# Scalable Automated Incrementalization for Real-Time Static Analyses

**Skalierbare automatische Inkrementalisierung für statische Analysen in Echtzeit**
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Diplom-Informatiker Ralf Mitschke aus
Groß-Gerau, Deutschland
Mai 2014 — Darmstadt — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Softwaretechnik

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Prof. Dr.-Ing. Reiner Hähnle

Tag der Einreichung: 29. April 2013
Tag der Prüfung: 26. Juni 2013

Scalable Automated Incrementalization for Real-Time Static Analyses
Skalierbare automatische Inkrementalisierung für statische Analysen in Echtzeit

Genehmigte Dissertation von Diplom-Informatiker Ralf Mitschke aus Groß-Gerau, Deutschland

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Prof. Dr.-Ing. Reiner Hähnle

Tag der Einreichung: 29. April 2013
Tag der Prüfung: 26. Juni 2013

Darmstadt — D 17

To my Parents

## Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den May 6, 2014

(Ralf Mitschke)

## Wissenschaftlicher Werdegang

|  |  |
|---|---|
| **Mai 2007 – Okt. 2013** | *Wissenschaftlicher Mitarbeiter am Fachgebiet Softwaretechnik* Technische Universität Darmstadt |
| **Okt. 1999 – Jan. 2007** | *Studium der Informatik – Dipl.-Inform.* Technische Universität Darmstadt |
| **Jun. 1998** | *Allgemeine Hochschulreife* Lichtenbergschule Darmstadt |

## Abstract

This thesis proposes a framework for easy development of static analyses, whose results are incrementalized to provide instantaneous feedback in an integrated development environment (IDE).

Today, IDEs feature many tools that have static analyses as their foundation to assess software quality and catch correctness problems. Yet, these tools often fail to provide instantaneous feedback and are thus restricted to nightly build processes. This precludes developers from fixing issues at their inception time, i.e., when the problem and the developed solution are both still fresh in mind.

In order to provide instantaneous feedback, incrementalization is a well-known technique that utilizes the fact that developers make only small changes to the code and, hence, analysis results can be re-computed fast based on these changes. Yet, incrementalization requires carefully crafted static analyses. Thus, a manual approach to incrementalization is unattractive. Automated incrementalization can alleviate these problems and allows analyses writers to formulate their analyses as queries with the full data set in mind, without worrying over the semantics of incremental changes.

Existing approaches to automated incrementalization utilize standard technologies, such as deductive databases, that provide declarative query languages, yet also require to materialize the full dataset in main-memory, i.e., the memory is permanently blocked by the data required for the analyses. Other standard technologies such as relational databases offer better scalability due to persistence, yet require large transaction times for data. Both technologies are not a perfect match for integrating static analyses into an IDE, since the underlying data, i.e., the code base, is already persisted and managed by the IDE. Hence, transitioning the data into a database is redundant work.

In this thesis a novel approach is proposed that provides a declarative query language and automated incrementalization, yet retains in memory only a necessary minimum of data, i.e., only the data that is required for the incrementalization. The approach allows to declare static analyses as incrementally maintained views, where the underlying formalism for incrementalization is the relational algebra with extensions for object-orientation and recursion. The algebra allows to deduce which data is the necessary minimum for incremental maintenance and indeed shows that many views are self-maintainable, i.e., do not require to materialize memory at all. In addition an optimization for the algebra is proposed that allows to widen the range of self-maintainable views, based on domain knowledge of the underlying data. The optimization works similar to declaring primary keys for databases, i.e., the optimization is declared on the schema of the data, and defines which data is incrementally maintained in the same scope. The scope makes all analyses (views) that correlate only data within the boundaries of the scope self-maintainable.

The approach is implemented as an embedded domain specific language in a general-purpose programming language. The implementation can be understood as a database-like engine with an SQL-style query language and the execution semantics of the relational algebra. As such the system is a general purpose database-like query engine and can be used to incrementalize other domains than static analyses. To evaluate the approach a large variety of static analyses were sampled from real-world tools and formulated as incrementally maintained views in the implemented engine.

## Zusammenfassung

Diese Arbeit schlägt ein neuartiges System zur einfachen Entwicklung statischer Analysen vor, deren Ergebnismenge inkrementell neu berechnet wird, um eine unmittelbare Rückmeldung der Ergebnisse in einer integrierten Entwicklungsumgebung (IDE) zu ermöglichen.

IDEs beinhalten eine Vielzahl an Werkzeugen die mit Hilfe statischer Analysen Probleme in Hinsicht auf die Qualität und Korrektheit von Softwaresystemen aufdecken. Allerdings sind diese Werkzeuge oftmals nicht imstande eine unmittelbare Rückmeldung der Ergebnisse zu liefern. Daher ist der Einsatz dieser Werkzeuge oft auf nächtliche Build-Prozesse beschränkt. Dies verhindert, dass Entwickler Probleme sofort bei deren Auftreten beseitigen können, also zu einem Zeitpunkt, an dem das Problem und die entwickelte Lösung noch gedanklich griffbereit sind. Um eine unmittelbare Rückmeldung zu ermöglichen, bietet sich die Technik der Inkrementalisierung an. Diese Technik nutzt die Tatsache, dass Entwickler meist nur kleine Änderungen am Quelltext der Software vornehmen. Daher können Analyseergebnisse schnell aufgrund dieser kleinen Änderungen neuberechnet werden.

Inkrementalisierung erfordert allerdings eine sorgsame Entwickelung der statischen Analysen. Daher ist die manuelle Entwicklung inkrementeller Analysen unattraktiv. Automatisierte Inkrementalisierung hilft dieses Problem zu beseitigen und ermöglicht es den Entwicklern solcher Analysen diese als Abfragen über die Gesamtmenge der Daten zu formulieren, ohne sich Gedanken um die Semantik der Inkrementalisierung zu machen.

Existierende Ansätze zur automatisierten Inkrementalisierung nutzen Standardtechnologien, wie deduktive Datenbanken. Diese bieten deklarative Abfragesprachen, benötigen aber auch große Mengen an Arbeitsspeicher, um die Gesamtmenge der Daten vorzuhalten. Diese Menge an Arbeitsspeicher wird permanent durch die Daten der Analyse belegt. Eine andere Standardtechnologie, die besser skaliert da sie wenig Arbeitsspeicher benötigt, sind relationale Datenbanken. Diese persistieren die Daten und nutzen nur wenig Arbeitsspeicher, um aktuelle Berechnungen durchzuführen. Allerdings benötigen relationale Datenbanken viel Zeit für Transaktion, die Auftreten wenn Daten in die Datenbank überführt werden. Die beiden Vorgestellten Technologien sind per-se nicht sonderlich geeignet, um statische Analysen in eine IDE zu integrieren, da die IDE bereits die Gesamtmenge der Daten vorzuhält und verwaltet. Eine Überführung der Daten in ein Fremdsystem ist daher ein unnötiger Arbeitsschritt.

Der in dieser Arbeit vorgestellte Ansatz bietet eine deklarative Abfragesprache und automatisierte Inkrementalisierung, benötigt allerdings nur das Minimum an Arbeitsspeicher, welches für die Inkrementalisierung vonnöten ist. Der Ansatz ermöglicht es statische Analysen als inkrementell gewartete Sichten zu deklarieren. Der zugrundeliegende Formalismus zur Inkrementalisierung ist die relationale Algebra mit Erweiterungen für Objektorientierung und Rekursion. Die Algebra erlaubt es das nötige Minimum an Daten zu bestimmen und zeigt auch, dass viele Sichten ohne das Vorhalten von Daten im Arbeitsspeicher wartbar sind. Diese Sichten werden "self-maintainable" (selbstwartbar) genannt. Um Selbstwartbarkeit für eine möglichst große Vielzahl an Sichten zu ermöglichen, wird eine Optimierung innerhalb der Algebra vorgeschlagen. Diese Optimierung nutzt Wissen aus der Domäne der zugrundeliegenden Daten. Die Optimierung arbeitet in ähnlicher Weise wie Primärschlüssel in Datenbanken, d.h., die Optimierung wird mit dem Schema der Daten spezifiziert und definiert welche Daten inkrementell in einem gemeinsamen Bezugsrahmen gewartet werden können. Dieser Bezugsrahmen ermöglicht Selbstwartbarkeit für alle Analysen (Sichten), die nur Daten innerhalb des gemeinsamen Rahmens korrelieren.

Der Ansatz ist als domänenspezifische Sprache in eine universelle Programmiersprache eingebettet. Die Implementierung kann als datenbank-ähnliches System aufgefasst werden, welches eine SQL-ähnliche Abfragesprache mit der Ausführungssemantik der relationalen Algebra besitzt. Das System als solches ist universell einsetzbar und nicht auf die Domäne der statischen Analysen beschränkt. Um den Ansatz zu evaluieren wurde eine Vielzahl unterschiedlicher statischer Analysen, aus bestehenden Werkzeugen, mithilfe unseres Systems als inkrementell gewartete Sichten nachimplementiert.

## Contents

## List of Figures

## List of Tables

# 1 Introduction

Static analyses are the foundation of many tools that enhance integrated development environments (IDEs) to catch software quality and correctness problems. Yet, they often fail to provide instantaneous feedback, making them unattractive for daily use by developers. Thus, many tools are restricted to nightly build processes and developers can not fix issues at their inception time, i.e., when the developed solution is still fresh in mind. Instantaneous feedback requires carefully crafted static analyses and is impeded by large amounts of analyzed data as well as the complexity of the analyses. This thesis proposes a framework for easy development of static analyses that are fit for daily use in an IDE.

The term *static analysis* refers to the processes of verifying (semantic) properties of a program based only on the knowledge of the source code (or compiled program), i.e., without executing the program. Traditionally, static analyses are found in compilers [AU77] and are concerned with optimizations for the compiled code, but also with error-checking using static *type systems*. Today the field of software engineering features a large variety of approaches that incorporate static analyses for defect assessment, including, but not limited to, detecting faults in languages with pointers and dynamic storage [DRS98], finding common bugs [HP04], checking style rules or coding conventions [Joh78], checking API usage rules [BR02], detecting architectural erosion [MNS95], or deriving (object-oriented) metrics for complexity [CK94].

Static analyses have been found to be good indicators for *fault-proneness* of software systems [NB05], i.e., for the probability of the occurrence of faults in the software [ZWN$^+$06]. Key findings are that static analysis defect density can be used as early indicators of pre-release defect density (i.e., estimates of system reliability) at statistically significant levels and can be used to discriminate between components of high and low quality.

*Instantaneous feedback* is a factor for increasing productivity in the software engineering process. Studies in industrial software practice find that faults are more costly to remove the later they are identified in the development process [BB01, BLS02]. In addition they find that (depending on the process maturity) up to 40 to 50 percent of effort in software projects can be spent on *avoidable work*, such as developers re-examining the relevant code and/or re-running expensive test cycles [SBB$^+$02]. Many software faults can be fixed at a lower cost or avoided altogether if detected earlier. Hence, instantaneous feedback can significantly improve software development productivity.

In an effort to move error detection into the earliest stages of the development process, many commercial and open-source static analyses tools provide feedback to developers inside the IDE, e.g., Checkstyle [Che13], FindBugs [Fin13], Hammurapi [Ham13], PMD [PMD13], and SemmleCode [Sem13]. Similar, many academic approaches feature an IDE integration [FLL$^+$02, HP04, YTB05, GYF06] or discuss integration as ongoing effort (e.g., [NB05, AHM$^+$08]). To achieve the best developer productivity, these tools should provide instantaneous feedback in *real-time* in the sense of "perceived without delay". Current IDE compilers typically provide feedback in real-time. Yet static analyses tools, in general, are not en-par with the real-time feedback an IDE compiler provides.

To achieve real-time feedback in the presence of large code bases, tools must use an *incremental* (re-)computation strategy of analysis results. An incremental computation utilizes the *principle of inertia* [GM99a], i.e., small changes in the input have only small impacts on the output. Thus, an incrementalized computation can deduce changes in the results very fast (for small input changes), in comparison to a re-computation over the full code base. Static analyses in an IDE favor incrementalization, since developers usually follow a manual edit-compile(-run/test) cycle that changes only small portions of the analyzed code base. Previous works on incremental program analysis have reported considerable performance improvements. A large amount of work has been done in the field of data flow analyses; a comparison is provided in [BR90]. To a smaller extent work has been performed in providing incrementalized pointer aliasing analyses [YRL99, VR01]. However, all the above works are manually written incrementalizations with specialized algorithms and data structures for incremental maintenance of the targeted problem. Manual incrementalization can be an acceptable approach for standard analyses. However, tools such as FindBugs or PMD feature many static analyses, e.g., Findbugs detects over 400 "bug patterns", and incrementalizing them all manually is a very time consuming and error prone process. Hence, an approach that provides automated incrementalization has considerable benefits when writing analyses, especially analyses that are specific to particular domains or frameworks.

The main benefit of automated incrementalization – not only for domain specific analyses – is a *reduction in the complexity* of developing the analyses. The developer only has to provide data structures and algorithms written with the full data set in mind. Hence, there is no need to manually design data structures that contain intermediate results of the previous computation. Intermediate results become a problem quite quickly. For example, an analysis that reasons over subtyping in an object-oriented program must maintain a data structure for the inheritance hierarchy or an analysis for inter-procedural data flow must maintain a data structure for the call graph, i.e., the calls between procedures where an edge represents a call and a node represents a procedure. Such data structures must be kept up-to-date which adds additional complexity and is error prone, since there incrementalization can require a specific order of updates for correctness, i.e., some intermediate results may depend on others. This problem becomes larger still, when using multiple incrementalized analyses that also depend on one another. For example if many static analyses require the inheritance hierarchy, the latter should only be computed once and then all dependent analyses should *re-use* the results. In a manual incrementalization approach such dependencies must be captured by some form of scheduling that runs each analysis at the right point in time, i.e., when all prerequisite analyses or intermediate results have concluded. An automated solution can alleviate the burden of scheduling from the analysis developers.

There are two distinct approaches to automated incrementalization, which have different qualities. The first utilizes a declarative specification (or query) of the static analysis, which can then automatically be executed with an incremental semantics instead of a full computation of the query (e.g., [EKS$^+$07]). The second is a rewriting technique that takes as input an imperative non-incremental program and tries to find an equivalent incremental program, which is also known under the term *dynamic programming* ([LT94]). In the following the term automated incrementalization is used to denote the first approach. The main benefit of the declarative specifications is that the primitives in a query have a distinct non-incremental semantics, which can automatically be translated to an equivalent incremental semantics. In contrast, the *dynamic programming* approach must find incremental semantics for arbitrary imperative programs. An additional benefit of the declarative specification is the easy applicability of query optimizations.

A first step towards automated incrementalization of static analyses was done in [SR05a] and [EKS$^+$07] on the basis of using incremental tabled logic programs [SR03]. The programs operate on a logic representation of the source code and analyses are defined as predicates over this representation. The authors of [SR05a] describe an implementation of incrementalized pointer aliasing analyses, while the authors of [EKS$^+$07] consider a broader range of incrementalized analyses. Note that logic programs are considered a valuable technique for declarative specification of static analyses in their own rights, i.e., apart from incrementalization [DRW96].

Logic language implementations (e.g., XSB[1]) have been noted for lacking scalability to larger code bases by Hajiyev et al. [HVdM06], since they require the logic representation of the source code to be completely in main memory. Large in this context means "> 10000" classes to be analyzed, which for example applies to the standard Java class library (JDK) or the code base of the Eclipse IDE[2], but also

1 http://xsb.sourceforge.net/

2 http://www.eclipse.org

to smaller projects that are analyzed together with required libraries. For such code bases a large amount of main memory is required simply for the base logic representation, e.g., over 1 gigabyte for the JDK. Even more memory is required to run the analysis, e.g., a machine with 4GB main memory was required for an analyses on the Eclipse code base in [HVdM06] and they considered only a subset of the whole program (concretely the parts that deal with dependencies). Hajiyev et al. propose to use standard relational database management systems (RDBMS), to overcome the scalability issue. An RDBMS can handle large representations, since relations are persisted and only a subset of the data – relevant for evaluating the current analysis – resides in main memory. Yet, this introduces a new bottleneck, since an RDBMS requires considerable transaction time for storing data in the database. Bottlenecks in transaction processing of RDBMSs have, for example, been discussed in [SMA+07] (especially for the traditional off-the-shelf RDBMS systems, such as Oracle, DB2, etc.). The impact for static analysis can be exemplified based on numbers reported in [HVdM06]. In these experiments transition of the data to an off-the-shelf RDBMS can take several seconds for a single class. Note that in [HVdM06] the fastest performing RDBMS requires approx. 45 minutes (and the slowest 1.5 hours) to store approx. 10000 classes. Thus the average transaction time per class is already 270 milliseconds and the work in [HVdM06] does not store the entire code base.

In summary, standard database technologies, such as RDBMS' or deductive databases (i.e., logic languages), are an ill match for incrementalized static analyses in an IDE; the RDBMS' for their high transaction time and the deductive databases for the high amount of memory required. In essence, there are only two properties of these systems that we wish to keep: (i) the declarative query language, for its ease of defining the analyses and allowing automated incrementalization and (ii) the in-memory computation model, for its speed in computing analyses results. In particular there is no need to retain an in-memory representation of the whole program, or move the representation of the whole program into a relational database. The IDE manages source code files or compiled bytecode files that already contain all the necessary information. Moving all the data into main memory – or to a relational database – (i) duplicates the information and (ii) maintains more information than is strictly necessary, since many changes are easily re-computed by re-analyzing the underlying source code files.

### Thesis

The goal of this thesis is to provide a comprehensive approach for automatically incrementalized static analyses that can produce answers in real-time and overcomes the scalability issue for main memory. The main contributions of this thesis are the definition and implementation of a database-like system and the definition of static analyses as a set of composable *views* in the system. The views are defined via declarative database queries and their results are incrementally maintained in-memory. However, the system retains in-memory only the nominally necessary information to incrementally maintain the views results.

The proposed approach defines views via relational algebra, which allows us to derive minimal measures for the information required to maintain the views by studying previous works. For example, the concepts of incrementally maintained views in relational and deductive databases [GMS93] and self-maintainable relational views [BCL89, GJSM96] provide a solid background. These works show that certain views can be maintained without storing any information at all.

Addressing the problem of incremental computation via relational algebra has a number of other advantages: (i) it bases the definition of incrementally updated analyses on a formal basis that is suitable for optimizations; (ii) it provides a compositional language for formulating and reusing incrementally maintained analyses; (iii) it allows to reason about performance issues – time as well as memory – on a "per operator" basis; and, (iv) it provides an extensible framework via the addition of new operators.

We implemented the proposed system as an embedded domain specific language (EDSL) inside a general purpose programming language (Scala[3]). The analyzed code base is represented as relations (defined in the EDSL), which contain objects that represent the entities in the code base, e.g., classes, methods or instructions. The representation is directly defined via classes in the host language, hence, no external specifications must be provided. Concretely, we implemented a representation of Java bytecode, over which we define static analyses. However, the results easily apply to static analyses of other programming languages and source code representations.

Using an EDSL has many advantages: (i) source code (and Java bytecode) parsers are written in general purpose languages and usually produce a suitable object representation of the entities in the code base. This representation can be reused with minimal integration, i.e., arranging them as relations in the EDSL, without transactional overhead for transferring entities to a different system; (ii) computations on objects can be performed by functions declared in the host programming language, hence, the EDSL retains the full expressivity and execution speed of an object-oriented programming language; (iii) the embedding can be made type-safe by reusing the type checking facilities provided by the host language.

[3] http://www.scala-lang.org/

## 1.1 Contributions of this Thesis

Previous works have considered incrementally maintained static analyses for the sake of increased runtime performance. This thesis is the first work to consider an automated incremental maintenance approach not only for runtime aspects, i.e., producing results in the shortest possible time, but also for the suitability in a real developer environment, i.e., the analyses may not excessively consume main memory in a developer's IDE. The following list gives an overview of the contributions of this thesis, which are then each discussed in detail in the following sections:

- The concept of incremental maintained database views is applied to the domain of static analyses, with the focus on self-maintainable views to allow scalable in-memory computations.

- The approach is implemented as an event-driven in-memory database for incremental view maintenance, which in itself is applicable beyond the scope of static analyses.

- The traditional database concepts for incremental maintenance are extended to enlarge the range of self-maintainable operations.

- The approach is evaluated in three different static analyses scenarios:
  - Lightweight analyses, i.e., bug and code smell detection based on heuristics; in addition object-oriented metrics are considered.
  - Bug and code smell detection using control and data flow analysis.
  - Architecture compliance checking between dependencies in source code and intended dependencies defined in high level architectural descriptions.

- The incrementalization of the architecture compliance checking approach allows us to define a novel modular specification language for the intended architecture. The modularity is furthered by the ability to incrementally check affected architectural modules. Without the incrementalization a modular approach was infeasible, since the changes required extensive re-computations for many modules. The latter is due to the fact that architecture compliances checking in itself is a whole program analysis, even when we modularize the specification into digestible parts that treat the rest of the program as a black-box.

### 1.1.1 Self-maintainable Views for Memory Efficient Static Analyses

Self-maintainable views stem from traditional relational databases and build on the observation that – for certain views – the result of a maintenance operation, i.e., insertion/deletion of a resulting tuple, can be determined solely based on the data contained in the input tuple. An example for a self-maintainable (domain specific) static analyses is the detection of explicit method calls to Java's garbage collection facility. Such calls are, for example, marked as a performance issue in FindBugs, since memory management can be time consuming (depending on the implementation garbage collector). The view ranges over all instructions in the source code, but is self-maintainable due to the fact that all relevant information resides in an instruction tuple, i.e., whether the instruction is a call and which method is called. Note, that w.r.t. the called method the analysis is self-maintainable, since it simply compares the call to a fixed set of methods, e.g., `System.gc()`.

The maintenance of the above exemplified view can be illustrated as follows. Editing of a method triggers a deletion of outdated instruction tuples and the insertion of new instruction tuples. Since the above view is self-maintainable, a deleted tuple can simply be re-checked for satisfying the above declared conditions of the view. In case of a positive match the tuple is removed from the results, where it is guaranteed to reside, since deletion always entails that a tuple was added at some prior point in time. Likewise, an added instruction tuple is checked and added to the results in the case of a positive match.

The major benefit of self-maintainable views is that they do not require to retain data of the underlying relation in memory, i.e., the data is not required to be permanently held in memory for subsequent computations. For example, the above view is defined over a relation containing – conceptually – all instructions in the analyzed source code. However, the view never requires to look up data in this relation, but merely processes added and deleted tuples. Thus, there is no need to retain the relation over all instructions in memory.

Note, that traditionally view maintenance is used to *materialize* (store) the results, which can be regarded as a form of caching for fast subsequent access. Materialization is traditionally linked to persisting the data in the database. However, in the course of this thesis we will use the term *materialize* to convey that the data is permanently required in memory.

To fully exploit the possibilities of self-maintainable views, the concept is extended to an event-driven approach that omits materialization of results completely and merely passes the added or removed results to interested clients. For example, in the case of an IDE integration the client would be a respective facility for visually displaying source code errors (or warnings), which in most modern IDEs will feature its own storage of the displayed data. Hence, there is no need for the view itself to provide storage – in terms of memory – and, thus, a self-maintainable view is completely free of materializations.

Another benefit of using self-maintainable views is that many views, while not being completely self-maintainable, filter out a large number of elements before they require any data in memory. Filtering is the general principle in the above exemplified view and is also in general self-maintainable. Hence, such views consume only a nominally required amount of memory, i.e., of all the tuples in the underlying relation only those relevant to the computation are materialized in memory.

### 1.1.2 An Event-Driven In-Memory Database for Incremental View Maintenance

As a technical contribution, we implemented our approach as an event-driven database-like system. The implementation provides incrementally maintained versions of all standard relational operators, as well as extensions to deal with complex objects that contain structured data in collections and extensions for recursion. In addition an SQL-inspired surface syntax is provided to define views. Such as it is, the system can be understood from two perspectives: the database perspective or the programming language perspective.

Form the database perspective we used the term *database-like*, since our system does not provide the standard services of traditional databases, e.g., transactionality for concurrent access. Nevertheless, we will use the term database throughout this thesis, after clarifying the distinguishing properties of the system here. To provide maximal memory efficiency, the database is event-driven, i.e., tuples are inserted or deleted via events and self-maintainable views only process events, rather than looking up data in stored tables. Data is materialized, i.e., permanently kept in memory, only when absolutely necessary for the incremental computation of a view's results. Due to the event-driven approach, the database is not per-se developed for ad-hoc queries, i.e., user supplied (SQL) queries that answer specific "one-time" questions over the relations stored in the database. Such questions require that any data that is of potential interest to users is available at all times in the database. Since potential interest allows no careful planning an RDBMS system stores all data. Instead, the incrementally maintained views can be thought of as data processors that maintain a fixed set of queries, which allows the database to plan and optimize the materialization of data in memory. Ad-hoc queries are still possible and even advantageous for some static analyses scenarios that need to define queries dynamically, as will be shown in the case study on architecture compliance checking. Such queries must, however, be planned during the design of the database relations, by declaring respective relations as materialized in the database. Naturally, the planning should also include a careful assessment of the amount of data necessary to be retained in memory. Furthermore, it is important to note that we do not require to backup data as traditional in-memory database do. The data that enters the database, e.g., the source (byte-) code files are already the backup, which is persisted by the IDE.

From the language perspective the system can be understood as a rich framework for defining computations over correlated collections, where we use relations as collections that do not really store the data but only propagate events. Note that from the end-user perspective the events are not visible, i.e., the SQL-like queries define how relations (collections) are correlated, but the automated incrementalization takes care of propagating the right events to obtain the correct results.

Regardless of the perspective, the system can be reused in other contexts than in static analyses. However, it is tailored towards a specific scenario, i.e., one where we have a large persisted set of data in a – possibly – domain specific format such as Java bytecode and want to define incrementalized computations over this data. That said; the embedded database can be helpful to incrementalize any approach that receives a large set of structured input data and performs a deductive transformation, i.e., a query, to structured output data. This can include traditional compilers that transform abstract syntax trees to compiled programs.

### 1.1.3 Extension of Self-Maintainable Views

Based on the observation that self-maintainability is enabled when all data required to compute a result is locally available inside the event for inserting or deleting a tuple, the approach can be broadened to allow a greater range of self-maintainable views. The key idea is to utilize domain specific knowledge that guarantees for all tuples that are relevant to a given view, to be inserted or deleted together. In this case the extension allows self-maintainability for views that correlate information across several tuples.

To illustrate the extension consider the Eclipse IDE that features an incremental compilation process with a granularity of entire classes. One (optional) analysis performed by the Eclipse compiler is to check that Java classes that implement a custom `equals()` method also provide a matching `hashcode()` method[4]. For the granularity of entire classes the results can be computed without any intermediate data structures, since the new version of the class is passed entirely to the analysis

[4] The Java specification states that equal objects must have equal hash codes, which is for example required for the correctness of using the objects in Java hash maps.

during an incremental update. Hence, all methods of the class are inserted together and the analysis can check whether tuples for both custom method implementations are present.

In the context of manually incrementalized static analyses it is very natural to perform analyses as the one exemplified above, without storing any intermediate data. Since the manual incrementalization has knowledge about the extent of a working unit, e.g., a class and all therein contained information, it can simply traverse the in-memory representation of the working unit to produce results. The argument also applies to smaller units, e.g., an intra-procedural data flow analysis can simply traverse all instructions in a method, since they are all contained in the same working unit.

The proposed extension can be thought of as enabling input granularity considerations – found in manually incrementalized approaches – for an automated incrementalization. In a traditional database approach such information is lost, since all entities of a certain kind, e.g., all methods or all instructions, are flattened into a single table to allow a unified reasoning over all said entities. For example reasoning over all methods, regardless of the class they are contained in.

The extension is provided in a generalized manner and, hence, not only applicable to static analyses. In short, the granularity can be convened via annotations on classes that define objects in the database. This treatment is similar to declaring primary keys on schema entities in a relational database. Any view that correlates tuples based on at least one annotated property is guaranteed to receive all insertion events or all deletions events of relevant tuples together.

The extension does not come without a cost. In essence there is a space-time trade-off. On the one hand, not storing intermediate results saves memory, especially for static analyses that correlate single instructions. On the other hand, updates to the database can now only be as fine-grained as the granularity guaranteed to the self-maintainable views. Coarser granularity means that many, possibly computation extensive, update operations must be (re-)performed. For example, if the granularity is at the level of classes, this means that if a single method in a class changed, all other methods are updated as well. Theoretically, the expense can be quite high, for example, if data flows for these methods need to be recomputed. However, there is no empirical data on how this affects performance in a real system, which is, hence, provided as part of this thesis.

### 1.1.4 Case Studies of Incrementally Maintained Static Analyses

To evaluate the approach three scenarios are considered that fall under different categories w.r.t. the characteristics of the analyses, i.e., computational complexity of the queries and materialized memory requirements. For all case studies the following points are evaluated:

**Memory Materialization** The overall memory consumed by permanently materialized data is evaluated for the queries in each case study. Furthermore, to evaluate the proposed extension for self-maintainable views, the memory consumption is compared with and without the extension.

**Non-incremental runtime** The incentive is that incrementalization does not amortize when the initial computation of results is exceedingly more expensive than a non-incremental computation. To evaluate the approach the incrementalized computation is compared to an implementation of the queries in the same language (i.e., Scala), yet using a classical traversal over the object structure of the source code.

**Incremental runtime** The runtime is measured across a large set of incremental changes that were recorded in a developer's IDE over 12 hours of work time. The measurement provides insight into how the approach behaves in a real-world setting. In addition the incremental runtime is measured with our proposed extension for self-maintainable views at the class granularity versus a method granularity. The incentive being that memory consumption can be decreased for the coarser granularity, yet the finer granularity – intuitively – yields better runtime performance. However there is currently no performance data that quantifies the degree, i.e., orders of magnitude, of the better performance and measures whether the coarser granularity is still fit to be considered real-time, i.e., executable within a bound of (a few) hundred milliseconds.

### Lightweight Analyses

In this category bug finders and code smell detectors utilize simple heuristics that use structural queries or evaluate simple instruction sequences, without considering control and data flow. Computationally these analyses are not very complex, since they largely consists of finding a specific set of elements and computing only a few correlations to other elements, e.g., finding classes with a custom `equals` method determining that a custom `hashcode` method exists. Another kind of analyses in this category are metrics, which typically perform structural queries, but have the additional characteristic of using aggregations, e.g., counting the number of entities with specific properties, such as incoming dependencies. Concerning memory requirements the analyses in this category are hard to generalize, since the analyses elicit quite diverse behaviors. Yet, as will be shown in the course of this thesis, in general their materialized memory requirements are quite low.

The lightweight analyses considered in the evaluation perform analyses taken from real-world tools. A large set of analyses stem from the tool *Findbugs*, which is a popular open-source bug-finding tool that analyzes Java bytecode. The metrics are sampled from the open-source tool *Metrics*[5], which is inspired by the object-oriented metrics found in [HS96] and [Mar03].

### Data Flow Analyses

A class of more concise bug finders utilizes data flow analyses to detect malicious code. Data flow analyses can answer a wide variety of questions such as uninitialized variables, reaching definitions or “live” variables (c.f. [NNH99] for a good overview of the topic). As part of the evaluation a basic form of intra-procedural data flow analysis is implemented, that determines the effect of instructions on an abstraction of the variables (and the stack) in the analyzed program.

The analysis is quite complex since it requires to iterate over all instructions inside a method to determine the static effect of instructions on an abstract representation for the state of variables and – in the case of Java – the stack. Hence, the inherent computational complexity stems from the fact that the instructions make up by far the largest portion of data in Java programs. Typically, the analysis also considers control flow, which describes all possible execution paths through a given program. Control flow is an additional source of complexity, due to the fact that it requires a treatment of cycles (e.g., loops) in the program. Cycles require a fixpoint computation to evaluate all possible abstract states that can arise through repeated application of the instructions in the cycle.

The results of the data flow analysis are used to encode several real-world bug detectors found in the tool *Findbugs*, which uses a similar form of data flow analysis. The resulting abstraction of the data flow can be quite cumbersome to use in the final Findbugs analyses. Simpler intermediate representations have been proposed for easier reasoning over data flows, for example, a 3-address based representation in [VRCG$^+$10]. However, transformation to such a representation requires an analysis similar to the one presented in this work. Hence, the presented analysis can be seen as a first step to evaluate the wide range of possible data flow analyses. Most importantly, the analysis is automatically incrementalized and due to the proposed extension also self-maintainable.

[5] http://metrics.sourceforge.net/

### Architecture Conformance Checking

In this analysis architecture specifications provide high level abstractions – termed ensembles – that group source code entities which conceptually belong together, as well as a set of architecturally intended dependencies between such ensembles. The structural dependencies found in the code base are then checked for compliance with the intended dependencies, based on the membership of code entities in ensembles.

Architecture compliance checking requires a global correlation of all source code dependencies. Such a correlation has a computational complexity comparable to the lightweight analyses, since only a few correlations are done in terms of operators. Yet the amount of data considered in this analysis is higher, due to the global nature of the correlation, which has an impact on the required computation times and more importantly also on the memory required for incremental maintenance.

Another peculiarity of this scenario is that the architecture specification itself is treated as an interactive system, i.e., architecture specification can be modified and are, hence, incrementalized. Thus, the database not only receives events triggered by editing source code, but also by editing the architecture specification. This scenario is unique, not only because two different sets of data are incrementalized, but because the architecture specification dynamically creates and modifies views over the source code, i.e., as ensembles.

### Summary of Results

The case studies show that the materialized memory for all sampled analyses stays below the memory required to materialize the data of the entire program. Compared to a materialization of the entire analyzed program in a deductive database the memory stays within boundaries of 5% - 10% with the exception of the architecture analysis. In the latter case a heavily indexed materialization took still only 20% - 25% of the memory required for an un-indexed materialization in the deductive database. The non-incremental runtime is comparable to our reference implementation in the majority cases. The computationally more complex analyses are approx. three times slower for computing the initial results of the incrementalization, which, however, quickly pays of (i.e., after three runs). The incremental runtime is in general fast and the majority of changes can be computed in less than 10 milliseconds and requires only 20-30 milliseconds for larger changes. Larger changes in our case means that we removed and added 20-30 methods with a total of approx. 2 000 - 3 000 removed (and the same amount of added) instructions. Finally, the class granularity can definitely be considered a good option for further memory improvement, albeit the method granularity offered much optimization potential for a larger number of queries. The runtime increase is visible, yet the majority of incremental changes for individual classes stayed within the time frame of 10 milliseconds.

### 1.1.5 Modular Definition and Checking of Source Code Against Architecturally Intended Dependencies

Existing approaches in architecture conformance checking require the whole architecture and its intended dependencies to be specified in one single specification. Based on the experience gained during a case study – performed as part of this thesis – w.r.t. modeling the architecture of real systems (e.g., Hibernate [BK04]), such approaches do not scale for large architectures. Scalability is an issue when treating large systems, i.e., with many architectural elements and dependencies, where architects want to focus on relevant parts of the system. Providing focused views is quite challenging, even when compact notations are used, e.g., dependency structure matrices [SJSJ05].

As part of this thesis a new approach to architecture conformance checking is proposed, that enables architects to define intended architectural dependencies in modular units termed slices. The incrementalization of the architecture conformance checking is an enabling factor for this approach and automation of the incrementalization has the advantage of easy development. Incrementalization is enabling such an approach, since slices are designed as independent units in terms of described dependencies and are required to be independently checked. Modularizing a non-incremental conformance check is not efficient, since architecture analyses are whole program analyses that consider a large amount of data and, hence, are expensive to run multiple times for each slice. Thus, an incremental approach is far better suited to and has to compute only small updates to slices. The automation eases the development of the analyses considerably, since it alleviates the need for a complex implementation of the update logic.

### 1.1.6 Publications

The following publications were created in the context of the research performed for this thesis:

1. R. Mitschke, A. Sewe and M. Mezini. Magic for the Masses: Safer High-level Low-level Programming through Customizable Static Analyses. In *Proceedings of the 1st workshop on Modularity in systems software*, *MISS '11*, ACM, 2011

2. R. Mitschke, M. Eichberg, M. Mezini, A. Garcia and I. Macia. Modular Specification and Checking of Structural Dependencies. In *Proceedings of the 12th International Conference on Aspect-oriented Software Development, AOSD '13*, ACM, 2013

3. P. Giarrusso, K. Ostermann, M. Eichberg, R. Mitschke, T. Rendel, C. Kästner. Reify Your Collection Queries for Modularity and Speed! In *Proceedings of the 12th International Conference on Aspect-oriented Software Development, AOSD '13*, ACM, 2013

## 1.2 Organization of This Thesis

The remainder of this thesis is structured into the following chapters:

**Chapter 2 – Background**
discusses the history and the applications of static analyses. Afterwards a general overview of different lines of database research is given and each is discussed w.r.t. the relationship to this thesis. Finally, the concept of incremental view maintenance is discussed in detail. The concept of incremental view maintenance is introduced via an illustrative example. In addition an overview of the design space for the problem of incremental view maintenance is provided and a survey of existing techniques and algorithms is given.

**Chapter 3 – Language-Integrated Database with Incremental View Maintenance**
introduces the event-driven in-memory database in detail. The chapter gives an introduction on how data is defined and manipulated. Then the supported relational operators are discussed; first w.r.t. their semantics and then w.r.t. their incrementalization.

**Chapter 4 – Query Language and Query Optimization**
discusses the SQL-style query language for the database. Furthermore the chapter introduces the optimizations to the database. First, traditional query optimizations are shortly discussed w.r.t. their added benefit in providing memory efficient incremental view maintenance. Then a novel optimization is introduced, that helps to further reduce the amount of memory required in memory, by introducing scopes for incremental computations. Finally, the chapter discusses issues in integrating the query language into a general-purpose programming language.

**Chapter 5 – Incrementalized Static Analysis Engine**
shows how the database is utilized to obtain a Static Analysis Engine (SAE). The respective data structures are introduced and example analyses are given that show how concrete static analyses are formulated using the SAE. Finally, the integration of the SAE into an IDE is discussed.

**Chapters 6-7 – Case Studies**
are presented to evaluate the performance of the approach w.r.t required memory storage and runtime. Chapter 6 discusses the lightweight analyses and the data flow analyses. Chapter 7 discusses architecture conformance checking.

**Chapter 8 – Conclusions and Future Directions**
concludes this thesis and discusses directions for future work to enhance the presented approach.

# 2 Background

This chapter first provides an overview of existing research in terms of static analyses and database systems. Both have been studied extensively since the 1970's and produced quite a large variety of related work. The presented overview is not intended to be comprehensive, but should rather serve as an outline of the main directions taken in these areas. Especially in the case of database systems, the chapter provides an overview w.r.t. different research directions that are closely related to the event-driven in.memory database presented in this thesis. Finally, the topic of incremental view maintenance in database systems has been researched at least since the 1980's. Hence, an overview and classification of the design space and a survey of incremental maintenance techniques is provided.

## 2.1 Static Analyses

The classical application of static analyses is found in compilers [AU77] that provide for example error-checking using static type systems, e.g., to detect mismatches between a procedure's formally specified parameter and the actual parameter. In addition compilers use data flow analyses during optimization, e.g., the live-variables analysis [Hec77] determines a variable to be live if it holds a value that is required in a subsequent computation; the optimizer knows that only variables alive at the exit of a block must be written to heap memory. The majority of compiler analyses are confined to a single procedure (*intra-procedural analysis*).

The traditional static analyses of compilers are conservative in order to retain *soundness*, where soundness in static analysis means that it does not miss any error in the program that can arise at runtime (cf. [Rus01]). Conservative in this sense means that the guaranteed properties are weak (yet easy to establish) compared to more useful properties. For example, there may be many paths through a procedure that do not correspond to any execution. Yet a static analysis can safely (conservatively) assume that all paths are executable [ASU86], which simplifies the problem and allows more efficient execution of the analysis.

With the wide adoption of languages such C and C++, the context of static analyses was broadened to diagnose faults in languages with pointers and dynamic storage. The focus in these analyses is to detect invalid pointer references, faulty storage allocations or usage of uninitialized memory [DRS98]. Such analyses typically consider the whole program (*inter-procedural analysis*). Analyses reasoning over pointers have to solve an *aliasing* problem, i.e., if any given two variables in the program can point to the same location in memory. These analyses operate on a model of the run-time state of the program and determine how the program manipulates this state. The model in any aliasing analysis must represent all memory locations via a finite number of objects. Since programs have many possible executions and many possibly used memory locations, a reasonable abstraction of all run-time states is used. In general such analyses have a trade-off between the abstracted-away state and the precision of the analysis (more conservative abstraction $\equiv$ less precise results). Due to the high number of states in a given program, such analyses are widely accepted to be very hard [LR91]. In fact many inter-procedural static analyses were shown to be *NP* hard (cf. [Mye81]).

Object-oriented languages are slightly more complex for inter-procedural analyses, since they rely on frequent method invocations with polymorphic subtyping. In general, a method is invoked on a variable (the receiver) and – due to subtyping and polymorphism – the runtime type of the receiver can yield different call targets. Since receivers are often objects passed as method parameters, the concrete runtime type of the receiver is also dependent on the context from which the enclosing method was called. Inter-procedural analyses can determine an approximation, i.e., a small set of possible classes that are concrete subtypes used throughout the execution. Such analyses have been proposed for optimizations in object-oriented compilers [DGC95], where statically known receiver types can be used to produce a compiled program that used fewer dynamic method dispatches. A broader class of analyses that use type information are *typestate* checkers, which have been applied to C [SY86] and also to Java [ASSS09]. Typestates can be seen as an extension to traditional types that include a notion of the state of a variable and the effect produced by invoking certain methods, which allows some temporal reasoning over the correctness of a program. In short, types define which operations can be performed on a variable and typestates when they can be performed. Typestate analyses can find a broad range of errors, such as uninitialized variables in languages with pointers, but also general adherence to contracts of APIs, e.g., reading from a file stream that has not been opened yet. Hence, static analyses can be used beyond the purpose of detecting programming errors. Adherence to contracts can have broad applications, such as finding security vulnerabilities (cf. [CM04] for an overview).

A different approach to error detection is pursued by [Joh78] (for C) and [AB01, HP04] (for Java). These approaches employ static analyses to find suspicious code patterns that have a high possibility to cause defects. The analyses are lightweight in the sense that they mostly rely on simple heuristics, and in some cases on intra-procedural data flow, but do not use inter-procedural analyses. In many cases these analyses are unsound, i.e., do not find all occurrences of a given class of errors, but try to identify the most common mistakes. For example Findbugs [HP04] features an “Open Stream” detector that finds Java in-/output streams, which are created in a method but not closed on all paths before the methods exits. Streams passed as argument to other methods are ignored. However, the lightweight analyses were shown to be useful and cover a good range of errors in real systems [RAF04].

Another set of – mostly – lightweight static analyses are software metrics, which provide a quantitative measurement of the complexity of the software’s design and implementation. Metrics in procedural and object-oriented languages have been proposed and studied as predictors for the fault-proneness of modules/classes [DMP02]. Metrics can be measured using very simple models, such as the number of lines of code, or using elaborate models, such as the cyclomatic complexity [McC76]. The latter is essentially a measurement of the complexity exhibited by the control flow graph of a method. In general, metrics are a very controversial topic, not only for the precision with which certain attributes of a software system can be measured, but more importantly how the measurements relate to the quality attributes, e.g., program defects, that they are supposed to model [KMB04]. While there are quite positive results (e.g., [DMP02]), some metrics, e.g. cyclomatic complexity, do not have statistically relevant correlation to defects [Hat12].

## 2.2 Database Systems

Relational database management systems (DBMS) are a technology that has been developed over several decades, starting with the seminal work by Codd [Cod70] on an algebra for managing data as a set of relations. Among the standard services provided by DMBSs are (i) *persistent storage* of very large amounts of data – not fit to retain in memory – and keeping it consistent over a long period of time; (ii) users can query the data and modify the data, using appropriate database languages, often termed *query language* and *data-manipulation language*; (iii) DBMSs provide *concurrent access* to the data, for queries as well as for modifications and (iv) *transactionality of modifications*, i.e., modifications are guaranteed to be performed in *isolation*, meaning independently of each other, and to be *atomic*, meaning each modification must either succeed completely or no changes are made at all. A good overview on the general concepts found in database systems is given in [UGMW01].

The concept of transactionality is now shortly discussed in detail, since it is the principle cause of long runtime for storing data in the database (with negative effects as discussed in the introduction and seen in [HVdM06]). Transactionality guarantees consistency in case of events such as system failure, violations of consistency constraints, network time-outs, and more. To support transactionality the database has a component for monitoring transactions and taking steps to recover from failed transactions. In a commercial DBMS this is typically done via *logging* [Gra92]. Logging records modifications to the data that would be made to the data. The modifications are only committed (i.e., stored) to the database in the case of successful transactions. Logs are typically persistent, so that in the case of a system failure, e.g., power outage (not including hard-disk failure), data is guaranteed to be retained. Hence, logging imposes considerable overhead for a transaction.

In the following, general lines of database research are discussed. The focus here is on (i) *deductive database systems* and *logic languages*, since these provide a broad range of expressible queries – thus making them attractive for formulating static analyses – (ii) *main memory databases*, since memory is a constraining quality for static analyses; (iii) *active* and *real-time databases*, since they are architecturally closest to the event-driven nature of the approach followed in this thesis; and finally (iv) *object-oriented databases*, since the embedding proposed in this thesis is done in the context of an object-oriented language and, hence, has several commonalities to OO databases, e.g., the expressiveness of the query language.

### 2.2.1 Deductive Database Systems and Logic Languages

Deductive database systems have evolved out of early logic programming languages [Llo87]. Logic programs can be seen as deriving logical consequences from a set of known *facts* via a set of first order logic predicates (or *rules*). Facts are usually represented as predicates with constant arguments (termed *ground atoms*). For example, the knowledge that Greg is a parent of Sally can be represented as the fact *parent(greg, sally)*, where *parent* is the predicate with constant arguments *greg* and *sally*. Using facts, knowledge is represented *extensionally*, i.e., by enumerating all arguments for which a given predicate is true. Rules contain a *head* term that can be derived when the predicates in the *body* are satisfied, i.e., the *head* is implied by the *body* (typically written as *head:-body* in PROLOG-style notation). For example a rule of the form

$$p \text{ :- } q_1, q_2, \ldots, q_n$$

can be read as "$q_l$ and $q_2$ and ... and $q_n$, implies p". Rules represent knowledge *intensionally*, i.e., by declaring how it can be derived from already ascertained knowledge via logical consequences.

Deductive databases can be understood as relational systems with a richer query language. In this interpretation, the facts in logic languages are tuples in a database relation, for example the fact *parent(greg, sally)* is a tuple *(greg, sally)* stored in a relation *parent*. The rules provide a form of querying the relations that is much richer than query languages such as SQL. In particular logic languages can express recursive queries, which traditionally were not part of SQL and even today are only provided with limitations [ISO11]. For illustration, consider the rules below that compute the ancestors relation. Ancestors are immediate parents (2.2.1.1), or recursively all parents of ancestors (2.2.1.2).

$$ancestor(X,Y) \text{ :- } parent(X,Y). \tag{2.2.1.1}$$

$$ancestor(X,Z) \text{ :- } parent(X,Y), ancestor(Y,Z). \tag{2.2.1.2}$$

From the query perspective, it is important to note that both deductive and relational systems share the characteristic of being declarative, i.e., of allowing the user to formulate a query that expresses which data is required, instead of a set of operations that compute the data. However, due to their inheritance from logic languages, deductive databases have many characteristics that set them apart from relational database systems; [RU95] provides a very good overview. Important relational database characteristics such as persistence, updates to the relations, or even transactional guarantees were not part of the first deductive databases. In particular persistence turns out to be quite difficult, since the logic languages follow a “tuple-at-a-time” processing strategy, whereas persistence in relational databases utilizes the set-orientation of relational operations for efficient disk storage. In addition, the “tuple-at-a-time” processing is a very different query evaluation technique and has (to some degree) lower run-time performance. Hence, a lot of effort was spent to achieve better query evaluation techniques (cf. [SSW94]).

### 2.2.2 Main Memory Databases

During the 1980’s a large amount of works considered the upcoming availability of large quantities of main memory. The focus in these works was mainly on the following topics (a more detailed discussion can be found in [GMS92]): *Transaction processing* has been reconsidered, since transaction can be completed faster when performed in memory. Large quantities of fast running transactions can be efficiently executed in a serial manner, which removes the cost of concurrency control. Data *access methods* for disk memory have been reconsidered, since main memory databases can directly retrieve the data via pointers in memory. Alongside the data access, the *indexing methods* have been simplified, e.g., variants of linear hashing have been efficiently used to index unordered data. In addition, memory efficient indexing for ordered data has been proposed in the form of T-trees; balanced binary trees where every tree node can hold multiple entries. *Data representation* was reconsidered w.r.t. the advantages of using pointers to reference values and store recurring (large) values only once on the heap. *Query processing* was also reconsidered, since the traditional relational databases focus on minimizing disk access, whereas in-memory access requires a shift towards efficient computations. Main memory databases still provide *failure recovery*, which remains a database service that requires storage on hard-disks. In essence backups (also termed *checkpoints*) are performed, that can – to some degree – be decoupled from transactions and perform disk I/O operations in larger chunks. Although a lot of research was performed in the area of main memory databases, there are few commercial system, e.g., Starburst, MonetDB, TimesTen. Applications using these systems are deemed to be limited to certain areas such as telecommunications and few other industries that require high performance [HYX11]. Nevertheless, interest in big main-memory data stores has been renewed in recent years. Yet, this trend utilizes multiple machines to broaden the pool of available memory. For example, SAP's Hana[1] main-memory database can utilize a cluster of servers as a single main-memory pool.

## 2.2.3 Active and Real-Time Database Systems

*Active databases* enable reactive behavior inside a DBMS via the definition of events (rules) that execute user specified functionality when they are triggered [PD99]. Traditional databases required to write external application code that effectively polls the database in the case of events or to ascertain whether some conditions are met by the data and institute appropriate actions. Active databases move the specification of the behavior into the database system, which allows both centralized and timely processing of events. Among the frequently mentioned examples that benefit from active databases are computer-integrated manufacturing, air-traffic control or stock-market trading [BH95]. The majority of active database systems defines reactive behavior via event-condition-action (ECA) rules. The semantics is the following: when an event occurs and the condition is satisfied, then the action is executed. Note that conditions are optional or implicit in some systems [PD99].

ECA-rule processing comprises several complex functionalities and, typically, the complexity is addressed by one or several distinct database component(s) for ECA processing. The design-space of components is quite diverse and features several dimensions (cf. [PD99]). Active databases can support a variety of *event types*, e.g., based on data manipulation, time triggers, external agents, or even composition of other (primitive) events [BH95]. The generality with which events are treated must be met by a general component for *monitoring* events and *triggering* corresponding rules. The processing of rules in relation to transactions can be performed differently,

[1] http://www.sap.com/hana

e.g., *immediately* when the conditions are met inside a transaction, *deferred* to the end of the transaction or *detached* from the triggering transaction, i.e., in a separate transaction. Multiple rules can be triggered at the same time; hence, a form of – possibly parallel – *scheduling* for executing actions is required. Another important issue is the timeliness of completing scheduled rules. This last issue is also the main point of contact between active databases and real-time databases [BH95, Eri99].

*Real-time databases* are traditionally concerned with the completion of transaction under given time constraints. The underlying idea is that data values, e.g., stock market prices, have a time semantics and are valid only for a short time interval. There are several implications of data validity that real-time databases address (cf. [Gra92]). First, *temporal consistency* means that transactions must be scheduled such that time constraints are honored, i.e., a transaction with a deadline for the data has a higher priority than transaction without a deadline. Second, the deadline of a transaction can sometimes be met using only an approximation of the results. Hence, the timeliness of a transaction can be achieved by relaxing the consistency guarantees of a database.

The introduction of validity intervals for data and scheduling techniques for temporal consistency are the central theme of real-time databases. Indeed, it has been noted that several misunderstandings on the concept of real-time databases exist, the most infamous being: *[...] that real-time [database] systems are synonymous with speed.* [SSH02].

### 2.2.4 Object-Oriented Database Systems

With object-orientation (OO) being on the rise in the 1980's the relationship between objects-oriented programming languages and databases was explored. From the perspective of OO languages, databases offered the ability to persist large amounts of objects (cf. [AH87]). Hence, one of the defining qualities of early approaches was the ability to persist objects on hard disks and load them into main memory on-demand via an object identifier (OID) – in essence a pointer – and then manipulate and navigate the in-memory representation. From the perspective of database research, object-orientation offered a basis to lift the restrictions implied by the first normal form [Ken83]. The first normal form essentially states that a tuple consists only of atomic values, which disallows data such as lists which can have variable lengths for different tuples. New models were already proposed to incorporate hierarchically structured data (cf. [AB93]). In this respect, object-orientation provided a new and general model that allows objects to contain other objects or even aggregations, i.e., groups of objects contained in collections such as arrays.

There are many issues in providing object-oriented database systems, which are discussed shortly in the following paragraphs. The focus here is on the underlying data model and querying capabilities. These are the properties of database systems that relate to the integration of relational querying with an OO language in the proposed embedded database of this thesis. In general, other database issues such as transaction management and failure recovery have also received special attention in object-oriented database systems. A more comprehensive and detailed overview is provided in [BM91] or [ZCC95]. These works also provide an overview of implemented systems and their design choices. Note, that the concept of an object-oriented database has been standardized by the ODMG (Object Data Management Group) [Cat00] and object-oriented-like extensions have been incorporated into the SQL standard (cf. [EM99] for an overview).

#### Object identity

The technique of loading objects via an identifier from disk into main memory requires a notion of *object identity*. An object must be identified via a unique and immutable value throughout its entire lifetime, while the object's attributes may change. Object identity in databases shares some similarity to the concept of object identity in OO languages. In particular both have two notions of object equivalence: (i) based on pointer equality, i.e., two objects point to the same memory location, or (ii) based on value equality, i.e., the attributes of two objects represent the same values.

Object identity is a quite far ranging concept for classical relational systems, since previously data consisted of tuples that only had value equality. Tuples were to some degree identifiable via their primary key. However, object identifiers are more than primary keys. The latter describe the values (columns) in a tuple that contain the data for which each tuple is unique. However, the data may change during the "lifetime" of a tuple, as long as the primary key does not collide with other tuples in the database. In this case the row is now uniquely identified by different data and can not be referenced using the previous values. Object identity allows to uniquely reference the *same* object at any time, even if attributes changed. In addition two objects can have the same attribute values, yet have different identifiers.

Object identifiers are typically managed by the database system and not by applications. This simplifies data definition, since users do not have to emulate a unique immutable value for identity, and simplifies client code, since the insurance of uniqueness for object identifiers and the maintenance of referential integrity is quite challenging.

## Object-oriented data definition

OO databases allow the definition of *classes* with *methods* and *inheritance* which are novel in the database context, yet are inspired by OO languages and very similar to these concepts in the programming language sense. Classes specify the structure of objects in terms of attributes (i.e., instance variables) and an instantiation mechanisms, that allows to create concrete objects adhering to the structure.

Methods provide data encapsulation by specifying a set operations on the data contained inside the objects. Most systems also provide a notion of *types* – in the programming language sense – that allows to specify interfaces of the operations separately from implementations. The latter is typically provided in classes. Note, that many OO databases are *strongly typed*, i.e., every operation is defined with typed operands.

In a concrete database, the encapsulation of data is often a tricky design choice that stands in contrast to one traditional goal of databases, which is that all possibly relevant data can be freely queried and correlated in new ways. When employing encapsulation only methods are visible, whereas the data and the implementation remain hidden, thus only data accessible via the methods may be freely queried. Since the attributes (i.e., the data) contained in objects are often the primary working units for applications most systems provide work-arounds, such as system defined get/set methods for each attribute.

Inheritance is provided by database systems in the classical OO sense that allows classes to specialize functionality from their super classes. Classes can specialize other classes with additional functionalities or by substituting functionalities. Single and multiple inheritance can be provided with the same implications as in OO languages, e.g., conflicting definitions. Inheritance is typically taken as a nominal *subtyping* relationship. Types are checked for compatibility w.r.t. the inheritance based type hierarchy, i.e., an object of the subtype can be assigned to a variable of the supertype. OO database systems – like OO languages – support dynamic method dispatch (i.e., *late binding*), that chooses the most specialized version of a method for a given object at runtime based on the object's class.

## Object-oriented query languages

One of the most interesting points in the combination of OO features and databases is the impact of OO on the query language of the database system. The traditional relational query languages select sets of tuples from relations. In OO databases a query selects a set of objects, which is quite similar to selecting tuples in a relational database. Yet, OO databases provide a new querying capability compared to relational databases. Given an object (via OID), the database can directly access referenced objects (via OID references). Note, that – from a query perspective – the latter is comparable to an extension of the relational system for hierarchically nested values. For illustration consider the following query:

*Find all employees that work in a department with*
*a budget greater than 500 000 (€)*

In a relational database where employees and departments are stored in separate tables, this query requires a join; which can be expressed as follows:

$$\{v \mid \mathit{Employee(v)} \;\wedge\; \exists u.\, \mathit{Department(u)} \;\wedge\; \mathit{v.DepartmentId} = \mathit{u.Id} \;\wedge\; \mathit{u.Budget} > \mathit{500\,000}\}$$

The query introduces an additional variable $u$ and a join predicate ($\mathit{v.DepartmentId} = \mathit{u.Id}$), to find corresponding departments and restrict them w.r.t. the budget declared for the query. Object-oriented queries can simplify the query, by formulating the restriction on the departments directly on a nested value. A *path expression* is used to denote the corresponding department object and the join can be thought of as being performed implicitly. The resulting query is a simplified version – in the sense of being more concise – of the above query, since less explicit variables and joins are required; it can be expressed as follows:

$$\{v \mid \mathit{Employee(v)} \;\wedge\; \mathit{v.Department.Budget} > \mathit{500\,000}\}$$

Path expressions can be used to increase the conciseness of the queries, in the sense of introducing less syntactic elements into the query. However, they do not add expressive power to the language, i.e., it is possible to express a semantically equivalent query using additional variables and explicit join predicates. Most object-oriented query languages provide a syntactic notation similar to path expressions [BM91].

**Classes and querying**

Classes (and types) in a database provide a novel mechanism for querying data via *extents*, i.e., the set of all instances of a given type (including all subtypes). Extents are in essence a reification of the type information in the set-theoretic sense. That means objects are classified as members of a set for their given type and the sets adhere to the subtyping relation. For example, if the class $A$ is a subtype of the class $B$ (written as $A <: B$), then the extent for $A$ is a subset of the extent for $B$ ($\mathit{Extent}_A \subset \mathit{Extent}_B$). Note, the use of $\mathit{Extent}_C$ is an abbreviated syntax for the extent of class $C$, as there is no common syntax.

Extents can be explicitly maintained by the database, which incurs some overhead, i.e., new object instances must be inserted (or deleted instances removed) from the corresponding extents. Whether or not all class extents are automatically managed by the database depends on the database system. Some systems also defer the choice to schema designers, and let them declare classes that require a managed extent.

Classes with extents provide a new form of querying, where tuples of different data "layout" can be returned. For illustration consider a database that stores *employees* with subtypes *consultants* and *engineers*. The subtypes can each store specific data, e.g., bonus payments for consultants, yet both have a common set of data, e.g., a monthly base salary. There are two issues here w.r.t. querying the data. First, extents allow users to retrieve all objects regardless of their concrete type and lets users declare queries based on the common data, e.g., the salary. Second, the specialization gives rise to a form of coercion (i.e., *typecasts*), as found in OO languages when dealing with heterogeneous collections. Ohori et al. [OBBT89] provide a good discussion on the topic and the implications for static type inference. For illustration consider the following query:

> *Find all employees of any kind (i.e., including subtypes) that have a monthly salary higher than 5 000 (€) and if they are consultants their bonus payments are greater than 2 000 (€)*

In an OO database the extent of employees can be explicitly denoted, e.g, as $\mathit{Extent}_{\mathit{Employee}}$ and the query can be formulated as follows:

$$\{v \mid \mathit{Extent}_{\mathit{Employee}}(v) \ \wedge \ v.\mathit{Salary} > 5\,000 \ \wedge \\ (v.\mathit{classOf}(\mathit{Consultant}) \ \rightarrow \ v.\mathit{asClass}(\mathit{Consultant}).\mathit{bonus} > 2\,000)\,\}$$

The query requires no special treatment for the field *Salary*, since all objects share this attribute. However, to deal with specialization, e.g., finding out if an employee is a consultant and querying their bonus, additional functions are needed for testing objects for particular types and for "down casting" objects. In the example, we use the functions *classOf* and *asClass* to perform these tasks. In essence the function *classOf* checks whether a tuple is of the given class and the function *asClass* performs a type cast to *Consultant*. The logical implication (→) is necessary for correctness and performs the cast only if the *classOf* check is successful, since the cast will yield runtime errors if performed on instances of other classes (save subtypes of *Consultant*). In essence the use of logical implication means that the query language has to provide some form of `if-then` control structure.

### Classes in query composition and view definition

Using classes in queries interferes with the free composition of queries allowed in the relational model. The classical composition allows query results as input to other queries, which is simple if only atomic values are supported. When classes and objects are used, the question arises if the returned values are objects and of which class. The same argument applies to the definition of views, which are in general complex queries and, hence, it is not always clear what type (or class) the elements in a view have.

The answer is simple for filtering queries as the one exemplified above, i.e., the objects of the underlying relation(s) are returned and retain their class information. However, the situation becomes more complex when queries are allowed to select a subset of the attributes in an object. When trying to treat this set of attributes as an object instance, it is unclear to which class the instance should belong. Potentially, there is a defined superclass that has exactly the same attributes; then the selection could be mapped to this class. Yet, the selection might as well yield a set of attributes for which no class is defined in the database.

A common approach to solve this problem is to introduce restrictions that avoid this situation. The database can restrict queries such that they retrieve either only a single attribute or all attributes (i.e., the whole object), which is often referred to as *object preserving* in literature. Other databases treat arbitrary attribute selections as tuples containing complex values; however, these tuples then do not have any methods defined on their data.

A different approach, followed by Heiler et al. [HZ90], is to define views via a query together with a class (or type) for the resulting objects. This approach makes the (reasonable) assumption that views are typically planned for by a database user, who can then as well provide a class definition (including methods) for the resulting objects. The implication of this approach is that the previously existing relational operators, e.g., a join, require extension to be able to create new objects (with object identifiers) instead of tuples containing values.

The most sophisticated solution is provided by *MultiView* [KR98], which automatically creates and manages *virtual classes* for a predefined set of relational operators. Virtual classes denote classes that arise due to the semantics of a query, instead of being specified by the user, i.e., there is no class definition in the system. Note that the term is ambiguous and not to be compared with virtual classes in strongly typed OO languages. In these languages the term virtual classes denotes a technique for defining general parameterized classes, such as lists or sets, and can be understood as an alternative to generics [MMP89]. The maintenance of virtual classes in an OO database is quite complex. Two techniques termed *object slicing* and *dynamic restructuring* of the stored objects are used during the maintenance. The slicing technique ensures that there is no fixed data layout. In essence all attributes are stored as pointers to a data structure that holds the value together with a reference to the class currently associated with the value. Consider for example that employees have a name attribute. The slicing technique stores a pointer to a structure containing the value of the name and a reference to the employee class. A query can then select only the name and other attributes from an employee. In this case the dynamic restructuring re-classifies all attribute instances to point to a new, i.e., virtual, class that contains only queried attributes.

## 2.3 Incremental View Maintenance

Incremental maintenance is based on the idea that a *view* of the data in a database can be efficiently maintained by only re-computing relevant parts of the view when changes are made to data.

A *view* is a relation defined as a query – in terms of the relational algebra – over *base relations*. The base relations are subject to direct modification (addition/deletion/update of tuples). A view is not subject to direct modification. Instead the result of the view is derived from the base relations. Thus a view can be seen as a function from base relations to a derived relation. The function is traditionally recomputed whenever the data of the view must be accessed [Sto75].

A *materialized view* [GM99a] can be obtained by storing the result of a view in the database. In essence a materialized view can be understood as a cached copy of the result. Consequently the access to the result of the view is much faster compared to re-computing the view's function every time the view is accessed. Furthermore, other views defined on top of the materialized view can be evaluated more efficiently, since the results of a materialized view can be indexed. Materialized views have a large range of applications, the most prominent and widely researched being data warehousing [JG12], where views are defined over relations that may be stored in different (distributed) database sources. Other applications include replication servers, data recording systems [JMS95], data visualization, and mobile systems [GM99a].

The efficient updating of materialized views is called *incremental view maintenance*, or *view maintenance* in short. The basic idea behind view maintenance is that updates to a database are small compared to the overall size of the database and only parts of the views change in response to updates in the base relations. The idea was termed "principle of inertia" [GM99b]. This principle is only a heuristic and does not apply in all cases, e.g., if all entries in a base relation are deleted it may be cheaper to re-compute a view that depends on this base relation. Yet the principle of inertia applies in most cases, making incremental view maintenance a worthwhile effort.

**Example view**

To illustrate the problem of view maintenance, consider for example a database for administering a university. In the following two base relations are defined that contain data on the lecturers of the university (2.3.0.1) and the staff that is tenured and when the tenuring was granted (2.3.0.2). The *Lecturers* relation contains a unique identifier *StaffId*, a *Name* and the *Age* for each lecturer. The *StaffId*, which is also used to identify lecturers in the *Tenure* relation, which stores the according date when the lecturer was given tenure.

$$\textit{Lecturers(StaffId, Name, Age)} \tag{2.3.0.1}$$

$$\textit{Tenure(StaffId, TenureDate)} \tag{2.3.0.2}$$

Figure 2.1 depicts an example instance of a database with tables for the two relations – *Lecturers* (2.1a) and *Tenure* (2.1b) – filled with exemplary data. Not all lecturers have tenure, for example the second entry in 2.1a has no corresponding entry in the tenure table.

| *StaffId* | *Name* | *Age* |
|---|---|---|
| 1 | Marinescu, E. | 47 |
| 2 | Rieble, D. | 35 |
| 3 | Glaser, T. | 41 |

**(a)** *Lecturers*

| *StaffId* | *TenureDate* |
|---|---|
| 1 | 09/01/2012 |
| 3 | 04/26/2002 |

**(b)** *Tenure*

**Figure 2.1:** Base relations in a university administration database

Given the above relations a view can be defined via the following SQL query:

```
1 CREATE VIEW TenuredLecturers(Name, TenureDate) AS
2         SELECT Name, TenureDate
3         FROM Lecturers, Tenure
4         WHERE Lecturers.StaffId = Tenure.StaffId
```

The query defines a new relation (line 1) that contains the lecturers *Name* and *TenureDate* (selected in line 2) from the respective base relations (line 3). To ensure that only tenured lecturers are contained in the view, the tables are correlated via a *join* (line 4) of the respective *StaffIds*.

A materialized view fully evaluates the join once and stores the result. Hence, applying the materialized view method on the data shown in Figure 2.1 would create the table shown below in Figure 2.2 as a result.

| *Name* | *TenureDate* |
|---|---|
| Marinescu, E. | 09/01/2012 |
| Glaser, T. | 04/26/2002 |

**Figure 2.2:** Materialized view of *TenuredLecturers*

Subsequent queries of the view are very quick, since they merely consist of reading the results stored in the materialized view. However, modifications of the base relations *Lecturers* or *Tenure* require examining the materialized view and determining if and how the view must be updated.

**General view maintenance algorithm**

The general algorithm that is responsible for incremental view maintenance can be summarized as follows. Consider $Q$ as a query over the base relation $R$, then $V$ is the (multi-)set consistent with the evaluation of $Q$ on $R$, i.e., $V \equiv Q(R)$. A set of modifications to the base relation $R$ consist off additions ($\Delta_R^+$) and deletions ($\Delta_R^-$). Updates to specific values of existing tuples, e.g., changing the name of a lecturer, can be modeled as a deletion (of the tuple with the old name) followed by an addition (of the tuple with the new name). The modifications change the base relation $R$ to

$$R' \equiv (R - \Delta_R^-) \cup \Delta_R^+$$

Naïve re-evaluation of the query computes $V' \equiv Q(R')$. Incremental view maintenance approaches are able to determine the net effect on the materialized view that stems from a change to a base relation. The effect is captured as additions and deletions to the view, i.e., $\Delta_V^+$ and $\Delta_V^-$. Hence, the view can be efficiently maintained by the following equation:

$$V' \equiv (V - \Delta_V^-) \cup \Delta_V^+$$

**View maintenance example**

For illustration consider following addition and deletion of tuples to the *Lecturers* relation:

$$\Delta^{+}_{Lecturers} = \{(4, \mathit{Foxon,\ R.}, 33)\}$$
$$\Delta^{-}_{Lecturers} = \{(3, \mathit{Glaser,\ T.}, 41)\}$$

The effect on the materialized view *TenuredLecturers* can be expressed as additions and deletions of tuples to that view. For the above example no new tuples must be added to the view, since the new tuple in $\Delta^{+}_{Lecturers}$ has no corresponding entry in *Tenure* The removal in $\Delta^{-}_{Lecturers}$ results in a removal to the view, since the lecturer with $\mathit{StaffId} = 3$ is not present in both base relations. Thus, given the above additions and deletions, the materialized view *TenuredLecturers* can be transitioned to a new database state *TenuredLecturers'* by the following formula:

$$\mathit{TenuredLecturers}' = (\mathit{TenuredLecturers} - \{(\mathit{Glaser,\ T.}, \mathit{04/26/2002})\}) \cup \emptyset$$

### 2.3.1 Design Space of Incremental View Maintenance

The design space for providing incremental maintenance of relational views can be classified along several dimensions that impact the efficiency of the approach. The following sections provide a classification of different design choices, exemplify their impact where appropriate, and provide an overview of the related work in the respective areas.

#### Maintenance Processing Strategy

There are three basic strategies for maintaining views (also non-incrementally). Once an update to the base relations occurs the following strategies are possible:

**Immediate** Updates are propagated instantly to all affected views. The drawback of the strategy is that a single update involves an unknown quantity of computation, i.e., depending on the number and complexity of affected views, the processing of the update can be costly. However, this strategy yields a minimal response time for accessing the results of the update in the maintained views.

**Deferred** Updates are not propagated but rather cached and the update is applied lazily once the view is retrieved the next time. This strategy moves the computational overhead for view maintenance away from the update and instead incurs the overhead during query answering. In addition data structures for caching the updates are needed, hence more space is consumed.

Periodic Updates are propagated at pre-defined intervals or during periods of low system activity. In contrast to the above two strategies, periodic updates loosen the consistency guarantees, i.e., at a given point in time the view may not be consistent with the actual values that were stored in the database, hence such views are also referred to as *snapshots*.

The majority of the literature on view maintenance considers the immediate strategy (cf., [GM99a]). Deferred view maintenance was for example studied by [CGL$^+$96] in a relational setting and by [AE99] for object oriented databases. A performance analysis of immediate vs. deferred update strategies in a persistent relational database is given by Hanson et al. [Han87]. They find that the choice for the best update strategy is highly application dependent. In particular the choice depends on the following parameters: (i) the ratio between data update and data retrieval operations, (ii) the selectivity of the predicates defining the view, (iii) the number of tuples written by each update (iv) the fraction of the data read (from hard-disk) during retrieval and (v) the cost of maintaining the sets of inserted and deleted tuples. They also show that for some views the performance of immediate and deferred maintenance are the same, i.e., the advantages and disadvantages of each strategy cancel each other.

The main advantage of deferred view maintenance is that fewer disk accesses to the stored copy of a view must be performed than during immediate view maintenance (pertaining to parameters (iv) and (v)). Hence, for main-memory database we can safely assume that immediate maintenance performs better (overall), since the main advantage of deferred view maintenance is not applicable

Periodic updates were proposed by [AL80] and provide an optimization for the performance of large distributed databases. In this scenario the advantage is that the large updates can be applied during periods of low system activity.

### Expressiveness of View Definition

The efficiency of the view maintenance approaches is largely determined by the database operators that need to be maintained. Hence, a database language that provides more sophisticated constructs, e.g., aggregation, allows the definition of views which are not as efficiently maintainable as other – more basic – constructs, e.g., selections. Intuitively, a richer view definition language requires specialized treatment of the richer semantics. The basis of relational algebra consists of select-project-join views (SPJ[2]) over set-valued relations. These views were studied by

[2] select-project-join queries are often referred to as *conjunctive queries* in literature

the earliest works on view maintenance [Pai82, SI84, BLT86, QW91] and SPJ views are supported by all successive techniques. The following extensions – loosely ordered from least to most expressive – have received special attention w.r.t. the incrementalization of their results.

**Duplicates** Most practical database systems are based on bag- (multiset-) valued relations. Hence, a treatment of duplicate values was introduced in [GMS93] for deductive databases and in [GL95] for relational algebra. In general, subsequent approaches are based on duplicate semantics. The results are also necessary for correct treatment of advanced operators, e.g., aggregations. For example, if the database for lecturers computes the average age – an aggregation – then the result in a set-valued relation would be wrong when at least two lecturers have the same age.

**Set Difference/Negation** Set difference operators are the algebraic prerequisite to formulating negations; as found in deductive databases. For example, consider a query that retrieves all lecturers that do not have tenure (from the example relations shown in the previous section). Informally, the query retrieves all lecturers and subtracts the (multi-)set of lecturers that do have tenure (cf. Sec. 3.3.5 for a formal treatment). The incremental maintenance of stratified negation for deductive databases was treated in [GMR95]. Stratified negation in essence states that negations may not be used inside recursions. A more formal definition of stratified negation can be found in [CGT89]. The algebraic treatment for relational algebra with duplicates was provided in [GL95].

**Aggregation** is typically used in data warehousing to build summary views, i.e., key performance indicators over the underlying data, such as average number of sales per marketed product. One example for an aggregation – over a single attribute – is the average age of lecturers in a university administration database. Aggregations in general can perform a grouping of the data, e.g., average age of lecturers grouped by the starting initial of the last name. In addition data warehouses typically consider aggregations across many dimensions (attributes) [AAD$^+$96]. The application of a single aggregate operator (without grouping) was treated in [GL95]. The work was extended by [QGMW96], who formulated a general incremental maintenance involving aggregation. The work in [PSCP02] treats the incremental computation of *non-distributive* aggregation operators, i.e., operators that can not be efficiently maintained due to their non-locality, e.g., *MIN*, *MAX* require access to the underlying dataset if the old minimum/maximum is deleted from the view.

**Transitive Closure/Linear Recursion** Traditionally, the relational algebra does not include any means for recursive queries. The *transitive closure* operator provides a limited form of recursion termed *linear recursion* (cf. [JAN87], for a good explanation of the relationship between transitive closure and linear recursion).

A linear recursive program is best exemplified in deductive databases. A program is called linear recursive if the logical implication for a literal is expressed recursively with only one occurrence of the implied literal in the antecedent. For example the following query denotes a path in a directed graph. A path exists if there is a direct *edge* between the vertices (rule 2.3.1.1) or if there exists an *edge* that connects to a known *path* (rule 2.3.1.2). The query is linear recursive, since the *path* literal is used only once in the antecedent of rule 2.3.1.2.

$$path(X,Y) \text{ :- } edge(X,Y). \tag{2.3.1.1}$$

$$path(X,Z) \text{ :- } edge(X,Y), path(Y,Z). \tag{2.3.1.2}$$

Although the transitive closure is a simple and well-understood operation, the efficient maintenance is impeded by the large number of tuples and high interconnectivity that cause unacceptable runtime and storage requirements. Hence, efficient treatments of the view maintenance problem have focused on providing optimal solutions for two important sub-problems of the general transitive closure that consider (i) only acyclic graphs and (ii) restricted queries, e.g., the above *path* query.

In [DT92] an algorithm is presented that derives non-recursive datalog programs to update right-linear-chain views over acyclic graphs. A right-linear-chain imposes a special restriction that requires the recursive call to be on the rightmost side of the rule and the variables to be ordered in a chain such that the rule has the form:

$$q(X,Z) \text{ :- } q_1(X,Y_1); q_2(Y_1,Y_2), \ldots, q_k(Y_{k-1},Z)$$

The algorithm in [DT92] handles insertions of edges, while an extension to deletions was given in [DS95], yet both only apply to right-linear-chain views. Other views require additional materialized data for incrementalization [DS00]. An example of a linear recursive view that is not a right-linear-chain is the same generation query, i.e., are two siblings in a tree – or directed graph – at the same depth. The same generation query is exemplified by the rules below (2.3.1.3, 2.3.1.4) as the predicate *sg*. Two nodes are in the same generation if they have the same parent (rule 2.3.1.3), or if they have two different parents who are in the same generation (rule 2.3.1.4). While the rules can be brought into a chain order by providing *parent* and *child* predicates, the rule 2.3.1.4 is not right linear and can not be transformed such that both chain ordering and right linearity are enforced.

$$sg(X,Z) \text{ :- } parent(X,Y), child(Y,Z). \tag{2.3.1.3}$$
$$sg(X,Z) \text{ :- } parent(X,Y), sg(Y,V), child(V,Z). \tag{2.3.1.4}$$

A general result on the maintainability of transitive closures is presented in [DLW96], who find that it is not possible to derive a non-recursive program for the maintenance of the transitive closure over an *arbitrary graph* in response to deletions, i.e., deletions in arbitrary graphs always require an intermediate storage for the incremental maintenance. Many algorithmic maintenance strategies for the transitive closure have been proposed; [KZ08] provides a good overview and comparison. The emphasis in these algorithms was on almost linear update time. No considerations were made w.r.t. the memory consumption – all algorithms require $\Omega(|Vertices|^2)$ space. In [Jag90] the authors proposed a compression strategy for maintaining the materialized view over the transitive closure that addressed the memory efficiency, however, the compression strategy does not apply to the various structures used to maintain the transitive closure in the various algorithmic strategies.

General Recursion is stronger than the transitive closure, since an arbitrary number of recursive predicate occurrences – and also mutual recursions – are possible. For example, using general recursion a simple and very declarative specification of transitivity is possible, as shown in the rule below.

$$p(X,Z) \text{ :- } p(X,Y), p(Y,Z).$$

General recursive queries are applied in ontologies – such as OWL – that have symmetric and transitive properties. For example, Volz et. al [VSM05] extend previously developed algorithms for maintenance of general recursive views and apply this to ontology languages that are operationalized via deductive databases.

The efficient maintenance of general recursive views was studied first in [Küc91, UO92, GMS93]. All three approaches are based on the principle of deriving a set of *delta rules*, i.e., a transformed version of the original rules used to efficiently compute only results that pertain to changes in the base relations. An algorithmic approach that (re-)derives sets of changed tuples on a relation-by-relation basis was presented in [HD92].

In general the above approaches assume that the results prior to a modification are stored and available for the view maintenance. Upon modification of the base relations the delta is used first to compute an overestimation of tuples that need to be removed from the derived results. Then alternative derivations for the new results are computed. A reverse approach is given in [SJ96], where the view results are not required to be available during the view maintenance process. Instead the approach exclusively uses the underlying relations.

### Information used During Maintenance

There are four basic structures that contain information which can be used during the maintenance of relational views. Which information is required depends largely on the expressivity of the language used to formulate the view, but also on algorithms and properties of the underlying relations. Furthermore there are cases where additional information can speed-up the maintenance process. Hence, a trade-off between time and space consumption is to be made.

**Deltas** are the tuples that are inserted, deleted or modified in the base relations. Maintenance of views that requires only information about deltas is very efficient, since no storage is required.

**Base relations** refer to the underlying base relations prior to the update of the database. Maintenance of views that require information about the base relations is more costly than using only deltas. First the base relations contain a large number of tuples, hence requiring much storage space. Second, some maintenance algorithms require finding tuples with specific properties in the base relations. Due to the fact that base relations are as a general rule larger than deltas, finding such tuples is a time-consuming operation.

**Materialized views** refer to the multiset of tuples the view contained prior to the update of the database. Using the materialized view incurs storage costs and lookup cost incurred for finding tuples with specific properties, similar to the base relations. However, the materialized views are as a rule much smaller, since not all elements in the base relations contribute to the result in the materialized view.

Auxiliary views refer to intermediate results that are stored in addition to (or instead of) the base relations and materialized views. For example, the base relations of a join can be indexed and the index stored as an auxiliary view instead of the base relations, which greatly speeds up the maintenance time.

The study of views that are maintainable using only the delta information has been conducted under the terms *autonomously computable* [BCL89] or *self-maintainable* views [GMR95, GM99a]. In addition [Huy96] presents a self-maintainability test for rules in deductive databases. Key findings in relational algebra are that selections and projections are always self-maintainable. Consider for example the selection of lecturers – from the definition in Figure 2.1a – that are below the age of 40 years. The respective view is defined below and filters the data in *Lecturers* based on the condition in line 4.

```
1 CREATE VIEW YoungLecturers(Name, Age) AS
2         SELECT Name, Age
3         FROM Lecturers
4         WHERE Lecturers.Age < 40
```

The self maintainability of the view *YoungLecturers* stems from the fact that selections are based on a local property of each tuple in the database. The selection condition can be checked for each inserted or deleted tuple in the *Lecturers* relation and if a tuple satisfies the condition it is added or removed from the view. Neither the base relations nor the previously computed materialized view are required to deduce the effect of the change.

Many of the more expressive language constructs are not self-maintainable and either require the underlying base relation, the previously computed view results, auxiliary data or a combination thereof. For example, a join between relations $A$ and $B$ – in the most general form – always requires access to the underlying base relations, since adding a or deleting a tuple (e.g., from $A$ ) always requires checking whether a corresponding tuple for the join condition exists in the joined relation (e.g. $B$). Consider for example the join presented in Figure 2.2, between *Lecturers* and *Tenure*. Here the addition/deletion of tuples to *Lecturers* requires to consult the *Tenure* relation and finding corresponding tuples for the join. A language construct that can be efficiently maintained using a combination of the base relation and the materialized view is the transitive closure (over acyclic graphs). With the combined data the view can efficiently be maintained in near linear time. It is interesting to note that the respective maintenance algorithm is non-recursive, even though the operator provides a form of recursion (a respective algorithm using SQL statements can be found in [PDR05]).

The use of auxiliary views is in some cases a necessity for an effective runtime performance of the view maintenance. For example [PSCP02] use axillary *automatic summary tables*[3] to efficiently maintain *non-distributive* aggregate views. Other views can also benefit from auxiliary views as shown in [QGMW96] and [VMK97] were auxiliary views can effectively be used to minimize the number of tuples in joins.

### Provided Modification Operations

The modifications to the database can be performed using different operations that manipulate the base relations. There are two different basic operations that can be offered for view maintenance:

**Insertions/Deletions** are the basic operations for performing updates to the database. The change to a view is maintained for a set of inserted or deleted tuples. All approaches offer these basic operations. Differences in performance can be eminent between insertions and deletions for some language constructs, e.g., the transitive closure can be maintained more efficiently for insertions than for deletions.

**Updates** can be supported as a first class concept or by modeling an update as a deletion followed by an insertion. The main advantage of modeling updates as an independent operation is the ability to reason over the attributes that are modified. This enables the view maintenance to identify tuples that do not contribute a change in the result of the current view. For illustration consider the join of *TenuredLecturers* shown in Figure 2.2. If the *Age* attribute of any lecturer is updated the results in *TenuredLecturers* do not change, since the respective attribute is not part of the view.

Update operations can not always be propagated across all language constructs, i.e., some constructs treat an update and respond with a set of deletions and insertions. The join operator is an example for such a language construct; joins can not propagate updates that affect the joined attributes. Consider again the *TenuredLecturers* view and an update to the *Lecturers* relation that changes the *staffId* of the lecturer *Marinescu, E.* from 1 to 4. The

[3] The term automatic summary table denotes an incrementally maintained (and materialized) view in the IBM DB2 database

corresponding change is a deletion of the entry *(Marinescu, E., 09/01/2012)* from *TenuredLecturers*.

Updates are treated in related work for specific language constructs. In [GJSM96], updates for SPJ views are discussed and the notion of attributes that contribute to a view are formalized under the term *exposed variables*. The authors in [UO92] consider update modifications to values in deductive databases. The authors in [CW91] discuss the finding that updates do not propagate across join operators; hence, they favor a general treatment of view maintenance as deletions followed by insertions. The majority of related work falls into the category, that treats updates in this fashion, e.g., [AE99, HBC02][4].

In [KR98] updates are discussed in the presence of managing "virtual classes" (cf. Sec. 2.2.4) as materialized views for an OO database system. The treatment of updates w.r.t. language constructs corresponds to the treatment in [GJSM96], i.e., updates to non-exposed variables are irrelevant to a view. In addition a complex registration process is proposed that analyzes all views and registers them in the database system if they need to be maintained due to an update of an objects attribute.

Note, that the term *update* can lead to confusion, since the term is sometimes used as synonym for a *change operation*, i.e., an insertion or deletion. For example the work in [GGMS97] discusses updates in the latter sense. The authors treat a general form of view updates as a merge operation in monoid homomorphisms[5], yet their updates correspond to the classical insertions and deletions. Consequently, throughout this thesis the term *modification* is preferred to convey a change operation, which can be an insertion, deletion or an update.

## 2.3.2 Incremental View Maintenance Techniques

Several approaches and techniques have been developed for the incremental maintenance of materialized views. The following sections provide an overview of the proposed solutions. *Counting algorithms*, are among the first works to consider incremental view maintenance. *Algebraic query rewriting* and *logic query rewriting* can be considered as a generalization of the incremental maintenance problem

4 The authors in [AE99] provide a treatment for deferred view maintenance that tracks a list of updated object, but the actual processing of updates in their algorithm treats updates as inserts and deletes.

5 The findings in [GGMS97] coincide with the classical works in the fact that there exists a set of language constructs that can not be maintained in the monoid homomorphisms without further auxiliary views.

for relational algebra and deductive databases. *Active rules* and *memoing/tabling* incorporate (or extend) existing database technologies for incremental maintenance. Finally, different *optimizations techniques* proposed in related works are discussed.

### Counting Algorithms

The basic idea of this technique is to keep a *multiplicity count*, i.e., the number of derivations for a tuple, as extra information. Upon insertion or deletion of a tuple the counter is incremented or decremented accordingly and tuples are deleted when their counter reaches zero. Similar variants were introduced by Shmueli et al. [SI84], Blakeley et al. [BLT86] and Gupta et al. [GMS93].

Counting algorithms are frequently cited under this name in literature. Given the above definition, the term seems to be a bit of a misnomer. Algorithmically the incrementation or decrementation of a multiplicity counter is not the maintenance technique. The actual maintenance is achieved by processing the underlying computational model – relational operators or logic rules – in correct ways w.r.t. modifications **and** by keeping a multiplicity counter. The latter is required for correct treatments of specific operators (rules).

The technique was introduced to maintain SPJ views in a set algebra [SI84, BLT86]. In this important subset of SQL, the multiplicity counters are necessary for a correct treatment of multiple derivations for the projection operator. In [SI84] specialized data structures are used to store multiple derivations. In [BLT86] general relations are used to store tuples, which are enriched with the counter information. In this approach an algorithm derives new SPJ expressions whose evaluation determines tuples that must be inserted to (deleted from) the view.

In [GMS93] the technique is applied to a deductive database with duplicate semantics. The multiplicity counter is derived from the multiplicity of the duplicate semantics for the deductive database and stored similarly to [BLT86]. The algorithm derives a set of *delta predicates* that use the old values of the base relations and views to determine a set of tuples that must be inserted to (deleted from) the view. The approach covers set union, stratified negation and aggregation.

### Algebraic Query Rewriting

The idea is to define different *change propagation expressions* for insertions and deletions from the base relations to the materialized view based on the original view expression. Change propagation expressions are defined using the operators of the relational algebra and can be simplified for insertions and deletions in different ways. The derived relational expressions compute the change to the view without doing redundant computation.

The idea was first introduced in [Pai82] under the term *finite differencing* and was used subsequently in [QW91] for view maintenance of SPJ views with set semantics. Griffin et. al [GLT97] provide a correction to the minimality result of [QW91] and extend the algebraic rewriting approach to a multiset algebra with aggregations and multiset difference in [GL95]. In this latter work the authors also provide interesting results pertaining to efficiency and show that for a very restricted class of views (no projections, no Cartesian products), the incremental computation is always more efficient. The algebraic approach was extended by [QGMW96] for views with aggregations.

### Logic Query Rewriting

This technique is used in deductive databases and is similar to algebraic query rewriting in the sense that a new logic program, i.e., an extended set of logic rules, is derived from the original logic program (rules). The main idea is the following. Given a view definition as a (set of) rule(s) in a deductive database as defined below (2.3.2.1), where a view $p$ is defined based on relations (predicates) $q_1$ to $q_k$. For brevity we omit a concrete variable binding, since the general idea can be transcribed to predicates of any arity.

$$p \text{ :- } q_1, q_2, \ldots, q_k \tag{2.3.2.1}$$

A set of maintenance rules for the new database state ($p^{new}$) can be derived for the insertions and deletions using three sets of delta rules, one for an overestimate of deleted tuples ($p^{del}$), one for a reinsertion of tuples with alternative derivations ($p^{red}$) and one for general insertions ($p^{ins}$). The rule derivation follows a specific schema that can in general be summarized as follows.

First the deletions are derived for each predicate $q_i$ by rewriting the query into a rule (as in 2.3.2.2) where the old state of all $q_x$ is unified with the deletions of $q_i$ (for all predicates $x \neq i$). The deletions $p^{del}$ stem from the union (i.e., disjunction) over deletions from all $q_i$ (2.3.2.3). The new database state consists of all tuples that are in $p$ and have not been deleted by $p^{del}$ (2.3.2.4).

$$p_i^{del} \text{ :- } q_1, q_2, \ldots, q_i^{del}, \ldots, q_k. \tag{2.3.2.2}$$

$$p^{del} \text{ :- } p_1^{del}; p_2^{del}; \ldots; p_k^{del}. \tag{2.3.2.3}$$

$$p^{new} \text{ :- } p, \neg p^{del}. \tag{2.3.2.4}$$

Second the re-derived tuples are all that have been deleted by $p^{del}$ but can be inferred through the new database states of the underlying predicates, i.e., $q_i^{new}$ (2.3.2.5). The new state of $p$ includes all re-derived tuples (2.3.2.6).

$$p^{red} \text{ :- } p^{del}, q_1^{new}, q_2^{new}, \ldots, q_k^{new}. \tag{2.3.2.5}$$

$$p^{new} \text{ :- } p^{red}. \tag{2.3.2.6}$$

Third insertions are treated in a similar manner as deletions by providing a rule for insertion to each underlying predicate $p_i^{ins}$ (2.3.2.7). The difference is that insertions are based on the new database state $q_x^{new}$ for all predicates $x \neq i$ that the rule $q_i^{ins}$ must unify with. Like deletions the set of all insertions $p^{ins}$ stems from the union of all insertions from all $q_i$ (2.3.2.8). Finally the new state includes all insertion to the database (2.3.2.9).

$$p_i^{ins} \text{ :- } q_1^{new}, q_2^{new}, \ldots, q_i^{ins}, \ldots, q_k^{new}. \tag{2.3.2.7}$$

$$p^{ins} \text{ :- } p_1^{ins}; p_2^{ins}; \ldots; p_k^{ins}. \tag{2.3.2.8}$$

$$p^{new} \text{ :- } p^{ins}. \tag{2.3.2.9}$$

The general idea was presented in [GMS93] as the *DRed* (Delete and Rederive) algorithm for stratified logic programs that can also use aggregations. Note, that the *counting algorithm* is presented in the same paper and can be seen as a special case of the DRed algorithm used for non-recursive views. A similar idea as the DRed algorithm was presented by [Küc91] for stratified recursive programs. However, the work was criticized for not generating safe rules [GJSM96] and not treating some special cases, such as duplicate rederivations, as efficiently as DRed.

In [UO92] the authors present a technique that rewrites queries with an explicit support for an update operation (rather than only modeling insertions and deletions). To enable efficient modeling of updates the concept of database keys was introduced for logic predicates, and updates are only allowed on non-key arguments of predicates. A key is modeled as a constraint that disallows arguments between two tuples to share the same value. For example, as shown in (2.3.2.10) $key^p(K)$ is the key constraint on $p$, where the first argument $K$ is the key and hence no two tuples may be stored using the same key. The program is translated into a new set of rules that use existentially quantified subexpressions on the keys.

$$key^p(K) \text{ :- } p(K,X), p(K,Y), X \neq Y. \tag{2.3.2.10}$$

From the maintenance rules summarized above it is easy to see that the general idea relies on the availability of the old database state of the view $p$ (i.e., rule 2.3.2.4). A modified version of the algorithm was given in [SJ96]. Their approach alleviates the reliance on the old state of the view and focuses only on the availability of the base relations.

### Active Rules

Ceri and Widom [CW91] study views from the practical perspective of extending an existing active database system with rules to support incremental view maintenance. The rules define respective SQL statements as actions that manipulate the view by inserting or deleting appropriate tuples. Hence, when a rule is triggered the according update to the view is performed by the database. Active rules for the insertions and deletions of tuples to the base relations are determined by a respective algorithm. The authors consider nested sub-queries with positive and negative existential quantification, as well as set union and difference operators. They define efficient incremental maintenance rules for views if key information about base relations is present. If no key information is present the views are re-computed in their entirety.

### Memoing/Tabling

The original idea behind *memoing* stems from a shortcoming in logic programming languages that use the SLD[6] resolution mechanism. SLD resolution basically implements a backtracking search through all rules that make up a logic program. This backtracking can be understood as evaluating the logic rules as procedure calls, with extension for the logic language, i.e., a rule can have multiple body definitions for the same head and parameters in the head may be variables that are unified in the body. Hence, speaking procedurally, this extension makes logic languages nondeterministic. Due to recursion, the procedure calls can easily enter infinite loops, even for simple – and meaningful – programs. Memoing is a technique that guarantees termination by storing intermediate results for already computed procedure calls, i.e., if the same call is made later in the computation the procedure is not re-executed but rather the stored result is used [War92, SSW94]. The term *tabling* is commonly used for the memoing technique in logic languages, as the already computed results are stored and retrieved from a result table. Tabling also

[6] SLD resolution is termed after the initial letters in "**L**inear resolution with **S**election function for **D**efinite programs"; cf. [Lif96] for a treatment of the SLD calculus

serves as an optimization technique that overcomes the fact that the SLD resolution technique may perform several calls repeatedly during backtracking.

The original tabling solution was not incrementally maintained and tables were effectively completely deleted in the presence of modifications. With tabling becoming an established feature of deductive databases, it was naturally extended with incremental view maintenance techniques. Note, that earlier works on view maintenance such as [Küc91, GMS93] maintain a storage similar to tabling, but were either not implemented or were custom extensions to existing implementations.

In [SR03] the authors present an adaptation of the DRed algorithm of [GMS93] that maintains tabled views. The approach uses a deferred maintenance strategy and is only applicable to a single view. The latter restriction is due to the fact that accessing the view refreshes the tables of all required predicates, but since the whole approach is a top-down deferred evaluation, the refresh is not propagated in any way to other views that depend on these predicates. Furthermore the approach has $O(n^3)$ space complexity if one assumes a program that effectively computes a graph reachability problem; where $n$ is the number of vertices in the graph. The additional space – reconsider $O(n^2)$ is required for transitive closure – is due to the fact that a vector of supporting facts is stored for each derived fact.

## Optimization Techniques

Several optimization techniques have been proposed to in the context of materialized view maintenance. The first two techniques are optimizations to (i) speed up evaluation and (ii) decrease (disk/main) memory for incremental maintenance. The last two techniques are concerned with providing optimal solutions in the presence of redefinitions of views or multiple independently defined views.

**Identifying irrelevant modifications**

The key idea is to provide tests that determine whether a particular modification affects a given view, i.e., if the modification is relevant. If the test is negative, i.e., the modification is irrelevant, no maintenance operations are necessary. However this only pertains to particular modifications and if the test fails another maintenance technique must be applied to maintain the view. In [BLT86, BCL89] a proposal is made to test SPJ views in relational algebra, which basically normalize the view definition and test the satisfiability of selection conditions. The approach was extended for logic programming languages in [Elk90], which also considers integrity constraints and simple recursive rules. In [LS93] the approach was further generalized and reduced to the equivalence problem for datalog programs. The approaches were adapted by [ABM09] for active XML documents.

Testing the satisfiability of predicates and the equivalence between queries is in general undecidable. The identification of irrelevant updates is thus only useful for simple cases that can be decided quickly. In general the test must be less expensive than determining – via some view maintenance technique – that the set of changes to the view is indeed empty. The main advantage comes into play when the modification is phrased as a simple query over a (potentially) large dataset. Consider for example the *YoungLecturers* view given in Sec. 2.3.1, where all lecturers have an age under 40. A database operation that is phrased as a query, which deletes all lecturers over 40, can easily be checked to have no relevance for the view *YoungLecturers* and is more efficient – for large datasets – than testing all deleted tuples for their relevancy w.r.t. *YoungLecturers*. However, a general decision when the test is advantageous is not discussed in the above approaches.

#### Efficient storage

To minimize the memory requirements of the incremental view maintenance for relational databases with persistence, the authors of [Rou91] propose a data structure termed *view caches*. The idea is to store only pointers to tuples in the underlying relations that contribute to a view, instead of storing the concrete values of the tuples in the view. The work was performed in a traditional stored database context and hence is mainly concerned with an optimal I/O criterion that the same buffer page of the underlying relation(s) is read only a minimum number of times. To combat the space complexity of incrementally tabled logic programs that use supporting facts the authors of [SR05b] propose a compression scheme that relies on sharing the supports. Actual savings vary for the defined views and input data, but can be considered close to tabling the results without supports.

#### Adapting views after Redefinitions

The main idea for this optimization is to maximize the reuse of results and auxiliary data in the case when the definition of a view is redefined with slight changes. The basic technique is discussed for relation algebra in [GMR95, MD96]. They discuss adaptations for different classes of changes, e.g., in the selection conditions or in joins, and show which additional information must be kept in order to react to redefinitions of views.

The authors of [VSM05] discuss changing rules for an incrementally maintained ontology language that is encoded in a logic database. Changes to the ontology change the rules of the logic program, i.e., the previously defined views. In [GIT11] the authors generalize the adaptations to a larger set of logic programs and relational algebra expressions. In particular they provide a framework for reasoning about queries with negation (or set difference).

**Optimizing queries using maintained views**

This query optimization takes materialized views into account and was proposed to speed up query processing time for arbitrary queries and not only the queries of the incrementally maintained views. The technique finds alternative formulations of queries – specified by the user without referring to a view – that incorporate already existing views maintained by the database system. As an optimization technique the approach also has a merit for the definition of materialized views, when views are defined via general queries that are partially maintained by other views. In this case, alternative formulations using existing views can help to reduce redundant maintenance work.

In [CKPS95] the authors present an extension to traditional query optimizations that takes information about maintained views into account. A similar discussion is provided for deductive databases in [LMS95]. The authors extend previous works by also considering the minimality, i.e., size of the expressions, and completeness, i.e., usage of only views and built-in predicates, of the alternative formulations.

# 3 Language-Integrated Database with Incremental View Maintenance

In this chapter the integration of incremental view maintenance into a host programming language is discussed. The incentive for the language integration is twofold. First, to provide a query (i.e., computation) capability that allows the use of relational operators found in databases for the sake of automated incrementalization. Second, to materialize in memory (i.e., store permanently) only the necessary amount of data that is required for the incremental maintenance. As shown in the previous chapter, incremental view maintenance has a long history of studies in the fields of relational and deductive databases. In particular, there exists a good understanding of how the different relational operators can maintain their results incrementally. From these insights we can deduce the minimal amount of materialized data.

It is instructional to understand the language integration as a combination of the following five points:

**Base relations** (or *extents*) represent a (multi-)set of objects – in the sense of objects provided by the host language – that can be queried. Extents are similar to tables in databases, or collections in programming languages. Yet, extents are only a logical representation of the contained data and never actually store the data.

**Incrementally-maintained relational operators** are provided for maintaining the results of complex queries over the extents. The operators store a necessary minimum of data to perform incremental maintenance, but not more.

**An SQL-inspired embedded domain specific language** is provided to specify queries, which are then compiled to a tree of relational operators. The language is comparable to those found in OO databases (cf. Sec. 2.2.4).

**Incrementally-maintained views** can simply be defined by storing the compiled query in a variable in the host language. This means that the OO extension mechanisms, e.g., subclassing, can be reused to make the definition of views more modular. For example, we can define an interface for views of specific types and give different implementations based on particular application scenarios. Note that a type termed `Relation` is the common supertype for all operators (i.e., compiled queries) and extents. Thus clients are not required to differentiate between compiled queries and extents, but can interchange them freely.

**Events** are responsible for performing the incremental maintenance. Maintenance starts by triggering events for the addition/removal/update of objects in the base relations. The incremental changes are then propagated through the operator tree and respective modification events are received by the view. Clients can use views as *materialized views*, i.e., their data is retained in memory and can be traversed similar to a list, or they can use a view without materialization and subscribe to the event system of the incrementalization to receive events whenever modifications to the results of the view are imminent.

All the above concepts are provided in the language Scala, which provides a very good integration for building embedded domain specific languages (EDSLs).

For illustration consider the example depicted in Figure 3.1. The example uses plain Scala (together with the proposed EDSL) and can be be compiled and run such as it is presented. A base relation over objects of type `Student` is declared in line 1 and bound to the variable `students`; the concrete instance is of the type `Extent` (line 2). A view of specific students is declared in line 4; the view is declared as materialized via the type `MaterializedRelation`. The respective query in line 5 defines which objects are selected as results of the view. The query selects entire student objects – denoted by `(*)` – from the base relation `students` that satisfy the condition of having a first name equal to `Sally`. The incrementalization is now performed automatically and the results are available via the variable `view`. Incremental modifications are made to the base relation in lines 8 and 10, where two new `Student` objects are added. After these modifications the execution of the line 12 prints the string `Sally`. Note that the `foreach` basically takes a first-class function and applies it to all elements in the collection; `foreach` is a standard Scala method for collections, which is also supplied for materialized relations.

The presented approach uses ideas from the traditional view maintenance techniques discussed in Sec. 2.3 to enable the incremental update exemplified above. The notable difference is that incremental view maintenance in databases is typically taken for materialized views, while the focus of this thesis is to perform as little materialization as possible. More precisely, materialized views in databases are seen as caches where the base relation and the views are materialized, whereas the data in the computation is transient. In contrast, we wish to keep only the necessary minimum of data required for the computation and treat the base relations and the views as transient data. For example, in Figure 3.1 we do wish to materialize the set of all students and the set of all students that are named “Sally”, but merely generate events if students named “Sally” are added or removed from the database. Note that the use of a materialized view in the example was made for illustrational purposes. Typically, the results of the views are only generated as events and interested clients can for example display them in a graphical user interface.

```
1 val students: Relation[Students] =
2   new Extent[Student]()
3
4 val view: MaterializedRelation[Student] =
5   SELECT (*) FROM students WHERE (_.firstName == "Sally")
6
7 val sally = new Student("Sally", "Fields")
8 students.add(sally)
9 val george = new Student("George", "Tailor")
10 students.add(george)
11
12 view.foreach(s => println(s.firstName))
13 // prints: "Sally"
```

**Figure 3.1:** Example of an incrementally maintained materialized view

Throughout this work the language Scala is used as the host programming language. Hence, all examples discussed in the following are expressed in the notation of Scala. The EDSL for writing queries in Scala provides complete type safety. As a simple example, type safety guarantees that the objects returned as results of the query in line 5 are of the type `Student`. As a more elaborate example, type safety guarantees that the conditions in the query are compatible with the objects found in the queried relation. In the above example the given condition must be phrased as a function that takes a parameter[1] contra-variant to the type `Student` (the type of the objects in the queried relation `students`).

Note that the notation of Scala is used in various discussions on typing issues. In this respect Scala reflects the notation of type annotations found in textbooks on type theory [Pie02]. In summary, the form $e : T$ is used to denote that the expression $e$ has the type $T$. Subtyping is expressed via the form $T <: S$ denoting that $T$ is a subtype of $S$. Parameterized types are denoted via square brackets, e.g. $R[T]$ denotes that $R$ is parameterized by type $T$.

The remainder of this chapter is organized as follows: In Section 3.1 the *data definition*, i.e., the layout of objects that can be queried via relations, is discussed w.r.t. the integration into the host programming language. Section 3.2 discusses the event system used for the *manipulation of data*. Section 3.3 introduces provided relational operators and discusses their semantics in the relational sense.

[1] In the concrete example an anonymous function is used that can be translated to a function taking a `Student` object as parameter

The *incremental maintenance* of the supported operators is discussed in section 3.4. Section 3.5 discusses the optimality of the incremental maintenance in terms of memory and runtime. Finally, we discuss further related works on incremental maintenance in section 3.6 The SQL inspired *query language* used to define automatically incrementalized views is discussed in the next chapter.

## 3.1 Data Definition

Object-oriented database management systems typically provide their own languages for data definition. In this thesis the database is directly integrated into an object-oriented programming language. Hence, no external data definition language is required. Instead, the definition of database objects is given directly as types in the language. Thus, objects stored in the database retain the safety of a strongly-typed host language. Database objects can be defined using the whole range of mechanisms provided by the host language, i.e., as classes, traits, object types or even structural types. The approach as a whole is not tied to a single programming language (i.e., Scala) and can in fact be conceived as being parameterized by the language. In the following paragraphs we discuss how data can be defined and how different database concepts are realized within the host language.

```
class Student(val firstName: String,
              val lastName: String,
              val grades: List[Int])
{
  def gradeAverage: Float = {
    grades.reduce(_ + _) / grades.length
  }
}

class Course(val title: String)

class Registration(val student: Student,
                   val course: Course)
```

**Figure 3.2:** Data definition for a student registration database

Figure 3.2 depicts a simple database schema for a registration system that records students and the courses they are registered for. The data in this database is declared using Scala classes. A student (declared in lines 1-8) has three attributes, a first and last name of type `String` and a list of her grade points of type `Int`. In addition a student has a method to compute the average of the grades (line 5-7). A course (line 10) has a title of type `String`. A registration (line 12-13) is a relation between students and courses, which is encoded as a class holding respective objects.

### 3.1.1 Object Properties

Objects are used in queries (and views) via the accessible attributes and methods that are defined for the types of the objects. In the following we denote all data that can be accessed for a given type as *object properties* or in short *properties*. A property can be a stored value, e.g., the name of a student (cf. Fig. 3.2), or it can be computed via a method, e.g., the `gradeAverage` of a student. We assume that the computation of the value has no side-effects. All properties are strongly typed due to their definition in the host language. Hence, properties are guaranteed to return values of the specified types.

We currently restrict properties to be *immutable* once an object resides in the database, i.e., was logically added to an extent. Instead of having mutable state an explicit event must be triggered to the extent holding the particular object, to signify an update of the object's values. This approach greatly simplifies the incrementalization, since all operators can rely on the fact that all modifications are received via a well-defined interface and results can not be invalidated due to unexpected changes in the state of an object.

```
1 val susy = new Student("Susy", "Fields")
2 students.update(sally, susy)
3
4 view.foreach(s => println(s.firstName))
5 // prints: ""
```

**Figure 3.3:** Example of an incremental update to object properties

For illustration, consider Figure 3.3 that updates an object in the `student` extent declared in Figure 3.1. The first name of the object `sally` is updated to `"Susy"` by notifying the database of the update (line 2). Instead of a state mutation, the updated value is passed as a new object (created in line 1). Hence, the view is notified of the changed value and the corresponding entry for `"Sally"` is removed. Changing the state of the object `sally` via mutable fields would invalidate the results in the defined view.

The treatment of properties as immutable is actually quite natural when comparing the approach with OO programming that uses collections such as hash sets or hash maps. Defining object equality (or object hash codes) via mutable fields is equally harmful as storing mutable objects in the database. In the case of collections potential users of the collections can experience strange results. For example, storing an object `o` in a hash set `s` and changing the state of `o` can yield a negative result for the test `s.contains(o)`. The book *Effective Java* [Blo08] provides a more detailed discussion on the topic.

Whether state is required to be mutable is also greatly dependent on the underlying scenario of how data changes. For example, immutable structures are sufficient for static analyses on bytecode, which was the motivating use-case for this thesis. The general workflow in this scenario is to read the bytecode that was generated by a compiler. The bytecode is parsed into a new immutable object structure. Previous versions of the bytecode (e.g., for a recompiled class) are also represented by an object structure, however, the parsing process does not try to identify objects in the previous versions and alter their state; this makes for a cleaner and more efficient design of the parser. Hence immutable state is sufficient and updates from the previous version to the next version are triggered as events on the database.

The incorporation of mutable object state is also a technical issue, which stems from the fact that a mutable state must somehow notify the incremental view maintenance of the changes. An explicit triggering of updates after mutating state – by sending an event to the database – is feasible, but requires additional boiler plate code. In essence an object's state mutation must be monitored (at least for objects participating in the incremental view maintenance) to trigger events to the database. Such a monitoring is not easily introduced with the common language semantics of most OO programming languages. Although languages based on bytecode as an intermediate representation (e.g., Java, Scala, C#), can be subject to instrumentation of the classes to introduce said monitoring and a respective trigger [TSDNP02].

### 3.1.2 Database Instances

A *database instance* consists of several *extents* and/or *views*. Logically an *extent* represents[2] a (multi-)set of objects instances of a specific type. Note that multisets allow duplicates of objects, which will be discussed in detail in Sec. 3.3. For example, a database instance for the data in Figure 3.2 consists of three extents $E_{Students}$, $E_{Courses}$ and $E_{Registrations}$ for each of the types `Student`, `Course` and `Registration`.

2 As noted in the beginning of this chapter extents do not actually store objects.

Extents can be queried similar to tables (relations) in traditional databases. Each property of an object corresponds to a column in a table and each object instance corresponds to a row.

For illustration, Figure 3.4 depicts a database instance with the data in the extents treated as being materialized. For example the database contains two students (Figure 3.4a) as rows and all properties are represented as a respective column, i.e., first and last names, grades – represented as a set of integers for the purpose of this illustration – and a grade average. Especially the grade average is not actually stored as an attribute in the object, but in terms of querying the data it can be accessed in the same manner as stored values.

| OID | firstName | lastName | grades | gradeAverage |
|---|---|---|---|---|
| $o_1$ | Sally | Fields | $\{1,3,2\}$ | 2.0 |
| $o_2$ | John | Doe | $\{4,1,1,3\}$ | 2.25 |

**(a)** $E_{Students}$

| OID | title |
|---|---|
| $o_3$ | Introduction to Computer Science |
| $o_4$ | Data and Knowledge Engineering |
| $o_5$ | Introduction to Software Engineering |

**(b)** $E_{Courses}$

| OID | student | course |
|---|---|---|
| $o_6$ | $o_1$ | $o_3$ |
| $o_7$ | $o_1$ | $o_4$ |
| $o_8$ | $o_2$ | $o_3$ |
| $o_9$ | $o_2$ | $o_5$ |

**(c)** $E_{Registrations}$

**Figure 3.4:** Example of a database instance for the student registration database

Objects in a database instance can have references to other objects. For example the `Registration` class is defined with a reference to a student and a course. Each entry in the registration extent uses the object identifiers (OID) to reference the respective student and course, i.e., the first row in 3.4c means that the student named “Sally Fields” is taking the course “Introduction to Computer Science”. Note that object identifiers are not actually stored or maintained in the database, but simply denote the object reference in memory used by the underlying programming language. The column OID is merely presented for illustration.

As noted in the beginning of this chapter, extents are simply accessible via variables of the host language. In a typical scenario users design a database with multiple extents (and views). In our scenario with the three extents defined in Figure 3.4, a database can simply be defined as single a class. For illustration, Figure 3.5 depicts an exemplary database definition. Using classes for database definition allows a very flexible design, since OO extension mechanisms, e.g., subclassing, can simply

```
class StudentRegistrationDatabase
{
  val students = new Extent[Student]
  val courses = new Extent[Course]
  val registration = new Extent[Registration]
  val view1 = ...
}
```

**Figure 3.5:** Definition of a student registration database

be reused to extend a database definition, e.g., via additional views in the a subclass. An instance of a database can then be obtained by instantiating the respective class. For global accessibility, i.e., throughout an entire program, singleton instances can be defined.

### 3.1.3 Object Identity and Object Equality

The proposed language integration treats object identifiers different from traditional OO databases. Object identifiers are not explicitly managed by the language integration, since they arise naturally by allocating objects in memory, i.e., each object can be uniquely identified by its address in memory and object references are pointers to said addresses. In contrast an OO database is required to manage these identifiers explicitly, since objects can either reside in memory or a persistent storage device and in the latter case must be retrieved efficiently.

Potentially, the host programming language provides (limited) access to object identifiers by providing an equality test that compares object references. For example, the Scala construct "`o1 eq o2`" returns **`true`** only if `o1` and `o2` reference the same object. However, for the purpose of memory efficient incrementalization tests based on object identity are actually detrimental. In a treatment via identity, the removal of an object would require that the exact same object be identified in the extent and removed from it, hence this is not possible without storing the object. For illustration consider the extent `courses` for objects of type `Course`. The course with the title "Introduction to Computer Science" was added at some prior point in time. Note that this knowledge is implicit in the database client application, i.e., the value in Figure 3.4b is not actually stored. The call `courses.remove(new Course("Introduction to Computer Science"))` removes the respective entry, even though the passed parameter is in fact a new (different) object.

Treatment by equality can be seen as a consequence of achieving minimal memory consumption for the incrementalization. Nevertheless, it is prudent to note the fact that – as an approach – treatment by equality implies that the complete state[3] of an object can be reconstructed by the database client without the help of the database. In the above example the complete title of the removed entry ("Introduction to Computer Science") was – somehow – known to the client. In the case of using the database for static analyses the client has this knowledge, since the data is input and modified from source code files (or compiled bytecode files). As a simple example, if an analyzed class is removed, the data is simply re-read from the respective bytecode file and removed from the database, prior to the final deletion of the file. Note that updates require a slightly more complex treatment which will be discussed in detail Section 5.5. The important thing to note is that the treatment by value equality is very flexible and does not require the extents to retain all the data. The concrete state can be stored on disk in a form dictated by the client application (as for example in the case of static analyses) or the state might be retained in memory by the client application in some other form, hence storing them in extents is a duplication. If this is not the case there is still the additional possibility to materialize the data. In summary, treatment by equality allows the design decision of where data is stored to be made by clients.

### 3.1.4 Subtyping and Querying

The types in the database definition retain the subtyping mechanisms of the host language, i.e., introduction of additional data/behavior via subclassing, mix-in composition and structural subtyping. For example, Figure 3.6 depicts two extensions to the class `Course`. The subclass `MasterCourse` (lines 1-3) represents a course taken by students during their masters studies. A `MasterCourse` can have a set of prerequisite courses which students must have already taken. A `LabProject` (lines 5-7) represents a hands-on training course, which can accommodate only a limited number of students per semester. Due to the subtyping rules of the host language both extension can be used whenever the common supertype `Course` is expected. Queries on this supertype can, for example, uniformly access the property `title` in a database containing all three classes.

Whether the extended properties (`prerequisites` or `studentLimit`) are accessible by a query depends on the type of the extent or view on which the query is specified. All extents and views are strongly typed and are parameterized by the type of the objects they contain. For typing purposes we can treat extents and views

[3] Or at least the parts of the state relevant for equality

```
class MasterCourse(val prerequisites: List[Course],
                   title: String)
        extends Course(title)

class LabProject(val studentLimit: Int,
                 title: String)
        extends Course(title)
```

**Figure 3.6:** Extension of the database definition via inheritance

uniformly as relations. The type of a relation is denoted by *Relation* $[T]$, which means that the relation contains objects of type $T$; from the typing perspective a relation is similar to a heterogeneous list. For example, the type of the extent $E_{Courses}$ is *Relation* $[Course]$. The properties available to views over a relation are those of the parameter type $T$. For example a view defined on the type *Relation* $[Course]$, can access the property `title`. If the relation contains objects of types `MasterCourse` or `LabProject`, their additional properties are not accessible.

Given a relation of type *Relation* $[T]$, access to particular subtypes $T^{'}$ of $T$ can be obtained as a view. The view must (i) filter the elements, i.e., all objects of type $T$ satisfying a predicate that they are instances of $T^{'}$ and (ii) downcast all elements to the particular subtype $T^{'}$. For illustration, Figure 3.7 depicts a view of all `MasterCourse` objects obtained from the base relation `courses`.

```
val masterCourses: Relation[MasterCourse] =
  SELECT ( (_:Course).asInstanceOf[MasterCourse] ) FROM
    courses WHERE (_.isInstanceOf[MasterCourse])
```

**Figure 3.7:** Providing object instances of subtypes as a view

Vice versa, given a set of $n$ relations of types *Relation* $[T_1]$, ..., *Relation* $[T_n]$ (where all $T_1 \dots T_n$ are subtypes of $T$), a combined view as *Relation*$[T]$ can be defined as the union of all relations. Hence, a complete relation over all subtypes or a set of relations for each subtype can be freely transformed into one another. Thus, a database can be designed with both a combined extent for the supertype and several distinct extents for the subtypes or only either of the two possibilities. Favoring one design over the other does not limit users of the database, but may require the definition of respective views if only one possibility is accessible in the predefined database design.

## 3.2 Data Manipulation

The data manipulation is designed with the intent of performing efficient maintenance of views. Since some views can be maintained without knowing the base extents from which they were derived, data manipulation is not as straightforward as in traditional databases, especially when treating deletions.

Databases can express modifications via a *data manipulation language*, which is for example part of the SQL specification. In SQL deletions of objects in a database can be phrased as a query, such as the following:

```
DELETE * FROM students WHERE firstName = 'Sally'
```

The net effect of a query in a data manipulation language can be evaluated into a delta of deleted objects from the underlying table, e.g., `students`. Such deltas are the basis for incremental maintenance algorithms as presented in [BLT86, GJSM96]. Given an extent $E$ and a set of modifications, a view maintenance algorithm uses deltas (sets of objects) for additions ($\Delta_E^+$) and deletions ($\Delta_E^-$) that are known to change the extent $E$ as follows:

$$E^{new} \equiv (E - \Delta_E^-) \cup \Delta_E^+$$

In this thesis the extents do not require explicit storage and concrete sets of objects can not be derived for queries such as presented above. Consequently, the data manipulation of extents is managed via explicit sets of objects and not by a data manipulation language. Thus the approach makes no assumptions whether the data is stored or only propagated to update views. According to this semantics methods for data manipulation are introduced for the type `Extent` as depicted in Figure 3.8. `Extent` defines methods that allow to manipulate single entries via `add/remove` or `update`. Multiple modifications can be provided as sets of additions, deletions or updates. The method `modify` is exemplified here (line 11), which allows to provide modifications of all three types together. Note the type `Iterable` is a Scala type that allows to iterate over an arbitrary collection. There are no restrictions on the data contained in the modifications, i.e., objects can be added multiple times if necessary.

As an interesting side-observation one might think that databases computations are based on sets of objects and, hence, the single valued manipulations are not of great importance. However, from the point of view of runtime performance, a treatment via single objects can be advantageous. The simple reason is that the

```
trait Extent[V] extends Relation[V]
{
  def add(v: V)

  def remove(v: V)

  def update(oldV: V, newV: V)

  ...

  def modify(additions: Iterable[V],
             removals: Iterable[V],
             updates: Iterable[V])
}
```

**Figure 3.8:** Excerpt of the Scala trait for base extents

aggregation of objects into collections also requires computation time, e.g., creation of nodes in a linked list, or entries in a hash set. If this computation is performed solely for the purpose of modifying the database, the runtime can decrease by factors of 3 to 5.

## 3.3 Definition of Relational Operators Using Functions as Parameters

The underlying execution model for which results are incrementally maintained is an adaptation of the relational algebra [Cod70]. The presented formalization is a comprehensive reformulation of relational operators with a uniform treatment using functions as operator parameters. The traditional relational algebra passes column names instead of functions to the various operators. For example, to signify the columns on which a join between two tables is performed.

The relational algebra consists of *relations*, which are sets of tuples, and *relational operators*, which query the relations and produce new relations as a result. Since all operators consume and produce relations they can be freely combined to form powerful queries. In addition, new operators can be defined that immediately work with the existing model. Hence, the relational algebra provides a very flexible formalism for expressing queries that supports composability.

Naturally, precursor formalizations were given in different related works. The effective relational algebra for using multisets in SPJ views and in set-theoretic operators was first presented by Dayal et al. [DGK82]. We also adopt the *multiset* approach where a multiset is a relation that can contain one or more copies of the same value, i.e., $\{a, b, b\}$ is a multiset with three elements. Multisets are the standard model for modern relational databases and SQL [ISO11]. Hence, queries can be formulated and understood by an audience familiar with standard database technologies.

In contrast to Dayal et al. [DGK82] we use freely definable functions, whereas they use column names. The object-oriented approach requires additional operators which were also provided by Shaw et al. [SZ90]. In addition, the typing rules are different from those found in related works. For example in the treatment by Kuno et al. [KR98], all operations produce new types (called virtual classes) as a result. Thus, for example the union over several relations containing objects of different types yields a new type that represents the union of said types. In our approach we require all types (for extents or views) to be defined as part of the host language. Using this approach type correctness can be proven for the defined views using the type checker provided by the host language.

The relational algebra operators provided for automatic incrementalization are described after a short discussion of the background on the formalization and the used notation. The semantics of each operator is introduced in prose and by a formal description using multisets. The operators are divided into four groups: (i) *basic operators* (Sec. 3.3.2), used in traditional relational algebra; (ii) set *theoretic operators* (Sec. 3.3.3), e.g., union; (iii) *advanced operators* (Sec. 3.3.4), e.g., transitive closure and recursion; (iv) *derived operators* (Sec. 3.3.5), which support common (sub-)queries for existential quantification.

## 3.3.1 Notation and Background

Multisets are formalized as sets of pairs where each pair consists of the value and an additional count for the occurrences of the value. Hence, $\{a, b, b\}$ is written as $\{\langle a, 1\rangle, \langle b, 2\rangle\}$. For simplicity in the below definitions we can assume that $\langle t, 0\rangle \in A$ is true if there does not exist a pair $\langle t, i\rangle$ in the multiset $A$, i.e., if the element is not in the multiset we obtain a count of 0. From an implementation perspective elements with a count of zero are removed from multisets. As discussed in Sec. 3.2 data manipulation relies on object equality rather than object identity. Consequently, the formalization regards two objects in a multiset as identical if the values of all their properties are equal. Hence, inserting an object to a multiset will increase the counter, if the multiset already contains an equal object. In practice this means that classes must implement respective `equals` and `hashcode` methods. This requirement is in the spirit of the underlying programming language, i.e., the same methods are required for correct treatment of objects in sets or hash tables and, thus, known to developers from working with these collection types.

Properties of objects, predicates, and transformations are uniformly treated as *functions*. For example, given an object $o : T$, then retrieving the property $v : V$ can be written as $o.v$ which returns a value of type $V$. Retrieving the property can be seen as a function from $T$ to $V$ which is denoted by the type $T \rightarrow V$. For example retrieving the title of a `Course` (as defined in Figure 3.2) is a function of type $\mathit{Course} \rightarrow \mathit{String}$. Likewise a predicate on objects of type $T$ is a function of the type $T \rightarrow \mathit{Boolean}$, that returns true for all objects satisfying the predicate. Finally transformations of the objects processed by an operator to a different output are treated as function $T \rightarrow O$ where $T$ is the type of the input and $O$ the type of the output.

We use a *lambda notation* for concrete functions supplied to the various operators. For example, $\lambda x.x$ is the function that takes a parameter $x$ and returns the parameter as a result, i.e., the identity function. The lambda functions are used without any type annotations for the sake of readability. The concrete types of parameters of a lambda function are always deducible by the respective relations on which the functions are used. For example, the function $\lambda s.\, s.\mathit{firstName} = \text{`}\mathit{Sally}\text{`}$ is a lambda function on students, which returns true if a students first name is “Sally”. The function can for example be used in a filter on a relation over student objects.

Where appropriate the Scala notation for tuple types is used, i.e., a tuple of two objects with types $U$ and $V$ is denoted by the type $(U, V)$. For readability by a non-Scala audience we denote access to the components of a tuple by properties *first* and *second*, i.e., if $o : (U, V)$ then $o.\mathit{first}$ returns a value of type $U$ and $o.\mathit{second}$ a value of type $V$. Tuples with more entries are treated accordingly, e.g., a tuple with three values $o : (U, V, W)$ can access the third component as $o.\mathit{third}$.

All operators are defined on multisets and the respective type for their operands is denoted by the type $\mathit{MultiSet}[T]$, i.e., a multiset that ranges over objects of type $T$. A set (denoted by $\mathit{Set}[T]$) is a specialization of a multiset, i.e., $\mathit{Set}[T] <: \mathit{MultiSet}[T]$ and thus all operators accepting multisets also accept sets. The differentiation between sets and multisets is advantageous for query optimization. For example, a (simple) optimization is that the operator for duplicate elimination (cf. Sec.3.3.2) transforms a multiset into a set and can thus be omitted if the operand is a set. Hence, we discuss which operators retain the set property of their operand.

### 3.3.2 Basic Operators

Let $A : MultiSet[T]$ and $B : MultiSet[V]$ be two multisets containing objects of types $T$ and $V$, respectively, then the basic operators of the relational algebra are defined as follows:

**Selection** $\sigma_\theta(A) : MultiSet[T]$, where $\theta$ is a boolean predicate of the type $T \to Boolean$ (i.e., the domain is $T$ and the range is $Boolean$). The selection yields all objects from $A$ that satisfy $\theta$. The selection preserves sets, i.e., if $A : Set[T]$ then $\sigma_\theta(A) : Set[T]$.

$$\sigma_\theta(A) = \{\langle v, i\rangle \mid \langle v, i\rangle \in A \wedge \theta(v)\}$$

**Projection** $\pi_\rho(A) : MultiSet[S]$, where $\rho : T \to S$ is a function that transforms the objects in $A$ to objects of type $S$. Multiple tuples can be transformed to the same result. Hence, the result may always be a multiset and the count of an object in the result is equal to the sum of all objects that are projected onto the same image.

$$\pi_\rho(A) = \{\langle v, k\rangle \mid \exists \langle x, n\rangle \in A \wedge v = \rho(x) \wedge k = \sum_{\substack{\langle a,i\rangle \in A \wedge \\ v=\rho(a)}} i\}$$

**Cartesian product** $(A \times B) : MultiSet[(T, V)]$ builds tuples of all combination of the objects in $A$ and $B$. The Cartesian product preserves sets if both input relations are sets, i.e., if $A : Set[T]$ and $B : Set[V]$ then $A \times B : Set[(T, V)]$.

$$A \times B = \left\{\left\langle (u, v), i * j\right\rangle \mid \langle u, i\rangle \in A \wedge \left\langle v, j\right\rangle \in B\right\}$$

**Join** $(A \ {}_\rho\!\bowtie_\varphi B) : MultiSet[(T, V)]$ builds tuples of all combinations of the objects $t \in A$ and $v \in B$, where the join condition $\rho(t) = \varphi(v)$ is true. The functions $\rho : T \to J_1$ and $\varphi : V \to J_2$ can return values of any types $J_1$,$J_2$ under the condition that an equality comparison operator is defined between $J_1$ and $J_2$.

The result of a join is equivalent to $\sigma_\theta(A \times B)$, where the filter function $\theta$ encodes the join condition.

The filter function takes a tuple procured by the Cartesian product – thus filter has the type $\theta : ((T, V)) \to Boolean$ – and evaluates the functions $\rho$ and $\varphi$ on the first and second component of the tuple. Hence, in the definition below we assume that the components can be accessed as properties *first* and *second* via the dot notation. The join operator preserves sets, as per the definition of selection and Cartesian product.

$$A \ {}_\rho\!\bowtie_\varphi B = \sigma_{\lambda t.\, \rho(t.\mathit{first}) = \varphi(t.\mathit{second})}(A \times B)$$

**Duplicate Elimination** $\delta(A) : Set[T]$, transforms a multiset $A$ into a set.

$$\delta(A) = \{v \mid \langle v, i\rangle \in A\}$$

**Aggregation** $\gamma_{\rho,\alpha}(A) : Set[(G, AV)]$, where $\rho : T \rightarrow G$ is a function that transforms the objects in $A$ to objects of type $G$, and $\alpha : MultiSet[T] \rightarrow AV$ is an aggregation function over multisets that builds a single value of type $AV$. Informally the aggregation forms groups of objects over the relation, where all elements in the group are transformed to the same value under $\rho$. Hence, the set of all groups is given by $\delta(\pi_\rho(A))$. The final results of the operator are the tuples containing the image of $\rho$ and the aggregate value of the respective group.

Typical aggregation functions are *COUNT*, *SUM*, *MIN*, *MAX*, that work over integers, but the operator works with any aggregation functions over any type. In the case the of the typical integer aggregation functions, we provide default implementations that are parameterized by an additional function $T \rightarrow Int$, e.g., $MIN(\lambda x\ x.age)$. The latter is similar to the treatment of aggregation functions in SQL where the function is parameterized by the column names over which the aggregation should be performed.

Note that the grouping function ($\rho$) can be considered optional, i.e., if no grouping is present the entire aggregation function is applied to the entire relation. Omission of the grouping function is semantically equivalent to using a grouping function that returns the same value for all elements, e.g., $\lambda x.\, 0$.

$$\gamma_{\rho,\alpha}(A) = \bigcup_{g \in \delta(\pi_\rho(A))} \Big\{(g, a) | a = \alpha(\sigma_{\lambda t.\, \rho(t)=g}(A))\Big\}$$

## 3.3.3 Set-Theoretic Operators

The algebra features *set operators* for *union*, *intersection* and *difference*, of multisets. The standard set-theoretic semantics can be extended in two different ways for multisets. The difference lies in the treatment of multiple occurrences of objects that are identical in their properties, i.e., all properties in two objects yield the same values. The first extension treats multiple occurrences of identical objects as *indistinguishable*. The second extension treats all objects as *differing*, even though they may be identical in their values. Both extensions have their utility.

The indistinguishable semantics is useful for formulating correct transformations of boolean predicates in selections. Recall that in relational algebra the combination of predicates using boolean operators is translated via the following formulas:

$$\sigma_{\theta_1 \vee \theta_2} = \sigma_{\theta_1} \cup \sigma_{\theta_2} \quad \sigma_{\theta_1 \wedge \theta_2} = \sigma_{\theta_1} \cap \sigma_{\theta_2} \quad \sigma_{\theta_1 \wedge \neg \theta_2} = \sigma_{\theta_1} - \sigma_{\theta_2}$$

These properties are carried over by the indistinguishable semantics of the set operators, since these operators reduce to their usual semantics if the operands are sets. Consider for example a query on the student registration database in Figure 3.4a, that selects all students named "Sally" or all students with an average grade below 3.0:

$$\sigma_{\lambda s.\, s.\mathit{firstName}=\text{'Sally'} \vee s.\mathit{gradeAverage}<3.0}(E_{Students})$$

Clearly this query should yield the first two rows in Figure 3.4a and each row should be present only once in the result. However, the disjunction in the predicate will yield true for both the first condition ($\mathit{firstName} = \text{'}\mathit{Sally}\text{'}$) and the second condition ($\mathit{gradeAverage} < 3.0$) for the row containing the student named "Sally". However, the union over two selections that evaluate the first and second condition separately must contain the row for "Sally" only once, i.e., both results are indistinguishable.

In contrast the differing semantics is required for correct unions over projections. In the relational algebra a union of two projections ($\pi_\rho(A) \cup \pi_\rho(B)$) can be transformed using the following formula.

$$\pi_\rho(A) \cup \pi_\rho(B) = \pi_\rho(A \cup B)$$

For illustration, consider $A = E_{Students}$ (as in Figure 3.4a), and let $B$ be the relation as shown below in Figue 3.9, and $\rho = \lambda s.\, s.\mathit{firstName}$. Then the result of $\pi_\rho(A \cup B)$

| OID | firstName | lastName | grades | gradeAverage |
|---|---|---|---|---|
| $o_{10}$ | John | Smith | $\{3, 4\}$ | 3.5 |

**Figure 3.9:** Additional student stored in relation B

should clearly be a the multiset $\{\langle \mathit{Sally}, 1\rangle, \langle \mathit{John}, 2\rangle\}$. However, in the union over two projections, the indistinguishable semantics treats instances of the name "John" as duplicate values and, hence, $\pi_\rho(A) \cup \pi_\rho(B) = \{\langle \mathit{Sally}, 1\rangle, \langle \mathit{John}, 1\rangle\}$.

To remedy the above situation a separate operator (denoted $A \uplus B$) is defined for union with the differing semantics. The differing semantics has the added benefit of being cheaper to maintain in materialized views. Note that the operators for set intersection and set difference reduce to trivial meanings under the assumption that all elements are differing, i.e., $A \cap B = \emptyset$ and $A - B = A$; thus they require no special treatment.

**Definitions**

Let $A : MultiSet[T]$ and $B : MultiSet[V]$ be two multisets containing objects of types $T$ and $V$ then the set-theoretic operators of the relational algebra are defined as follows:

Union (indistinguishable) $(A \cup B) : MultiSet[U]$, contains occurrences of objects that are contained in $A$ or in $B$ or both. Duplicates for equal objects occurring in both relations amount to the maximum of all duplicates in $A$ and $B$. The type $U$ of the elements contained in the union must be a common supertype of $T$ and $V$, i.e., $T <: U \wedge V <: U$. The type must be provided as part of the operator and object equality must be defined between elements of type $T$ and elements of type $U$ (or between $V$ and $U$). The indistinguishable union yields a set if both operands are sets, i.e., if $A : Set[T]$ and $B : Set[V]$ then $A \cup B : Set[U]$.

$$A \cup B = \left\{ \langle v,k \rangle \,|\, \exists i \exists j.\, \langle v,i \rangle \in A \wedge \left\langle v,j \right\rangle \in B \wedge k = \max(i,j) \wedge k \neq 0 \right\}$$

Union (differing) $(A \uplus B) : MultiSet[U]$, contains all occurrences of all objects from $A$ and $B$. Duplicates for equal objects occurring in both relations amount to the sum of all duplicates in $A$ and $B$. The type $U$ of the elements contained in the union must be a common supertype of $T$ and $V$, i.e., $T <: U \wedge V <: U$. The type must be provided as part of the operator. A notion of object equality is not necessary, since the differing union treats all objects as unique elements. The differing union yields a multiset in general. A set is only returned if both operands are non-overlapping sets, i.e., if $A : Set[T]$ and $B : Set[V]$ and $A \cap B = \emptyset$ then $A \uplus B : Set[U]$.

$$A \uplus B = \left\{ \langle v,k \rangle \,|\, \exists i \exists j.\, \langle v,i \rangle \in A \wedge \left\langle v,j \right\rangle \in B \wedge k = i + j \wedge k \neq 0 \right\}$$

Intersection $(A \cap B) : MultiSet[U]$, contains all objects that are both in $A$ and $B$. Duplicates for equal objects occurring in both relations amount to the minimum of all duplicates in $A$ or $B$. The type $U$ of the elements contained in the intersection must be a common supertype of $T$ and $V$, i.e., $T <: U \wedge V <: U$. The type must be provided as part of the operator and object equality must be defined between elements of type $T$ and elements of type $U$ (or between $V$ and $U$). Otherwise the result is guaranteed to be the empty set, since no combination of two objects from $A$ and $B$ can be considered equal. The intersection yields a set if either operand is a set, i.e., if $A : Set[T]$ or $B : Set[V]$ then $A \cap B : Set[U]$.

$$A \cap B = \left\{ \langle v,k \rangle \,|\, (v,i) \in A \wedge (v,j) \in B \wedge k = \min(i,j) \wedge k \neq 0 \right\}$$

Difference $(A - B) : MultiSet[T]$, contains all objects that are in $A$ and not in $B$. Duplicates for tuples occurring in both relations amount to the difference of all duplicates in $A$ and $B$. The types $T$ and $V$ of the elements in the operands must either be subtypes, i.e., $T <: V$ (or $V <: T$), or they must have a common supertype $U$ and object equality must be defined between elements of type $T$ and elements of type $U$ (or between $V$ and $U$). Otherwise the result is guaranteed to be the set $A$, since no combination of two objects from $A$ and $B$ can be considered equal. The difference yields a set if the first operands is a set, i.e., if $A : Set[T]$ then $A \cap B : Set[T]$.

$$A - B = \{\langle v, k\rangle \mid \langle v, i\rangle \in A, \langle v, j\rangle \in B \wedge k = \max(i - j, 0) \wedge k \neq 0\}$$

## 3.3.4 Advanced Operators

The operators (*UnNest* and *Nest*) deal with objects that contain collections as part of their definition[4], e.g., the `grades` attribute of the type `Student` as depicted in 3.2. Such objects can occur frequently in an OO design, when the data contained in the collection is deemed a logical constituent of the enclosing object. In the example in Figure 3.4 we modeled the data for a student's grades as a nested collection that naturally belongs to a student. In first-normal-form the student grades would be unnested as a "flat" relation. Figure 3.10 exemplifies the flattened relation of all student grades. We omit object identifiers for each entry for brevity.

| student | grade |
|---|---|
| $o_1$ | 1 |
| $o_1$ | 3 |
| $o_1$ | 2 |
| $o_2$ | 4 |
| $o_2$ | 1 |
| $o_2$ | 1 |
| $o_2$ | 3 |

**Figure 3.10:** Example of an extent ($E_{Grades}$) with unnested student grades

Each representation (nested or unnested) offers a concise form of querying for a certain scenario. The nested representation is more concise for queries that

4 A situation also found in relational database that are in "non first-normal-form" [JS82]

involve only the values in a collection belonging to one object. The unnested representation is more concise for queries that involve the values of all collections in all objects. Hence, the Nest and UnNest operators can transform a flat representation of database tuples into a collection that belongs to an object (Nest) and vice versa (UnNest).

For illustration consider our running example of students and their grades. The unnested relation $E_{Grades}$ works well with relational algebra operators that reason over all grades for all students. Consider for example the following query:

*Find the minimal grade given to any student*

In the relational algebra the query can easily be formulated as an aggregation using the *MIN* function over the unnested relation:

$$\gamma_{MIN(\lambda x\ x.grade)}(E_{Grades})$$

The query is not as concise if we only have the nested relation available, i.e. without support for for unnesting. In this case the nested relation must be queried by building the minimum of the grades for each student and then a minimum over the resulting values:

$$\gamma_{MIN}(\pi_{\lambda s\ MIN(s.grades)}(E_{Students}))$$

In contrast, the nested relation works well for queries that reason over the grades of a single student. Consider for example the following query:

*Find all students that have at least one grade equal*
*to 1 and one grade equal to 4.*

Given that the collections support appropriate abstractions, we can phrase the query such that the set $1,4$ is a subset of a student's grades as follows:

$$\sigma_{\lambda s\ Set(1,4).subset(s.grades)}(E_{Students})$$

The unnested relation requires a join to state the same query, since we can only reason over one grade at a time. Thus we have to find two entries, i.e., one for the grade 1 and one for the grade 4, and join them to see that they belong to the same student. The following operators briefly exemplify the join (the concrete join conditions are omitted):

$$\sigma_{\lambda s\ s.grade=1}(E_{Grades}) \bowtie \sigma_{\lambda s\ s.grade=4}(E_{Grades})$$

In summary, the Nest and UnNest operators can transform the database representation to allow more concise queries. In practice, we were mostly interested in the unnest operator, since it was more common that a query was better expressible in the unnested relation, while queries where the nested relation has an advantage – as exemplified above – were rare.

The *transitive closure* operator provides a limited form of recursion. Consider for example the class `MasterCourse` – as defined in Figure 3.6 – that has a prerequisite course. Since the prerequisite can also be a master course a transitive chain of prerequisite courses can exist. Queries using the basic and set-theoretic operators defined in the previous sections can only express selections on the direct prerequisite course; retrieved via the `prerequisites` property of the `MasterCourse`. A transitive closure operator can find all direct and transitively reachable prerequisites.

*Recursion* allows the results of an operator that depends on values from *A* to be used as new inputs for *A*. As noted in Chapter 2 the transitive closure also provides a limited form of recursion. Hence, it is important to differentiate these two concepts a little further. The transitive closure allows a recursive traversal over a graph formed by the objects contained in a relation. However, the underlying assumption is that all objects that form the graph are present as values in the relation. This allows to express a large category of recursive functions. Yet, it can not express functions that recursively depend on values computed as part of the recursion.

For illustration, consider the function `rec` in Figure 3.11. The function defines a

```
1 def rec(in: List[Data], fix: List[Data]): List[Data] = {
2   val next = new Data(transform(in.head))
3   if(!fix.contains(next))
4     rec(next :: in, next :: fix)
5   else if(in != Nil)
6     rec(in.rest, fix)
7   else
8     fix
9 }
```

**Figure 3.11:** Example for a fixpoint recursion

fixpoint recursion, where the initial input data is passed as a list `in` and the fixpoint is also passed as a parameter `fix` (line 1); typically with the value `Nil` when the recursion is started. In each step, a new value is computed as a transformation from the first element in the list (line 2). If the new value is not contained in the fixpoint (line 3) it is passed into the recursion as new input data (line 4). Otherwise, the rest of the input is processed (line 6) until no more elements are left in the input (line 8). The transitive closure cannot express a recursive addition of computed values, e.g., a call as found line 4. However, the important thing to note here is that the transitive closure can be seen as a special case of recursion, i.e., a recursive traversal over a set of edges, and can be maintained more efficiently in terms of the runtime of incremental updates.

**Definitions**

Let $A : MultiSet[T]$ be a multiset containing objects of types $T$ then the set extended operators of the relational algebra are defined as follows:

**Nest** $\nu_{\rho,\varphi}(A) : Set[(S, MultiSet[V])]$, where $\rho : T \rightarrow S$ is a function that transforms the objects in $A$ to objects of type $S$ and $\varphi : T \rightarrow V$ is a function that transforms the objects in $A$ to objects of type $V$. Informally the nesting groups objects in the relation, where all elements in the group are transformed to the same value under $\rho$. For each group a multiset of objects – containing elements transformed using the function $\varphi$ – is returned, that contains the nested values.

The operator can be reduced to a special form of aggregation (cf. Sec. 3.3.2). This is due to our broad treatment of aggregation, i.e., we consider arbitrary aggregation functions and not only aggregations over integers such as *COUNT*, *AVERAGE*, etc. The aggregation function $\alpha$ takes as input the multiset of the group and transforms the values using $\varphi$. Hence, the aggregation function is an application of the function *map* in the usual functional programming sense [Pie02].

Note that we give a definition of the nesting operator where the returned values can be transformed – via *map* – to construct new objects. These objects contain the nested values, e.g., as a property. The same effect could be achieved by defining a projection on top of a simple operator, i.e., storing the original objects in a collection and applying the *map* function in the projection. However, since all objects are processed by the operator in any case, the given definition is more efficient.

$$\nu_{\rho,\varphi}(A) = \gamma_{\rho,\lambda G.\,map(G,\varphi)}(A)$$

**UnNest** $\mu_{\rho}(A) : MultiSet[(T, V)]$ , where $\rho : T \rightarrow MultiSet[V]$ is a function that transforms the objects in $A$ to multisets containing objects of type $V$. The unnesting takes all values contained in the multiset returned by $\rho$ and combines them with the respective object of $A$, which contained the multiset. The operator returns a set if $A$ is a set and for all $t \in A$ the function $\rho$ returns a set.

$$\mu_\rho(A) = \{\langle (t,v), i * k\rangle \mid \langle t,i\rangle \in A \wedge S = \rho(t) \wedge \langle v,k\rangle \in S\}$$

Transitive Closure $TC_{\rho,\varphi}(A) : Set[(S,U)]$, where the objects in $A$ represent edges and $\rho : T \to S$ and $\varphi : T \to U$ are functions that are used to transform the edges to their respective start and end vertex. The *graph* over $A$ is then defined as $G_A = (V,E)$ where $E$ is the set of edges and $V$ the set of vertices, which are defined as:

$E = \{(v,u) | \exists t \in A \wedge v = \rho(t) \wedge u = \varphi(t)\}$

$V = \{v | \exists u.(u,v) \in E \vee \exists u.(v,u) \in E\}$

A sequence of vertices $(v_0, v_1, \ldots, v_n)$ is a *path* of length $n$ if $(v_i, v_{i+1}) \in E$ for all $i \in [0 \ldots n-1]$ and if $v_i \neq v_j$ for all $0 \leq i < j \leq n$. The sequence is a *cycle* of length $n$ if $v_0 = v_n$.

The transitive closure over $A$ is then defined via the existence of a positive path in $G_A$ (cf. [AMO93]). The result are always tuples of vertices, i.e., of type $(S,U)$, since – while original objects in $A$ are edges of type $T$ – a general transformation from a pair of vertices to objects of type $T$ can not be automatically derived. If such a transformation exists, it can be expressed as an additional projection on the result of the transitive closure. Note that the start and end vertex are not required to have the same type. During the construction of paths all objects are compared via object equality, hence there can be no runtime errors.

The transitive closure operator used in this thesis always treats the results as a set, i.e., multiple paths from one vertex to another are not treated as multiple results. In this respect the defined transitive closure differs from the treatment of right-linear-chain recursion in logic programming (cf. Sec. 2.3.1), where multiple paths yield multiple results. However, (at least) for the purposes of the static analyses defined in this thesis this treatment is sufficient.

$$TC_{\rho,\varphi}(A) = \{(u,v) | \exists \textit{ a path with positive length from } u \textit{ to } v \textit{ in } G_A\}$$

Note that we make no differentiation for cyclic or acyclic graphs w.r.t. the above defined set semantics, since the existence of a path of positive length is indifferent to cyclic or acyclic paths. Nevertheless, a treatment of cycles must be performed in the computation of results, which will be discussed together with the incremental maintenance of the operator in Section 3.4.3.

Recursion $REC(A, Op(A)) : MultiSet[U <: T]$, where $Op(A)$ is a combination of relational algebra operators that depend on $A$, i.e., have $A$ as one of their parameters. The recursion considered in this thesis is a fixpoint recursion that eliminates duplicate recursive derivations. Hence, recursive functions that would generate an infinite number of derivations are expressible and the finiteness is ensured by our treatment of recursion.

The recursion is basically a substitution in the tree formed by operators that depend in some way on the input relation $A$. The operators can form a complex expression which is denoted as $Op(A)$ in the following. To define recursion we introduce a notion of substituting operators in $Op(A)$ such that the recursion can introduce a cyclic dependency into the operator tree which lets results of $Op(A)$ be propagated back to $A$. Let $R$ be a tree of operators of the relational algebra. Then a substitution is denoted by:

$$Subst(R, Old, New) = R' : \textit{all occurrences of Old in R are replaced by New in } R'$$

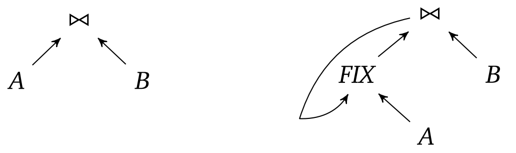
A recursive definition performs a substitution in the tree of *Op(A)* at the place where $A$ is used. The substituted value is the union of all original elements from $A$ together with the results from *Op(A)*. To introduce the notion of fixpoints the results of *Op(A)* are passed to an operator *FIX*, which can for now be thought of as taking the union of two inputs and removing all duplicates, i.e., *FIX(A,B)* $\equiv \delta(A \uplus B)$. Note that for an incremental computation the semantics of *FIX* is more complex. For type-safety the elements in *Op(A)* must be compatible to the elements in $A$. Hence, if the type of the elements in $A$ is $T$ then the type of the elements in *Op(A)* must be of some $U <: T$.

$$REC(A, Op(A)) = Subst(Op(A), A, FIX(A, Op(A)))$$

For illustration consider Figure 3.12, where the non-recursive definition of *Op(A)* is shown in Figure 3.12a and the substituted operator tree is shown in Figure 3.12b. The initial operator tree is defined by the expression *Op(A)* $= A \bowtie B$ – note that the concrete join-functions are omitted for brevity. Arrows indicate the direction in which data flows between the operators.

### Recursion and transitive closure

As noted earlier the transitive closure is a special case of recursion. Using the above definition we can phrase the transitive closure via a general recursion as shown in the equation below. Note that the results of the transitive



**(a)** Non-Recursive Definition: *Op(A)* $= A \bowtie B$ **(b)** Substituted Operator Graph: *Subst* (*Op*(*A*), *A*, *FIX* (*A*, *Op*(*A*)))

**Figure 3.12:** Substitution of Recursive Operators

closure are always tuples of vertices, i.e., of type $(S, U)$ where $S$ is the type of elements returned by $\rho$ and accordingly $U$ is returned by $\varphi$. Hence, the equality below is given by wrapping the transitive closure in a projection, where the projection function $\alpha$ transforms the tuples back into elements of *A*; in short, the type of $\alpha$ is $(S, U) \rightarrow T$. A formulation using only *TC* on the left hand side is also possible, but states equality to a recursion over a projection of *A* to vertices and not a recursion over *A* itself.

$$\pi_\alpha\left(TC_{\rho,\varphi}(A)\right) = \delta\left(A \uplus REC\left(A, \pi_{\lambda x.\,\alpha\left(\rho(x.first),\varphi(x.second)\right)}\left(A\ {}_{\varphi}\!\bowtie_{\rho} A\right)\right)\right)$$

The equality between transitive closure and recursion can be understood as follows: First the transitive closure contains all edges themselves, i.e., *A* plus all edges that can be found by recursively concatenating existing edges. Concatenation in this case means that $(u, v)$ and $(v, w)$ are concatenated to $(u, w)$. Note that the recursion will only find concatenated edges and not necessarily those in *A*, but the edges in *A* can be derived a second time in case of cyclic graphs. Hence, the complete result is wrapped in a duplicate elimination to make it comparable to the treatment of *TC*, where recursively concatenated edges that are already in *A* do not count as a new result.

The concatenation of edges via a join in the recursion is similar to the treatment of the transitive closure. In the transitive closures paths are built by joining edges, where edges are constructed from objects via the functions $\rho$ and $\varphi$. Thus, if there are two objects $o_1$ and $o_2$, the constructed edges are $\left(\rho(o_1), \varphi(o_1)\right)$ and $\left(\rho(o_2), \varphi(o_2)\right)$. A joined edge – or path – exists if the end vertex of the first edge is equal to the start vertex of the second edge, in short $\varphi(o_1) = \rho(o_2)$. In this case the resulting joined edge is between the start vertex of the first edge and the end vertex of the second edge, i.e., the result is $\big(\rho(o_1), \varphi(o_2)\big)$. In the recursion the equivalent to finding the existence of a new paths is the self-join on $A$ with the respective functions $\varphi$ and $\rho$, i.e., the join ${}_{\varphi}\bowtie_{\rho}$ will find corresponding objects such that $\varphi(o_1) = \rho(o_2)$. The result is a tuple $\big(o_1, o_2\big)$ from which the concrete new path is constructed via a projection. The recursion – as the transitive closure – runs until a fixpoint is reached, i.e.,until no new paths are constructed. The existence of previously computed paths is ensured via the fixpoint operator *FIX*. To provide a correctly typed recursive definition the constructed edge is transformed by $\alpha$. Thus the resulting objects of the recursion are always objects of the type that elements in $A$ have.

### Restrictions on recursion

It is a well-known fact in deductive databases that negation inside a recursion can lead to non-monotonicity. Hence, these systems only allow stratified negation, i.e., a negation may only be used inside a recursion if it does not depend on that recursively defined rule. In terms of operators in the relational algebra, this means that recursive definitions using set differences – the algebraic form of negation – are not allowed. For example, a recursive definition of $R = A - R$ is not allowed. The problem of non-monotonicity manifests itself in an oscillating behavior of the evaluation. For illustration consider the stepwise evaluation of $R$ shown in the table below. In the first step the value 1 is added $A$ and $R$ is empty. In the second step 1 is added to $R$, since $A - R$ evaluates to $\{1\} - \emptyset$. However, due to the recursion this means that the $R$ must be evaluated again with the new value. This third step reduces $R$ again to the empty set, since $\{1\} - \{1\} = \emptyset$. This situation is repeated endlessly, thus we can not find a faithful evaluation of $R$.

| | $A$ | $R$ |
|---|---|---|
| Step 1: | $\{1\}$ | $\emptyset$ |
| Step 2: | $\{1\}$ | $\{1\}$ |
| Step 3: | $\{1\}$ | $\emptyset$ |

It is less well known that for aggregations the same restrictions apply. For deductive databases this is not frequently discussed since not all systems allow aggregations. Nevertheless, the issue is discussed in the context of recursive extensions to relational databases (e.g. by Garcia-Molina et al. [GMUW08] who give an argument about aggregations and sums as we will use below). The argument is in essence the same as for negations. Consider a recursive definition of $R = \gamma_{SUM}(A \uplus R)$, i.e., the summation of all values from $A$ and

$R$. The stepwise evaluation would proceed as shown below. The interesting points are steps 4 and 5. In step 4 the value of the aggregation is updated recursively to the result 10. However, when the new result enters the recursion again in step 5, we have lost the previous value of $R$, i.e., the value 5 is no longer present in $A \uplus R$. Thus, the behavior of the recursive aggregation is non-monotonic.

| | $A$ | $R$ | $A \uplus R$ | $\gamma_{SUM}(A \uplus R)$ |
|---|---|---|---|---|
| Step 1: | $\{1,4\}$ | $\emptyset$ | $\{1,4\}$ | 5 |
| Step 2: | $\{1,4\}$ | $\{5\}$ | $\{1,4\}$ | 5 |
| Step 3: | $\{1,4\}$ | $\{5\}$ | $\{1,4,5\}$ | 10 |
| Step 4: | $\{1,4\}$ | $\{10\}$ | $\{1,4,5\}$ | 10 |
| Step 5: | $\{1,4\}$ | $\{10\}$ | $\{1,4,10\}$ | 15 |

Note that non-monotonicity in recursive aggregation is a tricky argument. Intuitively, there are a lot of day-to-day recursive computations that work towards a fixpoint and use operations similar to aggregations. The underlying assumption is always that the aggregation functions eventually converge to a single value. However, even such functions are problematic in the database setting and basically suffer from the problem that the fixpoint becomes self-sustaining. For illustration, consider the stepwise evaluation of $R = \gamma_{\cup}(A \uplus R)$ (shown below). In this setting the values in the database are sets and the aggregation is performed as set union, which eventually converges to a single value. In the below example we add elements $\{a\}$ and $\{b\}$ to $A$ until we eventually reach a fixpoint $\{a,b\}$ for the aggregation in step 3. In step 4 the fixpoint is added to $R$ and, since the aggregation is a convergent function the value does not change further. However, if we remove the value $\{a\}$ in step 5, no change is performed for the aggregation, since the fixpoint sustains itself. This behavior is very unintuitive and typically does not arise when using fixpoint computations in recursive programming, since the input data is usually not subject to incremental changes.

| | $A$ | $R$ | $A \uplus R$ | $\gamma_{\cup}(A \uplus R)$ |
|---|---|---|---|---|
| Step 1: | $\{\{a\}\}$ | $\emptyset$ | $\{\{a\}\}$ | $\{\{a\}\}$ |
| Step 2: | $\{\{a\}\}$ | $\{\{a\}\}$ | $\{\{a\},\{a\}\}$ | $\{\{a\}\}$ |
| Step 3: | $\{\{a\},\{b\}\}$ | $\{\{a\}\}$ | $\{\{a\},\{a\},\{b\}\}$ | $\{\{a,b\}\}$ |
| Step 4: | $\{\{a\},\{b\}\}$ | $\{\{a,b\}\}$ | $\{\{a\},\{b\},\{a,b\}\}$ | $\{\{a,b\}\}$ |
| Step 5: | $\{\{b\}\}$ | $\{\{a,b\}\}$ | $\{\{b\},\{a,b\}\}$ | $\{\{a,b\}\}$ |

## 3.3.5 Derived Operators

The following operators provide existential quantification over multiple relations. The semantics for existential quantification can in general be derived from the operators defined in the previous sections; hence, they are categorized as *derived operators*.

Queries with existential quantification act similar to a selection operator with a filter condition (discussed in Sec. 3.3.2). However, the filter condition is now based on existential quantification over a second relation, i.e., the existential quantification is a selection of all elements that also exist in another relation. Consider for example the relations $E_{Students}$ and $E_{Registrations}$ from Figure 3.4. Using existential quantification we can select all students that have registered for a course, i.e., there exists at least one entry in $E_{Registrations}$ for the respective student. Likewise we can select all students that have not registered for any course, i.e., there exists no entry in $E_{Registrations}$ for the respective student.

In standard database terminology the respective operators are termed *semi-join* (exists) and *anti-semi-join* (not exists).

**Definitions**

Let $A : \mathit{MultiSet}[T]$ and $B : \mathit{MultiSet}[V]$ be two multisets containing objects of types $T$ and $V$ then the basic operators of the relational algebra are defined as follows:

Semi-Join $(A\ {}_{\rho}\ltimes_{\varphi}\ B) : \mathit{MultiSet}[T]$ selects objects from $A$ such that for $t \in A$ and $v \in B$, the condition $\exists y.y = \varphi(v) \wedge y = \rho(t)$ is true. The functions $\rho : T \rightarrow J_1$ and $\varphi : V \rightarrow J_2$ can return values of any types $J_1, J_2$ under the condition that an equality comparison operator is defined between $J_1$ and $J_2$. The semi-join returns a set if the relation $A$ is a set.

In general the semi-join can be expressed as a join. The right-hand side is the relation $A$, the left-hand side is the set of values that are the images of the function $\varphi$ when applied to objects in $B$. Using a set – by duplicate elimination – on the left-hand side reflects the semantics, that at least one object has to exist in $B$ that matches a respective object in $A$. The outermost projection retrieves the elements from $A$ form the joined tuples.

$$A\ {}_{\rho}\ltimes_{\varphi}\ B = \pi_{\lambda t\ t.\mathit{first}}(A\ {}_{\rho}\bowtie_{\lambda t.t}\ \delta(\pi_{\varphi}(B)))$$

Anti-Semi-Join $(A\ {}_{\rho}\triangleright_{\varphi}\ B) : \mathit{MultiSet}[T]$ selects objects from $A$ such that for $t \in A$ and $v \in B$, the condition $\nexists y.y = \varphi(v) \wedge y = \rho(t)$ is true. The functions

$\rho : T \to J_1$ and $\varphi : V \to J_2$ can return values of any types $J_1, J_2$ under the condition that an equality comparison operator is defined between $J_1$ and $J_2$. The anti semi-join returns a set if the relation $A$ is a set.

In general the anti-semi-join can be expressed via a set difference and a semi-join. The right-hand side of the difference is the relation $A$, the left-hand side is semi-join of $A$ and $B$, i.e., we subtract from $A$ the (multi-)set of objects in $A$ that have a corresponding object in $B$.

$$A \;{}_{\rho}{\rhd}_{\varphi}\; B = A - (A \;{}_{\rho}{\ltimes}_{\varphi}\; B)$$

## 3.4 Incremental Maintenance of Relational Operators

The incrementalization works by building an operator tree in which each operator knows its immediate successors. The successors are notified of respective modification events, whenever the data of the underlying operator changes. Hence, the incrementalization works similar to an *observer pattern* [GHJV95]. The operators are registered as successors (observers) during the compilation of queries. Note that operators also know their predecessors and can – in essence – perform a non-incremental top-down evaluation using these references. This form of evaluation is used, for example, to initialize operators that are defined on top of materialized views which already contain evaluation results.

### 3.4.1 Observer Pattern Style Change Propagation

To illustrate the observer pattern style used for the maintenance of views, consider the definition of the `Observer` trait in Figure 3.13. The observer declares three atomic events for additions (line 2), removals (line 3) and updates (line 4). An operator registers itself as an observer and receives atomic modification events through any of these three methods.

In addition to atomic modifications each operator can receive a set of modifications – basically a transaction without any rollback functionality – via the method `modified` (line 6). A set of modifications is a triple consisting of additions, deletions and updates. The concrete modifications use a multiplicity counter, e.g., the type `Addition` includes a reference to the added value and a number indicating how often the value is added. Treating the set of modifications as an atomic event allows to enforce minimality conditions, e.g., no elements are added and immediately deleted afterwards. The minimality of the set of modifications will be treated in Sec. 3.4.3.

```
trait Observer[V] {
  def added(v: V)
  def removed(v: V)
  def updated(oldV: V, newV: V)

  def modifed(additions: Set[Addition[V]],
              removals: Set[Removal[V]],
              updates: Set[Update[V]])
}
```

**Figure 3.13:** Observer trait used during change propagation

Base relations (of type `Extent`) and operators of the relational algebra for multisets (presented in Sec. 3.3) are defined as instances of the trait `Observable` (cf. Figure 3.14).As such each base relation and each view – defined via one or more operators – can notify other views of respective changes. Notifications are triggered by respective notify methods, e.g., for additions (line 4); deletions, updates and sets of modifications work accordingly. The triggering of notifications happens in response to a received event and is specific to each operator. For example, when an operator receives an added event, the operator must determine which elements are added in response to the event. Note that additions can result in deletions and vice versa – depending on the current operator.

```
1 trait Observable[V] {
2   def observers: Set[Observer[V]]
3
4   def notify_added(v: V) {
5     observers.foreach (_.added (v))
6   }
7         ...
8 }
```

**Figure 3.14:** Excerpt of the trait Observable

To illustrate the notification process an excerpt of the projection operator is presented in Figure 3.15. The projection observes an underlying relation of the type `Domain` (line 3) and transforms the elements using a projection function (line 2). Hence, the result of the projection is of type `Range` and the operator as a whole is accordingly declared as a respective `Observable` (line 4). The event notification is comparatively simple. For example, when the projection is notified that a new element is added, the element is transformed and a respective notification is fired to all observers (line 7).

```
 1 class Projection[Domain, Range]
 2         (val projection : Domain => Range)
 3         extends Observer[Domain]
 4         with Observable[Range]
 5 {
 6   def added(v: Domain) {
 7     notify_added (projection (v))
 8   }
 9   ...
10 }
```

**Figure 3.15:** Excerpt of the Scala class for projections

Binary operators are treated in a slightly different way, since an observer is required for the left-hand side as well as for the right-hand side operand. Consider, for example, the class for the Cartesian product operator in Figure 3.16. The binary operator builds the cross product over elements from two different domains (`DomainA` on the left hand and `DomainB` on the right hand side of the operator). Hence, we require two differently typed observers (line 6 and 15) that also propagate changes slightly differently. For example, the addition of an object to the operand on the left hand side (line 7) results in new tuples for the added object together with all objects stemming from the relation on the right hand side. In the example this semantics is achieved by iterating over all objects in the right hand side operand (`foreach` in line 8). The resulting tuples are propagated by an anonymous function passed to `foreach` (line 9), that is called for every object `b` in the right hand side operand. The observer of the right hand side treats an addition accordingly by mirroring the behavior for the left hand side. Note that while we introduce internal observers, e.g., for the left- and right-hand side in the Cartesian product, we omit these observers in the following treatment of operators. That means we uniformly treat operators as successor without going into the details of internally used observers e.g., the Cartesian product operator is the successor of its left- and right-hand operand.

Each operator registers itself (or appropriate internal objects) as an observer to its operand(s) at the time a query is compiled. Optimizations of the query expression are considered before compiling the query and are discussed in Chapter 4. An operator can be passed multiple times as an operand, and such an operator hence

```
class CartesianProduct[DomainA, DomainB](
    val left: Relation[DomainA],
    val right: Relation[DomainB])
  extends Observable[(DomainA, DomainB)]
{
  object LeftObserver extends Observer[DomainA] {
    def added(v: DomainA) {
      right.foreach (
        b => { notify_added (v, b) }
      )
    }
    ...
  }

  object RightObserver extends Observer[DomainB]
}
```

**Figure 3.16:** Excerpt of the Scala class for Cartesian product

has multiple observers. This behavior basically reflects sharing of sub-queries, i.e., the result of a maintenance event is only computed a single time for the operator and then propagated to multiple observers.

### 3.4.2 Examples for View Maintenance

To illustrate the incremental view maintenance, consider the following two examples of views on the student registration database depicted in Figure 3.4.

**Example 1 – Selection of last names of a filtered set of students**

The first example filters all students that have an average grade below 3.0 and selects their last name as a result. The respective query using the defined operators is given as $view_1$ in the following equation:

$$view_1 = \pi_{\lambda s.\, s.lastName}(\sigma_{\lambda s.\, s.gradeAverage<3.0}(E_{Students}))$$

The result of the query for a (non-incremental) evaluation on the data presented in Figure 3.4a, is depicted in Figure 3.17. Both students in the extent $E_{Students}$ pass the test in the filter, i.e., their grade average is lower than 3.0. Hence, their last names are selected into the result.

The incremental evaluation is performed via events in the operator tree of $view_1$ depicted in Figure 3.18. The successor relation in the tree is depicted by respective arrows in the Figure, e.g., from $E_{Students}$ to the $\sigma$ operator. The tree of $view_1$ is very simple, since each operator only has a single successor. The data flow of the incrementalization starts by adding (removing) objects to (from) the extent $E_{Students}$. Respective events are then propagated to the successor (the $\sigma$ operator), which in turn filters the students and propagates events to the $\pi$ operator, if an object passes the filter.

| OID | value |
|---|---|
| $o_{11}$ | Fields |
| $o_{12}$ | Dow |

**Figure 3.17:** Result of *view*$_1$ for data in Fig. 3.4a

$$\begin{tikzcd}
\pi_{\lambda s.\, s.lastName} \arrow[d, dash] \\
\sigma_{\lambda s.\, s.gradeAverage < 3.0} \arrow[d, dash] \\
E_{Students}
\end{tikzcd}$$

**Figure 3.18:** Operator tree for *view*$_1$

To illustrate the incrementalization an event for adding a new object is depicted in Figure 3.19 as a UML sequence diagram. The event is received by the base relation $E_{Students}$ (leftmost lifeline denoted as `students`) and propagated to the operators defined by *view*$_1$. The top most operator ($\pi$ in Figure 3.18) of the view is marked in gray in the figure and basically represents the result. In addition the Figure includes a `MaterializedRelation` node that is registered as an observer of the view and stores the results. Reconsider that the view itself makes no a priori assumptions whether it is only an intermediate step in a computation or the final result. Hence, the top most operator does not store results by itself, but delegates storage to the materialized node.

The data flow of the depicted event starts with adding an object to the extent `students` as follows:

```
val v = new Student('John', 'Smith', List(2, 2))
students.add(v)
```

The `added(v)` event propagates from the extent to the `Selection` ($\sigma$) operator. The object defined above passes the test of the `Selection` – average grade 2.0 < 3.0 – and the event is then propagated to the `Projection` ($\pi$) operator. In the `Projection` the object is transformed using the provided operation $\lambda s.\ s.lastName$ into a new

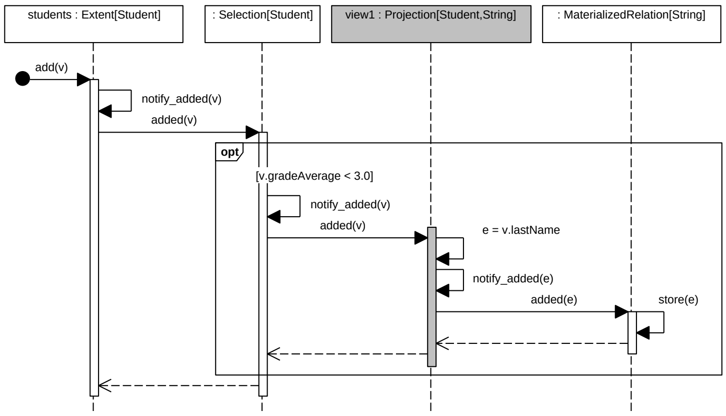


**Figure 3.19:** Data flow for propagating an addition from the base relation to *view*$_1$

value $e$. In the given example the transformed value $e$ is propagated to an instance of `MaterializedRelation` that stores the results of $view_1$.

As can be seen by the above example, the modifications are only propagated in the operator tree as far as they apply. Selections filter many elements out of the propagation process, e.g., all events (additions/deletions/updates) containing students that have an average grade above 3.0 are only propagated as far as the node of the selection operator. Other operators, such as duplicate eliminations, also propagate events only under certain condition, e.g., if an added element was not already present in the duplicate elimination.

**Example 2 – Selection of a filtered set of students registered for a course**

The second example features a more complex view with a binary operator. The view selects all students with an average grade below 3.0, which also have registered for a course. The respective query is defined below as $view_2$. The students are filtered similar to $view_1$ and the binary operator *exists* ($\ltimes$) is used to find students that also have a registration. The exists operator can be simplified to a join – as discussed in Section 3.3.5 – with the duplicate elimination ($\delta$) of registered students ($\pi_{\lambda r.\ r.student}(E_{Registrations})$) as the right hand operand.

$$\begin{aligned} view_2 &= \sigma_{\lambda s.\ s.gradeAverage<3.0}(E_{Students})\ {}_{\lambda s.s}\ltimes_{\lambda r.\ r.student}\ E_{Registrations} \\ &= \pi_{\lambda t.\ t.first}(\sigma_{\lambda s.\ s.gradeAverage<3.0}(E_{Students})\ {}_{\lambda s.s}\bowtie_{\lambda t.t}\ \delta(\pi_{\lambda r.\ r.student}(E_{Registrations}))) \end{aligned}$$

The incremental evaluation is performed by propagating events from the two base relations $E_{Students}$ and $E_{Registrations}$. Events are propagated from one base relation at a time via the operator tree depicted in Figure 3.20. Hence, the data flow starts either in $E_{Students}$ and propagates through the filter to the join or the data flow starts in $E_{Registrations}$ and propagates through the projection and the duplicate elimination to the join.

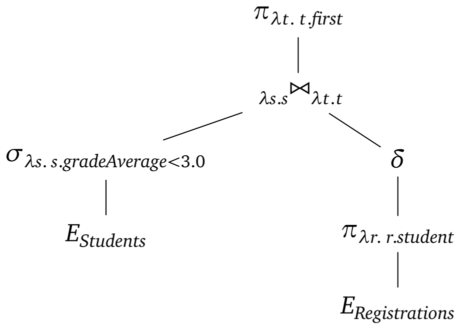

$\pi_{\lambda t.\ t.first}$
$_{\lambda s.s}\bowtie_{\lambda t.t}$
$\sigma_{\lambda s.\ s.gradeAverage<3.0}$
$\delta$
$E_{Students}$
$\pi_{\lambda r.\ r.student}$
$E_{Registrations}$

**Figure 3.20:** Operator tree for *view*$_2$

To illustrate the different data flows the sequence diagram in Figure 3.21 depicts one addition event for each base relation. The data flow is exemplified up to the join – marked in gray – which is positioned in the middle, such that the sequence diagram mimics the layout of the operator tree. Left of the join in the sequence diagram are the operators representing the left sub-tree of the join and the right hand is laid out accordingly. The base extents $E_{Students}$ and $E_{Registrations}$ are the outermost operators in the sequence diagram.

The first addition event is performed on $E_{Students}$ and propagation works similar to the propagation in Figure 3.19. Once the added object reaches the join it is internally added to the left hand side index (`addToLeftIndex(v)`). Note that the internal workings of the join are abbreviated here for space reasons. Conceptually the index is yet another observer of the underlying relation which notifies the join. From the added value and the right hand side index a (possibly empty) delta of new joined objects is computed (`delta = join(v, getRightIndex())`). Here we see that, in order to determine possible joined objects, the operator requires a materialization of all the data stemming from the right hand operand and the operator uses indexed data to efficiently determine joined tuples. After obtaining the delta, all observers of the join are notified for each added tuple (`notify_added(delta[i])`). As can be seen the addition of a single element can lead to multiple additions of results in a join operator. Note that a loop construct that emits atomic addition events was chosen in order to remain in this formalism and not interchange atomic valued events with sets of modifications.

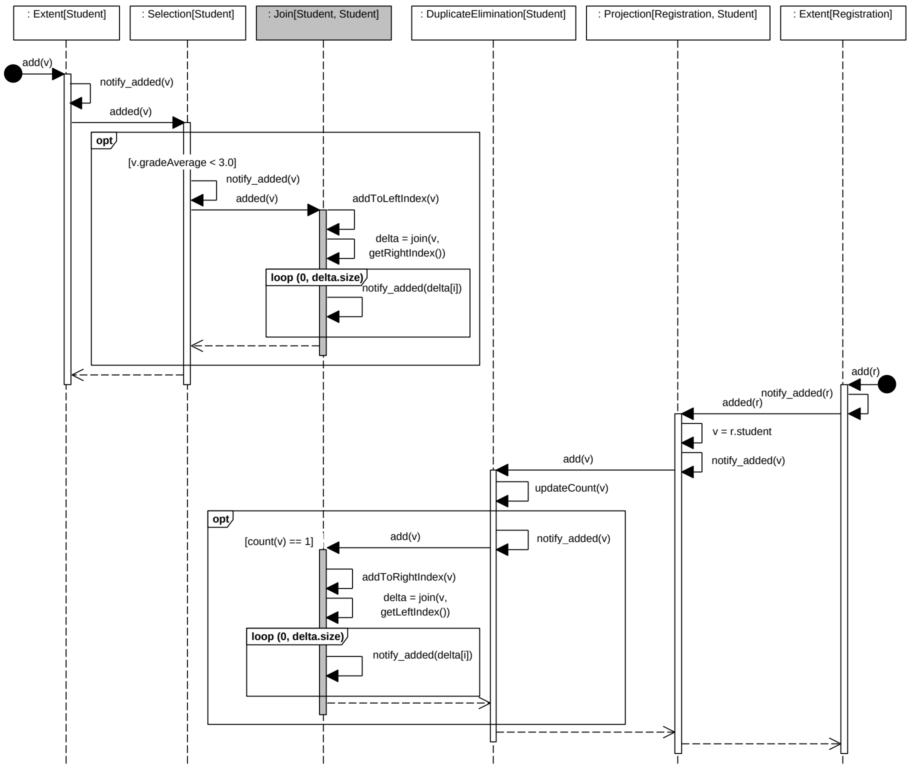


**Figure 3.21:** Data flow for propagating additions from the base relations to *view*$_2$

The second addition event is performed on $E_{Registrations}$. The first propagation is a projection which is similar to Figure 3.19. Afterwards the event is propagated to the duplicate elimination, which is one of the operators based on counting. As can be seen the duplicate elimination internally updates the count for the added element (`updateCount(v)`). An event is only fired if this was the first time the element was added to the duplicate elimination (`count(v) == 1`), otherwise the element was a duplicate and hence is already present as a result in subsequent observers. When the element is added for the first time, an according event is propagated to the join operator. As in the previous propagation from $E_{Students}$, the element is added to the index (but on the right hand side) and a delta of added objects is computed and propagated as results.

### 3.4.3 Change-Propagation Expressions

This section shows how each operator is maintained with respect to change events from underlying relations. The expressions obtained in this section are the minimal requirements for self-maintainability of each operator. In other words the expressions are derived such that a minimal amount of materialization is required to maintain one operator. The response to changes from underlying relations are discussed algebraically by giving a change-propagation expression for each operator, i.e., the expression basically models what happens in the observer pattern between a received event and a notification of subsequent observers. The algebraic discussion allows to reason over multisets of the underlying data that are required to be materialized. The implementation provided as part of this thesis also closely resembles the given propagation expressions. A treatment of optimizations for multiple operators is discussed at the end of this chapter.

In the following, the notation for modifications and prerequisites for the change-propagation expressions is discussed first. Then we discuss the maintenance of self-maintainable operators without auxiliary data. Finally, those operators that require some form of auxiliary data to be self-maintainable are discussed.

#### Notation of modifications

The change-propagation expressions are presented such that for each operator a set of modifications for addition ($\Delta^+$), deletions ($\Delta^-$) and updates ($\Delta^{upd}$) is derived. This basically reflects the three sets of modifications passed as parameters to the `modified` method of the observer depicted in Figure 3.13. Modifications always pertain to a specific relation (e.g., $A$), where additions and deletions are multisets – in the sense defined in Section 3.3 – of added and deleted elements (Equations 3.4.3.1 and 3.4.3.2) attached with a count of how many objects are added or deleted. Updates are sets of triples that contain the old value, the new value, and the number of updated objects (3.4.3.3). For abbreviation the combination of all three types of modifications for a given relation $A$ is written as a transaction ($T_A$), which is a triple of additions, deletions and updates to the relation (3.4.3.4).

$$\Delta_A^+ = \{\langle v, k\rangle \mid v \text{ is added } k \text{ times}\} \tag{3.4.3.1}$$

$$\Delta_A^- = \{\langle v, k\rangle \mid v \text{ is removed } k \text{ times}\} \tag{3.4.3.2}$$

$$\Delta_A^{upd} = \{\langle v, u, k\rangle \;\; v \text{ is updated to } u, \; k \text{ times}\} \tag{3.4.3.3}$$

$$T_A = \left\langle \Delta_A^+, \Delta_A^-, \Delta_A^{upd} \right\rangle \tag{3.4.3.4}$$

### Prerequisites for change-propagation expressions

The change propagation of all operators is defined in such a way that correctness and consistency of observing operators is enforced via the following two conditions: First, the set of deletions only contains objects that are already in the underlying relation. This means also that the objects are removed at most in the quantity in which they are present (3.4.3.5). Second, the set of updates contains only modification to objects that are also in the underlying relation, and no more objects can be updated than are present in the underlying relation (3.4.3.6).

$$\forall v,k.\ \langle v,k\rangle \in \Delta_A^- \wedge \langle v,i\rangle \in A \wedge k \leq i \tag{3.4.3.5}$$

$$\forall v,k.\ \langle v,u,k\rangle \in \Delta_A^{upd} \wedge \langle v,i\rangle \in A \wedge k \leq i \tag{3.4.3.6}$$

To obtain a minimal set of modifications – that are propagated through the operator tree – the following conditions must hold. First, the same objects are not deleted and then re-added by the same transaction (3.4.3.7). Second, updates are real updates, i.e., there are no tuples in the updates that have the same old and new value (3.4.3.8). In contrast to the earlier defined conditions, these latter two conditions can be loosened, which will be discussed in the context of optimization in Chapter 4. Note that loosening the condition 3.4.3.7 requires an internal treatment of modifications in the correct order. Deletions must be processed prior to additions, i.e., first the value is removed and then re-added. Due to this treatment a modification that adds and removes a completely new tuple is not allowed, since deletions are processed first and, hence, the very first condition (3.4.3.5) would be violated.

$$\Delta_A^+ \cap \Delta_A^- = \emptyset \tag{3.4.3.7}$$

$$\forall v,u,k.\ \langle v,u,k\rangle \in \Delta_A^{upd} \rightarrow v \neq u \tag{3.4.3.8}$$

### Notation for change-propagation expressions

A change-propagation expression is derived for each operator by determining the sets of modifications that must be propagated after applying a given transaction to an operator. Formally, the expression is derived for each set of modifications in a transaction, i.e., for the set of additions, deletions and updates. The modifications are derived as the results of three functions ($\Delta^+, \Delta^-, \Delta^{upd}$), which are defined for each operator (*op*), such that the following three equations hold. Note that the functions ($\Delta^+, \Delta^-, \Delta^{upd}$) can always depend on the whole transaction, e.g., additions may be derived from deletions or updates or vice versa.

$$op(A \uplus \Delta_A^+) = op(A) \uplus \Delta^+(op(T_A)) \tag{3.4.3.9}$$
$$op(A - \Delta_A^-) = op(A) - \Delta^-(op(T_A)) \tag{3.4.3.10}$$
$$op(A \hookleftarrow \Delta_A^{upd}) = op(A) \hookleftarrow \Delta^{upd}(op(T_A)) \tag{3.4.3.11}$$

First, the result of the operator after adding $\Delta_A^+$ to the base relation $A$ (via the union operator $\uplus$) is equal to adding the result of $\Delta^+$ for the transaction $T_A$ to the previous results of $op(A)$ (3.4.3.9). Second, the result of removing $\Delta_A^-$ from $A$ (via the set difference operator $-$) is equal to removing the result of $\Delta^-$ for the transaction $T_A$ from the previous results $op(A)$ (3.4.3.10). Third and finally, the result after the application of all modifications $\Delta_A^{upd}$ to $A$ is equal to the application of the result of $\Delta^{upd}$ for the transaction $T_A$ from the previous results $op(A)$ (3.4.3.11).

Note that the operator ($\hookleftarrow$) is introduced as a shorthand for applying an update to a relation, in order to keep the above definition simple. Formally, an update can be applied as shown in 3.4.3.12. The equation is a quite complex due to the treatment of multisets. The three conditions in each line in 3.4.3.12 basically do the following: first decrease the respective counts to remove old elements (first line), then add previously not existing elements (second line) and finally increase the count for previously existing elements (third line).

$$\begin{aligned} A \hookleftarrow \Delta_A^{upd} = \{\langle e,k\rangle \,|(&\langle e,i\rangle \in A \wedge \langle e,u,j\rangle \in \Delta_A^{upd} \wedge k = i - j) \vee \\ &(\langle v,e,j\rangle \in \Delta_A^{upd} \wedge \nexists \langle e,i\rangle \in A \wedge k = j) \vee \\ &(\langle v,e,j\rangle \in \Delta_A^{upd} \wedge \exists \langle e,i\rangle \in A \wedge k = j + i) \end{aligned} \tag{3.4.3.12}$$

Using the three equations (3.4.3.9 – 3.4.3.11) the net effect of a change event – in terms of modified objects – can be determined. The application of the three functions $(\Delta^+(op(T_A)), \Delta^-(op(T_A)), \Delta^{upd}(op(T_A))))$ yields basically the sets of elements that need to be propagated to the observers of an operator. In the following sections the result of these three functions is defined for each operator from Section 3.3. We distinguish two categories of operators:

**Self-maintainable** operators (Sec. 3.4.3) have a net effect that can be determined solely by using the delta of each change. Formally, the results of the three functions $(\Delta^+(op(T_A)), \Delta^-(op(T_A)), \Delta^{upd}(op(T_A))))$ only depend on values obtained from $\Delta_A^+, \Delta_A^-, \Delta_A^{upd}$.

**Non self-maintainable** operators (Sec. 3.4.3) have a net effect that either depends on the underlying relation (i.e., $A$) or further auxiliary data.

Note that a similar formalization was given by Griffin et al. [GL95] for SPJ views and the set-theoretic operators (i.e., not including aggregations and other advanced operators). This formalization did not include updates. Also, their focus was to obtain delta expressions for the sake of algebraic simplification. For example, using their formalization one can derive a minimal expression that only deals with additions of one side of a binary operator. Their simplifications assume the presence of materialized base relations. Instead we reuse parts of their formalization to derive minimal sets of data that must be materialized when base relations are not present.

## Self-Maintainable Operators

**Selections** are self-maintainable since basically each element in a transaction must pass the test ($\theta$). The set of additions is thus derived from all added elements that pass the test. Special care must be taken for updates. Here, all updated elements where the old value did not pass the test but the new value does, are also counted as additions. Deletions are handled similarly. Since only elements are deleted that were previously added, the deletions can propagate all elements that pass the test as these were surely added to the result of the selection in a previous transaction. Likewise updates count as deletions if the old value passed the test, but the new value does not pass the test. Updates are only propagated as update events if both the old and the new value pass the test.

$$\begin{aligned}
\Delta^{+}(\sigma_\theta(T_A)) =& \sigma_\theta(\Delta_A^{+}) \uplus \\
& \left\{ \langle u,k\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \wedge \neg\theta(v) \wedge \theta(u) \right\} \\
\Delta^{-}(\sigma_\theta(T_A)) =& \sigma_\theta(\Delta_A^{-}) \uplus \\
& \left\{ \langle v,k\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \wedge \theta(v) \wedge \neg\theta(u) \right\} \\
\Delta^{upd}(\sigma_\theta(T_A)) =& \left\{ \langle v,u,k\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \wedge \theta(v) \wedge \theta(u) \right\}
\end{aligned}$$

**Projections** are also self-maintainable and basically must only pass the additions and deletions as an addition or deletion of the projected value, i.e., the result of the transformation via $\rho$. Updates are only propagated if the old and new values are not projected to the same result. In addition, all updates that project the old and new values to the same result are counted as a single update.

$$\begin{aligned}
\Delta^{+}(\pi_\rho(T_A)) =& \pi_\rho(\Delta_A^{+}) \\
\Delta^{-}(\pi_\rho(T_A)) =& \pi_\rho(\Delta_A^{-}) \\
\Delta^{upd}(\pi_\rho(T_A)) =& \{\langle v,u,k\rangle \mid \exists\, \langle x,y,n\rangle \in \Delta_A^{upd} \wedge \\
& v = \rho(x) \wedge u = \rho(y) \wedge v \neq u \wedge \\
& k = \sum_{\substack{\langle a,b,i\rangle \in \Delta_A^{upd} \wedge \\ v=\rho(a) \wedge u=\rho(b)}} i\}
\end{aligned}$$

**Union (differing)** The union with differing semantics has the simplest change-propagation expressions, since the operator just forwards the additions, deletions or updates. This is due to the semantics of the operator, which computes a result by simply adding all elements from the left hand and right hand side operand. Since the operator is commutative the change-propagation expressions for a transaction on the right hand side operand are treated similarly.

$$\begin{aligned}
\Delta^{+}(T_A \uplus B) =& \Delta_A^{+} & \Delta^{+}(A \uplus T_B) =& \Delta_B^{+} \\
\Delta^{-}(T_A \uplus B) =& \Delta_A^{-} & \Delta^{-}(A \uplus T_B) =& \Delta_B^{-} \\
\Delta^{upd}(T_A \uplus B) =& \Delta_A^{upd} & \Delta^{upd}(A \uplus T_B) =& \Delta_B^{upd}
\end{aligned}$$

**UnNest** is self-maintainable, since the unnesting extracts a set of objects from a multiset that is locally contained in the added or deleted objects. Hence, additions and deletions merely take the added/deleted object, extract all elements from the contained multiset and propagated these as added or deleted elements. Updates are basically treated as deletions followed by insertions. This situation can not be avoided since we can not know – without further assumptions – that some objects contained in the unnested multisets of the update should be treated as old and new values of one another. One assumption that overcomes this problem is to treat the elements in the collection in an ordered sequence. In this case objects inside the unnested multisets can be correlated by their index in the sequence and we can propagate update events. However, this assumption is not generalizable. Nevertheless, the updates provide some benefit, since the extracted multisets can be analyzed to incorporate only elements that were not present before the update into the additions and remove only elements that are not present after the update.

$$\begin{aligned}
\Delta^{+}(\mu_\rho(T_A)) = \{\langle(t,e),k*n\rangle \mid \langle t,k\rangle \in \Delta_A^{+} \wedge S=\rho(t) \wedge \langle e,n\rangle \in S\} \uplus \\
\{\langle(u,e),k*n\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \wedge \\
S_{old}=\rho(v) \wedge \langle e,i\rangle \in S_{old} \wedge \\
S_{new}=\rho(u) \wedge \langle e,j\rangle \in S_{new} \wedge \\
n=j-i \wedge n>0\} \\
\Delta^{-}(\mu_\rho(T_A)) = \{\langle(t,e),k*n\rangle \mid \langle t,k\rangle \in \Delta_A^{-} \wedge S=\rho(t) \wedge \langle e,n\rangle \in S\} \uplus \\
\{\langle(v,e),k*n\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \wedge \\
S_{old}=\rho(v) \wedge \langle e,i\rangle \in S_{old} \wedge \\
S_{new}=\rho(u) \wedge \langle e,j\rangle \in S_{new} \wedge \\
n=i-j \wedge n>0\}
\end{aligned}$$

## Operators Maintainable with Auxiliary Data Structures

**Cartesian product** For the Cartesian product we require both underlying base relations for correct maintenance. This is due to the fact that for any added/deleted object a corresponding tuple must be built with *all* objects from the other relation. Hence, the view maintenance for either left or right hand operand requires the other relation to be materialized. The change-propagation expressions for additions and deletions simply build the Cartesian product over all additions and deletions together with the other relation. Updates can easily propagate over the Cartesian product. The respective expressions for updating $A$ build the set of tuples of the old value from the update and a value from $B$ and propagate this as an update to a tuple of the new value and the same value from $B$. The update to $B$ works accordingly by building tuples with objects from $A$.

$$\begin{aligned}
\Delta^{+}(T_A \times B) &= \Delta_A^{+} \times B \\
\Delta^{-}(T_A \times B) &= \Delta_A^{-} \times B \\
\Delta^{upd}(T_A \times B) &= \left\{\langle(v,b),(u,b),k*i\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \wedge \langle b,i\rangle \in B\right\}
\end{aligned}$$

$$\begin{aligned}
\Delta^{+}(A \times \langle T,B\rangle) &= A \times \Delta_B^{+} \\
\Delta^{-}(A \times \langle T,B\rangle) &= A \times \Delta_B^{-} \\
\Delta^{upd}(A \times T_B) &= \left\{\langle(a,v),(a,u),k*i\rangle \mid \langle v,u,k\rangle \in \Delta_B^{upd} \wedge \langle a,i\rangle \in A\right\}
\end{aligned}$$

**Join** For the join we also require the underlying base relations (as for the Cartesian product). In contrast to the Cartesian product, updates cannot easily be propagated over a join. The reason being that the join is based on looking up respective values via the functions $\rho$ and $\varphi$. For example, if an update to $A$ modified the value from $v$ to $u$ such that $\rho(v) \neq \rho(u)$ the objects now join with a completely different object from $B$. Hence, to keep the semantics simple the updates to a join are treated as deletions followed by additions.

$$
\begin{aligned}
\Delta^+(T_A \, {}_{\rho}\!\bowtie_{\varphi} B) =& \Delta_A^+ \, {}_{\rho}\!\bowtie_{\varphi} B \uplus \\
& \left\{ \langle u, k \rangle \mid \exists v. \, \langle v, u, k \rangle \in \Delta_A^{upd} \right\} \, {}_{\rho}\!\bowtie_{\varphi} B \\
\Delta^-(T_A \, {}_{\rho}\!\bowtie_{\varphi} B) =& \Delta_A^- \, {}_{\rho}\!\bowtie_{\varphi} B \uplus \\
& \left\{ \langle v, k \rangle \mid \exists u. \, \langle v, u, k \rangle \in \Delta_A^{upd} \right\} \, {}_{\rho}\!\bowtie_{\varphi} B
\end{aligned}
$$

$$
\begin{aligned}
\Delta^+(A \, {}_{\rho}\!\bowtie_{\varphi} \langle T, B \rangle) =& A \, {}_{\rho}\!\bowtie_{\varphi} \Delta_B^+ \uplus \\
& A \, {}_{\rho}\!\bowtie_{\varphi} \left\{ \langle v, k \rangle \mid \exists u. \, \langle v, u, k \rangle \in \Delta_B^{upd} \right\} \\
\Delta^-(A \, {}_{\rho}\!\bowtie_{\varphi} \langle T, B \rangle) =& A \, {}_{\rho}\!\bowtie_{\varphi} \Delta_B^- \uplus \\
& A \, {}_{\rho}\!\bowtie_{\varphi} \left\{ \langle v, k \rangle \mid \exists u. \, \langle v, u, k \rangle \in \Delta_B^{upd} \right\}
\end{aligned}
$$

**Duplicate Elimination** also requires to materialize the underlying relation. The set of deletions is built by removing all duplicates from the deletions and subtracting any elements that are still in the underlying relation $A$ after applying the deletions to $A$. In other words, a deletion is only propagated when the element reaches a count of 0 in $A$. Likewise additions are only propagated when the element was not already present in $A$ before the transaction. Updates can be propagated as updates, yet a check is needed that the new value is not a duplicate of a previously added element.

$$
\begin{aligned}
\Delta^{+}(\delta(T_A)) =&(\delta(\Delta_A^{+}) - A) \uplus \\
&\{\langle u,1\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \\
&\wedge \nexists \langle v,i\rangle \in A \wedge i-k=0 \\
&\wedge \nexists \langle u,j\rangle \in A\} \\
\Delta^{-}(\delta(T_A)) =&\delta(\Delta_A^{-}) - (A - \Delta_A^{-}) \uplus \\
&\{\langle v,1\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \\
&\wedge \exists \langle v,i\rangle \in A \wedge i-k=0 \\
&\wedge \exists \langle u,j\rangle \in A\} \\
\Delta^{upd}(\delta(T_A)) =&\{\langle v,u,1\rangle \mid \langle v,u,k\rangle \in \Delta_A^{upd} \\
&\wedge \exists \langle v,i\rangle \in A \wedge i-k=0 \\
&\wedge \nexists \langle u,j\rangle \in A\}
\end{aligned}
$$

Aggregation There are two issues in maintaining aggregations. First, what is the relevant data required due to the grouping (via the function $\rho$) of objects? Second, what is the relevant data required due to the aggregation (via the function $\alpha$) of the multisets obtained for each group?

**Maintenance of aggregate groups**

To answer the first question, the three cases for different modification can informally be distinguished as follows: Additions are imminent when new groups are obtained either from added objects or due to updates on objects. Deletions are imminent when the groups are not contained in $A$ after applying all updates (written as $A^{new}$ in the expression below). Updates are only imminent if there is a group that is present before and after the modifications. Hence, updates only pertain to obtaining new values from applying the aggregation function. However, it is interesting to note that updates to the aggregated values occur in response to additions or deletions, which makes the aggregation operator somewhat special – in general the other operators perform additions and deletions in response to updates and not vice versa.

$$
\begin{aligned}
\Delta^{+}(\gamma_{\rho,\alpha}(T_A)) =& \{(g,a) \mid (\ \langle v,i\rangle \in \Delta_A^{+} \\
& \lor\ \langle o,v,j\rangle \in \Delta_A^{upd} \land \rho(o) \neq \rho(v)\ ) \\
& \land g = \rho(v) \land g \notin \delta(\pi_\rho(A)) \\
& \land a = \alpha(\sigma_{\lambda t.\ \rho(t)=g}(A^{new}))\} \\
\Delta^{-}(\gamma_{\rho,\alpha}(T_A)) =& \{(g,a) \mid (\ \langle v,i\rangle \in \Delta_A^{-} \\
& \lor\ \langle v,o,j\rangle \in \Delta_A^{upd} \land \rho(o) \neq \rho(v)\ ) \\
& \land g = \rho(v) \land g \notin \delta(\pi_\rho(A^{new})) \\
& \land a = \alpha(\sigma_{\lambda t.\ \rho(t)=g}(A))\} \\
\Delta^{upd}(\gamma_{\rho,\alpha}(T_A)) =& \{\langle (g,a_{old}),(g,a_{new})\rangle \mid (\ \langle v,i\rangle \in \Delta_A^{+}\ \lor\ \langle v,j\rangle \in \Delta_A^{-} \\
& \lor\ \langle o,v,k\rangle \in \Delta_A^{upd} \land \rho(o) = \rho(v)\ ) \\
& \land g = \rho(v) \land g \in \delta(\pi_\rho(A)) \\
& \land a_{old} = \alpha(\sigma_{\lambda t.\ \rho(t)=g}(A)) \\
& \land a_{new} = \alpha(\sigma_{\lambda t.\ \rho(t)=g}(A^{new})) \\
& \land a_{old} \neq a_{new}\}
\end{aligned}
$$

As can be seen from the formal definitions – given above – existence of a group requires a test w.r.t. $\delta(\pi_\rho(A))$, i.e., one has to maintain the set of all groups obtained using the aggregations. Note that for deletions the value $\delta(\pi_\rho(A^{new}))$ is used. To capture this semantics without actually applying all modifications to $A$ (since we want to forgo completely materializing $A$), the operator uses an internal counter that stores how many elements in $A$ are contributing to a group.

### Maintenance of aggregation functions

The second question has a twofold answer that depends on the makeup of the aggregation function $\alpha$. Some aggregation functions, e.g., `Count`, can compute an updated value based on deltas, whereas other functions, e.g., `Min, Max` require the underlying group. The latter can be exemplified by considering an update that removes the value which was the minimum/maximum of the aggregation. To compute a new minimum/maximum the underlying group must be "scanned" to determine what the next higher/lower value is. Such functions have been termed *non-distributive* in [PSCP02] – note, there is no formalization in that work. In the terms of this thesis it is instructional to think of the aggregation functions as *not self-maintainable*, i.e., they require the materialization of auxiliary data for efficient incrementalization.

From the perspective of the change-propagation expressions given above, the impact of *not self-maintainable* functions is only imminent in the equation for $\Delta^{upd}(\gamma_{\rho,\alpha}(T_A))$. Note that for additions we know that $\sigma_{\lambda t.\, \rho(t)=g}(A) = \emptyset$, since the group was not contained in $A$ prior to the modification. Hence the value for $a$ can be computed using only the deltas. For deletions (and the value $a_{old}$ in updates) it would be sufficient to store the single aggregated value of $a$ for each value $g$ that uniquely identifies the group. Only for obtaining $a_{new}$ in the equation for updates the necessity can arise to query the whole underlying group – as exemplified for the `Min, Max` above.

Informally to re-compute a not self-maintainable function the operator must maintain an entire group, i.e., the multiset of elements in the group, for each value $g$ that uniquely identifies the group. A self-maintainable function requires as minimal auxiliary data the previously computed value $a_{old}$ for the group. Hence, both cases require some materialization, but the self-maintainable case requires less memory. Formally, the different requirements of (not) self-maintainable functions can be shown as follows. Each group is computed via the selection of elements given by $\sigma_{\lambda t.\, \rho(t)=g}$. The computation of the groups for $A^{new}$ is obtained from an old multiset of elements in the group ($A_g$), a multiset of additional elements $\Delta_g^{add}$ and a multiset of removed elements $\Delta_g^{del}$. Note that – for simplification of the formulas – updates are subsumed as additions and deletions (i.e., removal of the old value and addition of the new value) which are denoted as $\Delta_A^{add}$ and $\Delta_A^{del}$, in order to discriminate from the treatment of additions/deletions without updates.

$$\begin{aligned}
\sigma_{\lambda t.\, \rho(t)=g}(A^{new}) =& \sigma_{\lambda t.\, \rho(t)=g}(A \uplus \Delta_A^{add} - \Delta_A^{del}) \\
=& \sigma_{\lambda t.\, \rho(t)=g}(A) \uplus \sigma_{\lambda t.\, \rho(t)=g}(\Delta_A^{add}) - \sigma_{\lambda t.\, \rho(t)=g}(\Delta_A^{del}) \\
=& A_g \uplus \Delta_g^{add} - \Delta_g^{del}
\end{aligned}$$

For a self-maintainable aggregation function a homomorphism exists such that the function can be distributed over the set operators. The new result can then be obtained via: the old result ($\alpha(A_g) = a_{old}$) the result over the added elements ($\alpha(\Delta_g^{add})$), and the result over the deleted elements ($\alpha(\Delta_g^{del})$). The operators $\oplus$ and $\ominus$ are used to combine the results and are specific to the domain of the values for a given aggregation function.

$$\begin{aligned}
\alpha(\sigma_{\lambda t.\, \rho(t)=g}(A^{new})) =& \alpha(A_g \uplus \Delta_g^{add} - \Delta_g^{del}) \\
=& \alpha(A_g) \oplus \alpha(\Delta_g^{add}) \ominus \alpha(\Delta_g^{del})
\end{aligned}$$

For example, the count function would define $\oplus$ and $\ominus$ as addition and subtraction over integers.

$$\texttt{Count}(\sigma_{\lambda t.\, \rho(t)=g}(A^{new})) = \texttt{Count}_{old} + \texttt{Count}(\Delta_g^{add}) - \texttt{Count}(\Delta_g^{del})$$

Union (indistinguishable) For the union with indistinguishable semantics it is easy to see that both underlying relations are required for view maintenance. The number of elements that contribute to the result are based on the maximum of the number of elements in the left and right operand. Consider for example the maintenance of the union for adding elements to the left operand ($A$). The new maximum is built over the new number of elements on the left side ($A \uplus \Delta_A^+$) and the number of elements on the right side ($B$). Fortunately, given the change-propagation expressions from [GL95] we can find a better solution based on the multiset differences of $A - B$ and $B - A$. Hence, less storage is required compared to materializing $A$ and $B$, since effectively the intersection between the two relations does not require any storage. The basic idea behind these expressions is as follows: For additions, the change can only add more elements from $A$ if there was not already a larger number of elements in $B$. Recap that the definition of the union is based on the maximum of elements in $A$ and $B$. The expression $B - A$ contains all surplus elements from $B$. Hence, an addition is only propagated if $\Delta_A^+$ contains more elements than $B - A$. A similar argument holds for deletions. Deletions only propagate if the $A$ contains surplus elements not found in $B$. Hence, only the minimum ($\cap$) between $\Delta_A^-$ and $A - B$ is propagated as deletions.

Propagation of updates is modeled as deletions followed by insertions to keep the semantics simple. Intuitively an update can only be propagated as an update if $B$ did not have surplus elements of the old value and $B$ did not have surplus elements of the new value. Even then one has to determine how many surplus elements there are and if the old and new surplus do not match, some of the updates must be counted as additions or deletions. Hence, the intuitive semantics is to treat updates simply as deletions and insertions.

$$\begin{aligned}
\Delta^+(T_A \cup B) &= (\Delta_A^+ \uplus \left\{ \langle u, k \rangle \mid \exists v.\, \langle v, u, k \rangle \in \Delta_A^{upd} \right\}) - (B - A) \\
\Delta^-(T_A \cup B) &= (\Delta_A^- \uplus \left\{ \langle v, k \rangle \mid \exists u.\, \langle v, u, k \rangle \in \Delta_A^{upd} \right\}) \cap (A - B)
\end{aligned}$$

$$\begin{aligned}
\Delta^+(A \cup T_B) &= (\Delta_B^+ \uplus \left\{ \langle u, k \rangle \mid \exists v.\, \langle v, u, k \rangle \in \Delta_B^{upd} \right\}) - (A - B) \\
\Delta^-(A \cup T_B) &= (\Delta_B^- \uplus \left\{ \langle v, k \rangle \mid \exists u.\, \langle v, u, k \rangle \in \Delta_B^{upd} \right\}) \cap (B - A)
\end{aligned}$$

**Intersection** requires to materialize both underlying relations for view maintenance. Like the indistinguishable union the view maintenance can be reduced to change-propagation expressions based on $A - B$ and $B - A$. Also, propagations are treated as deletions and additions similar to the treatment in the indistinguishable union. The rationale behind the expressions is based on the fact that the intersection is defined via the min operator. Thus, an addition can only be propagated until the number of elements in $B - A$ is reached, i.e., the number of elements in $B$ minus the previously propagated elements from $A$. Deletions can only be propagated if they are in the intersection. Hence, all deletions that are also in $A - B$ are deletions of surplus elements from $A$ that are not also in $B$. Thus, only the elements that are also present in $A - B$ are propagated as deletions.

$$\Delta^+(T_A \cap B) = (\Delta_A^+ \uplus \left\{\langle u,k\rangle \mid \exists v.\, \langle v,u,k\rangle \in \Delta_A^{upd}\right\}) \cap (B - A)$$
$$\Delta^-(T_A \cap B) = (\Delta_A^- \uplus \left\{\langle v,k\rangle \mid \exists u.\, \langle v,u,k\rangle \in \Delta_A^{upd}\right\}) - (A - B)$$

$$\Delta^+(A \cap T_B) = (\Delta_B^+ \uplus \left\{\langle u,k\rangle \mid \exists v.\, \langle v,u,k\rangle \in \Delta_B^{upd}\right\}) \cap (A - B)$$
$$\Delta^-(A \cap T_B) = (\Delta_B^- \uplus \left\{\langle v,k\rangle \mid \exists u.\, \langle v,u,k\rangle \in \Delta_B^{upd}\right\}) - (B - A)$$

**Difference** – like the intersection and indistinguishable union – requires both underlying relations, but the change-propagation expression can also be reduced to differences. As a consequence, the difference operator basically stores the result as a materialized view. It is important to highlight the expressions for changes, since the set difference is an asymmetric operator and, hence, the expressions are slightly different from the other set operators.

The left-hand side expression ($T_A$) can be understood as follows. Additions are propagated ($\Delta^+$) after subtracting surplus elements from $B$, i.e., additions are only propagated for the number elements in $A$ that is greater than the number of elements in $B$. Deletions are propagated ($\Delta^-$) if they are also contained in the previous result, i.e., we remove everything in $\Delta_A^+$, but only as many elements ($\cap$) as we find in the result $A - B$. The same is true for updates, i.e., the number of propagated updates ($i$) from elements in $A$ is equal to the number we find in the result $A - B$.

The right-hand side expression ($T_B$) can be understood as follows. Additions are propagated ($\Delta^+$) if elements are removed from $B$ and there are more of the same elements in $A$, i.e., $A$ contains elements that are currently not in the result $A - B$ and that are now removed from $B$. As a change-propagation expression this is a bit tricky due to double negation. Essentially the subtraction of $B - A$ from $\Delta_B^-$ means that we remove from the propagation all elements that are not in $A$. Formulated as a positive statement this means that those elements that are in $A$ are propagated. Deletions are propagated ($\Delta^-$) if elements are added to $B$ and there are more of the same elements in $A$, i.e., the result $A - B$ contains elements that must be removed due to now being in $B$. This is straightforward in the change-propagation expression; we remove everything in $\Delta_B^+$, but only as many elements ($\cap$) as we find in the result $A - B$. In addition, updates on the right-hand are treated as deletions of old values and insertion of new values, since the result contains essentially elements from $A$ and updates in $B$ merely add or remove different elements from $A$.

$$\begin{aligned}
\Delta^+(T_A - B) =& \Delta_A^+ - (B - A) \\
\Delta^-(T_A - B) =& \Delta_A^- \cap (A - B) \\
\Delta^{upd}(T_A - B) =& \left\{ \langle v, u, i \rangle \mid \exists k . \langle v, u, k \rangle \in \Delta_A^{upd} \wedge \langle v, i \rangle \in (A - B) \right\}
\end{aligned}$$

$$\begin{aligned}
\Delta^+(A - T_B) =& (\Delta_B^- \uplus \left\{ \langle v, k \rangle \mid \exists u . \langle v, u, k \rangle \in \Delta_B^{upd} \right\}) - (B - A) \\
\Delta^-(A - T_B) =& (\Delta_B^+ \uplus \left\{ \langle u, k \rangle \mid \exists v . \langle v, u, k \rangle \in \Delta_B^{upd} \right\}) \cap (A - B)
\end{aligned}$$

**Nest** The nesting operator is defined in terms of an aggregation operator and, hence, incrementally maintained in the same way. Since the concrete aggregate functions are of great importance for the requirements of materialized data in the aggregation operator, the peculiarities of nesting are shortly discussed. The definition of the nesting operator – repeated below – defines an aggregation with a *map* function.

$$\nu_{\rho,\varphi}(A) = \gamma_{\rho,\lambda G.map(G,\varphi)}(A)$$

The function *map* takes a multiset and transforms the elements into a new multiset. As such the function is a self-maintainable aggregate function. Note that this is due to the fact that the transformation makes no assumption regarding ordering of elements, i.e., added or removed elements can just be added or removed to/from the previously computed result, without taking into account at which position they were added. However, while the function is self-maintainable the aggregation operator always requires the storage of the previously computed aggregate value, which is in this case the entire transformed multiset. Hence, nesting requires an amount of data comparable to a not self-maintainable aggregate function – depending on the size of the transformed elements the data can be slightly more or slightly less.

Transitive Closure can be maintained differently for acyclic and cyclic graphs. The change-propagation expressions are presented first for acyclic graphs. Then cyclic graphs are considered, which are maintainable using an extension of the basic idea used in acyclic graphs. The idea for maintaining the transitive closure can also be found in [DS95, DS00]. The former work also provides a (different) formalization and a correctness proof. For easier treatment the idea is presented here in the uniform formalization for all operators defined in this thesis.

In terms of materialized data, we can note that both types of graphs require a materialization of the result, i.e., all edges in the transitive closure, as well as an additional materialization of the edges in the underlying graph. Note that for both types of graphs updates are treated as deletions followed by insertions. For simplification the notation for additions and deletions with subsumed updates – already used in aggregations – denoted as $\Delta_A^{add}$ and $\Delta_A^{del}$ is used.

To further simplify the reasoning in the formulas the additions and deletions are directly expressed over the graph $G_A = (V, E)$ (cf. Section 3.3.4) and more specifically over the edges $E$ in the graph. Thus, the sets of added and deleted edges are defined as shown below. Note that the transitive closure yields a set of edges as result and, hence, additions are only processed once for each edge, i.e., if the edge is not already contained in $E$.

$$\Delta_E^{add} = \Big\{(v,u) | \exists t \in \Delta_A^{add} \wedge v = \rho(t) \wedge u = \varphi(t) \wedge (v,u) \notin E\Big\}$$
$$\Delta_E^{del} = \Big\{(v,u) | \exists t \in \Delta_A^{del} \wedge v = \rho(t) \wedge u = \varphi(t)\Big\}$$

### Maintenance of acyclic directed graphs

The basic idea for incremental maintenance of acyclic graphs revolves around concatenation of existing paths. Paths are already reified in the transitive closure as edges, i.e., if a path exists in the graph a corresponding edge from the start to the end vertex is contained in the transitive closure. The interesting fact to note is that the incremental maintenance does not require recursion but can be expressed as a finite number of concatenations of existing paths.

To simplify the change-propagation expressions, the concatenation of paths is declared as a function between sets of edges (3.4.3.13). In this formula a new path is generated by taking two existing edges where the end vertex of the first edge is the start vertex of the second edge.

$$E_1 \circ E_2 = \{(v,u) \mid (v,x) \in E_1 \land (x,u) \in E_2\} \tag{3.4.3.13}$$

Using the formula for path concatenation the maintenance of additions is phrased very simply as shown below. A (potential) addition is determined as the sets of edges for appending or prepending added edges to existing paths ($TC(A) \circ \Delta_E^{add}$ and $\Delta_E^{add} \circ TC(A)$) together with the set of edges obtained by concatenating two existing paths via an added edge ($TC(A) \circ \Delta_E^{add} \circ TC(A)$). For illustration consider Figure 3.22, where a new edge $(v_2, v_3)$ is added. The new transitive closure paths (marked as thick arrows) are derived from existing paths by prepending the edge (derives $(v_2, v_4)$), by appending the edge (derives $(v_1, v_3)$) and by concatenating existing paths (derives $(v_1, v_4)$).

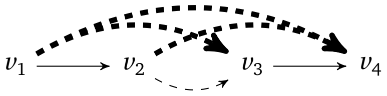


**Figure 3.22:** Deriving transitive closure paths after adding an edge

Concrete additions take into account that newly inferred paths are already in the transitive closure. Hence, the final set of propagated additions is obtained as the union of all added edges ($\Delta_E^{add}$), which is guaranteed to hold only new edges, and the set of inferred paths that were not already contained in the transitive closure. As can be seen from the equations, the maintenance of the transitive closure always requires a materialization of the complete result prior to the modifications (i.e., $TC(A)$).

$$TC_{added}(T_A) = TC(A) \circ \Delta_E^{add} \cup \Delta_E^{add} \circ TC(A) \cup TC(A) \circ \Delta_E^{add} \circ TC(A)$$
$$\Delta^+(TC(T_A)) = \Delta_E^{add} \cup \{e \mid e \in TC_{added}(T_A) \land e \notin TC(A)\}$$

Deletions are slightly more complex but use the same principle, i.e., path concatenation. Potentially, all paths that go through a deleted edge must be removed from the transitive closure. However, the transitive closure can contain different paths from a source vertex to a target vertex, i.e., paths that do not go through the deleted edge. Thus, for deletions one must determine a set of potentially deleted edges (termed $TC_{Suspicious}(T_A)$) and determine edges with alternative paths. For this reason a set of known trusted edges is built ($TC_{Trusted}(T_A)$), which contains all edges from the transitive closure that are guaranteed to be not suspicious; together with the edges found in the updated graph. The latter are by definition not suspicious. Alternative paths can – interestingly – be found by simple concatenations of two or three trusted edges into new paths. Hence, all alternative derivations of paths – i.e., edges in the transitive closure – are identified in the set $TC_{Alternative}(T_A)$. Note that these additional sets are not materialized; they depend on the concrete modifications and are determined during each incremental maintenance operation.

$$\begin{aligned}
TC_{Suspicious}(T_A) &= \Delta_E^{del} \cup TC(A) \circ \Delta_E^{del} \cup \Delta_E^{del} \circ TC(A) \cup TC(A) \circ \Delta_E^{del} \circ TC(A) \\
TC_{Trusted}(T_A) &= \left(TC(A) - TC_{Suspicious}(T_A)\right) \cup E^{new} \\
TC_{Alternative}(T_A) &= TC_{Trusted}(T_A) \cup TC_{Trusted}(T_A) \circ TC_{Trusted}(T_A) \\
&\quad \cup TC_{Trusted}(T_A) \circ TC_{Trusted}(T_A) \circ TC_{Trusted}(T_A)
\end{aligned}$$

The concrete set of deleted edges in response to the change modifications can now simply be phrased as follows. Deletions are all suspicious (i.e., potentially deleted) edges ($TC_{Suspicious}(T_A)$) that are not contained in the set of alternative derivations.

$$\Delta^{-}(TC(T_A)) = TC_{Suspicious}(T_A) - TC_{Alternative}(T_A)$$

From the definition of deletions and the intermediate sets for suspicious/trusted edges, it easy to see that the maintenance of the transitive closure not only requires the previously computed result ($TC(A)$), but also the set of edges in the graph ($E^{new}$). The latter are in this case already modified with all change events once they are used for the incremental maintenance. The edges from the graph are a strict necessity for correctness. Consider for example an edge $e$ is removed that has a lengthy path $(a, b)$ through $e$, hence $(a, b) \in TC_{Suspicious}(T_A)$. If the only alternative is a direct edge $(a, b)$ that is contained in the underlying graph, i.e., $(a, b) \in E^{new}$, then a treatment without adding $E^{new}$ to the set of trusted edges is incorrect.

Having established above that the operator needs to materialize sets of edges, there is an additional consideration, namely how edges are stored for efficient usage in the algorithms that encode the above expressions. Since the concatenation operator ($\circ$) is used to both prepend and append existing edges to paths, it is necessary to quickly retrieve existing edges via their start and via their end vertex. Hence, edges are stored as double adjacency list, i.e., for each vertex one can quickly retrieve the set of incoming and the set of outgoing edges. The space consumption is naturally higher compared to using single adjacency lists. However, using single adjacency lists, e.g., for outgoing edges, requires a a full traversal over all vertices to determine incoming edges.

### Maintenance of cyclic directed graphs

The transitive closure over cyclic graphs can not be maintained efficiently, i.e., in near linear time, for edge deletions without maintaining additional data structures [DP97]. Intuitively the reason is that deleted edges can be contained in a cycle and, hence, the maintenance algorithm must deal with situations in which a cycle is broken by the deletion. To perform this task, the maintenance algorithm requires a structure to quickly determine (i) if an edge (vertex) participates in a cycle and (ii) if a cycle is broken and (iii) which elements of a broken cycle are still reachable. The latter cannot strictly be determined via additional data structures for all cases of edge deletions. Hence, a traversal of the edges in the graph can not be avoided completely, but the impact can be mitigated, e.g., via a recursive depth-first search of only the elements in the cycle – and not the complete graph. Note that insertions can be handled in the same manner as in the acyclic case.

A large variety of algorithms and data structures for the maintenance of the transitive closure have been proposed with varying effectiveness; [KZ08] provides a good overview and comparison. The change-propagation expressions presented in the following are similar to the algorithmic idea found in Frigioni et al. [FMZ01]. The basic idea is to determine strongly connected components (SCCs), which are essentially subgraphs that correspond to all vertices in a cyclic path, i.e., every vertex in a strongly connected component can reach every other vertex. For each SCC a single representative vertex (termed super-vertex) is created and the complete SCC is replaced in the underlying graph by the super-vertex. After replacing all SCCs, the modified graph is guaranteed to be cycle-free. Hence, the original technique for computing the transitive closure over acyclic graphs can be reused.

The approach followed in this thesis is a bit simpler in so far as no complex substitutions are made and less data structures are used. The idea is to extend the definition of $TC_{Trusted}(T_A)$ to work with SCCs, which is best illustrated by an example. Consider the deletion of the edge $(v_4, v_2)$ depicted in Figure 3.23. Due to the deletion of the edge, all paths that go through the SCC are contained in $TC_{Suspicious}(T_A)$; especially the path $(v_1, v_5)$ is marked as suspicious. The reason why the change expressions of the acyclic graph maintenance do not work, is that the alternative paths cannot simply be derived by concatenation of *two* or *three* trusted paths. By the previous definition of $TC_{Trusted}(T_A)$, the set contains all paths ending at the SCC and all paths starting after the SCC, i.e., $(\ldots, v_2)$ and $(v_4, \ldots)$ are still trusted for any vertices in the graph not contained in the SCC, e.g., $v_1$ and $v_5$. But inside the SCC all paths are suspicious, since all paths in the cycle can go through the deleted edge $(v_4, v_2)$. Hence, trusted edges in the SCC are only the edges contained in the graph, i.e., $(v_2, v_3)$ and $(v_3, v_4)$. Now, to derive the – still valid – path $(v_1, v_5)$ the concatenation of *four* trusted paths is required. In general, an SCC can have paths of arbitrary lengths between end-points such as $v_2$ and $v_4$, which require more than four concatenations and, hence, the acyclic technique for re-deriving paths is insufficient.

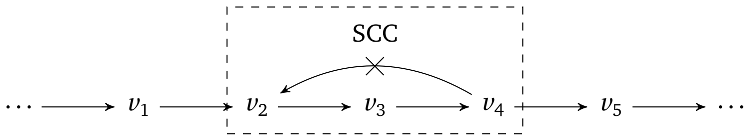

SCC
$\cdots \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow \cdots$

**Figure 3.23:** Deleting an edge in an SCC

To remedy the situation, the idea is to simply recompute trusted paths inside an SCC (and not further) via a DFS traversal. In the above example, this would yield the path $(v_2, v_4)$ as a trusted path. Afterwards the concatenation of two or three trusted paths is sufficient to maintain the transitive closure. The DFS traversal requires only a minimal materialization of additional data structures, namely the information in which SCC a vertex is contained. Apart from this the traversal only requires the edges in the graph, which are already maintained by the operator.

Formally, the data structure for determining participation in an SCC is a set that contains for each vertex a corresponding identifier of it's SCC as shown below. Let $V$ be the set of vertices for the underlying relation $A$ and $Id$ a set of identifiers. Then the set $SCC_{Id}(A)$ contains a set of pairs that links a vertex to an SCC identifier. The set must maintain the invariant that any two vertices that have the same identifier are also in a cycle. Note that not all vertices are required to be linked to an SCC, but only those that actually participate in a cycle. Hence, the size of the data structure is bounded by the number and size of the SCCs in a graph.

$$\begin{aligned}
&SCC_{Id}(A) \subseteq V \times Id \\
&\textit{Invariant: } \forall \langle u, id_u \rangle, \langle v, id_v \rangle \in SCC_{Id}(A). \\
&\qquad id_u = id_v \rightarrow (v,u) \in TC(A) \wedge (u,v) \in TC(A)
\end{aligned}$$

Given the above definition, the set of trusted edges can be extended; denoted as $TC_{TrustedSCC}(T_A)$.

$$\begin{aligned}
TC_{TrustedSCC}(T_A) = & TC_{Trusted}(T_A) \cup \\
& \{(u,v) \mid \exists \textit{ a path from } u \textit{ to } v \textit{ in } E^{new} \wedge \\
& \exists i \in Id.\ \langle u,i \rangle \in SCC_{Id}(A) \wedge \langle v,i \rangle \in SCC_{Id}(A)\}
\end{aligned}$$

The original set of trusted edges is retained and an additional set is added for all paths between vertices in the same SCC. The definition of $TC_{Alternative}(T_A)$ can then be substituted by a definition that uses $TC_{TrustedSCC}(T_A)$ instead of $TC_{Trusted}(T_A)$.

There are two important things to highlight in the presented approach. First, the identifiers for SCCs and the re-computation of paths in an SCC are only required if there are cycles in the graph. In the case of an acyclic graph, the algorithm and data structures automatically reduce to those found in the maintenance algorithm over acyclic graphs, i.e., $SCC_{Id}(A) = \emptyset$ and $TC_{TrustedSCC}(T_A) = TC_{Trusted}(T_A)$. Second, the approach naturally has a bad asymptotic complexity for an incremental update ($O(|E|)$), since the worst case is that the entire graph is a single cycle. Nevertheless the approach performs reasonably well for graphs with small SCCs.

As noted in the beginning the complexity for removing edges in an SCC can not be avoided entirely. In general, once an edge is removed that breaks the cycle of an SCC a re-traversal of the graph must be performed. The algorithm in [FMZ01] seeks to reduce the number re-computations by storing an additional *sparse certificate* for each SCC. A sparse certificate is a subgraph of the SCC that has the same set of vertices, but has fewer edges ($2k-2$ where $k$ is the number of vertices). The certificate guarantees the property that if there is a path between two vertices in the SCC, then there is also a path between the same vertices in the sparse certificate. One can think of the sparse certificate as the essentially required edges in a cycle, whereas multiple edges can exist that do not contribute to the cycle. Hence, deleting an edge not in the certificate guarantees that the cycle is not broken and no re-computation of the SCC is required. The certificate costs additional space and can only reduce the number of re-computations for graphs that contain such cycles with unnecessary edges.

Recursion As discussed in Section 3.3.4 the recursion performs a substitution in the operator tree. Conceptually, the substitution is quite simple. Each occurrence of the relation $A$ in *Op(A)* must be substituted by a recursive definition ($A_{Rec}$) given as:

$$A_{Rec} = FIX(A, Op(A))$$

The compiled relation for *Op(A)* must in turn be observed by the fixpoint operator (*FIX*). Given that the observer pattern allows to dynamically add or remove observers the construction of a recursion is relatively simple and the result of *Op(A)* in `opA`, the recursion can be constructed by the call:

```
opA.addObserver(fix)
```

The only issue for the query compiler is that the registration can only be performed after the compilation of the query *Op(A)*, i.e., during the construction of the operator tree we have no variable `opA` representing the result. Hence, the query compiler internally remembers the fixpoint operators and performs the recursive registration in a final compilation step.

The recursive edge introduced in the operator tree has no special semantics, i.e., elements are just propagated back in the tree. As a technical side note, the evaluation in a recursive operator tree requires large method call stacks, due to the fact that the propagation between the operators can produce rather large call chains. These chains can currently not be optimized by the compiler (e.g., via tail-call optimization), since recursion spans multiple observers with a generic notification mechanism. That means each operator iterates over the list of its observers and makes notification calls. Hence, the recursion spans multiple objects and cannot be optimized by the compiler. To allow the recursion to work with the default call stack sizes of the Java virtual machine, the *FIX* operator basically emulates its own stack, i.e., it records the recursive additions/deletions, but propagates them one item at a time.

To obtain correct incremental view maintenance a variant of the *DRed* (Delete and Rederive) algorithm [GMS93] (cf. Sec. 2.3) with derivation counters is used. Note that updates are treated as deletions followed by insertions in a recursion. For simplification we use the notation for additions and deletions with subsumed updates – already used in aggregations – denoted as $\Delta_A^{add}$ and

$\Delta_A^{del}$. To understand the treatment of recursion let us start with the insertion of elements.

### Maintaining insertions

Insertions are relatively straightforward. For the purpose of fixpoint recursion the set *Derivations*($A$) tracks the number of different derivations for each value in $A$. Thus, we can take the set of tuples with elements from $A$ and a counter and define the set *Derivations*(A) as a subset as follows:

$$Derivations(A) \subseteq A \times Int$$

An insertion is propagated if the fixpoint operator has not already encountered the value. This is formalized as follows:

$$\Delta^{+}\left(FIX(T_A)\right) = \left\{ v \mid \langle v, i \rangle \in \Delta_A^{add} \ \wedge \ (v,k) \in Derivations(A) \ \wedge \ k = 0 \right\}$$

For brevity in the above definition we can assume that $(v,k) \in Derivations(A)$ is always true and returns a count of zero if the element was not encountered before. When treating insertions the derivation counters must be updated. Basically the number is incremented each time the insertion of an element is encountered in to the operator, which can be formalized as follows:

$$\begin{aligned} &\forall \langle v, i \rangle \in \Delta_A^{add} \wedge (v,k) \in Derivations(A) \Rightarrow \\ &\qquad Derivations(A) = (Derivations(A) - \{(v,k)\}) \uplus \{(v, k+i)\} \end{aligned}$$

Note that each element is propagated exactly once, i.e., if an alternative derivation of the element is encountered, the element is not propagated again. Thus, any operators in the operator tree between the *FIX* operator and the recursive back edge will also contain the element only once. So far this treatment of additions is little more than an elaborate way to rule out multiple derivations. The number of alternative derivations will become interesting when we maintain deletions.

### Maintaining deletions

The basic procedure for deletions is to propagate all deleted elements through the operator tree, to determine all recursively derived values that are in turn deleted. The interesting point in the DRed algorithm is that for some elements we have alternative derivations. Hence, these elements must be re-inserted to determine which values can still be recursively derived. In other words the deletion phase is an over-approximation of all deleted elements and is followed by a rederivation phase that re-inserts elements to obtain a state where only the exact deletions are performed.

The deletion phase is a mirror of the insertion phase, with the difference that deletions are propagated. A deletion is propagated if the fixpoint operator has not already encountered the deletion of the value. The number of deletions is tracked in a set *Deletions*(*A*) as follows:

$$\mathit{Deletions}(A) \subseteq A \times \mathit{Int}$$

The change-propagation expression for deletions is straight-forward and formalized as follows:

$$\Delta^{-}\left(\mathit{FIX}(T_A)\right) = \left\{v \mid \langle v, i\rangle \in \Delta_A^{del} \;\wedge\; (v,k) \in \mathit{Deletions}(A) \;\wedge\; k = 0\right\}$$

Similar to derivations the counter for deletions is updated each time an element is encountered as deleted:

$$\begin{aligned}&\forall\, \langle v, i\rangle \in \Delta_A^{del} \;\wedge\; (v,k) \in \mathit{Deletions}(A) \Rightarrow\\ &\qquad\qquad \mathit{Deletions}(A) = (\mathit{Deletions}(A) - \{(v,k)\}) \uplus \{(v, k+i)\}\end{aligned}$$

Note that after the deletion phase we have effectively removed all deleted elements from the incrementally maintained operators that stand between the *FIX* operator and the recursive back edge to *FIX*. Hence, the rederivation phase consists of effectively propagating insertions for elements that must be rederived. In other words the rederivation phase is a second round of insertions, but one that is triggered from inside the *FIX* operator. The set of elements that must be rederived can be obtained as follows:

$$\begin{aligned}&\mathit{ReDerivations}(A) =\\ &\left\{v \mid (v,i) \in \mathit{Deletions}(A) \;\wedge (v,j) \in \mathit{Derivations}(A) \wedge j - i > 0\right\}\end{aligned}$$

In essence the set of rederivations contains all elements where the counter of deletions is smaller than the counter for derivations. Before the rederivation phase starts the counters for all derivations must be updated to remove the number of deleted elements as follows:

$$\begin{aligned}&\forall(v,i)\in \mathit{Deletions}(A)\ \wedge (v,j)\in \mathit{Derivations}(A)\Rightarrow\\&\qquad \mathit{Derivations}(A)=\big(\mathit{Derivations}(A)-\{(v,j)\}\big)\uplus\{(v,j-i)\}\end{aligned}$$

After having made this preparation the set *ReDerivations*$(A)$ is propagated to the operators in the recursion. Any new recursively derived values will enter the *FIX* operator again as an insertion and update the number of derivations accordingly.

### Materialized auxiliary data

In the above definitions for insertions and deletions we have declared three sets, i.e., for derivations, deletions and rederivations. In terms of permanently materialized data only the set for derivations is necessary. In other words after a deletion is completed the sets *Deletions*$(A)$ and *ReDerivations*$(A)$ are not needed anymore and thus the data is transient. Note that the rederivations are propagated recursively until no new elements are inserted, afterwards the control is returned to the method that started the rederivation phase and only then are these sets thrown away.

### Runtime efficiency

Incremental maintenance of recursive queries can be quite expensive. In essence it is a source of non-linearity for the time required to incrementally maintain operators. All other operators can quickly determine added or removed results based on state they maintain by themselves. But the recursion propagates changes through the operator tree to determine additions and deletions. Thus, the cost for incremental maintenance is determined by the complexity of the operator tree that is used inside the recursion.

Moreover, the recursion deletes an over-approximation of results and has a rederivation phase; hence, deletions are not treated in a minimal manner. The problem here is that the deletion and rederivation phases are also performed recursively. Note that the rederivation in itself is natural for a recursive operator. Consider for example the transitive closure which has a similar phase. In the transitive closure an over-approximation of edges is marked as suspicious and then alternative derivations are found by concatenating trusted edges. The difference between the transitive closure and the general recursion is that the transitive closure has an efficient way of finding alternatives by concatenations of two or three trusted edges. A transitive closure phrased as general recursion essentially performs a traversal over the graph during the rederivation phase, which is algorithmically much slower.

## 3.5 Discussion

In this chapter we have presented the foundations of a language-integrated database for incremental view maintenance. Incremental maintenance is achieved via operators of the relational algebra over multisets. The operators are expressive enough to formulate basic SPJ queries, set-theoretic queries, negations (via set difference), aggregations, existential queries, transitive closures and recursions. The following points deserve a closer discussion:

### 3.5.1 Memory Optimality of Incremental Maintenance

The queried data is represented as objects that are logically available as tuples in base relations, yet base relations do not actually materialize (store) the data. Instead data is materialized inside the operators only if required for incremental maintenance of the operator itself. Each operator has a change-propagation expression for incremental maintenance that materializes a minimal amount of data for the respective operator. Hence, the solution is optimal w.r.t. the auxiliary data used in each operator.

The overall optimality of the queries is in most cases reached when treating each operator in isolation. However, there are cases where a different treatment can be better. Consider for example the set-theoretic operators that rely on internal materializations of $A - B$ and $B - A$, where $A$ and $B$ are the underlying relations. If, for example, $A$ is already materialized, the auxiliary data is redundant and these operators can instead be maintained by directly materializing $B$. Note that the result of each operator is typically only materialized if explicitly requested by the user. Nevertheless, such cases can arise if a user has requested the materialization, or if $A$ is computed by an operator that must materialize its results. The latter is done, for example, in set differences or transitive closures. Yet, there is currently no data on how common such cases are. Indeed, they were practically non-existent during our evaluation of queries from the domain of static code analyses. Hence, for simplicity, we treat operators as presented in Sec. 3.4.3 and leave such optimizations as future work.

### 3.5.2 Runtime Optimality of Incremental Maintenance

The evaluation strategy followed by observer-style change propagation can be categorized as a *bottom-up evaluation*, which – in general – is considered inferior to a top-down evaluation. The bottom-up nature is inherent in the strategy, since the evaluation always starts at the base relations, i.e., the bottom, and propagates results to operators further up in the operator tree. The downside of this strategy is that each operator always produces a complete result for the defined query. If subsequent operators rule out resulting tuples, i.e., due to selection conditions, some results are not really required to be computed, i.e., unnecessary computations are performed. However, there are optimizations that can alleviate the problem found in bottom-up evaluation systems. One such optimization is to push down a selection to a lower level in the operator tree, hence, ruling out unnecessary results earlier in the computation. We discuss optimizations of the operator tree in the next chapter.

For an optimal runtime there is an interesting issue w.r.t. providing single-value modifications, i.e., atomic add/remove events, and the multiset based modifications used in the changed propagation expressions. In comparison, the single-valued modification is much faster (approx. 5 x) over large datasets even for trivial queries. The reason is that propagating the changes as multisets requires the creation of additional data structures, which are only used to notify the next operator. Notifications using single values are simple method calls. In other words, for a dataset of size *n* it is much faster to make *n* method calls then to construct a list of *n* results and making 1 method call.

### 3.5.3 Complexity of Integrating Incremental Maintenance

In terms of the complexity of incorporating incremental view maintenance into an application, a minor amount of integration must be performed. The view maintenance requires events for object modification that must be explicitly (manually) triggered as method calls to the base relations. Yet, this is well feasible if the number of base relations remains small compared to the number of defined views. For example, in the case of the static analyses formulated as part of our case studies, there are only four base relations and 50+ views – declared as part of the case studies.

The explicit triggering provides a very flexible form of integration into an application, since the application is free to perform non-incremental computations on the data. There is only one restriction on objects that participate in the incremental view maintenance, which is that state mutations must be explicitly communicated to the database as an update event and may not be performed outside of the incrementally maintained operators (cf. Sec. 3.1.1).

## 3.6 Related Work

A large variety of related work was already discussed in Chapter 2 with the focus on the background of static analyses, different database technologies and incremental view maintenance. This section addresses further related works w.r.t. incremental view maintenance in databases, that was not required as background knowledge. Furthermore, works on language integration for database queries are discussed.

**Database view maintenance**

In the context of OO databases, the MOVIE project [AFP03] provides a more fine-grained treatment of incremental modifications to objects with nested collection. The basis of these modifications is a very fine granularity for events, which can be summarized as `onInsertElementToCollection`, `onDeleteElementFromCollection`, and `onModifyAttributeInCollectionElement`. A formalization of such modifications for a subset of OQL was discussed in [Nak01], where OQL is reduced to monoid homomorphisms. Compared to the events defined in this thesis, the events in MOVIE allow more fine-grained incremental updates. We did not consider events at this granularity, since the primary target of our database are static analyses, where such events are hard to come by. However, extending our approach to a broader treatment is well feasible.

A different kind of database system has been introduced under the term *column-oriented database* (cf. [AMH08] for a conceptual overview and comparison to traditional row-oriented databases). The focus of these database systems is the optimization of query performance, where – in essence – column-oriented systems are more I/O efficient for read-only queries, but perform less well for write-only queries. This performance gain is mainly due to the fact that the database has to fetch (from a hard disk) only those columns (i.e., attributes) that are really accessed by a query, whereas row-oriented systems fetch all the data in a row. The research in column-oriented databases frequently cites an optimization technique termed *late materialization*, where rows are reconstructed (materialized) by joining together columns and these joins are performed as late as possible in a query's execution. Late materialization is conceptually different from the term materialization used in this thesis, since late materialization only refers to the reconstruction of row-like data after performing a query and is not tied to an incremental maintenance of the query's results. In contrast we use the term materialization to denote auxiliary data used during incremental maintenance of the query's results. Column-oriented database systems per-se offer a different angle at query processing that is orthogonal to incremental view maintenance. There are several implications of using column-oriented systems: (i) they require less main-memory during computation of results, since less data is loaded into memory, i.e., only required columns; (ii) the column-oriented requires additional column-based indices (on a hard disk) to back up this style of query processing; (iii) typically compression schemes are used in these systems, since column-based storage can be compressed better than row-based storage. Concepts, such as column-based indices and compression are currently not considered for the query processing in this thesis, but are worthwhile exploring in future works.

From a practical perspective it seems prudent to make a distinction between several applications of materialized views that are supported by commercial systems, e.g., Oracle Database[5], IBM DB2[6], etc. In these systems three basic use-cases employ materialized views. (i) pre-computation of query results for faster access, (ii) storage of data from several distributed database, and (iii) storage of summary tables to answer online analytical processing (OLAP) queries. Typically the term “materialized view” is used technically to refer to views that support the former two use-cases. In this sense the traditional databases use incremental view maintenance as a form of result caching, i.e., in the original sense of materialized views defined in Sec. 2.3, and also require a materialization of the base relation. The third use-case stems from the domain of data warehousing with federated databases and is an exception in so far as access to base relations is actively avoided by the incrementalization. In this setting the materialized views are available and maintained in the data warehouse. Yet, the base data must be accessed from remote databases that can have high access costs, e.g., if a high amount of data must be transferred over a network. Thus, it is beneficial to maintain the views without accessing the remote database. However, OLAP is a very specialized case that largely revolves around multi-dimensional aggregate queries [GM99a]. In this context materialized views can be used to pre-compute summary tables that increase the query performance of the data warehouse. View maintenance in OLAP also tries to achieve self-maintainability, yet, the view maintenance is to some extent orthogonal to traditional relational algebra and geared towards the operations for the underlying multi-dimensional aggregates.

It is also important to note that not all database systems support materialized view definitions. Especially NoSQL [Sto10] databases, e.g., Hadoop[7], often provide low-level abstractions, e.g., the map and reduce functions (or key-value stores),

[5] http://www.oracle.com/database
[6] http://www.ibm.com/db2
[7] http://hadoop.apache.org

which do not allow the definition of materialized views as supported by traditional relational databases.

Another (recent) development in databases is the focus on big data stores and that use parallelism to allow the database to cross multiple machines – in essence treating a cluster of servers as a single computation resource. For example, SAP's Hana[8] main-memory database can utilize a cluster of servers as a single memory pool. Likewise, the Hadoop MapReduce framework utilizes parallelism to perform computations on multiple servers, but uses persistent data. These approaches are viable for large business applications, yet the integration of static analyses into a developer's IDE calls for a lightweight approach.

The language integration proposed in this thesis can be considered lightweight in the sense that large(r) amounts of data can be processed on a single machine without large memory requirements and without using a fully-fledged database system. The language integration supports an incrementalization where the base relations and the results are transient and only auxiliary data that is required for the incremental view maintenance is materialized. The concept of self-maintainability plays a pivotal role in this approach, to identify minimal memory requirements. In itself the concept of self-maintainability was previously applied in database systems, as exemplified in the OLAP use-case above. Yet, our approach abandons traditional database models such as persistence and concurrent access and only keeps the query and optimization abilities of databases. Persistence and concurrency are not required in the setting of an integration of static analyses into a developer's IDE, since only one user on a single machine accesses data and the data is stored by default in the IDE.

Finally, the self-maintainable operators and the minimal auxiliary data identified in this chapter (cf. Sec. 3.4.3) are a good starting point, yet many standard operators – especially joins – do require materialization of data. Thus, we introduce an additional non-standard optimization that further reduces memory requirements in the next chapter. The proposed optimization enables scoping for the incrementalized data, such that no materialization is required for queries that correlate data within a single scope.

### Language integration for database queries

In recent years an increasing amount of works have provided *language-integrated query* capabilities, i.e., integrations of database queries into programming languages. The most widely known works are Microsoft Linq [MAB08] and Ferry [GMRS09], the latter was embedded into Scala as ScalaQL [GIS10]. The focus in these works is to integrate standard database technologies with general purpose programming

[8] http://www.sap.com/hana

languages to provide a better handling of persistence related code. The main issues addressed are (i) the type-correctness for database queries w.r.t. schemas in persistent databases and (ii) query optimization in the programming language (i.e., on the client side) to reduce the amount of persistent data moved between client and database.

The above works are orthogonal to this thesis in the sense that they can provide different query front-ends. We discuss more on this issue in the next chapter after introducing our own query language. It is important to note that the above cited works do not explicitly target incrementalization. Queries against in-memory collections could theoretically also be incrementalized in LINQ. However, the standard implementation (LINQ2Objects) performs no incrementalization. The commercial library LiveLINQ provides incrementalization in the form of caching of result collections. Furthermore, indexing is provided as an optimization for joins. All cached results, and indices, as well as the underlying base relations are then fully materialized in main-memory. The range of incrementalized operations provided by LiveLINQ is not well documented. However, it seems fair to state that LiveLINQ supports the basic and set-theoretic operators (cf. Sec. 3.3.2 and Sec. 3.3.3), since these are declared in the standard LINQ API. Nevertheless, there is no concrete documentation on how exactly individual operators are incrementally maintained. The available documentation on LINQ states that lists are used for the base relations and results (in conjunction with indices for optimizations), which conceptually means materialization of base relations as well as results. The underlying assumption is that base relations are materialized in-memory anyway (by fetching data from a database). Yet there is not information on auxiliary data or discussions on memory costs for the incrementalization.

The Java Query Language (JQL) [WPN06] is an approach that provides a declarative form of queries for the Java programming language. Yet, JQL is very specialized w.r.t. the supported relational operators – mainly joins. The approach uses standard optimization techniques – primarily join ordering – to improve runtime performance. In [WPN08] caching for JQL results is introduced together with incremental maintenance of the cache. Updates to JQL caches also assume the availability of the underlying base relations.

In general, language-integrated queries over in-memory collections are architecturally preferable over large distributed and parallel systems, when it comes to our use-case of integrating incrementalized static analyses into an IDE on a single computer. The two main reasons are that (i) there is no runtime cost for transitioning the data to/from servers over a network (and typically paying transaction costs for concurrent access, even though there is no concurrency) and (ii) the practicality of setting up a distributed system for a single developer is questionable.

However, the current language integrations share two characteristics: (i) the incrementalization is added as an “afterthought” which leads to a treatment of (ii) the incrementalization as a form of querying completely materialized collections and caching (i.e., materialization) of results. The latter is conceptually equivalent to treating the results as materialized views in databases. This means that in these approaches in-memory collections for base relations and for results are materialized alongside any auxiliary data structures required for view maintenance.

In contrast to the existing language integrations the treatment of incrementalization in this thesis materializes as little data as possible to allow scalability to large(r) amounts of data. In essence the base relations and the view results are seen as transient data and only the auxiliary data structures required for incremental view maintenance are materialized. The concept of not materializing base relations and – to a smaller extent – results is novel for language integrations, since the traditional programming model of working with collections is to have all the data available in memory. Indeed not materializing base relations is a very distinctive design decision that is viable only for a domain where the incrementalized data is available (e.g., persisted) outside of the running application. In this thesis the domain is static code analyses where the underlying data, i.e., the source code, is persisted by the IDE. However, the approach is easily applicable to other domains that revolve around file-based editing as for example model-driven tools.

# 4 Query Language and Query Optimization

This chapter presents the query language that is integrated into the host language Scala and discusses query optimizations in detail. In Section 4.1 we first present the query language via examples that cover the formulation of queries for the relational operators defined in the last chapter. Then the query syntax is compared to a standard SQL syntax and finally static type safety is discussed. In Section 4.2 several query optimizations are discussed. We cover the application of *traditional optimizations* and consider *optimizations for multiple views* that reuse overlapping parts of their queries, i.e., common subqueries. Then the relationship of the language integration to traditional query *optimizations of OO databases* is discussed. Furthermore we discuss how *indexing* is performed in our system, since indices play an important role for runtime, but should not require a complete materialization of data. Finally, we discuss a novel optimization that introduces *increment local operators*. Increment locality introduces a notion of scoping for data that is propagated from the base relations and used during incremental view maintenance. With this optimization views, that perform computations within said scopes, are self-maintainable, i.e., do not materialize any memory.

## 4.1 Query Language – IQL

The query language has a surface syntax inspired by SQL [CB74] and is termed **IQL** for “Incrementalized Query Language”. IQL provides keywords to formulate queries in typical SQL-style with `SELECT`, `FROM` and `WHERE` clauses. Queries formulated using IQL are then compiled into operator trees (cf. Section 3.4) whose results are automatically incrementalized. IQL is an embedded domain specific language that is integrated into Scala and, hence, is provided as a library.

The IQL syntax was developed with the following rationales in mind: First, the language must provide a type safe embedding into Scala. Second, the language should follow the standard SQL syntax where possible. Third, IQL must be expressive enough to formulate queries using the supported operators (defined in Section 3.3). However, IQL is not a complete transcription of the SQL language.

In this section the IQL language is presented in a mostly example driven manner. In addition, the type safety of the language is discussed and the key differences between IQL and SQL are highlighted and why IQL has to deviate from the standard SQL syntax.

The compilation from IQL to the relational operators is similar to a compilation from standard SQL to relational algebra. Query compilation is discussed in standard text books such as [GMUW08]. The general principles apply to IQL and a detailed discussion is omitted.

### 4.1.1 Defining Queries in IQL

All functions of the various operators used in the examples, e.g., selection functions, projection functions, which were previously formulated as $\lambda$ functions are now defined as (anonymous) Scala functions. The language is exemplified via queries on the extents $E_{Students}$ and $E_{Registrations}$, which are assumed to be available as objects `students` and `registrations` of the types `Extent[Student]` and `Extent[Registration]` respectively.

**Expressing queries using "dotless" function calls for keywords**

The basic definition of SQL keywords adopts a "dotless" function call style to mimic SQL style syntax. The basic Scala syntax rule that enables this is as follows: if there is an object and a function (message) called on the object, then the "dot" notation found in most object-oriented languages can be omitted. A respective query using the three basic SQL keywords is shown in Figure 4.1. The query formulates the view $view_1$ from the first example in Section 3.4.2 (p. 80).

```
SELECT ((_:Student).lastName) FROM students WHERE
    (_.gradeAverage < 3.0)
```

**Figure 4.1:** IQL query for selection of last names of a filtered set of students

The basic idea for the syntax is that each keyword is a method, called on an object representing the previous clause in the query. The `SELECT` clause is the staring keyword and, hence, uses a Scala **`object`** with an `apply()` method that takes the projection function[1] that transforms students to their last names. The produced clause defines a method `FROM` which can be called in the "dotless" style provided by Scala. The `FROM` takes one or more relations as parameters in the form of `Extents`, compiled `Relations` or uncompiled IQL queries. The produced `FROM` clause then defines a method `WHERE` which takes the selection function as a parameter. To

[1] Note that in standard SQL the `SELECT` clause defines projections, and the `WHERE` clause basically defines selections in the sense of the relational algebra operators

illustrate the general idea behind the syntax, the Figure 4.2 shows an expanded version with all the omitted syntactic elements from Scala.

```
SELECT.apply((_:Student).lastName).FROM(students).WHERE
    (_.gradeAverage < 3.0)
```

**Figure 4.2:** Expanded IQL query from Figure 4.1 with “dot” syntax

**Expressing queries using function definitions to omit type annotations**

Several of the functions, provided to the various operators via IQL, require type annotations for type safety. For example, the projection function in Figure 4.1 (line 1) has a type annotation `_:Student`, while the selection function (line 2) required no type annotation, i.e., used the `_` expression.

The type annotation in this position (and in several others) is required, because the Scala compiler can not infer any type for an anonymous function. The two rules of Scala that hamper the compiler from inferring types are (i) left to right deduction of types[2] and (ii) types are only deducible if the anonymous function is directly passed as a parameter to a method where all parameter types are known.

In the case of the above projection, the anonymous function is passed to the head of the expression, which means that the first rule hampers the compiler from deducing any type, i.e., there is no expression to the left that already can tell us what type the projection should have. In the case of the selection function both previous clauses have clearly stated that the objects of discourse are of type `Student`. The `SELECT` clause defines the type since an explicit function from `Student` to `String` was passed and the `FROM` clause defines the type of the objects since `students` is of the type `Extent[Student]`.

To alleviate the burden of type annotations a set of explicitly typed functions can be defined over the objects of discourse. For illustration Figure 4.3 depicts the above query using an explicitly defined function `lastName` (line 1). The function is passed as a parameter to the `SELECT` instead of the anonymous function (cf. line 2).

**Expressing SQL-style `SELECT` clauses**

The added benefit of using explicit functions is that one can achieve a type-safe and SQL-style definition for projections to multiple properties of the selected objects. In line 2 in Figure 4.4 the projection is defined using two functions that each take

2 The left to right deduction of types is a Scala specific problem. Other languages, such as Haskell, do not have this problem.

```
def lastName : Student => String = _.lastName
SELECT (lastName) FROM students WHERE ...
```

**Figure 4.3:** Shortened IQL query using function symbols

a `Student` object as parameter. The functions are implicitly converted by IQL to a function that takes a `Student` as parameter and returns a tuple of two strings.

```
def firstName : Student => String = _.firstName
SELECT (firstName, lastName) FROM students WHERE ...
```

**Figure 4.4:** IQL query of an SQL-style `SELECT` clause for multiple selected properties

**Expressing joins and join conditions**

Joins are expressed by passing multiple tables to the `FROM` clause and formulating one or more join conditions. Note that without join conditions such a query yields the Cartesian product of the tables. For illustration, Figure 4.5 depicts a self-join of the student relation that yields combinations of students that have the same first and last name but a different grade average. To indicate the join in the query, the extent `students` is passed twice to the `FROM` clause. The result of the query contains Scala tuples of two student objects. Note that the result is defined using the special keyword `(*)`, hence no projection is performed and the result is returned as defined for a join (or Cartesian product) operator, i.e., as a tuple of the joined objects.

```
1 SELECT (*) FROM (students, students) WHERE
2   (firstName === firstName) AND
3   (lastName  === lastName) AND
4   NOT (gradeAverage === gradeAverage)
```

**Figure 4.5:** IQL query for joining students with same name and different grade average

The query contains two positive join conditions, stating that the first names of both students must be equal (line 2) and the last names must be equal (line 3). The conditions are defined as functions that are supplied to a special IQL equality operator "===" that allows to specify equality between the conditions in an infix notation style. From the perspective of the query it is instructional to think of the equality operator as taking two functions, applying the left function to objects contained in the left relation in the `FROM` clause and applying the right function to objects contained in the right function. All objects that yield the same result under these functions are results of the join. In the case of the above query multiple equalities are defined in conjunction (keyword `AND`). Hence, objects have to agree in the returned values of all functions. The query contains a third condition (line 4) that is negated, which means the objects must yield different values under the given functions. Note that the negation (`NOT`) is constructed as a new object that is passed to the enclosing query. This technique increases usability by allowing to omit numerous parenthesis, e.g., the outer parenthesis in the expression `AND( NOT(...) )`.

**Expressing existential sub-queries**

Existential quantification is basically a filter condition on elements from a different query (as discussed in the respective operators in Section 3.3.5). Hence, existential quantification is formulated as sub-query[3] in SQL. The existential condition is initiated by the special `EXISTS` keyword, which is part of the `WHERE` clause and contains a complete sub-query of the form `SELECT`, `FROM`, `WHERE`. The sub-query specifies a join condition between objects in the sub-query relation and objects in the outer query relation. The join condition specifies the properties for elements in the outer query, such that there also exist elements in the sub-query with equal properties.

Note that the join condition can be omitted. In this case the outer query basically selects all elements of the outer relation, if the sub-query returns a non-empty result. However, this case was of less practical value for the static analyses formulated in this work and, hence, the (more interesting) example of using join conditions is presented.

Figure 4.6 illustrates this using the example query already shown in Figure 3.20 (p. 83), which selects a filtered set of students that have also registered for a course. The example uses two function definitions (for brevity and readability); (i) `registeredStudent` (line 1) retrieves the student object from a registration entry and (ii) `student` (line 2) is a short form of the identity function. Using the functions makes the query shorter and more readable than supplying anonymous lambda functions, since the latter would clutter the query with type annotations for the respective parameter and return types. The outer query uses the standard clauses presented earlier to select and filter all students that have a grade average below 3.0 (lines 3 and 4). The sub-query is surrounded by the `EXISTS` keyword (line 5) and the sub-query starts with the selection of all registrations (line 6). Finally, the

[3] Sub-queries are also often referred to as nested queries

```
1 def registeredStudent: Registration => Student = _.student
2 def student: Student => Student = _
3 SELECT (*) FROM students WHERE
4     (_.gradeAverage < 3.0) AND
5     EXISTS (
6       SELECT (*) FROM registrations WHERE
7         (registeredStudent === student)
8     )
```

**Figure 4.6:** IQL query for selection of a filtered set of students registered for a course

join condition declares that a the `registeredStudent` contained in a registration is equal to the `student` from the outer query (line 7). The join condition uses the same syntax as seen in a join of two tables. In the case of sub-queries the condition for elements of the sub-query is written on the left-hand side and the condition for elements of the outer query on the right-hand side.

**Expressing unnesting**

The unnesting basically takes a multiset – i.e., a Scala collection – stored in an object and makes the elements available as a flattened relation. Inspired by SQL, the keyword `UNNEST` is used to indicate the unnesting. Since the unnesting operator basically creates a new relation, it is only natural that the keyword is found as a substitute for a relation in the `FROM` clause.

For illustration the query in Figure 4.7 declares an unnesting that extracts the list of grades from students and makes them available as a relation. The relation has a result type of tuples of `Student` and `Int` (line 1), as declared in the respective operator in Section 3.3.4. Thus, as a result, each student is paired together with every grade stored in her list `grades`. The unnesting is defined via the `UNNEST` keyword, which takes two parameters; (i) the relation that contains the objects that hold the to-be unnested collection as a property and (ii) the function used to extract the collection from each object, e.g., `_.grades` in the example below.

```
1 val studentGrades : Relation[(Student, Int)] =
2     SELECT (*) FROM UNNEST (students, _.grades)
```

**Figure 4.7:** IQL query for unnesting the list of grades from students

### Expressing aggregations

An aggregation – in the general form – consists of two parts, (i) the aggregation function(s) for building a single value(s) from a multiset and (ii) the grouping function that segregates the objects in the underlying relation into groups, i.e., multisets that contain all objects which are equal under the grouping function. The separation of aggregations into two parts is also eminent in the standard SQL syntax, where the aggregation function is declared as part of the `SELECT` clause and the grouping function is an optional clause positioned after the `WHERE` clause. The IQL syntax adheres to this style of defining aggregations.

For illustration Figure 4.8 depicts an aggregation over the `studentGrades` relation, as defined in the previous example. The aggregation groups all students by their semester and selects the minimum of all grades students of the respective semester had. As in the previous examples several functions are used for brevity and readability. The functions in line 1 and 2 return the first and second component of the `(Student, Int)` tuple – returned as results in `studentGrades` – in more readable form. The function `semester` (line 3) is again a more declarative form of the identity function that transports the message that the integer value represents a semester. The first part of the aggregate is declared in the `SELECT` clause. IQL

```
1 def gradedStudent : (Student, Int) => Student = _._1
2 def grade : (Student, Int) => Int = _._2
3 def semester : Int => Int = _
4 SELECT (semester, MIN(grade)) FROM
5     studentGrades GROUP_BY (gradedStudent(_).semester)
```

**Figure 4.8:** IQL query for selecting the lowest grade of students by semester

has the convention that the aggregate functions are declared last in the clause. For example, the `MIN` function is declared as last function in line 4. Multiple aggregation functions can be given but must all be used after the general projection functions. The projections functions either take the objects in the underlying relation as a parameter or, if a grouping function is present, the values returned by the grouping function. Due to type-safety these alternatives are exclusive, i.e., if a grouping function is present the projections must adhere to the type returned by the grouping. In the example, students are grouped by a single function, which returns a student's semester as an `Int` value, hence only projections from `Int` values are possible. The grouping function is declared using the keyword `GROUP_BY` that starts a new clause (line 5). Note that grouping by multiple values is also possible and declared in the style of multiple functions in a projection. The result (and its type) is then a tuple of the values (types) returned by the grouping functions.

**Expressing transitive closures**

Transitive closures – like unnestings – generate new relations and are thus positioned in the FROM clause. For simplicity the keyword deviates from the SQL standard and the transitive closure is denoted by TC.

Consider for illustration the definition of the transitive closure depicted in Figure 4.9, which computes all transitive prerequisite courses of a master course. Master courses were shown in Figure 3.6 as an extension of normal courses. Each master course has a list of prerequisite courses, i.e., as field `prerequisites: List[Course]`. In the example the relation `masterCourses` is assumed to contain the data over all master courses. The direct prerequisite courses are obtained via an unnesting (line 8). Note that the unnesting generates a relation that contains tuples of `MasterCourse` and `Course` objects. The tuples are transformed by the query (line 7) into objects of the type `Prerequisite` (line 1) for easier treatment. We use the function `asPrerequisite` (whose definition is omitted) to transform the tuple.

```
 1 case class Prerequisite(
 2   course: MasterCourse,
 3   prerequisite:Course
 4 )
 5
 6 val directPrerequisites =
 7   SELECT (asPrerequisite) FROM
 8     UNNEST(masterCourses, _.prerequisites)
 9
10 val transitivePrerequisites =
11   SELECT (asPrerequisite) FROM
12     TC(directPrerequisites, _.course, _.prerequisite)
```

**Figure 4.9:** IQL query for the transitive closure over all prerequisite courses

The actual transitive closure is very simple (line 12) and is defined via the keyword TC together with three parameters. The first parameter is the underlying relation (`directPrerequisites`). The second and third parameters are the functions that retrieve a start and end vertex to interpret an object in that relation as an edge.

In the above example, the objects are interpreted as edges from a course to its prerequisite. The result is again transformed from a tuple to a prerequisite.

**Expressing recursion**

The definition of a recursive query is reminiscent of SQL's treatment of recursion – defined in the respective extensions of the SQL standard – and also features the keyword `WITH RECURSIVE` (denoted as a single keyword `WITH_RECURSIVE`).

For illustration, the transitive closure defined in the last paragraph is phrased as a general recursion in Figure 4.10. The example reuses the definition of `Prerequisite` and the relation `directPrerequisites`. The `WITH_RECURSIVE` keyword (line 9) takes two parameters. The first is the relation that should be substituted in the recursive query (cf. Sec. 3.3.4 w.r.t. the substitution semantics). The second parameter is the query that has the recursive dependency. All occurrences of the query given as the first parameter are substituted in the second query. The result of the keyword `WITH_RECURSIVE` is the result of the substitution.

```
 1 def course: Prerequisite => MasterCourse = _.course
 2 def prerequisite: Prerequisite => Course = _.prerequisite
 3
 4 def asTransitivePrerequisite:
 5   (Prerequisite, Prerequisite) => (Prerequisite) =
 6     (e => Prerequisite(e._1.course, e._2.prerequisite)
 7
 8 val recursivePrerequisites =
 9   WITH_RECURSIVE(
10     directPrerequisites,
11     SELECT DISTINCT (asTransitivePrerequisite) FROM
12       (directPrerequisites, directPrerequisites) WHERE
13         (prerequisite === course)
14   )
```

**Figure 4.10:** IQL query for a recursive traversal over prerequisite courses

The presented query is a concrete example of the general equality between transitive closure and recursion that was shown in Sec. 3.3.4. Two prerequisites are joined if the prerequisite of one equals a course with a further prerequisite (line 13). From the resulting tuples a new transitive prerequisite is constructed via the function `asTransitivePrerequisite` (lines 4 - 6). The results must be distinct (line 11) to ensure the termination of the recursion.

## 4.1.2 Comparison to SQL Syntax

The main difference of IQL to standard SQL is the usage of functions instead of identifiers to signify the column names to retrieve properties from an object. SQL is a standalone language that introduces its own syntax with a notion of identifiers. The language is then parsed and the parsed expressions can be checked for correctness by looking up the defined schema for the tables in the database. For example, in SQL, `students` is an identifier bound to a qualified table and `students.firstName` is bound to a qualified column, which is valid if the table `students` defines this column in its schema.

The usage of identifiers can be illustrated best by SQL's **`AS`** keyword, which allows to (re-)name columns explicitly. Figure 4.11 depicts a query that concatenates the first and last name of a student and provides it as the column named `fullName`. The identifier `fullName` is then a legal column name for the outer query.

```
SELECT fullName FROM(
  SELECT CONCAT(firstName,'␣',lastName) AS fullName
  FROM students
)
```

**Figure 4.11:** Example query that explicitly names a column in SQL

An embedded DSL such as IQL can not easily introduce a new notion of identifiers as they are found in SQL. This would entail writing a parser and type checker for SQL expressions, with knowledge of where identifiers are declared and where they can be looked up. For example, in the query in Figure 4.11 the parser must determine that the inner query declares an identifier (column name) `fullName` and that this identifier is valid for subsequent (outer) queries. In order to simplify the embedding of IQL, the notion of SQL identifiers is dropped and IQL uses first-class functions which can mimic column names.

**Transporting the meaning of functions used as identifiers**

Functions – in contrast to identifiers – can be anonymous and, hence, some queries may not transport the meaning of particular functions in `SELECT` clauses – in terms of values they represent. To ease the understanding for a query reader the IQL queries should be declared using meaningful function names as seen in the examples in the previous section. In addition, as we will see in Section 5, many queries are defined such that they return either attributes from the underlying objects or new objects of a specified type. Hence, there is no need to declare a lot of function names only to transport the meaning of a query. For example, instead of selecting a tuple of strings – which does not port the meaning of the strings – as seen in the query in Figure 4.4, the two strings can be wrapped in a new object that signifies the meaning via the declared field names. Figure 4.12 depicts the respective query as a transformation, which constructs a new object of type `Name` from a student. The name object retains the meaning of the values through the names of the fields in its declaration (line 1).

```
1 case class Name(firstName:String, lastName:String)
2 SELECT (s:Student) => Name(s.firstName,s.lastName)
3   FROM students
```

**Figure 4.12:** IQL query for a projection that constructs a new object

### Linking functions to relations

A further consequence of not having identifiers is that the link between the supplied functions and the queried relations is not explicit, i.e., the functions in join conditions and predicates do not explicitly mention the relation to which they are applied.

For join conditions a positional arrangement of the provided functions serves to establish a link to the queried relations. In an IQL query the left and right hand relations are linked to the left and right hand join conditions (as seen in Figure 4.5). In SQL, tables can be explicitly referenced via identifiers. For illustration Figure 4.13 depicts the join as an SQL query. The relations (SQL tables) are explicitly named as `s1` and `s2` (line 1). The join conditions can reference the tables via their identifiers and the position of the left or right hand table in the conditions is irrelevant; as seen in line 2 and 3 where the position of left-hand and right-hand side are switched. However, refraining from switching the position of conditions places no heavy burden on query writers.

```
1 SELECT * FROM students s1, students s2 WHERE
2   s2.firstName == s1.firstName AND
3   s1.lastName  == s2.lastName AND
4   ...
```

**Figure 4.13:** Example query that explicitly names tables in SQL

For predicates – in IQL – the link must be established via the types of the provided functions. For example, consider the query in Figure 4.14, where two relations are queried and a predicate is defined on each of them. There are three consequences of using the types to bind the predicates to specific relations. First, a positional

```
1 SELECT (*) FROM (students, registrations, registrations)
2 WHERE
3   (_.firstName == "Sally") AND
4   ((_:Registration).course.title == "Analysis I") AND
5   NEXT((_:Registration).course.semester == "Winter/2012")
```

**Figure 4.14:** IQL query using predicates on multiple relations

treatment is assumed as an "intuitive" solution, i.e., providing predicates in the order of the relations in the queries `FROM` clause. Otherwise users must carefully analyze a query to establish to which relation a predicate belongs. A free mixing of supplied functions – with respect to the order of the relations – is theoretically possible, but not supported by IQL. Second, for reasons of type inference, IQL requires completely typed functions to signify that the predicate is linked to the next relation. Thus, these functions can not use an untyped parameter name, e.g. `(s => s.firstName)` or the untyped placeholder, e.g. `_.firstName`. Note that IQL assumes that after a predicate of a specific type more predicates of the same type will follow, hence, predicates can make extensive use of Scala's untyped placeholder, which allows to write predicates in a very brief fashion. For illustration consider Figure 4.14. The first predicate (line 3) is declared as an untyped placeholder; IQL assumes the type is compatible to the first relation. The second predicate must be completely typed (line 4) to signify that the predicate reasons over objects of type `Registration`. The final point w.r.t. linking functions and relations is, that in extreme cases the query is defined with multiple references to the same relation or with references to relations of the same type. In this case type information is insufficient to indicate that a function is linked to the next relation. Note that this never happened in the static analyses defined in this thesis. Nevertheless, it can be supported via the keyword `NEXT` (line 5), which signifies that the predicated is linked to the next relation in the `FROM` clause.

### 4.1.3 Static Type Safety of IQL

The main idea for ensuring static type safety for IQL expressions is to provide a type safe encoding of the different IQL clauses (`SELECT`, `FROM`, etc.) into Scala and defer type checking to the Scala compiler. The basic principle followed in this approach uses parametric types in the declarations of each clause. The parametric types are bound to the types used in the query through the various parameters passed to each clause, i.e., the projection function(s), the concrete relations or the selection conditions. Type constraints on the bound parametric types ensure the applicability of parameters in subsequent clauses, hence making the whole IQL query type safe. A completed (and type correct) query is then parameterized by the type of the objects in the resulting relation, hence ensuring compositionality of queries. The full range of IQL clauses consists of a large number of classes; hence, the type safe embedding of IQL is only discussed to the degree necessary to understand the type constraints for a query.

**Basic type safety for a select-from query over a single relation**

To illustrate the approach Figure 4.15 depicts an excerpt of the respective types for the basic clauses `SELECT`, `FROM` and `WHERE`. The special object `SELECT` declares the method `apply` (line 2) that initiates the declaration of an IQL expression (cf. Section 4.1.1 for an example of the function call chain that constructs an expression). The parameter `projection` (line 3) is a function from `SelectionDomain` to `Range` that binds the two parametric types. For example, the function `((_:Student).lastName)` binds the types to `Student` and `String`. The `apply` method returns an object of type `SELECT_CLAUSE` which is parameterized by the domain and range types of the projection.

The type `SELECT_CLAUSE` (line 7) then defines the method `FROM` which takes the relation on which the query is performed; provided as a parameter (line 9). The type constraint in line 8 declares that the relation must contain objects of a subtype of the objects passed to the projection function declared in the select clause. For example, given the projection function for a student's last name, the relation must have objects of a compatible type, i.e., `Student` or subtypes of `Student`, since these types define `lastName` and, hence, are compatible. The method `FROM` returns a `FROM_CLAUSE` (line 10), which is parameterized by the type of the objects in the relation (`Domain`) and the returned objects of the projection (`Range`).

The `FROM_CLAUSE` (line 13) is the first clause that can conclude the definition of an IQL query, which is signified by inheriting from the type `SQL_QUERY` (line 14). In an `SQL_QUERY` the type parameter `Range` is bound to the type of the resulting objects. Alternatively, the query can further specify a `WHERE` clause (line 15). The `WHERE_CLAUSE` (line 19) allows to specify an arbitrary number of conditions by passing predicate functions to the methods `AND` and `OR` (lines 21 and 23), which take the domain objects of the relation as input and return a boolean value. Each condition again returns a WHERE_CLAUSE such that composition of many conditions can be done via a chain of method calls.

```scala
object SELECT {
  def apply[SelectionDomain, Range](
          projection: SelectionDomain => Range
  ): SELECT_CLAUSE[SelectionDomain, Range]
  ...

trait SELECT_CLAUSE[SelectionDomain, Range] {
  def FROM[Domain <: SelectionDomain](
           relation: Relation[Domain]
  ): FROM_CLAUSE[Domain, Range]
  ...

trait FROM_CLAUSE[Domain, Range]
    extends SQL_QUERY[Range] {
  def WHERE(predicate: Domain => Boolean
  ): WHERE_CLAUSE[Domain, Range]
  ...

trait WHERE_CLAUSE[Domain, Range]
    extends SQL_QUERY[Range] {
  def AND(predicate: Domain => Boolean):
    WHERE_CLAUSE[Domain, Range]
  def OR(predicate: Domain => Boolean)...
  ...
```

**Figure 4.15:** Excerpt of the type safe select and from clauses for a single relation

**Type safety for queries over multiple relations**

The basic principle of binding parametric types to various types – used in each clause – is also applicable to queries over multiple relations. The key idea for multiple relations is that for each number of relations a different type with a respective number of type parameters is needed[4]. This entails defining one type per

[4] This is similar to Scala's internal treatment of tuples, which can be declared as 2 to 22 comma-separated values enclosed in parentheses, which each construct a different type, e.g., `Tuple2`, `Tuple22`.

each clause for each number of parameters. For example the type WHERE_CLAUSE_2 signifies a where clause over two relations.

For illustration, Figure 4.16 depicts an excerpt of the clause types over two relations. The SELECT clause is constructed from a projection function that takes two parameters (line 3) and binds respectively more domain types in the type SELECT_CLAUSE_2 (line 4). The FROM and WHERE clauses are extended in a similar manner from the clauses over a single relation in Figure 4.15.

```
 1 object SELECT {
 2   def apply[DomainA, DomainB, Range](
 3         projection: (DomainA, DomainB) => Range
 4   ): SELECT_CLAUSE_2[DomainA, DomainB, Range]
 5   ...
 6
 7 trait WHERE_CLAUSE_2[DomainA, DomainB, Range]
 8     extends SQL_QUERY[Range]
 9 {
10   def AND (predicateA: DomainA => Boolean
11     ): WHERE_CLAUSE_2[DomainA, DomainB, Range]
12
13   def OR (predicateA: DomainA => Boolean)
14   ...
15 }
16
17 implicit def where2toNext[DomainA, DomainB, Range](
18      whereClause2: WHERE_CLAUSE_2[DomainA, DomainB, Range]
19     ): WHERE_CLAUSE[DomainB, Range]
```

**Figure 4.16:** Excerpt of the type safe declaration of predicates for multiple relations

Note that for this work we implemented clauses for up to three relations. More relations were never required during our case studies. In practice clauses for larger numbers of relations will be implemented and more importantly will be generated, since the types for clauses over different numbers of relations bear a large semblance and do not require manual implementations.

The main technical difference between single and multiple relations arises in the predicates that are passed as parameters to the where clause. Due to type erasure the types DomainA and DomainB (line 7) are not visible after compilation. The method declarations for AND and OR (line 10 and 13) are transformed – during type erasure – to methods with parameters of type Function1 (Scala's internal representation of a function with a single parameter). Hence, one cannot overload the same method, e.g., AND, to receive predicate functions over either domain type, since the overloaded methods are ambiguous after type erasure.

The problem of type erasure can not be alleviated by pure function definitions. Hence, IQL relies on Scala's implicit conversions, to "switch" types to the domain of the next relation. For illustration consider Figure 4.16. The AND and OR conditions in the clause over two relations allow to define conditions on DomainA (lines 10 and 13). The implicit conversion where2toNext (line 17) converts the clause to a where clause with one domain less than the predecessor and whose conditions can be defined on the next domain. Note that the implicit conversion is triggered when the supplied predicate function is not of the correct type, i.e., DomainB instead of DomainA.

In practice the scheme with explicit switching to the next domain means that conditions are defined in a style as discussed in the previous section, e.g., in Figure 4.14.

### Type safety for join conditions

The type safety for join conditions is fairly straightforward. As stated for the example in Figure 4.5, the join relies on a positional match between the left hand function to left hand relation – and the right hand function treated accordingly. The key point for join conditions is that IQL does not enforce type equality of the return types yielded by the functions used as join conditions. Type safety is not strong in this position, since the join is based on a notion of object equality, i.e., objects are transformed by the join condition functions and tested via equals(o:Object) Since, Java (and Scala) allow classes to declare their own notion of equality, the strong type safety would constrain users of the database in an unnecessary way.

Nevertheless the type safety is strong between the parameters passed to the join condition functions and the underlying relations from which these parameters come. Figure 4.17 depicts the types used in a join. IQL relies on implicit conversion of the first join condition function to an object (of type Comparator, which is omitted in the figure) that defined the special === method. The === method in turn takes the second function as a parameter and constructs an object of the type JOIN that represents the entire join condition. The conversion binds the types of the join condition functions (line 1) which are functions from DomainA to RangeA and DomainB to RangeB. A join condition can be passed to the methods AND (line 6) and OR (omitted in the figure). The domain types in the join have to conform to the domain types of the where clause. Note that the JOIN type naturally declares the domains to be contra-variant, i.e., a join allows the functions to be declared on

```
trait JOIN[-DomainA, -DomainB, RangeA, RangeB]

trait WHERE_CLAUSE_2[DomainA, DomainB, Range]
  ...
  def AND[RangeA, RangeB] (
     join: JOIN[DomainA, DomainB, RangeA, RangeB]
  ): WHERE_CLAUSE_2[DomainA, DomainB, Range]
```

**Figure 4.17:** Excerpt of the type safe declaration of conditions

supertypes Hence, a join on `Student` objects can for example declare a condition on a `Person` object, if `Person` is a supertype of `Student`.

### Type safety for join conditions between sub-queries and outer queries

In the type safety for joins between nested sub-queries and the outer query the basic assumption is that the sub-query is constructed individually, i.e., without receiving information from the outer query. For example in Figure 4.6 the sub-query on the `registrations` relation is constructed as an object and passed to the exists expression in the outer query. This style of query construction ensures compositionality and reuse, i.e., sub-queries can be defined once and reused in different contexts. For the sake of type safety this means that a sub-query that contains a join on an (unknown) outer query does not know the relation(s) of the outer query and hence does not know the type(s) of the elements in the outer query. Thus the basic idea for type safety is to bind the unknown types used in a join condition to a type parameter and check the type correctness when the sub-query is passed to an outer query.

To illustrate the type safety Figure 4.18 depicts the clauses and methods used to construct a valid sub-query. A `WHERE_CLAUSE` object can receive various sub-expressions as conditions (line 4), i.e., `EXISTS` over sub-queries, but also complex conditions phrased in parenthesis such as `c1 AND (c2 OR c3)`. An `EXISTS` subexpression is constructed via the corresponding `EXISTS` object (line 12) which in turn can be constructed from a sub-query with an unbound relation (line 15). Note that the term *unbound* is used to signify that the sub-query does declare a join condition over a relation not declared in its from clause. In terms of the Scala type system the type is concretely bound once a sub-query is constructed. A sub-query with an unbound relation is constructed via a where clause condition that receives an `UNBOUND_JOIN` parameter (depicted for `AND` in line 8). The `UNBOUND_JOIN` is a

```
trait WHERE_CLAUSE[Domain, Range]
  ...
  def AND(
     subExpression: SUB_EXPRESSION_1[Domain]
  ): WHERE_CLAUSE[Domain, Range]
  ...
  def AND[Unbound, RangeA, RangeB](
     join: UNBOUND_JOIN[Domain, Unbound, RangeA, RangeB]
  ): UNBOUND_WHERE_1[Domain, Unbound, Range]
  ...

object EXISTS
{
  def apply[Unbound, Range](
     query: UNBOUND_QUERY_1[Unbound, Range]
  ):  SUB_EXPRESSION_1[Unbound]
```

**Figure 4.18:** Excerpt of the type safe declaration of nested sub queries

special form of join condition and is implicitly constructed by a conversion from two functions via the === operator.

## 4.2 Query Optimization

Relational query optimizations perform a transformation of the relational expressions, i.e., the operator trees, to obtain a query that is cheaper in its evaluation. The transformation utilizes algebraic laws, e.g., commutativity and associativity, to find equivalent – and cheaper – relational expressions. Optimizations of relational queries are a long researched topic and are discussed in standard text books (e.g., by Garcia-Molina et al. [GMUW08]). In fact, the availability of well understood optimization techniques was a key decision for using relational algebra as a formalism for incremental maintenance.

In the following we first discuss the application of *traditional optimizations*. Such optimization can be applied to relational queries in general and not only for incremental maintenance. The main point here is to highlight optimizations that impact the amount of data that needs to be materialized during the incremental view maintenance. We then consider the case of *optimizing multiple views* to reuse overlapping parts of their queries, i.e., common subqueries. Then the traditional query *optimization of OO databases* is discussed w.r.t. providing an in-memory representation of objects. Furthermore we discuss how *indexing* is performed in our system, since indices should not require a complete materialization of data. Finally, we discuss an extension of traditional optimizations, that introduces *increment local operators*.

Increment locality introduces a notion of scoping for data that is propagated from the base relations and used during incremental view maintenance. A scope is evident in the sense that the data contains several objects, which are all modified together in a single transaction $T_A$, i.e., they are all propagated together during incremental view maintenance (as defined in Sec. 3.4.3). This scoping of data can be used to optimize operators that materialize auxiliary data, by making the materializations local to the scope of change, i.e., the auxiliary data is transient and must not be permanently maintained.

### 4.2.1 Traditional Database Optimizations

Query optimizations in traditional databases come in two forms. *Rewriting heuristics* are independent of the current state of the database and apply transformations that are in general considered to yield a cheaper query. Examples are: avoiding unnecessary duplicate eliminations ($\delta$) or applying selections ($\sigma$) as early as possible in a query (a.k.a. *pushing selections*). *Cost-based optimizations* require information about the state of a database, e.g., approximate size of relations, and assess the costs of several possible query executions for the given state in order to choose an optimal plan. For example, given two consecutive join operations $A \bowtie B \bowtie C$, the alternatives plans are to apply the first join and then the second $(A \bowtie B) \bowtie C$ or alternate the join order to $(B \bowtie C) \bowtie A$ (which is possible due to commutativity and associativity of the join operator). If $A$ and $B$ are known to be large and $C$ is small, a query optimizer avoids joining $A$ and $B$ first, since the join on two large relations can consume considerable time, and the result of joining $B$ and $C$ will most likely be small(er) relation again.

In the following the optimization for pushing selection is exemplified in more detail. Furthermore, a summary of the impact of traditional optimization techniques on incremental view maintenance and the materialized memory is given. In general, the techniques and algebraic laws used in traditional database optimizations are well known (cf. [GMUW08])). Hence, a comprehensive discussion of all techniques and algebraic laws is omitted here.

**Example query for pushing selections**

One of the most powerful optimization techniques in relational algebra is to push selections over other operators. For example, by pushing selections down over a join the joined relations are smaller and, hence, the join is less expensive.

The laws that allow selections to be pushed over joins are stated below and either allow to push the selection over one argument (4.2.1.1) or over both arguments (4.2.1.2). Informally, we can only push the selection down to a relation that has all the properties mentioned in the condition (Θ). In traditional relational algebra this assumption translates to checking that the relation has all the attributes (i.e., column names) used in the condition. This assumption can also be ported to typed functions as used in our definition of operators, as will be shown shortly.

$$\sigma_\Theta(A \bowtie B) = \sigma_\Theta(A) \bowtie B \tag{4.2.1.1}$$

$$\sigma_\Theta(A \bowtie B) = \sigma_\Theta(A) \bowtie \sigma_\Theta(B) \tag{4.2.1.2}$$

According to the above laws, selections can be pushed in both directions, i.e., up and down in the operator tree. In combination this means that we have an extremely powerful optimization, where we can use the first law (4.2.1.1) to push a selection from one side up, and then use the second law (4.2.1.2) to push the selection down to both sides.

For illustration consider the following query that expresses a join between students and registrations, where a selection filter for students that have a grade average above 3.0 is applied to the result of the join. Note that the result of the join is a tuple of students and registrations and the filter is applied on the first element of the tuple.

$$\sigma_{\lambda e.\, e.first.gradeAverage<3.0}\left(E_{Students}\ {}_{\lambda s.s}\bowtie_{\lambda t.\, t.student}\ E_{Registrations}\right)$$

To illustrate the applications of the laws for pushing selections, Figure 4.19 shows three different operator trees. In Figure 4.19a the selection is performed after the join, i.e., this operator tree corresponds to the above shown query. In Figure 4.19b the selection is pushed down to the left hand operator by law 4.2.1.1. In Figure 4.19c the selection is pushed down to both operators by law 4.2.1.2. The laws allow to move selections down to one operand, i.e., (a) → (b), or to both operands, i.e., (a) → (c). The laws are also applicable in both ways, i.e., we can move selections upwards from one operand over the join (b) → (a) and then push the selection down on the other operand (b) → (c).

$$\begin{tikzcd}
 & \sigma_{\lambda e.\ e.first.gradeAverage<3.0} & \\
 & \lambda s.s \overset{\bowtie}{} \lambda t.\ t.student \arrow[u] & \\
E_{Students} \arrow[ur] & & E_{Registrations} \arrow[ul]
\end{tikzcd}$$

**(a)** Selection on a joined value

$$\begin{tikzcd}
 & \lambda s.s \overset{\bowtie}{} \lambda t.\ t.student & \\
\sigma_{\lambda s.\ s.gradeAverage<3.0} \arrow[ur] & & E_{Registrations} \arrow[ul] \\
E_{Students} \arrow[u] & &
\end{tikzcd}$$

**(b)** Pushed down selection to one operand

$$\begin{tikzcd}
 & \lambda s.s \overset{\bowtie}{} \lambda t.\ t.student & \\
\sigma_{\lambda s.\ s.gradeAverage<3.0} \arrow[ur] & & \sigma_{\lambda r.\ r.student.gradeAverage<3.0} \arrow[ul] \\
E_{Students} \arrow[u] & & E_{Registrations} \arrow[u]
\end{tikzcd}$$

**(c)** Pushed down selection to both operands

**Figure 4.19:** Pushing selections over joins

**Using general functions as conditions**

When treating selection conditions as filter functions, we can push filters to the earliest position in the operator tree where the variables used in the filter function are provided by the relation. For example, we can push down the condition from Figure 4.19a, which deals with student objects, to both sides, because one side is the extent of students and the other side are the registration objects that hold a reference to student objects. Yet, we could not push down a filter involving registrations to the side of $E_{Students}$.

The treatment via functions requires some transformations to move operators in the tree. As can be seen in Figure 4.19, the join conditions used in the different operator trees are not the exact same functions, but the operator trees are semantically equal. For illustration, consider the condition in (a), which can be transformed to the condition in (b) by using the information how the joined value was constructed as shown below.

$$\begin{aligned}
&\lambda e.\ e.first.gradeAverage < 3.0 \\
=\ &\lambda s.\lambda r.\ (s,r).first.gradeAverage < 3.0 \\
=\ &\lambda s.\lambda r.\ s.gradeAverage < 3.0 \\
=\ &\lambda s.\ s.gradeAverage < 3.0
\end{aligned}$$

The parameter $e$ in the first line was constructed by building a tuple of students and registrations $(s, r)$. Hence, we can rewrite the original condition to a function with two parameters and replace all occurrences of $e$ by the tuple $(s, r)$. After that it is a matter of simplification (i.e., partial application) to deduce that $(s, r).first = s$. Finally we have to consider that the parameter $r$ of the function is not used anymore in the body, which allows us to push the selection, since $E_{Students}$ can only provide student objects, i.e., values for the parameter $s$.

The push down to the relation $E_{Registrations}$ (Figure 4.19c) is only correct, since the join condition used entire student objects from $E_{Students}$. The respective transformation takes into account the condition $s = r.student$, hence we can substitute $s$ in the below equations.

$$\begin{aligned} & \lambda e.\ \mathit{e.first.gradeAverage} < 3.0 \\ = & \lambda s.\lambda r.\ \mathit{(s,r).first.gradeAverage} < 3.0 \wedge s = r.student \\ = & \lambda s.\lambda r.\ \mathit{(r.student, r).first.gradeAverage} < 3.0 \\ = & \lambda r.\ \mathit{r.student.gradeAverage} < 3.0 \end{aligned}$$

In general, a different join condition can lead to situations where the selection is not allowed to move down. For example, the condition $s.name = r.student.name$ does not allow us to make a substitution as seen above. Hence, there is no simplification such that the resulting function does not depend on $s$ and thus moving the selection is illegal. Indeed, pushing down the selection to registrations would yield incorrect results, if done as seen in Figure 4.19c. The reason is that we could have registrations for students with a higher grade average that have the same name as some students with a lower grade average. Tuples of such students would not be in the results if the selection were pushed down. Note that pushing down a selection is still legal if the selection condition involves only the joined property, e.g., *name*.

**Impact on incremental view maintenance**

In general, all optimization techniques that reduce the number of tuples used in subsequent operators also reduce the amount of memory that needs to be materialized. For example, pushing down selections is a very powerful optimization, especially in conjunction with joins, since it allows to perform joins over less tuples. For incremental view maintenance this means that the joins – which require materialization of the underlying relations (cf. Sec.3.4.3) – must materialize less data. Hence, there is a huge impact by filtering data as early as possible. Indeed, if no data is ever filtered at all, and the operators for incremental view maintenance require a materialization, then the entire base relations are materialized and there is no gain in memory efficiency. Other optimizations that decrease the number of tuples are the join ordering or pushing duplicate eliminations over joins.

## 4.2.2 Optimization for Multiple Views

Multiple views potentially share some common parts in their defining queries. An optimization can detect these common parts and use a single *shared subquery* instead of repeatedly evaluating the subquery, i.e., after the optimization multiple views share a part of the operator tree. Subquery sharing allows to define views in a very modular way, that is completely independent of each other. At the same time the optimization provides an improved evaluation of these queries.

Optimizations for multiple queries have been pioneered by Sellis [Sel88] and the topic has received much attention – in slightly different form – in the context of data integration and data warehouses. The problem addressed in the latter context is that a conjunctive query $Q$ should be answered using views $V_1, \dots, V_n$. Several algorithms have been proposed to address this problem ([KA11] proposes an algorithm and gives a good overview of existing works). In essence the algorithms must *rewrite* the original query $Q$ into a union of conjunctive queries that only use the views.

In the following we exemplify the optimization of sharing subqueries in multiple views and then discuss its implications in view definitions and incremental view maintenance.

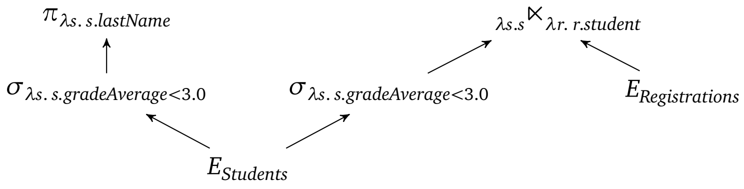
**Example of subquery sharing**

For illustration, consider the following two views that were defined in Section 3.4.2 as examples of view maintenance.

$$
\begin{aligned}
view_1 &= \pi_{\lambda s.\, s.lastName}(\sigma_{\lambda s.\, s.gradeAverage<3.0}(E_{Students}))\\
view_2 &= \sigma_{\lambda s.\, s.gradeAverage<3.0}(E_{Students}) \;{}_{\lambda s.s}\ltimes_{\lambda r.\, r.student}\; E_{Registrations}
\end{aligned}
$$

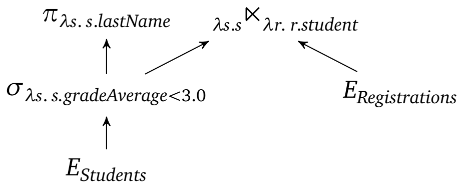
The queries defining $view_1$ and $view_2$ share a common part, namely the selection over students ($\sigma_{\lambda s.\, s.gradeAverage<3.0}$) which has exactly the same filter function. Hence, instead of performing the filter evaluation separately for each view, it can be shared and thus only be evaluated once for both views.

The optimization rewrites the operator tree and substitutes shared subqueries by a single instance of the subquery. For the above two views the operator tree is rewritten as depicted in Figure 4.20. The original operator tree, where each view is evaluated in isolation is shown in Figure 4.20a. The operator tree where the subquery is shared is depicted in Figure 4.20b.



$\pi_{\lambda s.\ s.lastName}$
$\sigma_{\lambda s.\ s.gradeAverage<3.0}$
$\sigma_{\lambda s.\ s.gradeAverage<3.0}$
$_{\lambda s.s}\ltimes_{\lambda r.\ r.student}$
$E_{Registrations}$
$E_{Students}$

**(a)** Base operator tree (without shared subqueries)



$\pi_{\lambda s.\ s.lastName}$
$_{\lambda s.s}\ltimes_{\lambda r.\ r.student}$
$\sigma_{\lambda s.\ s.gradeAverage<3.0}$
$E_{Registrations}$
$E_{Students}$

**(b)** Shared subquery in operator tree

**Figure 4.20:** Comparison of base operator tree to a tree with a shared subquery

**Applicability of subquery sharing**

In general, the optimization requires a notion of equality between the operator trees (i.e., the subqueries) as well as the functions used in the various operators. In the example shown in Figure 4.20 the optimization was easily applicable, since the functions are syntactically equal and only a single operator is reused. In general, less obvious rewritings can also yield shared subqueries. However, the effort to find such rewritings is considerably greater. Especially, equality between the functions is hard to decide. Levy et al. [LMS95] have shown that finding a rewriting of a query to uses the results of existing views is NP-complete.

One also has to consider that rewriting – for view reuse – is orthogonal to finding optimized queries for single views. In other words, we can find a rewriting that allows views to reuse subqueries, yet each view in isolation would have a different optimized operator tree. Hence, such a rewriting system must carefully weigh global optimum versus local optimum.

As a final remark to applicability, note that the sharing of subqueries in IQL must currently be enforced manually (by the query writer), i.e., by defining views in terms of the results of another view. For example two queries can explicitly select different elements from *view*$_2$ and thus explicitly share the subquery. Indeed declaring complex views (queries) as reusable abstractions is a natural pattern when using the views in a programming language context. Hence, the ratio between subquery sharing and reusing common abstractions is not quite clear, i.e., it is not clear how many views are expressed in terms of common abstractions vs. how many views implicitly share the same subquery. Thus, we deem the incorporation of subquery sharing an interesting extension to the query compiler, but have limited ourselves to identifying potential uses for subquery sharing and manually rewriting them to an optimized version. The evaluation in Chapter 6 includes a discussion on how many potential sites for subquery sharing were found.

**Impact on incremental view maintenance**

In general, the incremental view maintenance benefits most if complex queries are reused, which also materialize auxiliary data. In the example in Figure 4.20 a very simple filtering is reused, which does not require materialization of data. However, if the entire result of $view_2$ were reused by other queries the savings would be great, since the existential quantification requires to materialize data.

Nevertheless, the shared subquery in Figure 4.20 has a benefit for query runtime and, hence, the runtime of the incremental view maintenance. The expected savings in terms of runtime are, however, highly dependent on the filter condition, i.e., is it cheap or expensive to evaluate, and the amount of data filtered by the operator.

## 4.2.3 Comparison to OO Database Optimizations

Traditional OO databases typically have an optimization step that deals with *path expressions* (cf. Sec. 2.2.4, p. 23) and must be performed prior to algebraic query optimization (cf. [CD92]). The problem with path expressions is that referenced objects must be retrieved from a different extent that is physically stored on a hard disk. Hence, path expressions are often referred to as *implicit joins*. The optimization of path expressions allows an OO database to retrieve only those referenced objects that are absolutely required to evaluate a query. An object can contain multiple other references that do not contribute to the query results and, hence, are not retrieved from persistent storage. Path expressions can be optimized by rewriting them into equivalent algebraic expressions, i.e., joins. In contrast, the in-memory representation of objects used in this thesis does not require such an optimization, since objects can be efficiently retrieved via references to memory locations.

**Path expressions as implicit joins**

For illustration consider the following query that selects the last names of students referenced in a registration.

$$\pi_{\lambda s.\, s.lastName}\left(\pi_{\lambda r.\, r.student}\left(E_{Registrations}\right)\right)$$

In an OO database the referenced student object must be retrieved from a respective extent ($E_{Students}$) that resides in persistent storage. The corresponding query for retrieving the object is a join as depicted in Figure 4.21. The join operator is denoted as *OOJoin* to distinguish it from the join we have used so far.

$$\begin{tikzcd}
 & \pi_{\lambda s.\, s.lastName} & \\
 & {}_{\lambda s.\, s.self}OOJoin_{\lambda r.\, r.student} \arrow[u] & \\
E_{Students} \arrow[ur] & & E_{Registrations} \arrow[ul]
\end{tikzcd}$$

**Figure 4.21:** Algebraic equivalent of an OO path expression

The resulting operator tree features a new join and – in a more complex query – this join is again subject to optimizations via query rewriting. Note that the implicit joins are only required for object references. OO databases also contain atomic values, e.g., integers, which are stored as part of an object and do not require further query rewriting.

### Impact on incremental view maintenance

The use of direct in-memory object references allows queries to be executed faster since no implicit join must be performed. Hence, incremental view maintenance is also performed faster. In terms of memory requirements for a single reference, e.g., the student in a registration, an implicit join is naturally more expensive. To be efficient the join requires indices over $E_{Students}$ and $E_{Registrations}$. In contrast, the pointer reference is a small memory cell in a registration object, i.e., typically the address is stored using 4-bytes in 32-bit architectures and 8-byte in 64 bit architectures. Yet, we need to consider that all other referenced objects are also retained in memory as long as the object storing the pointer remains in memory. For example, if the registration object features several more object references, that are never used in a query, they are retained in memory.

The current implemented system requires users to declare data as objects, which may contain any number of references. However, database designers should be aware that pointers can always be replaced by explicit joins. Hence, there is a user-driven trade-off between fast execution and – possible – materialization of irrelevant data through object references. The trade-offs are further discussed at the end of this chapter (cf. Sec. 4.3.5), together with a possible extension for moving the decision of materialization to the query optimizer.

### 4.2.4 Indexing

Joins are efficient only if the underlying relations are indexed. Otherwise a join requires a full traversal of both relations to identify matching objects. The indices used in our system are all based on hash tables. We assume that the reader has some familiarity with hash tables and omit a detailed discussion.

The important thing to note is that creating an index materializes all objects of the underlying relations and requires additional memory for the index structure, i.e., the hash table. Hence, indexing is a very memory intensive operation. Since the overall design goal for our database is to require as little memory as possible, indexing is performed as late as possible in the operator tree.

Consider for example the following query that joins students and registrations by the first and last name of the student. Note that multiple equality conditions are translated to a function that creates matching tuples for each value that must be equal. For example for the given query two indices are created. The index on the left hand side stores student objects that can be efficiently retrieved by asking for the tuple *(s.firstName,s.lastName)*, where *s* is a student object. The index on the right hand side does the same for registration objects.

$$\begin{tikzcd}
& {}_{\lambda s.(s.firstName,s.lastName)}\bowtie_{\lambda r.\ (r.student.firstName,\ r.student.lastName)} & \\
\sigma_{\lambda s.\ s.gradeAverage<3.0} \arrow[ur] & & E_{Registrations} \arrow[ul] \\
E_{Students} \arrow[u] & &
\end{tikzcd}$$

The indexing is performed right before the join operator, i.e., as late as possible. Thus, the left-hand side index only contains students that passed the filter condition. In general, such indices are also referred to as *filtered indices* in the literature.

Note that filtered indices are traditionally specified by the database designer in the database schema. However, the filtered indices we wish to use are naturally tied directly to the queries. Hence, we allow queries to be compiled by default with respective filtered indices for all contained joins.

#### Impact on incremental view maintenance

As already discussed performing indexing is an operation that consumes much additional memory. However, not performing indexing also has severe consequences for the runtime of incremental maintenance. Consider for example that in the above shown operator tree there is no index for registrations and a single new student object is added that also passes the filter. To determine whether there is a registration the whole multiset of registrations must be traversed. This traversal is repeated many times, if objects are added as single events. If multiple students are added together the traversal can be done once for the entire change set, but still each registration object must be compared to all added students.

### 4.2.5 Optimization Based on Increment Locality

The key idea behind the optimization is to perform a substitution of an operator that requires materialized data by an analogous operator that uses only a transient materialization. Such an operator can compute the same result as the analogous materialized version, if all data used during the computation is in the scope of one change propagation transaction.

Before we exemplify such a translation, let us shortly discuss in which scenarios this optimization is applicable and why traditional databases do not offer such an optimization. In general, scenarios for this optimization have three premises. First, there must be a large set of data, which is not stored in the database, but rather persisted externally in a large set of files, which can be brought into a basic structured form, e.g., via parsing, suitable for formulating queries that are of interest to end-users. Second, the basic structured form is not permanently required by the end-user or the data changes too frequently, which makes permanent storage unattractive. In short, the premise is that there is a large set of data that contributes to results but has no inherent value in being permanently stored in the database. Third and finally, there must be results that can be obtained using a single file, i.e., the scope of the optimization; otherwise the optimization is not applicable.

The static analyses discussed in the following chapters satisfy these three premises. First, the data is stored externally, i.e., as source code files or compiled code files. Second, the code files contain a lot of information, for example in the form of single instructions which are not directly of interest to end-user, who rather wishes to receive information on whether the given instructions violate certain conditions, which would be formulated as a query. Third, there are conditions that can be ascertained by analyzing a single code file, e.g., a single method can be invalid if there is an instruction that dereferences a variable that contains the value **`null`** instead of an object.

Traditional databases assume that all data that is stored in the database has some inherent value to the end-user and can be retrieved anytime or correlated in new ways via ad-hoc queries. In addition, the assumption that there is some external storage whose content is edited outside of a database is inherently tied to a single-user scenario, whereas traditional databases seek to provide concurrent access. Hence, such databases do not offer an optimization based on increment locality.

### Example for increment locality

To exemplify the optimization let us consider an extension to the database for administering a university that was used throughout the last chapters. The extension introduces course descriptions for the curriculum of the university. Note that we use the example for the sake of staying in this simple domain. Administrative databases for universities are rather traditional in the sense that all data should be persisted as relations in the database. Nevertheless, let us consider that our university stores each course description in a single external text file. The files adhere to a certain structure to allow a parser to identify values, such as the title of the course, one or more lecturers, and the material recommended for the lecture, e.g., books. Let us further assume that the description files are stored in some form of document management system (DMS). When a course description is changed the new description file is uploaded to the DMS.

Note that the DMS has no knowledge about what changed inside the file. Hence, it parses the complete predecessor version of the file, as well as the new version. Instead of running some application specific code to determine changes on the parsed result, the parser simply fires events to the base relations in the database that signify tokens found in the old file as deleted and tokens in the new file as added. Hence, the identification of relevant changes and modified results is performed by the database in accordance to the currently defined views over the course descriptions.

For this example we define three basic types of data (cf. Figure 4.22): course descriptions with a title of the course (line 2), lecturers found in a course description (line 4) and books recommended in the course description (line 8). These three types of data are logically available as extents (lines 13-15). To perform an optimization based on increment locality, the scope of the increment must be provided by the database designer via annotations on classes. In the example below, the scope of one increment is a single course description, hence, the class `CourseDescription` is annotated as `@LocalIncrement`.

To exemplify the local increment optimization, consider a query for all lecturers that recommend a book for their course that was written by themselves (cf. Figure 4.23). The query is a straightforward join over the two relations `lecturers` and `books` and will select tuples of lecturers and books, i.e., the result shows which lecturer recommend which of her own books. The join conditions state that the book must be recommended in a course given by the lecturer (line 2) and that the lecturers name must be the same as the name of the book's author (line 3). Note

```
@LocalIncrement
case class CourseDescription(title: String)

case class Lecturer(
  course: CourseDescription,
  name: String)

case class RecommendedBook(
  course: CourseDescription,
  authorName: String,
  title: String)

val descriptions: Extent[CourseDescription]
val lecturers: Extent[Lecturer]
val books: Extent[RecommendedBook]
```

**Figure 4.22:** Data definition for course descriptions

that the functions `course`, `name` and `authorName` select the respective properties of the underlying objects; their definition is omitted for the sake of brevity.

```
SELECT (*) FROM (lecturers, books) WHERE
  (course === course)   AND
  (name === authorName)
```

**Figure 4.23:** IQL query for lecturers together with all books they wrote and use in their own course

An ordinary join has no knowledge about increment locality and, hence, must materialize the data contained in lecturers and books. Yet, it stands to reason that we can find a possible combination in a single course description. In other words, a pair of a lecturer and book that both satisfy the two conditions can be found without considering the data of other course descriptions. Hence, we can substitute the join ($\bowtie$) by an operator with the same relational semantics ($\bowtie^L$) that deletes ("forgets") the internally used auxiliary data once the end of a change propagation transaction is reached. The underlying property that allows us to perform the substitution is that the data declared as the scope of a local increment, i.e., the `CourseDescription`, is used as one of the join conditions (line 2). If the query is phrased without the quality of course descriptions the substitution is not valid. Note that the query would translate to: “retrieve all lecturers together with all books they wrote, which are used in any course”. Note furthermore the use of the phrase “any course”, which implies that all course descriptions must be searched by the query and not only those in a single course description.

### Applicability of increment locality

As already shown in the above example a substitution by a local increment version is not always possible. Table 4.1 summarizes the conditions that must be satisfied to replace an operator. Note that a substitution only makes sense for the operators that require auxiliary materialized data (cf. Sec. 3.4.3). There are two side conditions that must be satisfied in order to have a correct substitution. The first side condition states that the substitution is only valid if the instances of the type annotated with `@LocalIncrement` either stem directly from base relations or the instance may have undergone only object preserving transformations. In other words we are not allowed to make the substitution if the instance were instantiated by some transformation, i.e., by a projection. The simple reason is that one can always return an object `new CourseDescription(...)` from a projection, even for objects from base extents that are never changed during a local increment. In most cases this condition is easy to verify, e.g., in Figure 4.23 both inputs of the join are base relations.

The second side condition states that the instances of the local increment type must be unique. Increment locality only guarantees that if a result exists it will be deduced from data in one increment. Yet, there may be more results found by correlating data between increments. Consider for example, that we have two course descriptions with the same title, i.e., they are not unique. Both courses are given by the same lecturer, but only in one course description does she recommend her book. The ordinary join would find two objects from `lecturers` that can join with one object from the relation `books`, i.e., each of these joins is a result. The increment local join only finds the one result that is contained in the course description where the book is recommended. Afterwards, the operator “forgets” the fact that the book was recommended and, hence, produces only one result.

### Impact on incremental view maintenance

Increment locality can have a huge impact on the data required to be materialized, depending on the used operators and the size of the data. For example, in the join over lecturers and books in course descriptions, we can have thousands of course descriptions and never require any data to be materialized.

| Substituted Operator | Condition |
|---|---|
| $\bowtie \Rightarrow \bowtie^L$ | At least one join condition is an equality check that is performed on a type annotated with `@LocalIncrement`. In terms of the functions shown in the operator tree, both join functions either return objects of the `@LocalIncrement` type or tuples where the same component of the tuples in both join functions are objects of the `@LocalIncrement` type. |
| $\gamma \Rightarrow \gamma^L$ | The grouping function returns objects of a type annotated with `@LocalIncrement` or tuples where one component are objects of the `@LocalIncrement` type. |
| $TC \Rightarrow TC^L$ | The functions that determine start and end of an edge both return objects of a type annotated with `@LocalIncrement` or the functions yield tuples where the same component of the tuples in both functions are objects of the `@LocalIncrement` type. |
| $op \Rightarrow op^L$ | The underlying operators that are the inputs for *op* are all substituted by a local version. Note that to check this property the self-maintainable operators are also logically marked as transaction local. For example, a selection is performed after an increment local join is marked as increment local and, hence, further operators after the selection can be substituted. However, the self-maintainable operators are not replaced with different versions in the compiled query. |

**Table 4.1:** Conditions for substituting an operator by a local version

It has to be noted, that the increment locality can in turn have a negative impact on runtime. The reason is that the increment local scope must always be changed in its entirety, even if no change was made to the underlying data. For example, if a course description has unchanged lecturers and books the result of the join must be recomputed nevertheless. The concrete impact on runtime performance depends on the input data and the size of the increment local scope chosen. The trade-off that is made here is quite tricky and has two important points: First, a smaller scope for local increments (or none at all) allows to have more fine-grained maintenance events. For example, when using course descriptions without a scope and a single lecturer is changed, only the single event for updating the lecturer must be processed. In short we can say that bigger scopes require more re-computation (i.e., more events are processed) and smaller scopes require less re-computation. As a second point, we also have to see that the fine-grained maintenance events do not come for free. They require the knowledge of what exactly changed from the last version. For example in the course description scenario, there is a file that needs to be parsed and, hence, the change must be computed between two parsed outputs. To compute the change all data in the old and new versions must be correlated, e.g., to find out that only single lecturer changed we must compare all the books in the old version to all books in the new version. Such a comparison can also take some time and, hence, the re-computation w.r.t. the entire scope can be as efficient at least for some views.

## 4.3 Discussion

In this chapter we have presented the declarative query language IQL. IQL queries are compiled to the incrementally-maintained operators defined in the previous chapter and, hence, provide the same expressivity. In other words, using IQL we can define basic SPJ queries, set theoretic queries, negations (via set difference), aggregations, existential queries, transitive closures and recursions. The following points deserve a closer discussion.

### 4.3.1 General Query Optimization

Writing a fully-fledged SQL query optimizer was beyond the scope of this thesis. We were rather interested in the semantics for incremental maintenance of relational operators and their minimal memory requirements. Nevertheless we considered optimizations in Sec. 4.2; albeit not all are currently performed automatically.

In IQL selections are performed as early as possible in a single query, i.e., also before joins. There is no need to push down selections in an optimizer, since IQL queries as shown in Sec. 4.1.1 already produce an operator tree, where the selections stand out as functions on the left or right hand side relation of a join. Nevertheless, a fully automated optimizer can cover pushing selections up and down or over multiple queries, i.e., sub-selects. Likewise there is no automated re-ordering of joins. Indexing is performed automatically during query compilation. The substitution for increment locality is currently performed by instructing the query compiler to produce an increment local version of a query.

The issue of a fully-fledged query compiler is (mostly) a technical matter. The optimizer requires a first class reification of the Scala functions used in the various operators. We have explored the use of such a reification for optimizing non-incremental queries using Scala collections and for-comprehensions [GOE$^+$13]. The concepts are well portable to a fully-fledged SQL query optimizer.

### 4.3.2 Using an Embedded Language Instead of Plain SQL

The embedding of IQL into Scala has two main advantages, namely (i) type safety at compile-time for the written queries and (ii) compositionality with the host language, i.e., the code is written in Scala and existing abstractions can be integrated from Scala into IQL or vice versa. While IQL was inspired by SQL and bears a strong resemblance, it is nevertheless not identical to the SQL standard. Thus, a slight burden is placed on users of IQL to learn how different queries can be transported from a pure SQL syntax to IQL.

An alternative design would be to support the SQL standard syntax, e.g., as plain character string in Scala. The SQL standard features several extensions, e.g., for recursive queries, such that a plain SQL syntax can express all IQL queries. Such a design looses static type safety, yet could enhance the initial acceptance by users familiar with SQL. Nevertheless, we did not explore such a design. The focus of this thesis was mainly on the semantics of incremental maintenance and the application of incremental maintenance to the domain of static analyses. Indeed the concepts can be applied regardless of the concrete query language and as such can be seen as parameterized by different query languages. However, it is important to note that static type safety is very advantageous for correctness and has additional benefits when using an IDE, e.g., query writers can use content assistant proposals that are derived from the static types.

### 4.3.3 Using the Host Language vs. Using the Query Language

The presented query language is very expressive and features a large variety of incrementally maintained operators, including the definition of recursive queries as found in logic languages. Yet, the query language is embedded in a general-purpose host language and – due to the very general treatment where operators work on arbitrary functions – the host language can be used instead of the query language in many places. Naturally, we can ask ourselves, which alternative should be used, the host language or the query language?

### Using the host language

Consider a query that uses the function `contains`[5] defined in Figure 4.24. The function performs a recursive traversal over a list and can be used as part of a selection (line 14) or any other operator that requires a function, e.g., a projection.

```
1 def contains[T](list: List[T], element: T) = {
2   if(list == Nil){
3     false
4   }
5   else if(list.head == element){
6     true
7   }
8   else {
9     contains(list.rest, element)
10  }
11 }
12
13 SELECT (*) FROM students WHERE
14  (s:Student) => contains(s.grades, 1)
```

**Figure 4.24:** Using a function of the host language

The two important issues to note here are that (i) to be able to use a function from the host language, the input to the function must be locally (i.e., completely) available from the underlying object in the database, e.g., the list of grades in the student object, and that (ii) the incremental maintenance treats the function as a black box and is oblivious to the complexity of the function, in this case a recursive traversal. The second point implies that the function is recursively evaluated during incremental maintenance, e.g., upon insertion of an element the function `contains` performs the entire recursion once and upon deletion the recursion is completely recomputed again. Intuitively this is a desirable behavior, since incremental maintenance of recursive queries always entails materialization

[5] The function `contains` is defined here for illustration purposes. The function is of course defined in Scala's collection API and can furthermore use a looped iteration instead of recursion.

of data. Thus, for locally available data and recursive functions that are fast to recompute, query writers retain the benefit of low memory consumption.

**Using the query language**

In the last paragraph we used a recursive function to test containment in a list. However, the use of a recursive function does not imply that the equivalent is always a recursive query. For illustration consider Figure 4.25, where an equivalent query is defined.

```
SELECT (*) FROM students WHERE EXISTS (
  SELECT (*) FROM studentGrades WHERE
    (student === gradedStudent) AND
    (grade == 1)
)
```

**Figure 4.25:** Declarative formulation of Figure 4.24

The query uses the unnested relation `studentGrades` (defined as an IQL example in Figure 4.7) to find students that have at least one grade equal to 1. The query is non-recursive, but where did the recursion go? The answer is that the unnesting to `studentGrades` performs a complete traversal over all grades and makes them available as a flat relation.

There are two important issues tied to this question that also raise further questions. First is the issue of performance. Intuitively the host-language version performs better, since (i) the recursion stops once the element is reached, whereas the unnesting on student grades traverses the entire list of grades and (ii) the existential quantification requires additional auxiliary data for incremental maintenance, i.e., a join and duplicate elimination (cf. Sec. 3.3.5)[6]. Second is the issue of declarativity, i.e., the query does not assume how the data is computed but merely states what data is computed. In contrast the function in the host language defines one fixed computation of how the containment of a value is tested in a list. The important point is that, for the declarative query, the performance issue must be seen in new light. The way "how" results are computed is always subject to optimizations as shown in Section 4.2. Furthermore, there might be alternative representation for the data. For example, the `studentGrades` can be a flat relation in the first place, and not a derived unnesting from a student's grades. In this representation incremental maintenance is much faster in the query language than in the host

[6] Note that the query can be simplified, but still there is a need for auxiliary data

language. The query language compiles to operators that test each single grade in isolation when it is inserted and thus no re-traversals over grade lists are performed.

Declarative queries easily allow optimizations or changing the data to alternative representations. Hence, there are two further questions: First, if we have a function in the host language, can we find an equivalent declarative computation where all optimizations are still applicable. Second, if we have a given data representation, can we find an equivalent representation that is better suited to the incremental computations that are performed, e.g., using an flat representation of student grades instead of the nested data structure. Both questions must be seen in a larger context and will be discussed in the next sub sections. Finding equivalent computations effectively means we use that host language as a query front-end instead of giving declarative queries in IQL. Finding equivalent representations means that we provide an object-oriented definition of the data structures and can find suitable relational data structures.

### 4.3.4 Using the Host Language as a Query Front-End

Currently queries are formulated in IQL, which means using the provided primitives for correlating data, such as joins or existential quantification. The advantage of using the primitives of IQL is that they are declarative, which means they do not assume an incremental or non-incremental computation model, but can easily be translated to both.

The biggest issue that precludes us from simply using the host language is that it inherently defines computations via non-incremental functions. Hence, to use the host language requires an automated program transformation that translates non-incremental functions into incremental functions. For illustration consider, the `gradeAverage` declared for students in our database for university administration. The `gradeAverage` is a function that traverses the entire list of grades and produces the average. The traversal is shown below for illustration; note the original definition in Figure 3.2 used higher-order functions that perform the same traversal.

The above function must be translated into an incremental function that can deal with additions/deletions (and possibly updates) to the input data, i.e., changes to the collection `grades`. The incremental function must be automatically derived, which entails finding change propagation expressions similar to those presented in Sec. 3.4.3. In the above example, the function deals with a collection as input, hence the incremental version must at least have change propagation expressions

```
def gradeAverage: Float = {
  var sum = 0
  for(grade <- grades) {
    sum = sum + grade
  }
  return sum / grades.size
}
```

**Figure 4.26:** A non-incremental computation of the grade average

for adding elements to the collection (4.3.4.1) and deleting elements from the collection (4.3.4.2).

$$f(A \uplus \Delta_A^+) = f(A) \oplus \Delta_f^+ \tag{4.3.4.1}$$

$$f(A - \Delta_A^-) = f(A) \ominus \Delta_f^- \tag{4.3.4.2}$$

The target to which we want to translate must be an equivalent to the incremental version of the aggregation function `AVG`. Note that the incremental function is provided as a primitive by the database; hence the incrementalization features some optimizations. The change expression for the average is exemplified below:

```
 1 class AVG {
 2   var avg: Float
 3   var size: Int
 4
 5   def add(v:Int) = {
 6     size = size + 1
 7     avg = avg + (v - avg) / size
 8   }
 9   ...
10 }
```

The incremental version stores the results of `avg` and `size` of previous computations (lines 2 and 3). Furthermore the function `add` (line 5) declares the incremental update for adding elements (for simplification the example shows the addition of a single element). Respective modifications to `size` and `avg` are performed in this function (lines 6 and 7). As can be seen the average actually requires two results, i.e., `avg` and `size`, which are maintained in a single function as an optimization. Furthermore, the computation of the new `avg` is an optimized expression that does not have to compute large sums. A naïve or automatically deduced approach might not find this expression, but simply deduce an incremental change expression for the `sum` and `size` used in Figure 4.26.

Finding equivalent incrementalized functions automatically can be done, as shown for example in the works by Liu et al. [LT94, LS03, LSLR05] under the term *dynamic programming*. Yet, the automatic derivation is not easy since all applied transformations must be semantics preserving. Furthermore, these incrementalized functions perform computations over a single data structure. Hence, it is not immediately clear how this method would translate functions to queries that correlate several structures in the form of multisets, e.g., via joins or existential queries. Indeed it has been noted as an open problem by Liu to establish an exact connection between the dynamic programming technique and algebraic incrementalization as formulated by Paige et al. [Pai82] (cf. 2.3.2).

### 4.3.5 Object-Oriented vs. Relational Data Structures

In general, there is a dichotomy between object-oriented and relational design of data structures. Object references, i.e., pointers in memory, can be expressed via joins and vice versa.

The alternative designs, i.e., an OO design with references and a relational design that uses joins, can be considered from three alternative perspectives. First, from the *query perspective* the OO design allows more concise queries, since object denotations (i.e., path expressions) are syntactically much shorter than joins (cf. discussion in Sec. 2.2.4). Second, from the *runtime perspective* direct in-memory object references allow faster object retrieval. Note that this is true only if the objects are in-memory; a persistent database must perform the joins "under the hood" (cf. Sec. 4.2.3). However, this point is tied to the questions of expressing computations on the data in an object as non-incremantal functions, e.g., the `gradeAverage` shown in Figure 4.26. In the case of such non-incremental definitions the relational design has the immediate advantage that queries are declared in IQL and, hence, are automatically incrementalized. Third, from the *memory perspective* there are intricate trade-offs between both designs. On the one hand, a single reference is more memory efficient than the memory materialized in a join. Note however, that we have defined new optimizations, hence the join might be performed in a local increment and thus not materialize memory at all. On the other hand, an object reference keeps all referenced objects alive (i.e., materialized in memory), whereas the relational design will materialize only objects required for view maintenance.

The two main questions to discuss here are (i) when should we favor the OO or the relational design, i.e., under which circumstances is there an advantage for either design, and (ii) given the goal of using the database for a clearly specified set of views, can an automated optimization be performed to transform a given design to the alternative, if it is expected to perform better.

**Which design should be favored?**
As discussed above the relational design features increased potential for incrementalization and has the added benefit of only retaining in memory the data that is needed to answer the defined views. Thus, the relational design seems like a clear winner. However, the second point is quickly diminished if the entire data is used in the queries. In this case the following considerations should be made.

*Use an OO representation if the expected dataset is small* and the queried properties can be re-computed fast. For example, if each student only has a small number of grades, the computation is comparatively fast. In this case, incremental maintenance of the computed value loses some of its advantages, since the machinery for incremental maintenance also requires time, i.e., the change propagation of values and the computations inside the operators (e.g., the aggregation in `gradeAverage`) are additional overhead. Note that the OO representation typically allows to formulate the queries in a self-maintainable way. Thus, the data is only materialized inside the OO representation, but no additional requirements arise in the relational operators. For example, the selection of students with particular grade averages (shown below) is self-maintainable.

```
SELECT (*) FROM students WHERE (_.gradeAverage < 3.0)
```

*Use a relational representation if the expected dataset is huge* and let the query compiler optimize the materialization. This is naturally a tricky assessment, since the possible optimizations are highly dependent on the defined views. In the worst case a complete materialization can not be avoided. However, the important thing to note here is that typically optimizations are possible. In Sec. 4.2 we presented traditional database optimizations as well as additional non-standard optimizations that can reduce memory consumption where traditional optimizations fail. The important thing to note here is that the OO representation is not suitable for these optimizations, since the data is kept alive via object references in any case.

*Use a relational representation if the (incremental) computation is inherently non-modular*, i.e., the modification of a value yields changes pertaining to other (unreachable) values. This is typically the case when using operators that correlate data to form results, e.g., joins and other binary operators, or the transitive closure. For example, the transitive closure over all master course prerequisites (cf. Figure 4.9) performed an explicit unnesting to obtain a relational representation that features each prerequisite as a tuple with the master course and the prerequisite course as components. The OO representation of having a list of prerequisites stored inside a master course is not suitable for an efficient incremental computation of the transitive closure. The reason is that when we add a new prerequisite course, the transitive closure of other master courses can change. For illustration consider Figure 4.27 where two master courses are created and their prerequisites changed – for ease of reasoning using state mutation. The first modification (line 4) can easily be incorporated into an incremental result, i.e., by adding an edge $(mc2, mc1)$ to the transitive closure. However, the second modification (line 5) is non-modular. The object of the modification is *mc1* and we must add $(mc2, c)$ to the result, yet the object *mc1* has no knowledge about the existence of *mc2*. Hence, the incremental change can not be computed using only the list of prerequisites, but requires an external traversal over all master courses to find other master courses that have *mc1* as a prerequisite. Note that in the case of the transitive closure the entire dataset is materialized anyway. However, keeping the references inside each object that participates in the transitive closure is duplicate work that can be avoided.

```
1 val mc1, mc2: MasterCourse
2 val c: Course
3
4 mc2.prerequisites.add(mc1)
5 mc1.prerequisites.add(c)
```

**Figure 4.27:** Modifications that require non-modular computation (in the transitive closure)

**Automated transformation between OO and relational representations**

In short such transformation can be performed, but not without a highly complex program transformation. In essence this problem is akin to using functions in the host language as a front end to query formulation. A complete discussion of this transformation is beyond the scope of this thesis – whose focus is the incrementalization of static analyses by expressing them in the provided EDSL. Nevertheless, we shortly outline the general idea in the following.

The issue for such a transformation is that the OO representation uses localized data, e.g., one list of grades per student object. The relational representation uses global data, e.g., a list of all grades for all students. Conceptually, these two representations can be converted into one another by the nesting and unnesting operators. For example, assuming a globally defined relation for grades `database.grades` and a function `gradeList` for selecting the list of grades from a nesting operation, a respective query for the localized version can be expressed as follows.

```
class Student(...){
  def grades =
    SELECT (gradeList) FROM
      NEST(database.grades, _.student) WHERE
        (_.student == this)
}
```

It is important to note, that such a conversion should be performed only conceptually. That is, we do not want to instantiate and maintain the query presented above, since this would entail maintaining one query per student object. Instead, we want to use `grades` as a property of student objects, as if the list of grades was an instantiated set of objects. The actual query that expresses how the list of grades for each student is constructed can be optimized away.

## 4.4 Related Work

This section compares IQL with other language-integrated query systems. We discussed these systems already in Sec. 3.6 w.r.t. incrementalization; here the focus is on the syntactic differences. In addition we discuss related work on optimizing incremental view maintenance.

**Comparison to language-integrated query systems**

Existing language-integrated query systems, e.g., LINQ [MAB08], Ferry [GMRS09], and ScalaQL [GIS10] have deep roots in functional programming and are influenced by earlier works on functional query languages, e.g., HaskellDB [LM99] or Kleisli [Won00].

In general, the language integrations are based on (monad) comprehensions (cf. Gray et al. [GKKP10]), which can be understood as succinct notation for collection operations. The simplicity of the notation lies in the fact that many SQL-operators, such as selections and projections have immediate translations to comprehension operations, such as `filter` and `map`. The usage of filter or mapping functions is already made explicit in the comprehension, yet the comprehension does not presume a particular evaluation. Thus comprehensions – like IQL – are declarative and favor optimizations. For example, the Scala compiler transforms comprehensions into function calls to the default collection implementations for map, filter etc., but other transformations (and optimizations) of the comprehensions are possible.

While the comprehensions are very succinct for some operators, others have no succinct one-to-one translation, e.g., groupings and aggregations or existential quantifications, i.e., semi-joins. Thus the comprehension expressions for these operators are more verbose. Hence, these operators benefit from further high-level syntax, e.g., `GROUP_BY` or `EXISTS`. This approach is taken by IQL, and to a lesser degree also by LINQ. For some operators an SQL-style query syntax is provided by LINQ, while others are used as collection operations. For example, the set union or the minimum element (aggregation) are declared as function in the type of collections, e.g., `col1.Union(col2)` or `col1.Min(stud => student.grade)`. In IQL all operations are decoupled from the actual collections (relations), but the general idea of providing a high-level syntax is the same. IQL goes further than LINQ, since we provide advanced operators such as semi-joins, transitive closure and recursions.

The IQL syntax is currently mapped to relational algebra operators. However, monoid comprehensions are more general and can also be used as a target to represent (and optimize) the queries. In particular, monoid comprehensions have been cited as a simple and powerful formalism for unnesting of sub-queries [GKKP10]. Nevertheless, it is not immediately obvious which traditional optimizations can be performed in the monoid comprehensions and which require a representation of the relational algebra operators. The exploration of this connection is interesting as future work.

### Optimizing incremental view maintenance

Optimizations are concerned with two primary scenarios. (i) optimization of the incremental maintenance expressions to increase runtime performance and (ii) optimization of the amount of materialized data.

The work on optimizing incremental view maintenance expressions was pioneered by Griffin et al. [GL95] for a practical relational algebra with multiset semantics. Kawaguchi et al. [KLMR98] present an optimization for nested (collection) data structures using “nested descriptor indices”, which are similar to traditional indices used in relational joins. In the work by Kawaguchi the indices are persisted data, i.e., pointer structures into persisted tables and do incur materialization costs, however, these are not explicitly discussed. We did not yet consider performance issues during updates to nested collections. However, we deem the incorporation of indices to treat updates to nested collections as interesting future work.

A very recent and interesting work on incremental view maintenance is the DBToaster project [AKKN12]. DBToaster is an optimization framework that takes SQL queries, and compiles them into procedural C++ code that performs the incrementalization. The framework uses an incremental view maintenance technique called higher-order delta processing, which can be understood as an extension of traditional delta queries as found in [GL95]. Instead of simplifying algebraic expressions as in [GL95], the deltas are compiled to highly optimized code. The work is orthogonal to this thesis in supporting more aggressive optimizations. Nevertheless, it assumes the availability of base relations for incremental view maintenance, hence, it is not immediately clear how much of the optimizations are portable to our approach.

The works by Hull et al. [HZ96] and Quass et al. [QGMW96] optimize the amount of materialized data and are thus most closely related to our work. Hull et al. minimize the materialized data in a distributed data warehouse setting, through (partially) virtualized views, which means that (some) attributes are not stored by the materialized view. The incentive is that general access to the distributed source relations (i.e., base relations in our terminology) of the data is expensive, yet some source relations are not frequently updated. Depending on the operators used in defining the views, only the source relations (or specific attributes in these relations) required during frequent re-computations are materialized. The decision of what to materialize is driven by the database designer and communicated via annotations on the database schema, similar to the increment local annotations introduced in Sec. 4.2.5. This approach is orthogonal to the increment local optimization used in this thesis. In our use-case of static analyses we typically change all base relations (i.e., all sources). Hence, such an optimization is not profitable. Yet, in other scenarios the approach by Hull et al. can certainly improve memory consumption.

Quass et al. also consider the problem of materializing data in a distributed data warehouse setting. They use referential integrity constraints, i.e., primary and foreign key relationships, to determine irrelevant data tuples. The work has a special emphasis on views declared via joins. Referential integrity can be thought of as a kind of existential constraint. For example, if a database contains relations for stores and items sold at each store, we can only insert sold items if a corresponding store exists. Assuming that a view is built over the items sold in particular stores (a join over a subset of the stores to a subset of the sold items), incremental maintenance can be optimized as follows: (i) if a new sold item is inserted, an existential query can filter out items that will not join with any of the selected stores, and (ii) if a new store is created we know that existing items can not join to a new store. Hence, only the subset of the items that can actually join with the selected stores is required to be maintained. This subset is then materialized as a (smaller) auxiliary view. In general, the approach determines all relevant auxiliary views via the integrity constraints. The approach assumes a special form of existential operator (i.e., semi-join in our terminology), compared to the operator introduced in Sec. 3.3.4. In the existential operator of Quass et al. the right-hand side is always materialized and the left-hand side is transient (comparable to a selection operator), the correctness of this operator is ensured by the integrity constraints. As an optimization the approach is also orthogonal to the increment local optimization. In summary, the range of queries that can be optimized is smaller (i.e., only operators on primary and foreign keys) and the optimization on integrity constraints still materializes data; though less than an unoptimized version. In practice we found little application of the optimization that uses referential integrity constraints. The constraints themselves can be found in static source code analyses. For example, a method is always contained in a class; hence, a join from methods to a particular set of classes could be optimized. Yet, we used object references from methods to enclosing classes; hence, a query that selects method for classes with particular properties is expressible as a simple selection.

# 5 Incrementalized Static Analysis Engine

In this chapter the event-driven embedded database (introduced in the previous chapters) is utilized to develop an incrementalized *Static Analysis Engine* (SAE). The goal of the SAE is to provide users with a rich framework for formulating static analyses whose results are automatically maintained by the engine. The incrementalization is used to provide real-time feedback w.r.t. analysis results that are shown in an IDE and the results are automatically updated when developers perform changes to the code base. The analyzed code considered in this thesis is Java bytecode. Hence, plain Java programs, but also programs in languages that build on Java bytecode, e.g., Scala, Clojure[1], Groovy[2] etc., can be analyzed.

The purpose of this chapter is twofold. First, we discuss the *representation* of Java bytecode and define the *base relations* of a database that will be queried. Second, we show how static analyses are formulated over these base relations, using IQL (cf. Sec. 4.1) as a query language. We also introduce *general views* that are predefined for all users of the SAE and exemplify how more elaborate analyses can be defined using these views. Finally, we discuss how the SAE is integrated into an IDE.

## 5.1 Java Bytecode Representation and Base Relations

A Java program is represented as objects in memory that capture its basic structure, i.e., classes, methods, fields and instructions. The representation considered in this thesis is derived from Java bytecode and can be divided into three main areas. First, as a basis, all Java programs build on a representation of the types used in the program. Types are used at all levels of the representation, for example to declare the type of a class, the parameter types of a method, or the type of the receiver of a method call instruction. Second, the representation consists of objects for the basic entities in a Java program, i.e., objects representing classes, methods and fields Finally, the representation of Java code at the level of bytecode instructions is considered.

In order to read Java bytecode the toolkit BAT [BAT13] is used, since it is written in Scala and integrates nicely into the embedded database. BAT is utilized in a manner that reuses several structures from the toolkit. Integration with other bytecode toolkits, such as ASM [BLC02], is also possible and would only require a minimal effort.

[1] http://clojure.org/

[2] http://groovy.codehaus.org/

## 5.1.1 Representation of Java Types

Java types are encoded using a type safe approach in Scala via the type hierarchy depicted in Figure 5.1. The type hierarchy is directly reused from the underlying bytecode toolkit (BAT). The type hierarchy discerns two main types. First, the `VoidType` is valid only as a return type of a method, but can not be used in any other position, e.g., as the type of a method's parameter. Second, the `FieldType` can be used in all other positions, i.e., as a method parameter, as a return type or as a field type. Note that the term `FieldType` stems from the Java language specification. The `FieldType` is further discriminated into the `BaseType`, used for Java primitive types such as `int`, `double` etc., and the `ReferenceType`, used for object and array references. Reference types have specific attributes for the type they represent. The `ObjectType` contains a string for the class name it represents. For example to represent the type `java.lang.Exception` an `ObjectType` is constructed with the class name[3] as follows:

```
val Exception = ObjectType("java/lang/Exception")
```

The `ArrayType` contains a reference to the type of the components stored in the array – multidimensional arrays are represented with components `ArrayType`. For example an array of strings (`java.lang.String[]`) is constructed as follows:

```
val stringArray = ArrayType(ObjectType("java/lang/String"))
```

Memory efficiency is of great importance for the overall representation of types, since (references to) types make up a considerable portion of the represented Java program. Hence, all objects of type `BaseType` and `VoidType` are singletons [GHJV95], e.g., there is only one object representing the type `int`. This also reflects the nature of these types since primitive types (or the void type) are the same in the whole program. Usage of singletons is enforced at the language level via Scala's the **`object`** construct [OSV10], e.g., the following code declares the type for integers to be a singleton.

```
object IntegerType extends BaseType
```

Due to the guarantee on the language level, singletons are also used when static analyses construct instances of a specific type, e.g., to check that a method signature contains an `int` parameter.

Memory efficiency is also considered by BAT when constructing object types and array types. The same type can potentially be used in many places, e.g., the type `String` can be used as a parameter or return type for a multitude of

[3] Note that the bytecode representation uses the character "/" instead of "." to separate identifiers

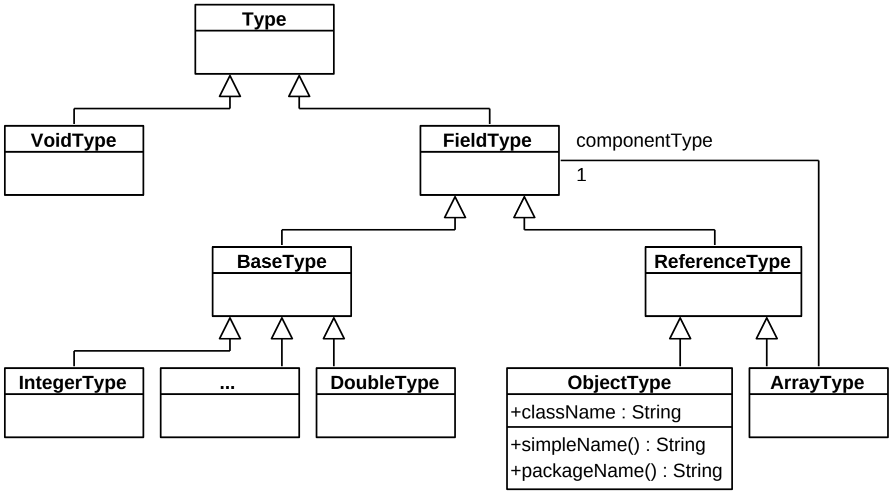


**Figure 5.1:** Representation of the Java type system

```
1 object ObjectType
2 {
3   val cache: Map[String, ObjectType] = Map()
4
5   def apply(className: String) = {
6     cache.getOrElseUpdate(className,
7                                new ObjectType(className))
8   }
9 }
```

**Figure 5.2:** Flyweight pattern for creation of `ObjectTypes`

methods, or as field type in many classes. Hence, object types and array types are constructed via a Flyweight pattern [GHJV95], which is basically a cache for previously constructed types. For illustration the Flyweight pattern for `ObjectTypes` is depicted in Figure 5.2. The `cache` (line 3) is a map from the previously constructed class names to their type representation. Types are constructed via the method `apply` (line 5). The name `apply` is a special method name, which allows to call the

Flyweight factory via the shorthand `ObjectType("java/lang/String")`, i.e., the `apply` is implicitly added as a method call by Scala.

### 5.1.2 Representation of Java Entities

The basic Java entities (class, method and field declarations) are each represented as distinct classes that encapsulate all relevant information pertaining to each entity.

**Classes**

The `ClassDeclaration` contains the structural information w.r.t. the type information of a class and the inheritance hierarchy and is depicted in Figure 5.3. Note that a class is also the largest scope for which we can guarantee that modifications are made as a single local increment. Hence, we annotate the class declarations with `@LocalIncrement` (line 1). The `classType` (line 3) is the type of the declared class. The `superClass` represents the type of the super class this class declaration extends (line 4). Note that for the special case of type `java.lang.Object` no superclass is defined, hence the field is declared as a Scala `Option` that allows an analysis to check whether the super class is defined or not. The `interfaces` (line 5) represent the set of interfaces (as types) implemented by the Java class. Class declarations also contain the minor and major version (lines 6 and 7) of the Java language in which the class was compiled. Note that some analyses are only relevant for programs compiled in earlier or later versions of Java. Visibility and other modifiers are encoded as a single value in the `accessFlags` (line 8) that stores all modifiers as bits set in the integer value. The `ClassDeclaration` provides suitable abstractions to recover the individual modifiers. For example the method `isPublic` (line 10) performs a bitwise operation to test whether the `accessFlags` contains the modifier that declares the class as `public`.

**Class members**

Method and field declarations capture the relevant information pertaining to the declaration of a respective member in a class. Note that a method declaration captures information relevant only to a method's signature and modifiers. The code of a method will be treated in the subsequent paragraph on the representation of Java Instructions. For illustration, the definition for method declarations is depicted in Figure 5.4.
The depicted trait `DeclaredClassMember` (line 1) allows to abstract over methods and fields, since they share some commonality, i.e., being declared in a class and having specific access modifiers such as `public`. The class `MethodDeclaration` (line 8) contains a reference to the declaring class (line 9), the `name` of the method

```
@LocalIncrement
case class ClassDeclaration(
  classType: ObjectType,
  superClass: Option[ObjectType],
  interfaces: Set[ObjectType],
  minorVersion: Int,
  majorVersion: Int,
  accessFlags: Int)
{
  def isPublic: Boolean =
        (ACC_PUBLIC & accessFlags) != 0
  def isAnnotation: Boolean = ...
  ...
}
```

**Figure 5.3:** Representation of Java class declarations

```
trait DeclaredClassMember {
  def declaringClass: ClassDeclaration
  ...
  def isPublic: Boolean
  ...
}

case class MethodDeclaration(
  declaringClass: ClassDeclaration,
  name: String,
  returnType: Type,
  parameterTypes: Seq[ValueType],
  accessFlags: Int
) extends DeclaredClassMember
{
  def isNative: Boolean = ...
  ...
}
```

**Figure 5.4:** Representation of Java method declarations

(line 10), the return type (line 11) and the method parameters (line 12). The latter are an ordered sequence of types as they appear in the method signature. In addition the `accessFlags` (line 13) are stored and treated similar to the `accessFlags` in a class declaration. Note, that the `accessFlags` are complete w.r.t. the specification of the Java language specification[4]. For example, methods can be declared as `native` (line 16) Fields declarations are treated similar to method declarations, but have only the field's type instead of a return type and parameter types. Note that fields also have different `accessFlags`, e.g., `transient`.

### Relational representation vs. OO representation

The representation of Java entities as defined by the SAE is inherently relational. For example, the set of all methods will be made available as an extent and, hence, each method is represented by a single object that uniquely identifies the represented method in terms of (i) declaring class, (ii) name and (iii) signature. The object-oriented representation that is the default output of BAT contains an object graph for traversing the data. For example, a class contains a list of its methods and fields. These objects are then only uniquely identified in the context of the containing class, i.e., they contain only name and signature.

Since the relational representation contains merely necessary facts to identify the represented entity, it does not keep alive unnecessary objects that are filtered out by queries over the database. Note that we intentionally included more data in the entities than is strictly necessary. For example, a class contains a list of interfaces, which we will later also make available as a relation. Nevertheless, this data contributes only small amounts of memory, while keeping it in the representation simplifies query writing.

### Local increments

As noted earlier a class is the biggest scope for which we can use the optimization w.r.t. local increments. By using the `@LocalIncrement` annotation we can now, for example, join two class members, e.g., two methods, via their `declaringClass` property and the join will not require any materialization of data. Note that the second valid scope we will later use in the evaluation is the method scope, which is declared by using the annotation on the `MethodDeclaration` entity instead of the class declaration.

[4] The Java Language Specification can be found at http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

## 5.1.3 Representation of Java Code

The Java code of each method requires a representation of the instructions, i.e., the code executed for a method. All instructions are represented as objects of type `Instruction`. The overall interface is very narrow, i.e., in essence each instructions has an opcode that identifies which instruction is to be executed (cf. Figure 5.5).

```
trait Instruction {
    def opcode: Int
}
```

**Figure 5.5:** Interface for Java bytecode Instructions

Individual instructions are singleton objects, if they do not contain further information, e.g., the instruction `ALOAD_0` (cf. Figure 5.6 line 1) that loads a reference on the stack from the local variable table at index zero. Instructions that contain further data – depending on how they are used in the program – are represented as case classes that contain said data as fields, e.g., the instruction `ALOAD` (cf. Figure 5.6 line 5) that also loads a reference from the local variable table, but uses the index indicated by the field `lvIndex`.

```
1 case object ALOAD_0 extends Instruction {
2     def opcode: Int = 42
3 }
4
5 case class ALOAD(lvIndex: Int) extends Instruction {
6     def opcode: Int = 25
7 }
```

**Figure 5.6:** Concrete Representation for Java bytecode Instructions

The complete code of a method is packed together as an array of instructions. The rationale of BAT is to provide all relevant information in a compact format that closely resembles the Java bytecode, i.e., in the bytecode all instructions are represented as a byte array. The object representation stores the actual instructions in the array and implicitly stores the *program counter* (pc) for the instructions via the index in the array. Note that each instruction has a length of one or more byte and the pc of a Java program is basically the index in the byte array of instructions.

The index is then used in control flow related instructions, e.g., a goto instruction provides the index (pc) at which the execution should continue. Figure 5.7a depicts a simple program for a constructor that calls the super constructor of the type `java.lang.Object`. The call instruction (invokespecial) has a length of 3 bytes; hence the next pc is 4. Figure 5.7b depicts the resulting array for this simple method. The object representation fills indices at which no instruction resides with the value **`null`**.

| pc | instruction |
|---|---|
| 0 | aload_0 |
| 1 | invokespecial java/lang/Object.<init> |
| 4 | return |

**(a)** Bytecode Instructions

| |
|---|
| `ALOAD_0` |
| `INVOKESPECIAL(ObjectType("java/lang/Object"), "<init>", MethodDescriptor(Nil, VoidType))` |
| **`null`** |
| **`null`** |
| `RETURN` |

**(b)** Array Representation

**Figure 5.7:** Representation of a method's instruction array

The representation as an array of instructions is very compact and contains all relevant information. However the representation is not suitable for declarative definitions of analyses and is also not amenable to optimizations such as indexing for specific instructions. Nevertheless, this representation allows a compact definition of base relations, i.e., the entire code is made available in one extent. A suitable transformation using the incrementalized operators provided by the database is considered in Section 5.3.

The complete code – in the above described array form – is defined by the SAE as an object of type `CodeInfo` (cf. Figure 5.8) that holds all relevant information of a method's code. The code info object defines to which method the code pertains (line 2), the maximum stack height (line 3), the maximum amount of local variables (line 4), the registered exceptions handlers (line 5) and finally the instructions (line 6). The maximum stack height and maximum local variables are invariants generated by the Java compiler that allow optimized execution of the Java bytecode, without re-analyzing the code to determine these values. An exception handler is a designated block of code, at which the execution continues when an exception is thrown. Each handler can treat a different type of exception and the correct handler is determined when an exception is thrown.

```
case class CodeInfo(
  declaringMethod : MethodDeclaration,
  maxStack: Int,
  maxLocals: Int,
  exceptionHandlers: Seq[ExceptionHandler],
  instructions: Array[Instruction]
)
```

**Figure 5.8:** Representation of a method's code block

## 5.1.4 Base Relations

The SAE is defined using only four base relations. All other relations are derived from the data found in these relations. The minimal number of base relations simplifies the integration between the parsing of a Java class file and the database. The four relations are depicted in Figure 5.9. The first three relations capture the different Java entities, i.e., classes (line 3), methods (line 5) and fields (line 7). The final relation captures the code information for each method (line 9). Note that Figure 5.9 depicts the interface that a concrete database has to implement. The interface can be used to access the relations in queries. However. concrete instance can define more (e.g., derived) relations.

All four base relations are declared with the type `SetRelation`. This signifies the guarantee that each element in these relations will be present only once. For example, there can be only one class declaration for each type such as `java.lang.String`. Likewise, there can be only one method with a specific signature, i.e., method name, parameter types and return type, for a given class; since the class is a declared as part of a `MethodDeclaration`, the elements in methods are unique. The information that the extents are sets can be used by the query compiler for optimizations

Note that the framework does not enforce this guarantee, since such a guarantee can only be asserted by storing all elements and checking each addition. Since the underlying execution model is based on the fact that the elements should not be stored in the base relations, elements are added to the base relations and propagated

```
trait BytecodeBaseRelations
{
  def classDeclarations: SetRelation[ClassDeclaration]

  def methodDeclarations: SetRelation[MethodDeclaration]

  def fieldDeclarations: SetRelation[FieldDeclaration]

  def code: SetRelation[CodeInfo]
}
```

**Figure 5.9:** Base relations for static analyses

to derived relations without any checks. Nevertheless, the guarantee is enforce by the Java compiler and since the static analyses are based on Java bytecode generated by a Java compiler there is no need to re-assert this guarantee.

## 5.2 Example Analyses on Base Relations

Using the base relations defined in Figure 5.9, the first (simple) static analyses can be formulated. The following two examples are lightweight analyses derived from the tool Findbugs [HP04]. For this thesis a larger set of Findbugs analyses was sampled and formulated as SAE queries. The whole range of sampled queries and their performance characteristics are discussed in Chapter 6. Note that all queries are formulated with the entire dataset in mind. Incrementalization is performed automatically due to the defined relational operators.

**Example 1 – Protected field in final class**

The first analysis detects cases of field declarations with confusing visibility modifiers. A field is in violation of the query, if it is declared as **protected** while the declaring class is declared as **final**. The rationale behind this check is that the class declaration does not allow any sub classes; hence the field can never be accessed by any sub class and should either be declared as package visible or as **private**. The respective SAE query is depicted in Figure 5.10. To illustrate the relationship between a query and the underlying database, the query is defined as the result of a method named `apply` that constructs the incrementally maintained query for a given `database` (passed as a parameter in line 1). The result is a relation of type `FieldDeclaration` (line 2), which can then be used to report violating field declarations to the end user.

```
1 def apply(database: BytecodeBaseRelations):
2   Relation[FieldDeclaration] =
3 {
4   SELECT (*) FROM (database.fieldDeclarations) WHERE
5     (_.declaringClass.isFinal) AND
6     (_.isProtected)
7 }
```

**Figure 5.10:** SAE query for protected field declarations in final classes

The query is defined on the base relation `database.fieldDeclarations` (line 4) and selects all field declarations `(*)` that pass the test formulated in the `WHERE` clause. The formulation of the test is simple and declarative; the declaring class must be have the modifier **`final`** (line 5) and the field itself must be declared as **`protected`** (line 6). Due to the integration of the EDSL with the type inference of Scala anonymous functions can be used to declare the tested properties.

**Example 2 – Public finalizer defined**

The second analysis detects method declarations of Java finalizer methods (`void finalize()`), with dubious visibility modifiers. The general purpose of finalizers is to perform cleanup operations such as releasing system resources before an object is collected by the VM's garbage collector. The contract of Java declares finalizers as **`protected`**, which is due to the fact that these methods may be overridden in sub classes, but should not be called from external objects. Hence, declaring a finalizer as `public` is dubious and checked by the example analysis.

The query (cf. Figure 5.11) is defined on the base relation `methodDeclarations` (line 4) and selects all method declarations that pass the test formulated in the `WHERE` clause. The tests are defined in a similar declarative style as in the previous example. The declared method must have the name `"finalize"` (line 5), be declared as `public` (line 6) and adhere to the signature of the Java finalizer methods, i.e., have a return type of `void` (line 7) and no parameters (line 8).

## 5.3 Extended Representation and Derived Relations

In the following, two common areas of information are discussed that are used in many analyses and therefore declared as derived relations in the `BytecodeDatabase`.

```
def apply(database: BytecodeBaseRelations):
  Relation[MethodDeclaration] =
{
  SELECT (*) FROM (database.methodDeclarations) WHERE
    (_.name == "finalize") AND
    (_.isPublic) AND
    (_.returnType == VoidType) AND
    (_.parameterTypes == Nil)
}
```

**Figure 5.11:** SAE query for publicly declared finalizer method

The first area is the inheritance hierarchy. The second is a relational representation of a method's instructions, which is suited for optimizations by the database.

**Inheritance hierarchy**

In the base relations the information regarding subclassing and inheritance from interfaces is captured inside each class declaration (cf. Figure 5.3). However, many static analyses have a special interest in reasoning over the transitive closure of all subtypes, hence in the following a view `subtypes` is derived for this purpose. The basic process for deriving such a relation is to unnest the information w.r.t. interface inheritance from the class declaration, and building the transitive closure – via the respective operator – over a union of subclassing and interface inheritance. The important fact in this process is that the view `subTypes` can be declared using only self-maintainable operators without auxiliary storage, except for the final step which is the building of the transitive closure. In the following a detailed account of the derivation process is given for illustration.

The first step of the derivation is to find a general representation of the information w.r.t. subclassing and inheritance from interfaces. This is necessary since the transitive closure is defined over a single relation, where each object in the relation represents an edge. Hence, the information is uniformly transformed to a subtype-supertype relationship. Figure 5.12 depicts the respective Scala class that is used in the following to capture all inheritance based relationships.

The second step is to transform the base relations into the uniform representation. The definition of respective queries is depicted in Figure 5.13. The view `classInheritance` (line 1) declares a transformation for the superclass inside a class declaration to an inheritance relation. The view `interfaceInheritance` (line 6) declares the transformation that unnests the interfaces from the class

```
case class InheritanceRelation(
  subType: ObjectType,
  superType: ObjectType)
```

**Figure 5.12:** Representation of the inheritance relation

declarations and maps them to an inheritance relation. The unnesting is declared in line 9 using the keyword `UNNEST`. The interfaces function of type `ClassDeclaration => Set[ObjectType]` unnests the set of interfaces. The query is translated to an unnest operator ($\mu$) and a projection ($\pi$). The results of the unnesting – i.e. the interfaces – are passed as a parameter `i` (line 7) to the projection function – declared in the `SELECT` clause – together with the declaring class `c` and transformed to an inheritance relation.

```
1 lazy val classInheritance = SELECT (
2   (c: ClassDeclaration) =>
3     InheritanceRelation(c.classType, c.superClass.get)
4 ) FROM classDeclarations WHERE (_.superClass.isDefined)
5
6 lazy val interfaceInheritance = SELECT (
7   (c: ClassDeclaration, i: ObjectType) =>
8     InheritanceRelation(c.classType, i)
9 ) FROM UNNEST(classDeclarations,_.interfaces)
```

**Figure 5.13:** Relational views of subclassing and interface inheritance

The final step is to build the transitive closure over all inheritance relations and is depicted in Figure 5.14. The view `inheritance` (line 1) builds the union over the inheritance relations for subclassing and interface inheritance. Since the two relations are distinct, they can be declared with the self-maintainable indistinguishable `UNION ALL` keyword that is translated to the self-maintainable operator $\uplus$. The transitive closure `subTypes` (line 5) is built using the `TC` keyword (line 8). The first parameter passed to `TC` is the underlying relation (`inheritance`). The second parameter determines the start and end vertex of an edge in the relation. In this case the start vertex is the declared as the `subType` and the end vertex as the `superType`. Note that the transitive closure can also be built by interpreting the edge from super to subtype. Yet, since the edges are declared inside a class in the sub-to-super-type fashion, i.e., the subtype `A` is declared as **`class A extends B`**, this interpretation seems more natural. The result of the transitive closure operator contains tuples of two `ObjectTypes`, since the operator can not know how to translate these tuples into any other meaningful edge representation. Hence, a projection is declared on top of the closure that transforms the tuples back into the uniform `InheritanceRelation` representation. Note that this is not a necessity, yet is done to achieve a uniform representation over all relations that allow to reason over inheritance.

```
lazy val inheritance =
  SELECT (*) FROM classInheritance UNION ALL
  SELECT (*) FROM interfaceInheritance

lazy val subTypes = SELECT (
  (edge: (ObjectType, ObjectType)) =>
    InheritanceRelation (edge._1, edge._2)
  ) FROM TC(inheritance)(_.subType, _.superType)
```

**Figure 5.14:** View of the transitive closure over the inheritance hierarchy

As can be seen in the above definitions, all queries used in the definition of the `inheritance` relation are compiled to self-maintainable operators ($\mu, \pi, \uplus$). Hence, the transformation to `inheritance` comes at no cost in terms of memory. The transitive closure operator itself must store the tuples that represent edges, however the `inheritance` relation is comparatively small, as will be shown in the later evaluation in Section 6.

**Relational view of instructions**

The goal of the relational view is to transform the arrays of instructions presented in Section 5.1.3 into a relational form, where every instruction is a single tuple in the relation. A relational representation of instructions allows efficient queries over all instructions, such as filtering for specific (groups of) instructions. The relational representation is also designed to provide explicit information about an instructions program counter (*pc*) and successor instructions. The former is present in the array representation only implicitly via the array index where the instruction resides. The latter is tedious to find since the array can contain an arbitrary number of null values between instructions, hence one must always think about the byte length of an instruction. Both situations preclude queries from simple and declarative reasoning over program counters or successor instructions, hence the relational view makes the information explicit. Using the explicit information allows indexing for queries that relate instructions, e.g., finding the target pc of a `goto` instruction.

The relational representation uses a type hierarchy with concrete instructions types that are all a subtype of `RelationalInstruction` (depicted in Figure 5.15). This type contains all relevant information for each single instruction. All instructions contain the methods in which the instruction is declared (line 4), which is the basic requirement to later find only instructions in the same method. Each instruction has a `pc` (line 5) that stores the program counter and can be used for control flow analyses. The `seqIndex` (line 6) numbers each instruction consecutively to allow quick access to precursor or successor instructions, without having to reason about specific instruction lengths.

```
1 trait RelationalInstruction
2         extends Instruction
3 {
4  def declaringMethod: MethodDeclaration
5  def pc: Int
6  def seqIndex: Int
7 }
```

**Figure 5.15:** Relational representation of an instruction

The transformation from the array representation to the relational representation is a two-step process, which is depicted in Figure 5.16. First, the array is transformed to a list of relational instructions. This step includes making the implicit information, i.e., program counter and sequence index, explicit and providing the information in which method the instruction was declared. In addition the list is filtered to omit the **`null`** entries from the array. The step is depicted for illustration in lines 1-19. Since the first step is intermediate, the result is declared as a private value (line 1), thus not to be used directly by the static analyses.

The illustration of the array iteration serves two purposes. First, to underline the workload of analyses that deal with instructions. The instructions make up – by far – the largest part of data in a Java program and, hence, iterations over all instructions are time-consuming. The work performed during the iteration is done only once in the SAE, but contributes to the overall runtime. Second, to illustrate the code for an iteration over instructions. During the evaluation in the next chapter, we compare the SAE to a non-incrementalized implementation in BAT. Some analyses, such as the dataflow analysis, require writing code similar to the one shown above. As we will see the declarative version is much shorter and more concise.

```
 1 private lazy val relationalInstructionsList =
 2   SELECT ( (codeInfo: CodeInfo) => {
 3     var pc = 0
 4     var seqIndex = 0
 5     val length = codeInfo.instructions.length
 6     var result: List[RelationalInstruction] = Nil
 7     while (pc < length) {
 8       val instr = codeInfo.instructions(pc)
 9       if (instr != null) {
10         result = asRelationalInstruction(
11                        codeInfo.declaringMethod,
12                        pc, seqIndex,
13                        instr) :: result
14         seqIndex += 1
15       }
16       pc += 1
17     }
18     result
19   }) FROM code
20 
21 lazy val instruction =
22   SELECT (*) FROM UNNEST(relationalInstructionsList, _)
```

**Figure 5.16:** Relational view on instructions

The actual translation to relational instructions is performed in a single loop over the instructions (line 7). Filtering of the **null** entries (line 9) is performed together with the transformation (line 10) in a single pass. The transformation asRelationalInstruction simply constructs a new RelationalInstruction object for a given instruction, based on the instructions type, e.g., aload_0, invokevirtual. The second step after the transformation is simply to unnests all relational instructions from the list using the unnesting operator of the database (line 22). Since the unnesting is performed on a list of relational instructions, the unnesting requires no extraction of a multiset from an underlying object, but already receives a multiset, i.e., the list of instructions. Hence the unnesting is declared as identity (UNNEST(..., _)).

Using the above declared instruction relation the database can be efficiently filtered for specific instructions. For example, Figure 5.17, depicts two views that filter all static method calls (line 1) and all instructions that read a field (line 6). The SAE uses the type system to filter instructions of a specific subtype. All static method calls are of the type `INVOKESTATIC` – signifying the `invokestatic` bytecode – and hence a respective `instanceof` filter is applied (line 4). The results are cast to the `INVOKESTATIC`, which provides information such as the type of the receiver of the call and the name of the called method. The SAE also features type abstractions for multiple instructions that share a commonality. For example, the instructions that read fields are signified by the `FieldReadInstruction`, which is a supertype for the two concrete bytecode instructions that read fields (`getstatic` and `getfield`). Again the respective view filters instructions by their type (line 9) and casts them (line 7) to make additional properties available, such as the name of the field.

```
1 lazy val invokestatic = SELECT (
2     (_: RelationalInstruction).asInstanceOf[INVOKESTATIC]
3   ) FROM (instructions) WHERE
4     (_.isInstanceOf[INVOKESTATIC])
5
6 lazy val readfield = SELECT (
7     (_: RelationalInstruction).asInstanceOf[FieldReadInstruction
8   ) FROM (instructions) WHERE
9     (_.isInstanceOf[FieldReadInstruction])
```

**Figure 5.17:** Selected relational views for specific instructions

The key advantage of the representation in Figure 5.17 is that the iteration over all instructions is performed only once. The alternative is that each analysis works directly on the array of code. For example, one can easily imagine that an analysis that reasons over static method calls performs it's own filtering via the instruction array, e.g., `code.instructions.filter(_.isInstanceOf[INVOKESTATIC])`. However, the function `filter` must perform an iteration over all instructions similar to the loop in Figure 5.16. Hence, performing many such iterations in each analysis is time consuming and the relational representation is more efficient.

## 5.4 Example Analyses on Derived Relations

Using the derived relations defined in the previous section, more elaborate static analyses can be formulated. The following two examples are derived from the tool Findbugs, similar to the previous examples. While more elaborate than the previous examples, the presented checks on the code are still considered lightweight analyses, in the sense that no control-/dataflow is required. The first example reasons over the inheritance hierarchy. The second finds a specific pattern of instructions, i.e., a sequence of instructions with specific properties. Hence, the second example requires some notion of a method's code but not the full control-/dataflow of the code.

**Example 1 – Clone defined in non-cloneable class**

The first analysis finds situations in which a class declares a Java `clone()` method for object duplication, but does not implement the interface `Cloneable`. By Java's convention if an object does not implement the interface `Cloneable` a `CloneNotSupportedException` is thrown. Hence, this situation must be avoided.

The analysis is based on finding declarations of the clone method where the interface `Cloneable` is not a supertype of the declaring class. Since the `Cloneable` interface can be declared in superclasses, the analysis requires the whole inheritance hierarchy, i.e., the transitive closure over all subtypes. Note that the correctness of this analysis depends on the available classes with which the database is populated, since – due to the dependence on the whole inheritance hierarchy – classes can be missed if their superclasses are not analyzed in the database. For example, the class `CharacterIterator` is defines as cloneable and resides in the JDK. If an analyzed class defines a clone method and is a subclass of `CharacterIterator`, but classes in the JDK are not part of the database, the class is reported as a false positive. However, this limitation exists also in the Findbugs tool and we encoded the analysis as it was found there.

```
1 SELECT (*) FROM methodDeclarations WHERE
2   (_.name == "clone") AND
3   (_.parameterTypes == Nil) AND
4   (_.returnType == ObjectType("java/lang/Object"))
5   NOT ( EXISTS (
6     SELECT (*) FROM (subTypes) WHERE
7       (subType === declaringClassType) AND
8       (_.superType == ObjectType("java/lang/Cloneable"))
9   ))
```

**Figure 5.18:** SAE query for clone defined in non-cloneable class

The query for the analysis is depicted in Figure 5.18 and depends on the base relation of all method declarations (line 1) and the derived `supTypes` relation of all transitive subtypes (line 6). From the method declarations all methods are selected that are implementations of clone, i.e., have the name `"clone"` (line 2), are parameterless (line 3) and return an object of the type `java.lang.Object` (line 4). These selected clone methods are filtered by the `NOT EXISTS` keyword (line 5). The filter finds all methods where no inheritance relation exists, such that the subtype is the type of the declaring class (line 7) and the supertype is the interface `Cloneable` (line 8).

**Example 2 – Primitive boxed and unboxed for coercion**

The second analysis finds situations in the code where primitive values are boxed, i.e., wrapped into a respective object such as `Integer` for `int` values, and then immediately unboxed to a different primitive type. For example, given a variable `float f;` a boxing and unboxing is performed to coerce the `float` to an `int` by the following statement:

```
int i = new Float(f).intValue()
```

In Java the above computation should be performed by a direct primitive coercion that is cheaper in its execution, e.g., using the statement:

```
int i = (int) f
```

The actual analysis must only be performed for classes compiled for Java 5, since this version introduced automatic boxing of primitives, which can lead to confusion for untrained developers. Note that other situations – apart from coercion – where boxed types are immediately unboxed are also checked by Findbugs, but reported as different errors.

The analysis revolves around finding an `invokespecial` instruction – the constructor call, e.g., `new Float(f)` – immediately followed by an `invokevirtual` instruction – the call to perform the coercion, e.g., `intValue()`. The analysis – such as it is presented – is derived from the original Findbugs implementation and can be seen as only a heuristic for finding boxing followed by unboxing. In general an object can be stored in a variable and unboxed later, which requires a more elaborate analysis based on the method's dataflow. The query applies two more heuristics also found in the original Findbugs implementation. The first heuristic captures the right constructors by stating that the class of the constructed object resides in the package `java.lang`, e.g., `java.lang.Float` or `java.lang.Integer`, and receives a primitive value as its single parameter. The second heuristic captures the right method calls for the coercion by stating that all methods are basically parameterless and their name ends with `Value`, e.g., `intValue()` or `floatValue()`.

The respective query is depicted in Figure 5.19 and performs a join between the derived relations `invokeSpecial` and `invokeVirtual`. (line 1) to find immediately

```
1  SELECT (*) FROM (invokeSpecial, invokeVirtual) WHERE
2    (declaringMethod === declaringMethod) AND
3    (nextIndex === index) AND
4    (receiverType === receiverType) AND
5    NOT (firstParameterType === returnType) AND
6    (_.declaringClass.majorVersion >= 49) AND
7    (_.parameterTypes.size == 1) AND
8    (_.parameterTypes(0).isBaseType) AND
9    (_.receiverType.packageName == "java/lang") AND
10   ((_: INVOKEVIRTUAL).parameterTypes == Nil) AND
11   (_.name.endsWith("Value"))
```

**Figure 5.19:** SAE query for primitives that are boxed and unboxed for coercion

consecutive instructions. There are three join conditions, stating that (i) instructions must reside in the same method (line 2), (ii) instructions must follow each other immediately, i.e., the next index after the first instruction is the index of the second instruction (line 3) and (iii) methods must be called on the same type (line 4), e.g., the statement `new Float(f).intValue()` produces a constructor call to `Float` and the call to `intValue()` on an object of type `Float`. To check that a coercion performed, the joined instructions are compared by the parameter type of the constructor and the return type of the `...Value()` call. A coercion is performed if the parameter and return type are not the same (line 5). The check in line 6 states that the Java version must be greater or equal to Java 5 (which is encoded as the number 49 in Java class files). The `invokespecial` instruction must fulfill several properties; (i) for coercion the constructor must receive a primitive type as its sole parameter (lines 7 and 8), (ii) for the above mentioned heuristic, the constructed class must be in the package `java.lang` (line 9). Finally, the heuristic for the coercing method call checks the method to be parameterless (line 10) and of a name that ends with `Value` (line 11).

## 5.5 IDE Integration

The goal of the integration is to provide visual feedback of the static analyses results and automatically notify the end-user (developer) of changes in the results. Figure 5.20 depicts a high level overview of the SAE's integration into an IDE. The visual feedback is provided by IDE views (uppermost component), that register themselves as observers to various static analyses. The observers are notified of all changes in the results of the analyses via the API shown in Figure 3.13 in Section 3.4.1 and take appropriate actions to visualize the results. For example in the case of the Eclipse IDE, results are shown in the Eclipse “Error View”. In the following the components that the visualization relies on are discussed from top to bottom as shown in the overview in Figure 5.20.

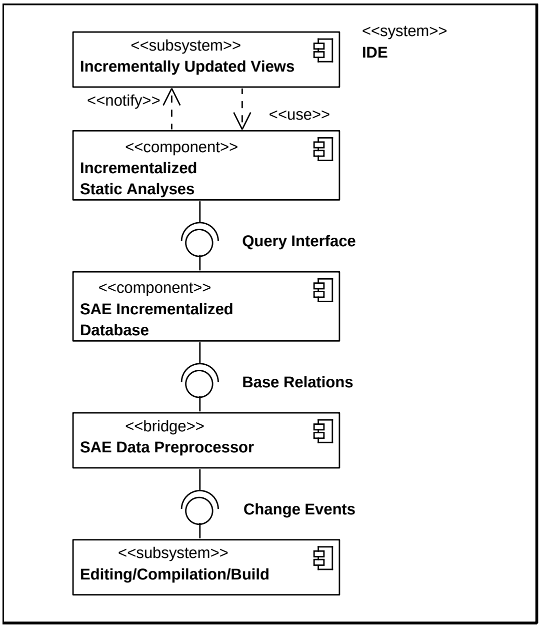


**Figure 5.20:** High-Level overview of the SAE IDE integration

## Incrementalized static analyses

The analyses are represented by the queries defined on the Java bytecode as shown in the four examples in the previous sections. Each query can be instantiated to a compiled and incrementally maintained operator tree via an `apply` method as shown in Figure 5.10. The IDE then registers appropriate observers to a compiled query. Note that the assumption is that the IDE is written in the same programming language as the host language of the database, or based on the same execution platform, i.e., a Java VM. Hence, efficient integration is possible via direct method calls and no communication overhead between the database and the IDE exists. Thus the analyses in the presented form can be efficiently integrated into any Java-based IDE such as Eclipse[5], NetBeans[6] or IntelliJ[7]. From these Java-based IDE's we have integrated the SAE into the Eclipse IDE, but the concepts are generalizable to other IDE's as well.

To allow the IDE to choose which analyses are to be executed, a list of available analyses is provided. For example in the case of the Findbugs analyses – from which the previous examples are drawn – a list of available analyses is provided via a respective identifier for each analysis stemming from the Findbugs website[8]. The IDE – in the case of Eclipse a respective Plugin – then instantiates the analyses, registers itself as an observer and delegates visualization, e.g., to the Eclipse "Error View".

For the IDE integration analyses can also be de-constructed when they are not needed any further or should not be displayed for other reasons. The deconstruction entails the clearing of the chain of operators that contributes to the query result until a base relation is reached. Conceptually, the idea is to traverse the registered observers top-down and remove all observers that contribute to the result. Since operators can contribute to the results of more than one analysis, the traversal is stopped, if – after the removal – further observers are registered to an operator. There is a small catch, namely that the process should not de-construct the derived relations defined in Section 5.3. There queries are currently compiled once during the creation of the database and should not be de-constructed. For example, if the query in Figure 5.18 is the only query that uses the `subTypes` relation, the `subTypes` query must not be de-constructed even if (currently) no further analyses have registered observers to it. Hence, a cleanup operation is defined for each query that stops the top-down traversal at specified child operators.

```
1 def clean(db: BytecodeDatabase,
2           query: Relation[MethodDeclaration]) = {
3   query.cleanObserversWhile(
4     _ != db.subTypes
5   )}
```

**Figure 5.21:** Deconstruction of the query for clone defined in non-cloneable class

For example Figure 5.21 depicts the deconstruction for the example in Figure 5.18. The call to `cleanChildrenWhile` (line 3) de-constructs all operators until a specified

[5] http://www.eclipse.org

[6] http://netbeans.org

[7] http://www.jetbrains.com/idea

[8] http://findbugs.sourceforge.net/bugDescriptions.html

child is reached. In this case the query depends on the `subTypes` relation, which is not visited due to the check in line 4.

### SAE incrementalized database

All queries are formulated on a single incrementalized database. Concretely the database corresponds to a single instance of the class `BytecodeDatabase`. The IDE is then responsible for providing data to the base relations (cf. Section 5.1) of the database. The data corresponds to the Java program currently developed by the user of the IDE, together with any libraries that must be checked by an analysis.

The idea of the integration is to provide data via the code compiled by the IDE for a given Java program. During the development in the IDE each editing of code triggers the IDE's compiler and a new class file is generated (given that the code compiles). The class file is then parsed and its data added to the base relations of the SAE. Since, the main motivation of the SAE is to have a minimal memory overhead, the deletion and update of the base relations revolves around re-reading the old versions of a class file and generating deletion/update events instead of addition event. Hence, the analyzed data can be seen as persisted in the form of class files.

For the integration, the `BytecodeDatabase` provides several methods for reading class files compiled by the IDE. Figure 5.22 depicts the API, which features methods for adding (line 3), removing (line 8) and updating (line 10) the base relations from the data found in the respective class file. The methods all behave similarly; a class file is read (parsed) by a `reader` (line 4). Different readers exists that generate add/delete or update events on the base relations during the parsing of a class file. In addition, libraries used by a Java project are added to the analysis process via respective methods that read a whole archive – jar file – via the methods in line 13 and 15.

To trigger respective calls for adding and removing class files an integration into the IDE's build process is necessary. This integration entails tracking additional information, such as where the IDE stores compiled classes. Since this is very IDE specific, the task is performed by a dedicated component called *Buildsystem Bridge*.

### SAE buildsystem bridge

The buildsystem bridge registers to the low level services of the IDE (bottommost component), e.g., source code editing, compilation or other build facilities and translates the received events into calls to the `BytecodeDatabase`. Notifications from said facilities are provide by most modern Java IDE's, i.e., Eclipse, IntelliJ or NetBeans. The bridge has two important tasks. First, the bridge tracks where the IDE stores class files for a given project. Thus the bridge can translate modification events to concrete file streams that are parsed and added to the `BytecodeDatabase`.

```
class BytecodeDatabase{
  ...
  def addClassFile(stream: InputStream) {
        val reader = ClassfileAdditionReader(this)
        reader.readClassfile(stream)
  }

  def removeClassFile(stream: InputStream)
  ...
  def updateClassFile(streamOld: InputStream,
                      streamNew: InputStreams)
        ...
  def addArchive(stream: InputStream)
        ...
  def removeArchive(stream: InputStream)
  ...
}
```

**Figure 5.22:** API for providing data of analyzed code

Second, the bridge performs a bookkeeping over already compiled class files. Since the remove (and update) methods to the database require the old (and new) class file both must be readable at the same time. In the case of the Eclipse IDE this poses the practical problem that the compiler writes the new class file and then produces a change event. Hence, the previous version is lost without a form of bookkeeping. Thus the bridge provided for the Eclipse IDE stores for each class file a “shadow” file that represents the previous state of the compiled class. In practice, this process produces practically no overhead, since the class file as a whole is simply copied.

# 6 Incrementalized Static Analyses for Software Quality Assessment

To evaluate the SAE, a rigorous performance measurement is performed over a large variety of queries that assess the quality of a software system in terms of correctness and maintainability. The queries presented in this chapter represent queries with largely different characteristics that employ *lightweight analyses*, *metrics* and *intra-procedural dataflow*.

The lightweight analyses focus on (i) searching for declarations of classes, methods, or fields with specific properties, (ii) inspecting the type hierarchy or (iii) locally analyzing method implementations for specific instruction patterns, without resorting to a full control-/dataflow analysis of the methods (i.e., analyzing the effect of bytecode instructions on the stack).

The metrics are frequently cited as indicators for code styles that are problematic during software maintenance. The metrics are mainly included, since they provide a set of queries that excessively uses aggregations, i.e., metrics are inherently based on the notion of counting different relations between entities in Java (byte-)code.

The intra-procedural dataflow analyses are a means to find bugs more precisely. The analysis performed in this chapter is a low level form of intra-procedural dataflow analysis that basically models the stack and local variables of a Java program and evaluates the effect of instructions on this model. Note that intra-procedural dataflow analyses as found in static analyses text books (e.g., [NNH99]) use higher level mathematical abstractions to formulate the analyses, but are in principle comparable.

To obtain a realistic evaluation of the SAE the measurements are performed on real-world queries. The first set of real-world queries stems from the tool *Findbugs* [HP04], which is a popular open source bug-finding tool that analyzes Java bytecode. Findbugs was selected as a source of queries because the Findbugs analyses represent typical non-trivial queries over Java bytecode and is used as the source for lightweight and intra-procedural analyses. The second set contains object-oriented metrics as they are found for example in the open-source Java tool *Metrics*[1]. The Metrics tool is primarily inspired by the object-oriented metrics found in Henderson-Sellers [HS96][2] and Robert C. Martin [Mar03].

The actual queries for the evaluation were chosen in the following manner. In Findbugs we sampled queries in two batches that together contain 24 queries from

[1] http://metrics.sourceforge.net/

[2] Henderson-Sellers is a very good text book on object-oriented metrics that also includes corrections to some metrics that can be found in earlier published papers

approx. 400 analyses in Findbugs. The first batch contains 12 manually selected queries that were chosen mainly to get a good distribution over the queried entities (classes, fields, etc.). Subsequently, a second batch was randomly selected to avoid any bias in the range of sampled queries. This second batch consisted of 19 additional Findbugs queries, from which 12 do not use control-/dataflow analyses and hence fall into the measured category of lightweight analyses. The other 7 randomly selected queries are presented in the context of control-/dataflow analyses. From the Metrics tool, we manually selected queries with different properties that (i) reason over all dependencies found in the bytecode, (ii) count and correlate multiple properties and (iii) count elements in the inheritance hierarchy.

In the following the basic evaluation procedure is defined first and the datasets on which the evaluation is performed are characterized. Then the different analyses are evaluated in detail in the order of *lightweight analyses*, *metrics* and finally *intra-procedural dataflow*. For each of these analyses the general shape of the used queries is characterized first before presenting the results of the evaluation.

## 6.1 Evaluation Procedure

To evaluate the SAE's performance for executing the above defined static analyses, the runtime and memory consumptions are measured by running the analyses on a variety of existing code bases.

All measurements were performed on a 4-core/8-thread Intel Core i7-870 processor with 2.93 GHz and 4096 MiB RAM running a 64-bit version of GNU/Linux (Debian "Squeeze" 6.0.4, kernel 2.6.32). Each analysis is executed multiple times, which allows us to assume that the arithmetic mean of the execution times and memory consumptions follow a Gaussian distribution [GBE07]. There are several factors of the Java virtual machine (JVM) that are diminished by multiple executions. First, the just-in-time compiler (JIT) of the JVM also consumes time and memory for compiling executed methods. Second, frequently executed methods are re-compiled with higher optimization levels by the JIT. All performance measurements are averaged across 100 invocations with a warm-up phase of 50 invocations. The latter are not counted into the measurements. Overall this achieves a steady-state performance with a coefficient for variation below 3.0 % for all measured analyses. Hence, all measured values are sufficiently stable so that a detailed presentation of error bars will be omitted.

Each analysis is evaluated using the following three steps, which are each discussed in more detail in the following sub sections. First, the SAE is compared to a non-incrementalized analysis in BAT for its performance w.r.t. the initial execution time of the analyses over large datasets. Second, the SAE's incrementalization is measured to analyze that the *principle of inertia* indeed holds and incremental computations are fast. This second step also compares the impact of the `@LocalIncrement` optimization w.r.t. the runtime. Two different scopes for the optimization are compared: (i) entire classes and (ii) entire methods. Third, the evaluation measures the memory that is permanently materialized by the SAE to hold auxiliary data. This third step also measures the impact of the `@LocalIncrement` optimization (defined in Sec. 4.2.5) on memory materialization. Note that for all three steps we use the SAE in a fully indexed mode, i.e., every join will index relevant data from the underlying relations. Thus we have measured the best possible runtime along with the highest possible memory allocation.

### 6.1.1 Non-incremental Runtime

The initial runtime of the analyses on large datasets is compared to a non-incremental implementation. The latter was developed using an object-oriented representation of the bytecode, which is the default output of the byteocde toolkit BAT. The comparison is performed to ensure that the time required for the initial computation of the incrementalized analyses is comparable to the non-incrementalized analyses. A negative result in this evaluation would mean that the performance gain of incremental computation does not amortize quickly, e.g., if the incrementalized analyses require an initial computation that is ten times as high, we could simply run the non-incrementalized analyses ten times, before incrementalization amortizes itself.

The re-implementation in BAT was chosen mainly to obtain a uniform comparison. First we used analyses from different tools, i.e., Findbugs and the Metrics tool, which are very differently implemented. Second, a direct comparison to the tools is not possible in a rigorous and fair manner. Findbugs, for example, has bug detectors that are not fully modularized. A visitor pattern is used, where each visitor receives the underlying bytecode data and analyzes it for a large variety of bug patterns. For example many bug patterns related to fields are implemented in one visitor, which includes checking for unread fields, as well as self-assignments to variables where an identical named field exists.

As part of the evaluation it was ensured that the implementations are comparable to the original Findbugs and Metrics implementations. Note that we omit a presentation of the number of detected bug or the values for the metrics. In general, the implemented analyses yield the same results as their counter-parts in tools. Albeit, in rare cases Findbugs uses intricate heuristics that filter out some results, but are hard to reproduce due to the non-modular implementation of the tool. Nevertheless, the implementations in BAT and the SAE produce the exact same results. Furthermore, it should be noted that the number of bugs is not a relevant indicator for runtime performance. In fact, it can be misleading, since an analysis can perform a large amount of computation to conclude that there are 0 bugs and vice versa a large amount of bugs can be detected without heavy computations.

**Example of a comparable BAT analysis**

To give a short account of what the SAE is compared against, a small example is in order. The object-oriented representation in BAT is fairly straightforward and features `classFiles` as the top-level abstraction, which include lists of fields and methods. Each method includes an array of instructions as we have seen in Section 5.3. Type information is represented in the same manner as in the SAE, since the BAT representation was simply reused for the SAE (cf. Section 5.1.1). The queries are formulated as for-comprehensions over the object-oriented structure. For example, Figure 6.1 depicts the same query as previously shown in Figure 5.10 that finds protected fields in final classes.

```
def apply(classFiles: List[ClassFile]) =
for (
  classFile <- classFiles if classFile.isFinal;
  field <- classFile.fields if field.isProtected
) yield (classFile, field)
```

**Figure 6.1:** BAT query for protected field declarations in final classes

In some cases the analyses require more elaborate auxiliary structures, such as the inheritance hierarchy or all fields that are read at least once in the program. These structures are in general computed prior to an analysis (as hash sets or hash maps) and are either used inside the for-comprehensions, or are used as the input to the comprehension (in cases where only selected elements are searched). This ensures that no unnecessary computations are duplicated as sub-loops in each iteration of a for-comprehension and ensures competitiveness to the fully-indexed SAE. Note that it is exactly this usage of auxiliary data structures that complicates a manual incrementalization of the analyses.

**Measured values for comparing to non-incrementalized analyses**

For the evaluation the non-incremental runtime is measured once for each analysis in a particular category, i.e., each lightweight analysis, each metric, etc. Measuring each analysis in isolation allows us to identify which queries exactly perform better or worse in comparison to BAT. The total time for all analyses is not compared, since

BAT is naturally at a disadvantage here, i.e., each analysis in BAT is phrased as a single for-comprehension and iterations over the data are re-performed by each analysis.

The runtime is measured – for both the SAE and BAT – from an end-user perspective, that is, from reading and parsing the respective bytecode until the analysis results are delivered. While the parsing time is not relevant for the individual analyses there are three main reasons to measure the time in this manner. First, the end-user perspective indicates how long an IDE user must wait until results are delivered and thus more aptly reflects the real-world scenario. Second, the SAE is tightly integrated into the parsing process, i.e., during parsing respective event are triggered that already perform computations and yield analyses results. In contrast BAT builds a complete in-memory representation and traverses the object structure to generate a list of results. These two schemes are architecturally different and naturally require different runtimes. BAT traverses structures twice, once during construction and once during the analysis, whereas the SAE only traverses the structure once during construction and then only propagates events. Separating the SAE from the parsing process would be unfair to the architecture of the SAE. For illustration, in this scenario one would simply read the whole structure into memory and then traverse the structure and trigger events to all base relations. However, this scenario includes a traversal of the entire dataset, whereas BAT will traverse only the parts of the dataset relevant for the respective analysis. Thus, the measurement would include more time for the SAE and less for BAT. Third and finally, the inclusion of the parsing process is the natural work that has to be re-performed by BAT, since it has no inherent support for incrementalization.

### 6.1.2 Incremental Runtime

The incremental runtime is compared to the non-incremental case. The runtime is compared for executing all analyses of a particular category together, e.g., all lightweight analyses. Naturally, the expectation is that incremental runtime should be much smaller compared to a full (non-incremental) computation. Hence, we evaluate the orders of magnitude saved by incremental view maintenance. Furthermore, the absolute values for the incremental runtime are used to evaluate, if incremental maintenance can really be considered real-time, or in which cases it fails.

A second comparison takes into account the `@LocalIncrement` optimization using two different scopes. First the class scope is measured, since this is the largest possible incremental scope and, hence, provides the largest possibility for saving memory, i.e., all analyses confined to a single class require no materialized memory. Yet the class scope also requires the largest amount of re-computations, i.e., all class members such as methods and fields are re-analyzed. The second scope are method declarations, thus methods in a recompiled class require a re-analysis only if they have really changed w.r.t. their instructions; other methods are not re-analyzed.

**Measured values for incrementalized analyses**

The incremental runtime is obtained by replaying a large set of change events recorded in a developer's IDE. A change event can include several classes that were re-compiled by the IDE. The measured time reflects how long it takes to incrementally maintain the analysis results for each complete change event, i.e., all contained classes. These values are compared to a non-incremental runtime obtained by using the SAE. The SAE is used (and not BAT), since BAT is unfit to run all analyses together, i.e., each analysis is phrased as a single for-comprehension and iterations over the data are re-performed by each analysis. All runtimes (incremental and non-incremental) are measured from an end-user perspective, that is, from reading and parsing the respective bytecode until the analysis results are delivered.

The second comparison (`@LocalIncrement` scopes) measures three values. First the time to incrementally maintain the whole change set by removing and adding all classes and all therein contained members (class scope). Second the time to incrementally update methods in the classes only if they have changed (method scope). The method scope requires some pre-computation, i.e., for each class the difference between the old methods and the new methods – in terms of instructions – must be computed. The runtime for the pre-computation is included in the measurements for two purposes. First, it naturally affects the overall runtime of using the method scope. Second, this work must be performed in general for any IDE integration where the IDE typically does not produce fine-grained events, such as changes for a single method.

### 6.1.3 Memory Materialization

This third step of the evaluation compares the amount of memory that is materialized for incremental maintenance to an approach where the base relations are fully materialized. The values for the incrementalized analyses are compared to the values for materializing each dataset entirely in memory. The latter measurement includes two values. One for simply reading all data into the SAE with materialized base relations and one for reading the data into a deductive logic engine (SWI-Prolog[3]). The latter is possible, since BAT is able to produce a logic representation of the Java bytecode. Hence, we can compare how a deductive database handles

[3] http://www.swi-prolog.org/

the large amount of data. The overall memory requirements for the complete materialized datasets are presented in the next section together with a general characterization of the datasets.

Note that we use SWI-Prolog and not XSB for our comparison. XSB would have been preferable, since it also provides incrementalization, whereas SWI does not. However, we were unable to use XSB for the larger datasets, since the engine ran out of memory. Furthermore, note that the values for materialized base relations in the deductive database are an optimistic measure. In particular, these systems would require additional indexing in order to be competitive for analyses that use joins. Hence, the evaluation actually favors the compared approaches, i.e., SWI. Nevertheless, a measurement with incorporating special indices for a fair comparison is hard, since – to be fair – the indices should be filtered indices such as the SAE uses. An indexing of the complete base relations is infeasible for all approaches (even for the SAE), since we have for example expressed joins over consecutive instructions (cf. the boxing analysis in Figure 5.19) and indexing 4 000 000+ instructions as found in our datasets requires enormous amounts of memory.

**Measured values for memory materialization of incrementalized analyses**

For the evaluation the materialized memory is measured in general for all analysis in a particular category. The memory comparison takes into account three values for the incrementalized analyses of the SAE. First the memory required due to the auxiliary data in the change propagation expressions defined in Sec. 3.4.3. Second, the memory required when the `@LocalIncrement` optimization is considered at the scope of entire classes and third the memory for the same optimization at the scope of methods. Where optimizations are possible the memory is measured for each optimized analysis in isolation, which allows us to identify which queries provided the most savings.

## 6.2 Overview of Measured Datasets

Throughout this chapter the following datasets are used for the evaluation. The comparison for non-incremental runtime and materialized memory are performed over two large datasets: (i) the runtime library of the JDK (version 7, windows 64 bit) and (ii) the Scala compiler (version 2.9.2). The runtime of the incremental maintenance is measured using a set of real-world changes performed by student developers that worked in our lab.

## 6.2.1 Non-Incremental Runtime and Memory Materialization

Both the JDK and the Scala compiler (Scalac) are large in terms of number of classes, methods and instructions. Table 6.1 summarizes the overall amount of data contained in each dataset, together with the time required for parsing the bytecode and the amount of memory required, if the entire dataset were materialized.

| Dataset | # elements | | | | parse | mat. mem. (MB) | |
|---|---|---|---|---|---|---|---|
| | classes | fields | methods | instr. | time (s) | SAE | SWI |
| JDK | 18 663 | 76 399 | 165 512 | 4 505 310 | 1.97 s | 466,8 | 1 160,0 |
| Scalac | 7 710 | 20 137 | 66 531 | 808 084 | 1.60 s | 92,5 | 295,4 |

**Table 6.1:** Overview of the benchmarked dataset

The number of elements ($2^{nd} - 5^{th}$ columns) and the time required for parsing ($6^{th}$ column) are rather self-explanatory. To obtain a measure of how in-memory databases with materialized base relations deal with the datasets, we have materialized all base relations in the SAE ($7^{th}$ column) and in the logic engine SWI ($8^{th}$ column). Note that the $7^{th}$ column is used solely for comparison and the memory is not required by the SAE. The value in the $8^{th}$ column, however, is a strict requirement when using SWI, i.e., the amount of memory is always materialized and permanently blocked on a developer's machine.

Note that the measurements for the materialized memory represent only the base relations. That means that we did not consider special indexing on the data. However, XSB – and other deductive databases – automatically index atoms based on their first parameter, e.g., the atom *parent(john, sally)* can be quickly retrieved for the value *john*, which is a good match for some queries, yet others would require a lookup for the value *sally*.

Overall, it is interesting to see that the logic representation consumes considerably more memory compared to the SAE. Partially this is due to the above mentioned indexing on the first parameter of an atom. Furthermore the SAE benefits from the object-oriented representation of data. For example, the same types – in terms of the same objects that represent them – can be referenced in many places in the bytecode, e.g., in method parameters as well as in method call instructions. The logic engine simply treats them as different atoms.

## 6.2.2 Incremental Runtime

The change set used to measure incremental runtime reflects 12+ hours of developer time on a small scale project that initially comprised 381 classes at the level of

Java bytecode, i.e., including inner classes, and increased to 389 classes during the course of development. Table 6.2 depicts the initial size of the project and the time required to parse the bytecode (in milliseconds).

| # elements | | | | parse time |
|---|---|---|---|---|
| classes | fields | methods | instr. | (ms) |
| 381 | 2 563 | 900 | 54 712 | 80.82 |

**Table 6.2:** Initial data of the incrementally analyzed project

Changes were recorded in the Eclipse IDE and a change event was recorded whenever Eclipse compiled a class. Compilation is performed when the developer saves the edited class(es) or when Eclipse performs re-builds of a project. A *change event* consists of all classes that are compiled together at a particular time, e.g., if Eclipse performs a full rebuild the change event consists of all classes in the project. The *change set* contains all recorded events and originally consisted of approx. 250 replayed events. However, we reduced the presented data, since such a large amount of values becomes unwieldy. First, we removed approx. 50% of the events, since they only perform a small line number change in a single class, i.e., due to changing comments. Second, we removed another 30% of the events that all perform only a single code change in a small method. All these events have a very low runtime; yet showing them all only clutters the presentation without giving further insights. We left representatives in the data, but were more interested in the performance when actually more computation takes place. The final change set features 52 events, of which two are large rebuilds by Eclipse, which were included intentionally to illustrate the limitations of incremental maintenance.

The data in the change set is characterized in tables 6.3 and 6.4. The first column in the tables sequentially numbers all change events. The numbering is later used as the x-axis for the measured incremental runtime. Hence, the x values can be used to put the measured runtime values into perspective with the necessary information from the change set.

Note that a real-world change set is more complex to characterize than artificially induced changes, such as “a single method was removed”. Hence, as a short overview, we can say that the change set contains a diverse range of events ranging from (i) changes in only one or two very small methods (e.g., $x = 16$), over (ii) a medium number of 10-30 methods (e.g., $x = 23$) and includes two large re-builds ($x = 17$, $x = 44$). The latter are not complete rebuilds of all classes. The analyzed project is subdivided into several Eclipse projects, which can each be rebuilt by themselves or in groups. Yet the change at $x = 17$ comes very close to a full rebuild.

| $x$ | $\Delta$ | class scope ($\circ$) |  |  |  | method scope ($\blacktriangle$) |  |  |  | $\blacktriangle/\circ$ |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  | $\Delta_m^{add}$ | $\Delta_m^{del}$ | $\Delta_i^{add}$ | $\Delta_i^{del}$ | $\Delta_m^{add}$ | $\Delta_m^{del}$ | $\Delta_i^{add}$ | $\Delta_i^{del}$ |  |
| 1 | 1 | 23 | 21 | 1176 | 1077 | 3 | 1 | 278 | 179 | 20% |
| 2 | 1 | 4 | 4 | 190 | 408 | 1 | 1 | 10 | 228 | 40% |
| 3 | 1 | 4 | 4 | 404 | 190 | 1 | 1 | 224 | 10 | 39% |
| 4 | 1 | 3 | 4 | 341 | 401 | 0 | 1 | 0 | 60 | 8% |
| 5 | 4 | 28 | 28 | 1303 | 1787 | 3 | 3 | 10 | 484 | 16% |
| 6 | 1 | 3 | 3 | 270 | 130 | 1 | 1 | 150 | 10 | 40% |
| 7 | 1 | 23 | 23 | 1026 | 927 | 2 | 2 | 109 | 10 | 6% |
| 8 | 1 | 23 | 23 | 1160 | 1026 | 1 | 1 | 134 | 10 | 7% |
| 9 | 1 | 4 | 3 | 408 | 270 | 2 | 1 | 288 | 150 | 64% |
| 10 | 1 | 21 | 21 | 2798 | 2750 | 2 | 2 | 956 | 908 | 33% |
| 11 | 2 | 25 | 23 | 1189 | 1160 | 3 | 1 | 68 | 29 | 4% |
| 12 | 2 | 23 | 25 | 1160 | 1189 | 1 | 3 | 29 | 68 | 4% |
| 13 | 2 | 25 | 23 | 1189 | 1160 | 3 | 1 | 68 | 29 | 4% |
| 14 | 2 | 23 | 25 | 1160 | 1189 | 1 | 3 | 29 | 68 | 4% |
| 15 | 1 | 22 | 24 | 1115 | 1184 | 0 | 2 | 0 | 69 | 3% |
| 16 | 1 | 2 | 2 | 63 | 16 | 1 | 1 | 57 | 10 | 83% |
| 17 | 312 | 2127 | 2079 | 36335 | 36680 | 52 | 4 | 2095 | 739 | 4% |
| 18 | 5 | 56 | 52 | 2529 | 2451 | 14 | 10 | 2059 | 2009 | 80% |
| 19 | 1 | 11 | 10 | 320 | 303 | 3 | 2 | 171 | 96 | 42% |
| 20 | 6 | 62 | 67 | 2743 | 2800 | 12 | 17 | 2105 | 2230 | 77% |
| 21 | 6 | 67 | 62 | 2992 | 2743 | 16 | 11 | 2193 | 1913 | 70% |
| 22 | 1 | 24 | 24 | 1033 | 1214 | 1 | 1 | 11 | 192 | 9% |
| 23 | 1 | 24 | 24 | 1188 | 1033 | 8 | 8 | 623 | 468 | 49% |
| 24 | 1 | 26 | 25 | 1227 | 1188 | 3 | 2 | 152 | 113 | 11% |
| 25 | 1 | 2 | 2 | 81 | 34 | 1 | 1 | 75 | 28 | 88% |
| 26 | 3 | 54 | 54 | 2058 | 2508 | 8 | 8 | 1124 | 1501 | 57% |

**Table 6.3:** Overview of the change set

In detail the tables contain the necessary information to characterize the workloads performed during each incremental change. The $2^{nd}$ column denotes the overall size of the change set in terms of number of classes recompiled by Eclipse. The $3^{rd} - 6^{th}$ columns characterize the workload when performing incremental maintenance at the class scope and the $7^{th} - 10^{th}$ columns include the same characterization for the method scope. The two scopes will in the following be denoted by $\circ$ (class scope) and $\blacktriangle$ (method scope). For each scope four values are listed:

$\Delta_m^{add}$ number of added methods

| $x$ | $\Delta$ | class scope (∘) | | | | method scope (▲) | | | | ▲/∘ |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $\Delta_m^{add}$ | $\Delta_m^{del}$ | $\Delta_i^{add}$ | $\Delta_i^{del}$ | $\Delta_m^{add}$ | $\Delta_m^{del}$ | $\Delta_i^{add}$ | $\Delta_i^{del}$ | |
| 27 | 3 | 54 | 54 | 2190 | 2058 | 8 | 8 | 1183 | 1124 | 53% |
| 28 | 1 | 24 | 24 | 1243 | 955 | 7 | 7 | 678 | 390 | 48% |
| 29 | 1 | 24 | 24 | 1273 | 1243 | 1 | 1 | 41 | 11 | 2% |
| 30 | 4 | 64 | 65 | 2797 | 2824 | 4 | 5 | 1023 | 1108 | 37% |
| 31 | 1 | 2 | 2 | 34 | 81 | 1 | 1 | 28 | 75 | 88% |
| 32 | 1 | 2 | 2 | 81 | 34 | 1 | 1 | 75 | 28 | 88% |
| 33 | 2 | 23 | 24 | 10 | 1263 | 23 | 24 | 10 | 1268 | 100% |
| 34 | 2 | 24 | 23 | 1263 | 10 | 24 | 23 | 1268 | 10 | 100% |
| 35 | 11 | 55 | 57 | 263 | 346 | 10 | 12 | 10 | 107 | 19% |
| 36 | 8 | 87 | 91 | 5741 | 6073 | 11 | 15 | 2776 | 3052 | 49% |
| 37 | 1 | 21 | 21 | 1119 | 1060 | 1 | 1 | 221 | 162 | 17% |
| 38 | 1 | 21 | 21 | 1060 | 1119 | 1 | 1 | 162 | 221 | 17% |
| 39 | 3 | 30 | 30 | 1685 | 1631 | 1 | 1 | 55 | 1 | 2% |
| 40 | 1 | 21 | 21 | 909 | 1068 | 1 | 1 | 11 | 170 | 9% |
| 41 | 1 | 21 | 21 | 1078 | 909 | 1 | 1 | 180 | 11 | 10% |
| 42 | 1 | 21 | 21 | 909 | 1078 | 1 | 1 | 11 | 180 | 10% |
| 43 | 1 | 21 | 21 | 1078 | 909 | 1 | 1 | 180 | 11 | 10% |
| 44 | 175 | 1265 | 1258 | 27355 | 27355 | 20 | 13 | 4208 | 4154 | 15% |
| 45 | 1 | 21 | 21 | 898 | 1078 | 1 | 1 | 37 | 180 | 11% |
| 46 | 5 | 11 | 15 | 10 | 192 | 11 | 15 | 10 | 192 | 100% |
| 47 | 5 | 15 | 11 | 342 | 10 | 15 | 11 | 342 | 10 | 100% |
| 48 | 1 | 19 | 21 | 717 | 898 | 2 | 4 | 84 | 265 | 21% |
| 49 | 3 | 29 | 30 | 1425 | 1630 | 1 | 2 | 10 | 282 | 9% |
| 50 | 3 | 29 | 29 | 1626 | 1425 | 1 | 1 | 201 | 10 | 7% |
| 51 | 11 | 90 | 90 | 1100 | 1749 | 19 | 19 | 10 | 717 | 25% |
| 52 | 1 | 19 | 19 | 717 | 10 | 19 | 19 | 717 | 10 | 100% |

**Table 6.4:** Overview of the change set (cont.)

$\Delta_i^{add}$ number of added instructions in the added methods

$\Delta_m^{del}$ number of removed methods

$\Delta_i^{del}$ number of removed instructions in the removed methods

The characterization is focused in methods and instructions, since these contribute the highest amount of work during incremental maintenance. The change set also features changes to fields, e.g., renaming a field. However, these changes are rare and not discussed in detail for brevity. Also for brevity all the above values include updates as deletions followed by insertions.

Finally, the difference between class scope (○) and method scope (▲) requires some discussion. At the class scope all methods – and all instructions – of all classes in a change event will be re-analyzed. Hence, the $3^{rd} - 6^{th}$ columns can be understood as the number of methods and instructions in the changed classes, which are naturally rather large numbers. At the method scope basically all methods that have not changed, i.e., have the same name, same signature and exact same instruction sequence, are not re-analyzed. Hence, the $7^{th} - 10^{th}$ columns can be understood as showing those methods that really have changed.

In the majority of cases the method scope contains only a small fraction of the changes found at the class scope. Hence, we can indeed expect that the method scope will perform better in terms of runtime for the incremental maintenance. However, there are cases where both scopes are equally sized, e.g., $x = 33$ and $x = 34$. Since the method scope must be pre-computed first these events can be expected to have a larger runtime in the method scope than in the class scope. For easy comparison of the method scope to the class scope, the last column in each table shows the ratio between the number of changes at the method scope and the number of changes at the class scope (▲/○). For example at $x = 1$ the method scope contains only 20% of the changes found at the class scope. The ratio is simply computed by taking the quotient over the sum of all changes in each scope, i.e., the sum over all additions and deletions of methods and instructions.

## 6.3 Lightweight Findbugs Analyses

In the following the measured lightweight analyses from the Findbugs tool are first briefly described w.r.t. their semantics – in terms of detected bugs – and the queries are characterized w.r.t. the underlying relations and used operators. An overview of the analyses is given in Table 6.5 and Table 6.6 for the manually the randomly selected queries respectively. Both tables are identical in their layout. The first column contains an abbreviated identifier for the query. The identifier is based on an original identifier that can be found in Findbugs[4]; the tool categorizes detected bug patterns by these identifiers. The Findbugs categorization already groups related bug patterns together, e.g., the prefix *CN* denotes bug patters related to violating Java's conventions for defining a `clone()` method. The presented abbreviated identifiers share the prefix found in Findbugs; a mapping to the original identifier can be found in Appendix A. The second column features a short description of the bug pattern detected by each analysis. The description provides an idea of the detected bug patterns. The third column lists the relations used by each analysis and the fourth column the operators used to formulate the queries based on these relations.

[4] http://findbugs.sourceforge.net/bugDescriptions.html

Note that the relations are the either the base relations (declared in Section 5.1) or the derived relations (declared in Section 5.3), i.e., the queries are not “broken down” into queries to only base relations. The derived relations are included since they provide standard abstraction (relations) used by many queries and provide no further optimization potential.

The queries for the selected analyses are not all discussed in detail. The listed bug patterns mostly represent bad practices or violations of Java conventions. The exceptions are the analyses *EQ*, *DMI* and *UR*, which detect situations that can lead to incorrectly executing code. An in-depth discussion was presented for four of the analyses; as examples for formulating static analyses in Section 5 (in the order *CI*, $FI_1$, $CN_3$, *BX*). Most of the remaining analyses have similar qualities, in the sense that they either (i) filter specific elements (as seen in *CI*, $FI_1$), (ii) find specific elements where some other element does exist or does not exist (as seen in $CN_3$) or find specific sequences of instructions (as seen in *BX*).

There are three notable queries (all found in Table 6.6) that perform different and/or more complex analyses compared to other queries. First, the $FI_2$ analysis counts the number of instructions in a method as a heuristic to find methods that only perform a super call. Second, the *SIC* analysis finds inner classes that do not require an reference to their enclosing object. The analysis combines checking that the outer **`this`** field is never read or written and that the constructor of the inner class does not use the parameter containing the instance of the outer class; except to initialize the outer **`this`** field. The latter is checked via a heuristic that counts accesses to the parameter, i.e., exactly one access is required to initialize the field. Third the *UR* analysis features many joins: the query must find a field that is declared and read in the same class ($\bowtie$); the reading method must override a super class method ($\bowtie$); the super constructor must call the reading method ($\bowtie$); and the super constructor must actually be called from a constructor in the subclass ($\bowtie$).

### 6.3.1 Non-incremental Runtime

The runtime for each single analysis is measured once for SAE and once for BAT. All measurements are performed using the JDK and Scalac libraries introduced in the basic setup (cf. Sec.6.2). The results for the manually and randomly selected Findbugs analyses are given in the tables 6.7 and 6.8. The two tables have the same layout where the first column contains the identifier of the respective analysis, the second to fourth column contain the results obtained for the JDK where the second column is the time measured for the SAE, the third column the time measured for BAT and the fourth column contains the ratio between the SAE and BAT times.

| Id | Description | Used Relations | Ops. |
|---|---|---|---|
| *CI* | Class is final but declares protected field (cf. Sec. 5.2) | `fieldDecl.` | $\sigma$ x 1 |
| $CN_1$ | Class implements Cloneable but does not define a clone() method | `methodDecl.` `subTypes` | $\sigma$ x 2 $\pi$ x 1 $\triangleright$ x 1 |
| $CN_2$ | clone() method does not call super.clone() | `methodDecl.` `instructions` | $\sigma$ x 3 $\pi$ x 1 $\triangleright$ x 1 |
| $CN_3$ | Class defines clone() but doesn't implement Cloneable (cf. Sec. 5.4) | `methodDecl.` `subTypes` | $\sigma$ x 2 $\pi$ x 1 $\triangleright$ x 1 |
| $CO_1$ | Abstract class defines covariant compareTo() method | `methodDecl.` `subTypes` | $\sigma$ x 3 $\pi$ x 2 $\ltimes$ x 1 |
| $CO_2$ | Covariant compareTo() method defined | `methodDecl.` `subTypes` | $\sigma$ x 2 $\pi$ x 2 $\ltimes$ x 1 |
| $DM_1$ | Explicit call to garbage collection; extremely dubious except in benchmarking code | `instructions` | $\sigma$ x 4 $\pi$ x 2 $\uplus$ x 1 |
| $DM_2$ | Call to the unsafe method runFinalizersOnExit | `instructions` | $\sigma$ x 2 $\pi$ x 1 |
| *EQ* | Abstract class defines covariant equals() method | `methodDecl.` | $\sigma$ x 1 |
| $FI_1$ | Finalizer should be protected by convention, not public (cf. Sec. 5.2) | `methodDecl.` | $\sigma$ x 1 |
| *IM* | Dubious catching of IllegalMonitorStateException | `code` | $\sigma$ x 1 $\pi$ x 1 |
| $SE_1$ | Class implements Serializable, but its superclass does not define a void constructor | `classDecl.` `methodDecl.` | $\sigma$ x 3 $\pi$ x 1 $\ltimes$ x 1 $\triangleright$ x 1 |
| *UuF* | Class declares an unused field, i.e., no read or writes are performed on the field | `fieldDecl.` `instructions` | $\sigma$ x 2 $\pi$ x 1 $\triangleright$ x 1 |

**Table 6.5:** Description and characterization of *manually* selected Findbugs analyses

| Id | Description | Relations | Ops. |
|---|---|---|---|
| *BX* | Primitive value is boxed then unboxed to perform primitive coercion (cf. Sec. 5.4) | `instructions` | $\sigma$ x 4<br>$\pi$ x 3<br>$\bowtie$ x 1 |
| *DMI* | Integer argument (with wrong bit layout) provided in call to Double.longBitsToDouble(long) | `instructions` | $\sigma$ x 4<br>$\pi$ x 3<br>$\bowtie$ x 1 |
| *DP* | Privileged method called outside of a doPrivileged block | `instructions`<br>`classDecl.` | $\sigma$ x 3<br>$\pi$ x 1<br>$\triangleright$ x 1 |
| *$FI_2$* | Finalizer does nothing but call superclass finalizer; determined via the existence of the call and a heuristic that counts the number of instructions | `instructions` | $\sigma$ x 3<br>$\pi$ x 1<br>$\ltimes$ x 1<br>$\gamma$ x 1 |
| *ITA* | Method performs inefficient call on toArray(Object[]) by providing zero-length array argument | `instructions` | $\sigma$ x 6<br>$\pi$ x 4<br>$\bowtie$ x 2<br>$\ltimes$ x 1 |
| *$MS_1$* | Field should be package protected, since it is never read from outside of the package | `fieldDecl.`<br>`instructions` | $\sigma$ x 4<br>$\pi$ x 1<br>$\triangleright$ x 1 |
| *$MS_2$* | Static public field should be final | `fieldDecl.` | $\sigma$ x 1 |
| *$SE_2$* | Non-serializable class has a serializable inner class | `classDecl.` | $\sigma$ x 3<br>$\bowtie$ x 1<br>$\triangleright$ x 1 |
| *SIC* | Anon. inner class can be refactored into a static inner class. Note requires naming the inner class | `classDecl.`<br>`fieldDecl.`<br>`instructions` | $\sigma$ x 5<br>$\pi$ x 2<br>$\bowtie$ x 1<br>$\triangleright$ x 2<br>$\gamma$ x 1 |
| *SW* | Certain swing methods needs to be invoked in Swing thread | `instructions` | $\sigma$ x 2<br>$\pi$ x 1 |
| *UG* | Declaration of synchronized set method, but unsynchronized get method | `methodDecl.` | $\sigma$ x 2<br>$\bowtie$ x 1 |
| *UR* | Uninitialized read of a field, due to access in overridden method that is called from super constructor | `methodDecl.`<br>`instructions` | $\sigma$ x 4<br>$\pi$ x 2<br>$\bowtie$ x 4 |

**Table 6.6:** Description and characterization of *randomly* selected Findbugs analyses

The final three columns contain the results obtained for the Scalac and mirror the previous three columns in their layout.

| *Id* | time (s) - JDK | | | time (s) - Scalac | | |
|---|---|---|---|---|---|---|
| | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ |
| *CI* | 2.1 | 2.2 | x1.0 | 1.6 | 1.6 | x1.0 |
| $CN_1$ | 2.2 | 2.2 | x1.0 | 1.7 | 1.6 | x1.1 |
| $CN_2$ | 2.8 | 2.2 | x1.3 | 1.9 | 1.6 | x1.2 |
| $CN_3$ | 2.3 | 2.2 | x1.0 | 1.7 | 1.6 | x1.0 |
| $CO_1$ | 2.3 | 2.2 | x1.0 | 1.7 | 1.6 | x1.1 |
| $CO_2$ | 2.3 | 2.2 | x1.0 | 1.8 | 1.6 | x1.1 |
| $DM_1$ | 2.9 | 2.7 | x1.1 | 1.9 | 1.8 | x1.1 |
| $DM_2$ | 2.7 | 2.5 | x1.1 | 1.8 | 1.7 | x1.1 |
| *EQ* | 2.0 | 2.2 | x0.9 | 1.7 | 1.6 | x1.1 |
| $FI_1$ | 2.1 | 2.2 | x0.9 | 1.6 | 1.6 | x1.0 |
| *IMSE* | 2.1 | 2.5 | x0.8 | 1.7 | 1.7 | x1.0 |
| $SE_1$ | 2.2 | 2.2 | x1.0 | 1.7 | 1.6 | x1.1 |
| *UUF* | 3.2 | 2.8 | x1.2 | 2.0 | 1.7 | x1.2 |

**Table 6.7:** Non-incremental runtime comparison of manually selected Findbugs analyses

Overall, we can establish that the runtime performance does not differ much for the SAE and BAT when running the analyses. The ratio between the SAE and BAT runtime is ±10% in the vast majority of the analyses. Some analyses even perform better in the SAE, especially those that benefit from indexing as for example the *BX* analysis, which joins consecutive invokespecial and invokevirtual instructions (cf. Sec. 5.4). Other analyses that fall into this category are *DMI*, *ITA*, *UG* and *UR*. Although it has to be noted that indexing is not overly prominent due to the fact that analyses in BAT also make use of hash tables for frequent data lookups (as discussed in Sec. 6.1.1). If BAT were to use an approach purely based on iterating over the data, i.e., without such optimizations, the benefit of indexing in the SAE would be more prominent.

Many of the manually selected analyses are very short running and the parsing time dominates the results quite heavily. For example, the analyses *CI*, $CO_*$ and *EQ*

| *Id* | time (s) - JDK | | | time (s) - Scalac | | |
|---|---|---|---|---|---|---|
| | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ |
| *BX* | 3.1 | 4.6 | x0.7 | 1.9 | 2.0 | x0.9 |
| *DMI* | 3.0 | 4.6 | x0.6 | 1.8 | 2.2 | x0.8 |
| *DP* | 2.8 | 2.9 | x1.0 | 1.8 | 1.8 | x1.0 |
| $FI_2$ | 3.0 | 2.2 | x1.4 | 1.8 | 1.6 | x1.1 |
| *ITA* | 3.6 | 4.4 | x0.8 | 2.0 | 2.1 | x0.9 |
| $MS_1$ | 3.2 | 4.0 | x0.8 | 1.9 | 1.8 | x1.1 |
| $MS_2$ | 2.1 | 2.2 | x0.9 | 1.7 | 1.6 | x1.1 |
| $SE_2$ | 2.3 | 2.2 | x1.0 | 1.8 | 1.6 | x1.1 |
| *SIC* | 3.2 | 3.8 | x0.8 | 1.9 | 1.8 | x1.1 |
| *SW* | 2.8 | 2.4 | x1.2 | 1.8 | 1.7 | x1.1 |
| *UG* | 2.2 | 2.3 | x0.9 | 1.7 | 1.6 | x1.1 |
| *UR* | 4.0 | 4.6 | x0.9 | 2.1 | 1.9 | x1.1 |

**Table 6.8:** Non-incremental runtime of comparison of randomly selected Findbugs analyses

require almost no time for the actual computation of results. Recall that parsing time for the JDK was 2 seconds and for the Scalac 1.6 seconds. The computations of analysis results are truly performed in a manner of tens of milliseconds. One might argue that the ratios are distorted by this, since the overall time is dominated by parsing and not the analysis. Nevertheless, the comparison between the SAE and BAT still shows when the SAE performs better or worse, e.g., the $CO_*$ values for the JDK are still marked with 0.1 seconds overhead. Also, the argument swings both ways, i.e., in cases such as *EQ* where the SAE performs better the comparison would yield a higher ratio for the SAE.

Finally, it is important to note that the ratios can differ between the analyzed datasets. The difference stems from the fact that the actual computation is not linear in size, but depends also on the makeup of the data. For example, the instructions that potentially have boxing errors *BX* are nearly non-existent in the Scala compiler. Note that this does not mean that there are far more results in the JDK. There are 3 bugs in the JDK versus 0 in the Scala compiler. The issue is simply that less computation was performed in the Scala compiler to obtain this result.

The incremental runtime for the lightweight analyses is depicted in Figure 6.2. The measured times for each event are plotted on a logarithmic scale y-axis that shows the time taken to compute the incremental update in milliseconds. The x-axis shows the change events in the order in which they appeared when the developer made changes in the IDE. For simplicity the events are numbered from 1 to 52, though the time between events naturally differs. The value at $x = 0$ denotes the initial computation performed on the entire code base (right after starting the IDE).

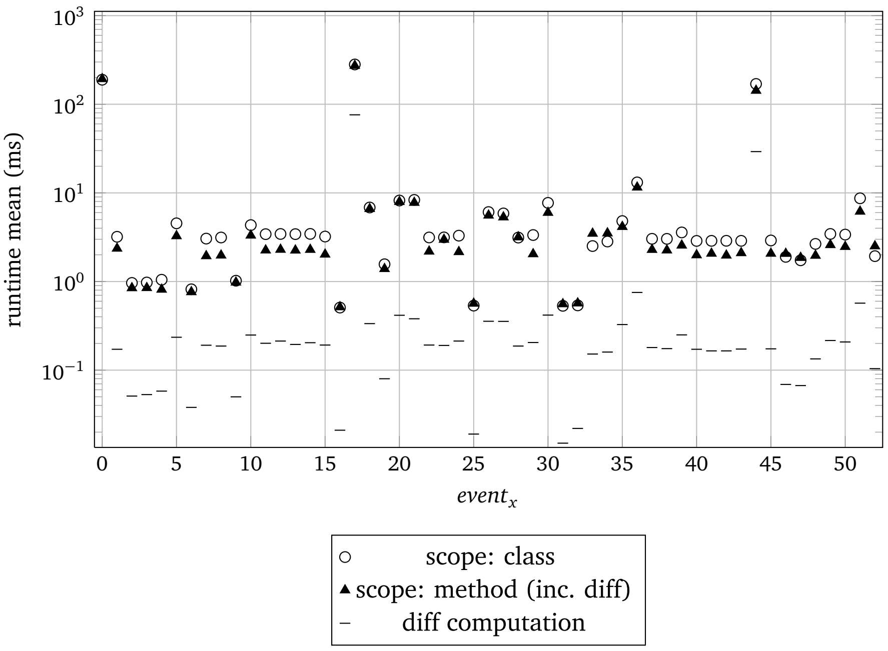


**Figure 6.2:** Incremental runtime of lightweight analyses

The majority of the incremental computations is performed between 1 and 10 milliseconds and is thus two orders of magnitude faster than the initial computation. Note that we omitted 80% of the data, since it all falls into the category of the value at $x = 25$ or below. However, that really places the vast majority in the complete dataset in a range below 1 millisecond. The two major outliers in the range above $10^2$ milliseconds are the re-builds initiated by the IDE. As discussed in Sec. 6.2 both are not complete rebuilds. The first re-builds approx 3/4 of the classes and the second approx. 1/2. They fulfill the expectations on the incrementalization in the sense that incremental updates for half of the code ($x = 44$) should be in the range of the runtime for the initial code base, since 1/2 of the time is spent for maintaining deletions and the 1/2 of the time is spent for insertions.

Between the class scope and the method scope for the `@LocalIncrement` optimization there is no big difference. The method scope performs better in general, which is to be expected. The general trend is very clear, and the reason is that many analyses filter out methods early or do not require extensive computations on each method in a class. Hence, the results are easily maintained by the class scoped incrementalization while the method scoped incrementalization must ascertain that a method has indeed not changed, which is only marginally faster to compute. There are three outliers that place the method scope above the class scope; $x = 33$, $x = 34$ and $x = 52$. The outliers are also not unexpected and exist due to the fact that the time to deduce that the method is irrelevant for the analysis is smaller than the time to deduce that the method has changed and is irrelevant.

### 6.3.3 Memory Materialization

The presented analyses exhibit a diverse range of queries that have very different requirements on the materialized memory. Tables 6.9 and 6.10 provide a classification of the queries w.r.t. their self-maintainability and the potential for the `@LocalIncrement` optimization (cf. Sec 4.2.5). Table 6.9 features the manually selected queries and Table 6.10 the randomly selected queries. Both tables list the queries by their identifiers in the columns, and show the self-maintainability ($1^{st}$ row), the potential to use the `@LocalIncrement` annotation optimization ($2^{nd}$ row). Furthermore the potential for a subquery sharing optimization is shown in the $3^{rd}$ row. The latter is included in the evaluation to obtain a measure on how important subquery sharing is (overall and for memory sharing) and will be discussed separately.

Of all the manually and randomly selected queries, there are 9 fully self-maintainable queries. Which means 36% of all sampled queries (i.e., roughly one third) consume no memory per construction. The increment local optimization is applicable to 8 queries in total. In these queries there are three main types of join comparisons that can benefit from increment locality. The first category is joins between method declarations and therein contained instructions, as for example the $CN_2$ query joins[5] `clone()` declarations and call instructions inside that method. The second category is joins on instruction sequences, as for example in the *BX* query,

[5] The join in $CN_2$ is performed as part of the existential quantification.

|  | *CI* | $CN_1$ | $CN_2$ | $CN_3$ | $CO_1$ | $CO_2$ | $DM_1$ | $DM_2$ | *EQ* | $FI_1$ | *IM* | $SE_1$ | *UuF* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Self-Maint. | ✓ |  |  |  |  |  | ✓ | ✓ | ✓ | ✓ | ✓ |  |  |
| `@LocalInc.` |  |  | ✓ |  | ✓ | ✓ |  |  |  |  |  |  |  |
| Subquery-S. |  | ✓ | ✓ | ✓ | ✓ | ✓ |  |  |  | ✓ |  | ✓ |  |

**Table 6.9:** Self-maintainable and optimizable queries (manually selected)

|  | *BX* | *DMI* | *DP* | $FI_2$ | *ITA* | $MS_1$ | $MS_2$ | $SE_2$ | *SIC* | *SW* | *UG* | *UR* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Self-Maint. |  |  | ✓ |  |  |  | ✓ |  |  | ✓ |  |  |
| `@LocalInc.` | ✓ | ✓ |  |  | ✓ |  |  |  |  |  | ✓ | ✓ |
| Subquery-S. |  |  |  | ✓ |  | ✓ | ✓ | ✓ |  |  |  |  |

**Table 6.10:** Self-maintainable and optimizable queries (randomly selected)

where instructions are joined with to the next instruction by their sequence index. The final category is joins between elements inside a single class. The *UG* query for example joins setter and getter methods inside the same class and looks for pairs where setters are synchronized but getters are not.

### Subquery sharing

As part of the evaluation of the lightweight analyses we explicitly looked for the amount of queries that can benefit from subquery sharing. The subquery sharing optimization is enabled by the fact that many of the analyses are performed on the same elements, yet with small differences in the bug patterns. The queries for the common elements are typically trivial and, hence, developers might declare them redundantly. For example the $FI_*$ queries look for methods that have the name "finalize" (cf. Sec. 5.2 for the complete query). In contrast the analyses in other categories depend on some complex queries that are in general not repeated, but declared as a re-usable view.

There are 11 queries in total that share a subquery. In Tables 6.9 and 6.10 sharing is counted only if the analyses exhibit some commonality besides using the same derived relations (cf. Sec. 5.3), i.e., for this evaluation the focus is to find whether queries elicit some commonality, besides using common abstractions such as the `subTypes` relation. The Findbugs analyses indeed elicit further commonality. In the sampled analyses the commonality is eminent between analyses that look for similar bug patterns. In essence one can already deduce subquery sharing from the identifiers, e.g., all queries labeled $CN_*$ share a common subquery. This is not surprising, since we adopted the identifiers from Findbugs, which categorizes queries quite well by the detected bug patterns. Most analyses share a single subquery. The

$CN_1$ and $CN_3$ analyses are the only analyses that share two subqueries, i.e., one query for all subtypes of `Clonable` and one for all classes that implement `clone()`.

Subquery sharing can also affect the materialized memory. However, only the subqueries shared by the $CN_*$, and the $CO_*$ queries consume memory. The $FI_*$, $MS_*$ and $SE_*$ analyses share a common selection of elements, which is self-maintainable, e.g., the $FI_*$ analyses share the selection of all `finalize()` method declarations.

### Overall materialized memory

The comparison between the memory required by the SAE vs. the materialization of base relations and vs. the materialization in the logic engine is summarized in Table 6.11. The table contains one row for each dataset and the memory materialized by the SAE without `@LocalIncrement` optimization ($2^{nd}$ column) as well as the materialized memory with the optimization ($5^{th}$ column). All measurements include other optimizations such as subquery sharing. The `@LocalIncrement` optimization is considered at the best level, i.e., the class scope. A differentiation of scopes for each optimized query is discussed shortly. The measured values are compared to materializing all data in the base relations in the SAE (columns 3 and 5), as well as to the memory for materializing base relations in the logic engine (columns 4 and 6) – see Sec. 6.2 for the actually compared values.

Overall the memory used by the SAE is already very low. Approximately 1/4 of the data is materialized, which means the analyses make do with only 1/4 of all actually available information. This reduction in memory required no special optimization, save pushing selections early into the evaluation, which was performed by construction in the IQL queries. Note again that the analyses were written with the whole dataset in mind. Hence, by construction of the queries the SAE has ruled out 3/4 of the data as unnecessary without the end-user having to think about what to materialize. The latter would have been very difficult in the case of the lightweight analyses, since these use very diverse datasets. In terms of the `@LocalIncrement` optimization we can reduce the data by approx. another 50%. This leaves a very low memory overhead of $10 - 14\%$ compared to the materialization of the complete base relations. In comparison to the SWI system the SAE naturally performs even better leaving us with $4 - 5\%$ of the in-memory data required by the logic engine.

Note again, that the comparison is actually in favor of the materialized base relations and the logic engine, since we did not measure indexed data for these cases. In other words the SAE already consumes less memory and is completely indexed where necessary.

| JAR | Mem. Plain (MB) | | | Mem. Opt. (MB) | | |
|---|---|---|---|---|---|---|
| | SAE | $\frac{\text{SAE}}{\text{Base}}$ | $\frac{\text{SAE}}{\text{SWI}}$ | SAE° | $\frac{\text{SAE}^\circ}{\text{Base}}$ | $\frac{\text{SAE}^\circ}{\text{SWI}}$ |
| JDK | 118.71 | x0.25 | x0.10 | 50.10 | x0.10 | x0.04 |
| Scalac | 25.15 | x0.27 | x0.09 | 13.16 | x0.14 | x0.05 |

**Table 6.11:** Mat. memory of lightweight analyses with and w/o `@LocalIncrement`

| *Id* | Mem. JDK (MB) | | Mem. Scalac (MB) | |
|---|---|---|---|---|
| | Plain | Opt | Plain | Opt |
| $CN_2^\blacktriangle$ | 1.66 | 0.03 | 1.16 | 0.02 |
| $CO_1^\blacktriangle$ | 15.47 | 13.04 | 7.18 | 5.72 |
| $CO_2^\blacktriangle$ | 15.37 | 13.30 | 7.16 | 5.73 |
| $BX^\blacktriangle$ | 3.70 | 0.03 | 1.22 | 0.04 |
| $DMI^\blacktriangle$ | 2.05 | 0.02 | 1.22 | 0.01 |
| $ITA^\blacktriangle$ | 46.53 | 12.90 | 8.45 | 5.80 |
| $UG^\circ$ | 7.65 | 0.07 | 1.17 | 0.05 |
| $UR^\circ$ | 49.9 | 26.96 | 12.13 | 6.66 |

**Table 6.12:** `@LocalIncrement` optimization by query

### Closeup of the optimization

To see how the different scopes affect the memory materialization Table 6.12 lists the detailed savings for each of the optimized queries. The table lists the optimized queries by their id ($1^{st}$ column) together with an annotation whether the scope required is the class (∘) or the method scope (▲). The $2^{nd}$ and $3^{rd}$ columns show the unoptimized and optimized memory requirements for the JDK and the $4^{th}$ and $5^{ht}$ columns for the Scalac respectively.

Overall only two analyses require the class scope. Note that classes are the bigger scope and include the method scope, which means the other six analyses are optimized in both scopes. At the class scope, the *UG* query correlates synchronized and unsynchronized methods. Thus the analysis is basically for free after the optimization. We have measured small residues of memory, yet they have error margins that place them around 0. The *UR* analysis finds uninitialized field reads from fields in the same class, hence these read instructions and fields are joined for free. The analysis performs further correlations that are not for free, i.e., to methods in the super class, yet the optimization is the second largest contribution to the overall savings by a single query.

Most optimized queries contribute small amounts to the total saving discussed earlier. The *$CN_2$*, *BX* and *DMI* queries are basically for free (as the *UG*), yet the amount of instructions that contribute to these bugs, is not very large. The *$CO_*$* queries require a large amount of memory for the materialization of the type hierarchy, and apart from that also did not require much memory for the materialization.

The largest amount of saved memory is contributed by the optimized *ITA* query, which makes sense in the light of the instructions that are searched by this analysis. It searches for patterns of `toArray()` calls, paired with array creations and `iconst_0` which loads the constant 0 onto the stack. Especially in the JDK these instructions are often found, hence the large savings. In the Scalac these instructions are not as dominant and there are only smaller savings.

In total all queries add up to a respectable sum as seen in Table 6.11. Note that the total saving is smaller than the sum over the individual savings in Table 6.12. This is natural, since each individual measurement includes elements, such as class and methods declarations that may be materialized by other queries.

## 6.4 Metrics

In the following the measured analyses from the Metrics tool are described w.r.t. their semantics – in terms of measured values – and the queries are characterized w.r.t. the underlying relations and used operators. An overview of the metrics is given in Table 6.13 The table is identical in layout to the description in Sec. 6.3.

The first two analyses (*CA*, *CE*) perform a high amount of filtering and building of unions as a result of computing the dependencies between different classes. Dependencies stem in the majority from instructions, e.g., method calls, but to a smaller degree also from signatures, i.e., method parameters or implemented interfaces. The latter are both unnested first ($\mu$) to obtain a uniform flattened representation of all dependencies. The heavy use of unions combines all different kinds of dependencies into a single relation (which is also reused across both analyses). Both metrics consider dependencies at the level of classes and multiple dependencies at lower levels, e.g., multiple method calls, are subsumed, hence, the duplicate elimination ($\delta$). The *$LCOM^*$* metric has to perform three separate aggregations and join them together, since the actual value of the metric is based on (i) the total number of fields, (ii) the total number of methods and (iii) the number of methods that actually use a field. The *DIT* metric is the simplest and just counts how many edges the transitive closure to `java.lang.Object` has. This metric only

| Id | Description & (Findbugs Id) | Used Relations | Ops. |
|---|---|---|---|
| *CA* | *Afferent Coupling* determines for a given package the number of classes outside the package that have dependencies to classes inside the package | `classDecl.` `methodDecl.` `fieldDecl.` `instructions` | $\sigma$ x12 $\pi$ x 3 $\uplus$ x13 $\mu$ x 2 $\delta$ x 1 $\gamma$ x 1 |
| *CE* | *Efferent Coupling* determines for a given package the number of classes inside the package that have dependencies to classes outside the package | `classDecl.` `methodDecl.` `fieldDecl.` `instructions` | $\sigma$ x12 $\pi$ x 3 $\uplus$ x13 $\mu$ x 2 $\delta$ x 1 $\gamma$ x 1 |
| *LCOM** | The (lack of) cohesiveness of a classes methods, determined by measuring the overlap of class attributes used by all methods in a class. The measured value is normalized in the interval [0..1] and signifies *lack of cohesion*. Hence, a value close to 1 suggests that the class is better split into two (or more) cohesive classes. | `methodDecl.` `fieldDecl.` `instructions` | $\sigma$ x 3 $\pi$ x 2 $\uplus$ x 1 $\bowtie$ x 2 $\gamma$ x 3 |
| *DIT* | Depth of inheritance tree for declared classes | `classDecl.` | *TC* x1 $\gamma$ x 1 |

**Table 6.13:** Description and classification of selected metrics

takes into account direct class inheritance, i.e., interfaces are not considered, which is the definition used in the Metrics tool.

## 6.4.1 Non-Incremental Runtime

The results for the evaluated metrics are shown in Table 6.14. The table follows the layout discussed in Sec. 6.3. Overall, the runtime performance between the SAE and BAT is more distinguished than in the lightweight analyses. BAT outperforms the SAE by a factor of 1.4 to 2.3 for the more complex metrics. The simple metric *DIT* has the same runtime in both systems.

The reason for the discrepancies is that SAE has to maintain groupings and update the results of aggregated values many times. For example, the *CA* analysis groups all source code dependencies by their target package. Each event that adds a respective dependency must find the respective group and update the value of the aggregation.

| *Id* | time (s) - JDK | | | time (s) - Scalac | | |
|---|---|---|---|---|---|---|
| | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ |
| *CA* | 7.0 | 3.0 | x2.3 | 2.8 | 1.8 | x1.5 |
| *CE* | 6.8 | 3.0 | x2.3 | 2.7 | 1.9 | x1.5 |
| *DIT* | 2.2 | 2.2 | x1.0 | 1.7 | 1.7 | x1.0 |
| *LCOM** | 5.0 | 2.6 | x1.9 | 2.4 | 1.7 | x1.4 |

**Table 6.14:** Non-incremental runtime comparison of metrics

In comparison, BAT only computes a set of of dependencies once, traverses them all and categorizes them by their target package during the traversal. Hence, there are no updates to the aggregated values. This behavior is most eminent in the $CA$ and $CE$ aggregations since here groups are maintained by packages, of which there are few and thus many updates must be made to the aggregated value of a single group. The $LCOM$ metric groups members inside a class, which means more groups and less updates to the aggregation value of a single group. Finally the $DIT$ metric groups classes in the inheritance hierarchy, where there are not as many elements to group – classes contribute the smallest number of elements – and groups tend to be small, hence no apparent difference in runtime performance arises.

### 6.4.2 Incremental Runtime

The measurements for the metrics (cf. Figure 6.3) are almost indistinguishable from the measurements of the lightweight analyses (cf. Figure 6.2) in terms of incremental runtime. Overall the runtime actually slightly decreased for the metrics. The decrease seems intuitive, since we only considered 4 metrics and 25 lightweight analyses. Yet, it is interesting to note that the decrease is not very prominent. This means that a single metric is a rather expensive computation, and we can search for a larger number of lightweight bug patterns in the same amount of time. This expensiveness is also eminent in the definitions of the metrics (cf. Figure 6.13), e.g., the two metrics *CA* and *CE* are based on a rather complex query to computes the dependencies.

Between the class and the method scope a more prominent differentiation is visible compared to the lightweight analyses. This is expected, since metrics consider every single method in their computations. In contrast, the majority of the lightweight analyses is concerned with very specific methods, e.g., `clone()`.

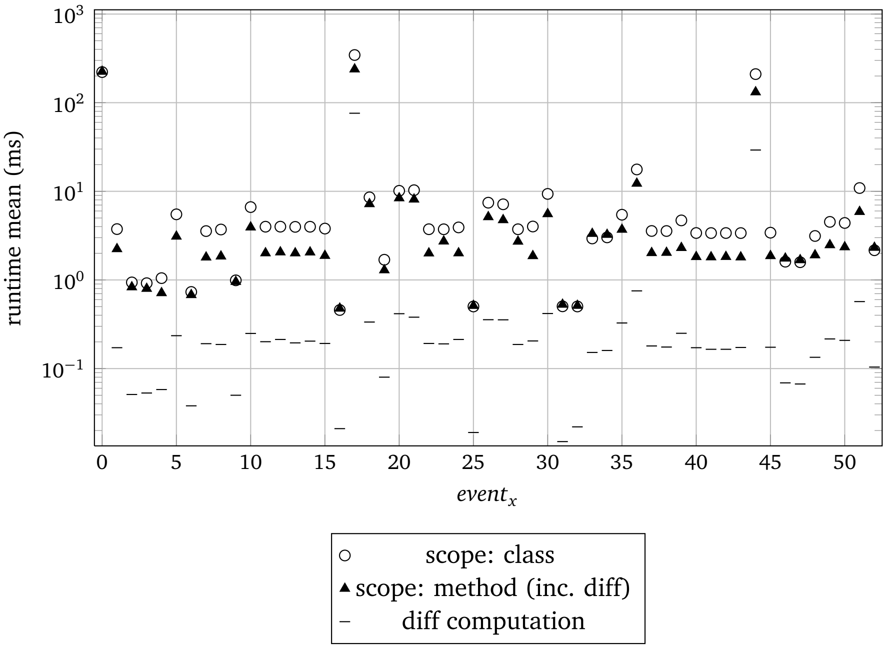


**Figure 6.3:** Incremental runtime of metrics

## 6.4.3 Memory Materialization

The presented metrics are in general very efficient w.r.t. memory materialization (shown in Table 6.15). Even without further optimizations only 7 - 18 % of the complete data is used. The reason for the overall low memory requirement is that much of the data can be efficiently filtered out early. For example, the *CA* and *CE* metrics analyze all dependencies, yet the metrics only count dependencies between packages, hence a vast majority of dependencies is filtered by selections. The JDK requires a markedly lower fraction of the data than the Scalac. The reason is that the metrics are dominated by materializations of all class data, yet the Scalac features a much larger amount classes compared to other elements in the data set, such as instructions. In other words, the Scalac contains a little less than half the amount of classes as the JDK, yet it contains less than a fourth of the number of instructions. Hence, the ratio to the overall amount of elements in each data set is different.

Of the presented metrics only *LCOM** has the potential for the `@LocalIncrement` optimization. The *LCOM** metric only computes values confined to a single class, e.g.,

| JAR | Mem. Plain (MB) | | | Mem. Opt. (MB) | | |
|---|---|---|---|---|---|---|
| | SAE | $\frac{\text{SAE}}{\text{Base}}$ | $\frac{\text{SAE}}{\text{SWI}}$ | SAE° | $\frac{\text{SAE}^\circ}{\text{Base}}$ | $\frac{\text{SAE}^\circ}{\text{SWI}}$ |
| JDK | 32.47 | x0.07 | x0.03 | 22.80 | x0.05 | x0.02 |
| Scalac | 16.69 | x0.18 | x0.06 | 11.44 | x0.12 | x0.04 |

**Table 6.15:** Mat. memory of metrics with and w/o `@LocalIncrement`

counting methods that use fields declared in the same class. Hence, the optimization can be performed at the class scope (∘) and the entire metric requires no memory after the optimization. Using the optimization approx. 1/3 of the memory can be saved.

## 6.5 Intra-Procedural Findbugs Analyses

The analysis used here is a stack analysis similar to the one used in Findbugs. Stack analysis is a data flow analysis that computes the effect of a Java bytecode program on the stack and the local variables. The results of the stack analysis are uniformly represented as states. A state defines the makeup of the stack and the local variables at any given point of the program execution, i.e., at the point of any given instruction. The states of the stack analysis are abstractions of the real values in the execution. Every state uses "items" on the stack an in the local variables that store type information (e.g. int, String etc.) and additional information such as creation location (the instruction at which the item has been created). The stack analysis relies on a control flow analysis that determines for each instruction the set of predecessor instructions that are executed before. In essence the analysis determines the edges in the control flow graph and makes them available as a relation.

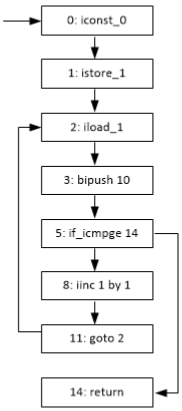
### 6.5.1 Control Flow Graph

The control flow graph is computed quite easily. There is a set of specific instructions in Java bytecode that manipulate the control flow, i.e., conditionals, gotos, or table lookups (switch case). Note that goto is a construct on the bytecode level, whereas conditionals and table lookups have a corresponding statement in Java source code.

To exemplify the control flow graph Figure 6.4 shows a simple Java Program (6.4a) and the corresponding bytecode and control flow (6.4b). There are two instructions in the bytecode that manipulate the control flow. The `if_cmpge`

```
1  int a = 0;
2  while( a < 10 ){
3    a++;
4  }
```

**(a)** Example Java Program



**(b)** Java bytecode and Control Flow of the Program

**Figure 6.4:** Example of a control flow graph

instruction corresponds to the test in the head of the while loop (line 2 on the Java side), yet the condition is phrased negatively, i.e., the loop will stop if the value of `a` is greater or equal to 10. The `goto` marks the end of the loop and jumps back to the beginning. The important thing to note is that some instructions have multiple predecessors, e.g., the `iload_1` instruction at $pc = 2$ and some have multiple successors, i.e., the `if_cmpge` instruction.

The computation of the control flow is mostly straightforward, an excerpt is depicted in Figure 6.5. The computation selects different instructions that manipulate the control flow. For example the goto instructions (i.e., unconditional branches) (line 1). The selections are all based on the type of the particular instructions (line 3). All control flow instructions are joined with other instructions in the code (line 7) under the condition that (i) they are in the same method (line 8 and (ii) the target pcs of the control flow are the pcs of the particular instructions (line 9). For example in the program depicted in Figure 6.4, the join shown below finds an edge from the `goto` at $pc = 11$ to the `iload_1` at $pc = 2$. The complete control flow graph (cfg) is the union over several such joins, i.e., for all the different kinds of branch instructions together with all regular instructions, i.e., instructions that are simply executed in the sequence in which they appear in the bytecode. Note that the complete control flow is slightly more complicated, since exceptions are taken into account. Exceptions basically break the regular instructions into smaller blocks that can jump to the exception handler, i.e., the catch block. However, their treatment just adds a further sequence of joins and regular instructions are then filtered, to ascertain they are not part of exceptional control flow. The important thing to note is that all these concepts are well expressible in the SAE.

```
val unconditionalBranches =
  SELECT (asGotoInstruction) FROM (instructions) WHERE
    (_.isInstanceOf[GotoBranchInstruction])

val cfg =
  SELECT (controlFlowEdge) FROM
    (unconditionalBranches, instructions) WHERE
      (declaringMethod === declaringMethod) AND
      (branchTargetPc ===  instructionPc)
  UNION ALL
  ...
```

**Figure 6.5:** Excerpt of the control flow graph computation

### 6.5.2 Dataflow Analysis

The computation of the dataflow is in essence a fixpoint computation over the possible states for each instruction. That means for each instruction we take the state of the previous instruction in the control flow graph and calculate the effect of the instruction on that state. Since some instructions have multiple predecessors, i.e., jumps from instructions further down in the cfg, this process is repeated until no new states can be inferred.

#### States and instruction effects

States in our analysis represent triples of (i) the current instruction, (ii) the stack before the current instruction is executed and (iii) the local variables before the instruction is executed. The stack and local variables each contain a list of items that contain type information and the information at which program counter the item was created. Given an edge in the control flow graph we can simply deduce the state before the next instruction in the graph, by applying the changes from the current instruction. To illustrate the application of state changes, Figure 6.6 depicts a very small excerpt of how new states are computed. In the example, the current instruction pushes the constant zero on the stack. Hence, the state before the next instruction (line 4) is the old stack with the pushed constant (line 6) and an untouched copy of the local variables (line 8). The item on the stack contains the type information (`IntegerType`) as well as the pc at which it was created. The complete set of cases that are transformed is rather large, since it must deal with all instructions found in the Java Language Specification, yet all cases follow the principle of the example.

```
 1 def nextState(edge: ControlFlowEdge, oldState: State): State =
 2   edge.current.instr.opcode match {
 3     case 0x03 => //ICONST_0
 4       State(edge.next,
 5             oldState.stack.push(
 6               Item(IntegerType, edge.current.instr.pc)
 7             ),
 8             oldState.variables)
 9     ...
10   }
```

**Figure 6.6:** Excerpt of computing instruction effects on the state

**Upper bounds for states**

It is important to note that the amount of states can be become quite large; hence, for efficiency we need a notion of combining states to an upper bound. The large amount of states is due to an exponential growth in the possible paths through a method. For example, an if-instruction will have two possible control flows (i.e., if and else branch). Yet a method with 2 sequential if-instructions will have two possible states after the first branch and both can enter the second if-instruction. Hence, there will be 4 possible states after the second branch. A repeated computation of possible states multiplies exponentially for such possible branches. To alleviate this problem states are combined and an upper bound is computed for the data in the states.

Upper bounds are typically formulated as part of the domain (i.e., the abstraction of the state) on which a particular dataflow analysis operates. For example a standard live variables analysis builds a set of variables that is used at a particular point in the program. The important fact is that, if multiple possible live variable sets reach an instruction (i.e., due to branching) they are combined – to an upper bound – via set union. Thus, for a given instruction the live variables analysis can be used as a single input and a single output set can be computed. Note that live variables are determined in a backward fashion, i.e., from the end of a program to the start and in general treat blocks of instructions, but the argument remains the same.

For the purpose of our stack analysis we use a simple approach to upper bounds, which is sufficient to detect the bugs found by Findbugs (at least for the sampled analyses), yet is in general not very precise. A complete description of the upper bound is omitted for brevity, yet we give a brief description. In short, if an instruction has multiple predecessor states we build an upper bound for the local variables of these states and stacks in the following way: Given two lists of items that represent the current local variables (or stacks), we compare each two items – at the same index in the list – w.r.t. their values for their type information and their program counter. If both are equal, the upper bound is an item with the same values. In case of differing type information, the result contains a bound of `AnyRef` for reference types (e.g., `String`) and of `Any` for value types (e.g., `int`). In the case of differing pcs, their result contains a value of $-1$ (for undefined). In terms of precision this means that states are only precise if their storage is performed in a sequence of instructions with a single entry and a single exit point (also called a basic block).

While some precision is lost, the important part for this evaluation was to declare an intra-procedural dataflow analysis that is comparable to standard analyses in terms of the performed operations, i.e., computing an upper bound. For example, a standard single static assignment form (SSA) also computes a – different – upper bound. SSA uses an abstraction termed $\Phi$-function to denote variables that were modified in different control flows. The $\Phi$-function can be understood as a different form of upper bound that does not loose information. Yet, the basic step that must be performed by the analysis is the same, i.e., if at a given point in the execution two possible states arise, an upper bound must be computed.

### Fixpoint computation

The formulation of the fixpoint for the dataflow is very short and succinct. However, the computation of upper bounds does complicate matters. Figure 6.7 declares the dataflow analysis in IQL, which features a query that computes the start states for each method (line 1) and a recursive query that performs the fixpoint computation to deduced states for all instructions (line 7)

The start states are computed via a function `startState` (line 2) that takes as input the first instruction of each method (selected via the condition in line 4) and

```
1 val startStates =
2   SELECT (startState) FROM
3     (instructions, codeAttributes) WHERE
4       (_.pc == 0) AND
5       (declaringMethod === declaringMethod)
6
7 val dataflow =
8   WITH_RECURSIVE (
9     startStates,
10    SELECT (UpperBoundState) FROM (
11        SELECT (nextState) FROM
12          (cfg, startStates) WHERE
13            (current === instruction)
14    ) GROUP_BY instruction
15  )
```

**Figure 6.7:** Fixpoint recursion of the Dataflow

in addition requires information about the maximum size of the stack and the local variables which is obtained via a join with the relation `codeAttributes` (line 3).

The fixpoint computation (`dataflow`) is a recursive query (line 8), where the recursion is performed on the start states (line 9), i.e., any deduced state enters the recursion again as a new "start" state. The fixpoint computation is split into two queries, an outer aggregation for the upper bounds (line 10) and an inner query that actually deduces the next states (line 11). We discuss the inner query first. The next state is computed via the respective function shown in Figure 6.6. To deduce the next state, the current state is joined to the control flow graph (line 12), where the current instruction of the edge in the graph equals the instruction for which we have computed the current state (line 13). The computation of upper bounds must be performed by an aggregation function (line 10). We use the function `UpperBoundState`, which is an incrementally maintained aggregation function that takes a collection of states and returns the single upper bound. The aggregation groups states by the current instruction (line 14). Thus if we have deduced multiple states for the same instruction they are categorized into the same group.

It is important to note that for the computation of upper bounds we require an aggregation inside the recursion. Note that a definition without the aggregation is also correct, but its practicability depends on the input programs. For example the JDK features methods that have 3 000+ instructions with many branches, which explode the number of states to 4 000 000+ states. In this case the analysis takes minutes for only one method. In our definition of recursion we have stressed that the use of aggregation functions inside a recursion is forbidden (cf. Sec.3.3.4 for the definition of the recursion operator). Nevertheless, such aggregations can surely be computed, since standard static analyses frameworks rely on them. There are two properties of the dataflow query that allow us, yet to use aggregations. First, the `UpperBoundState` will always converge to one upper bound value. In our case convergence is simple to verify, since the upper bound looses concrete path information as discussed earlier. Second, the dataflow will be computed in a local increment scope. This means that the fixpoints can be computed independently for deletions and additions. Thus the semantics discussed in Sec.3.3.4, where the fixpoint will essentially support itself indefinitely, i.e., will never be removed, is actually not a problem. The fixpoint will be computed once for deletions and once for additions and after the scope of the method (or the class) is left the data is removed from main-memory.

### 6.5.3 Findbugs Analyses

The dataflow query depicted in Figure 6.7 was used as the basis for 7 bug detectors from Findbugs. Table 6.16 gives an overview of the different analyses. In general the queries are rather simple once the dataflow is computed, i.e., each makes only a single selection ($\sigma$). However, it must be noted that the simplicity is in some cases due to additional information that we store inside the items used to emulate data on the stack (and in the local variables). For example, the $SA_1$ requires the knowledge whether the data was read from a field. The specific property is stored in an item as `item.isReadFromField`, along with other properties such as `item.mayBeNull`. It is not strictly necessary to carry the information in the items; however it greatly simplifies query writing. The decision to carry this information in items affects the runtime of the analyses only in so far that the general computation of the dataflow performs more work, while each individual analysis does less work. Hence, – if at all – we have raised the overall runtime (for the sake of simpler query writing), since each analysis can effectively filter out items before computing the same information.

### 6.5.4 Non-incremental Runtime

The runtime for each single dataflow analysis is measured once for SAE and once for BAT. The BAT implementation performs in essence the same work as the fixpoint computation shown in Figure 6.7. However, the BAT version performs an iteration over an array containing all instructions (as shown in Sec. 5.1.3). The iteration is

| Id | Description | Used Relations | Ops. |
|---|---|---|---|
| *DL* | Synchronization on a boxed primitive constant, e.g., of type Integer. | `dataflow` | $\sigma$ x 1 |
| $DMI_2$ | The method `toString()` is called on an array variable (the generated string is in general unintelligible). | `dataflow` | $\sigma$ x 1 |
| *RC* | Use of == or != operator on references should be replaced by `equals()` method. | `dataflow` | $\sigma$ x 1 |
| *RV* | The return value of a method is ignored. | `dataflow` | $\sigma$ x 1 |
| $SA_1$ | Value of a field is compared with itself. | `dataflow` | $\sigma$ x 1 |
| $SA_2$ | A self assignment to a local variable; e.g. `x = x` | `dataflow` | $\sigma$ x 1 |
| *SQL* | A call to `set...()` on a prepared statement with the parameter index given as 0 (note, indexes start at index 1). | `dataflow` | $\sigma$ x 1 |

**Table 6.16:** Description and classification of *dataflow* Findbugs analyses

performed twice; once to compute predecessor instructions for the control flow and once to compute actual dataflow. The computation of next states and upper bounds is performed by the same functions as in the incrementalized versions; hence the performed work is comparable.

The results for the non-incremental runtime over the JDK and Scalac datasets are shown in Table 6.17. Overall, the times for the individual analyses are very similar, because dataflow is computed for all methods, which is the most dominating operation.

In comparison the SAE takes approx. 3 times as long as the implementation in BAT. This increase is comparable over both measured datasets and all measured analyses. The additional time is naturally due to the fact that the SAE performs real recursions that propagate results through the operator tree, whereas the BAT implementation performs a simple looped iteration over the instructions. Nevertheless the total time taken by the SAE is not considerably worse, i.e., the incrementalization already pays off after three changes.

### 6.5.5 Incremental Runtime

The maintenance of the dataflow for all methods is naturally a more expensive operation and takes longer than the incremental computations for the lightweight

| $Id$ | time (s) - JDK | | | time (s) - Scalac | | |
|---|---|---|---|---|---|---|
| | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ | SAE | BAT | $\frac{\text{SAE}}{\text{BAT}}$ |
| $DL$ | 49.91 | 15.92 | x3.1 | 14.32 | 4.10 | x3.5 |
| $DMI_2$ | 46.79 | 15.81 | x3.0 | 14.57 | 3.99 | x3.7 |
| $RC$ | 48.71 | 15.61 | x3.1 | 14.55 | 3.97 | x3.7 |
| $RV$ | 45.22 | 15.66 | x2.9 | 11.91 | 4.35 | x2.7 |
| $SA_1$ | 47.92 | 15.66 | x3.0 | 14.73 | 4.41 | x3.3 |
| $SA_2$ | 48.11 | 14.81 | x3.3 | 14.77 | 4.23 | x3.5 |
| $SQL$ | 48.28 | 16.15 | x2.99 | 13.46 | 3.94 | x3.4 |

**Table 6.17:** Non-incremental runtime comparison of intra-procedural dataflow analyses

analyses or the metrics. The runtime for the incremental changes in our dataset is depicted in Figure 6.8.

As expected all changes have a higher runtime compared to the previous measurements, however the majority of the changes can still be computed in a time frame of 10 milliseconds. Another notable difference is the greater gap between class and method scope. This is also not unexpected, since the class scope recomputes dataflows for all methods in a given class, whereas the method scope has to consider only a smaller number of methods. Nevertheless, the class scope is still a choice under which the majority of the changes can be recomputed in real-time.

Note that there are some distinctive outliers for the events at $x = 4$, $x = 16$, $x = 31$ and $x = 32$, where the method scope takes longer than the class scope, which was not the case in the previous measurements. In these events only a single class with very few and very small methods was changed. Hence, even the class scope is recomputed quickly. For comparison the prominent differences at $11 < x \leq 15$ are due to events that changed classes with 20+ methods.

### 6.5.6 Memory Materialization

Without optimizations the memory materialization for the intra-procedural dataflow is quite enormous. As shown in Figure 6.7 we join control flow edges to states, perform an aggregation and compute states recursively. Without any optimizations this means that we have to materialize the control flow edges, the states, the

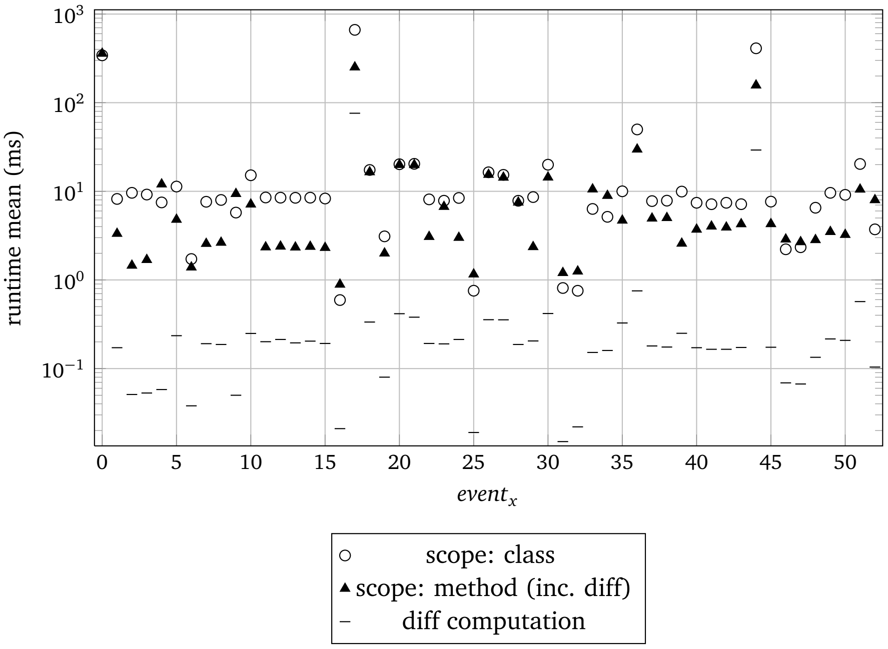


**Figure 6.8:** Incremental runtime of intra-procedural dataflow analyses

groupings of states by instructions and keep track of recursively computed values for the delete and re-derive (DRed) algorithm.

The only optimization that helps us in this respect is the local increment scope. The scope basically reflects the knowledge that an intra-procedural dataflow will never require values from other methods and, hence, the computation results are not required to be retained in memory. Note that in this case the method scope (▲) is sufficient, but the class scope serves just as well to reduce the amount of memory. Also note that for the optimization to be applied automatically – according to the definition in Sec. 4.2.5 – we have to rewrite the join condition in Figure 6.7 (line 13) slightly, i.e., instead of checking equality on entire instructions we phrased the condition as: the instructions have the same class, the same method, the same pc, etc. However, a more sophisticated check of the respective `equals` method for instructions can automatically derive this information.

The memory required by the dataflow analysis with and without the optimization is depicted in Table 6.18. Note that the dataflow dominates the memory for all Findbugs analyses and, hence, no detailed discussion w.r.t. each analysis is required.

| JAR | Mem. Plain (MB) | | | Mem. Opt. (MB) | | |
|---|---|---|---|---|---|---|
| | SAE | $\frac{\text{SAE}}{\text{Base}}$ | $\frac{\text{SAE}}{\text{SWi}}$ | SAE▲ | $\frac{\text{SAE}^{▲}}{\text{Base}}$ | $\frac{\text{SAE}^{▲}}{\text{SWI}}$ |
| JDK | OoM | - | - | 0.06 | x0.00 | x0.00 |
| Scalac | OoM | - | - | 0.03 | x0.00 | x0.00 |

**Table 6.18:** Mat. memory of data flow analyses with and w/o `@LocalIncrement`

The notable fact is that without the optimization the Java Virtual Machine runs out of memory (OoM) for both datasets. With the optimization the dataflow analysis is basically for free. Although as in previous sections we have measured small residues of memory.

The “out of memory” values in our measurement deserve some further perspective. Note that this only means that materialization and indexing of all states and control flows is too memory intensive. In general there are certainly other ways of incrementally re-computing dataflows, even in databases without our local increment optimization. An alternative solution can even be given in the SAE. The solution is to use the array of all instructions as BAT does and write the dataflow analysis as a single function that iterates over this array and produces an array of possible states[6]. In this case the dataflow computation becomes a simple projection, i.e., projecting the array of instructions to an array of states, which is also a self-maintainable operator and does not require materialized memory. However, this means that a part of the analysis, i.e., the array transformation, is written in an imperative style. Hence, the case we make here is for an end-to-end declarative specification of queries, which are phrased with the full dataset in mind. In this setting the local increment scope is required to make the dataflow analysis feasible. The scope can – in this setting – be understood as a high-level indicator to the database on how a query over the full dataset can be broken down into computations over smaller blocks of the dataset. This declarativeness has the advantage that we rely less on the concrete data representation. For example, the array of all instructions is not strictly required by the SAE. As discussed in Sec. 5.1.3 we keep it as a residue of the current bytecode parser. Deeper integration into the parser can alleviate our dependence on this array and even improve performance, since instructions can directly be propagated instead of doing a cumbersome unnesting of the array.

[6] In traditional databases this computation can for example be performed using stored procedures.

## 6.6 Discussion

In this chapter we have applied the SAE to three categories of static analyses, lightweight bug detectors, metrics and bug detectors using intra-procedural dataflow. The SAE allows to declaratively specify all these analyses while materializing only a necessary minimum of data. The following points deserve some closer discussion:

### 6.6.1 Comparing to the Incremental Runtime in a Deductive Database

The measured incremental runtime for the SAE was currently not compared to a deductive database. The main reason is that, even if the deductive database were to perform better, it still is based on materialization of base relations. The amount of memory used for this data alone is quite high. Moreover, to be competitive the deductive database must use indices as we have used in the SAE. The additional memory for indexing makes this approach more unattractive. Furthermore, such a comparison must take into account that we provide end-to-end measurements, i.e., including parsing time and creating a relational representation. Hence, for using a standard deductive database engine additional time is required to transition data from our bytecode parser to the engine. Given that we have seen comparatively short runtime windows (below 10 milliseconds) for several hundreds of instructions, the communication between parser and deductive database is an impeding factor to achieve comparable results. Nevertheless, such a comparison is interesting as a future work, but must also be performed with great care to obtain a performant implementation in the deductive database. It has been noted by Hajiyev et al. [HVdM06] that the elegant logic rules must be polluted with mode annotations (to mark input and output variables) or usages of the cut operator to control backtracking, otherwise the programs are comparatively slow.

The use of aggregations inside the dataflow deserves some discussion, since we said that logic languages are being used for static analyses, yet, we were only able to give a valid execution model for the dataflow query due to the specific circumstances of the local increment optimization. In general, aggregations are seldom mentioned when describing static analyses in logic languages. Mendez et al. [eNH07] discuss fixpoint iteration explicitly and show a logic-language representation of bytecode that features all instructions in a list, which is comparable to our array representation. Yet, their focus is to provide a novel algorithm for fixpoint iteration and not to use standard deductive databases.

Other related works, either do not discuss the aggregation step, e.g. [WACL05], or use a logic-representation where this information is already computed, e.g., [EKS$^+$07] use an SSA form where the aggregation according to $\Phi$-functions is already performed. Still others must be noted for considering different domains that are already expressed as logic facts tailored towards the specific problem, e.g., [BJ03] encode call graph algorithms or [WL04] consider points-to analyses with precomputed representations for variable reads/stores. Nevertheless, Saha et al. [SR06] propose an extended DRed algorithm that is said to treat aggregations and non-stratified negations for incremental maintenance of tabled datalog. Yet, how this algorithm can be applied to make aggregations available for general recursions in our database – and thus the SAE – is not entirely clear yet. The algorithm relies on a first-class representation of logic programming calls with partially instantiated variables, e.g., $p(3,X)$, which have no correspondence in our approach.

### 6.6.2 Inter-Procedural Data Flow

The analyses in this chapter have not considered inter-procedural data flow analyses. Nevertheless, these analyses are well expressible using the SAE. In general, such analyses require a recursive analysis of called methods. Unlike intra-procedural data flow an aggregation of results is typically not required. We have performed some preliminary experiments for inter-procedural data flow by formulating incremental call graph construction algorithms. The call graph is the basis for further inter-procedural analyses. The early experiments were conducted using an algorithm termed rapid type analysis and the results are encouraging. The non-incremental performance places the SAE in a range below 2 times that of a reference implementation (with an absolute value of approx. 2 seconds for the computation). However, the incremental maintenance must be studied more carefully, since removal of methods cascades along the recursively computed method calls. The memory consumption for call graph construction is naturally greater than for any of the analyses evaluated in this chapter, since method calls are globally correlated data and can not benefit from the local incremental scope optimization. Nevertheless, a fully indexed and and recursively computed call graph for the Scala compiler dataset – used in this chapter – required only approx. 50% of the fully materialized data in the SAE (approx. 48 MB).

### 6.6.3 Non-Incremental Runtime

In the measured analyses the SAE is competitive to a non-incremental approach that uses iterations over an object-oriented structure (BAT). The non-incremental approach was augmented with hash tables and hash maps for recurring data, which (i) reflects a real-world programming model for the analyses and (ii) ensures that the approach is competitive to the indexing used in the SAE. The lightweight analyses have shown a relative even match, where both approaches are sometimes better and sometimes worse; yet always in a small measure of approx. 10% of the overall runtime. The metrics and intra-procedural analyses have shown an increased ratio of the runtime that places the SAE in a range of 2-3 times slower. Yet, this measure is quickly amortized in the light of incrementalization.

### 6.6.4 Incremental Runtime

The incremental runtime for all analyses was extremely small. The majority of the changes to the code base were re-computed in a manner of 10 milliseconds. Even the intra-procedural dataflow analyses stayed close to 10 milliseconds and required only 20-30 milliseconds for larger changes. Larger in our case means changes that removed and added 20-30 methods with a total of approx. 2 000 - 3 000 removed (and the same amount of added) instructions. The largest change that we did not consider a re-build was ($x = 36$) which added and removed approx 90 methods in 8 classes with respective numbers for added and removed instructions at approx. 6 000. This change still took only 50 milliseconds for the intra-procedural analysis.

In a broader context we can take a quick look at how representative the methods in our change set are, by using statistical data obtained by Collberg et al. [CMS07]. They place the average size of Java methods – in terms of instructions – at 33.2 with a very high standard deviation of 156 (their statistics were obtained using a very wide set of Java Programs). The methods we analyzed in our change set had on average 45 instructions and a maximum of over 200 instructions, which places us well into that average.

This means that the incremental maintenance of the SAE can be considered real-time at least for the shown analyses when using Java programs as input that have typical average method sizes. In such programs we can incrementally maintain small changes in 10 milliseconds and smaller refactorings covering approx. 10 classes will be computed in 100 milliseconds and probably less time. Nevertheless, it is hard to generalize the incremental maintenance performance, since Java Programs can have a large variety of methods sizes and, hence, more diverse workloads.

### 6.6.5 Materialized Memory

The memory required by the individual analyses is in general low compared to the overall amount of data found in the code base. The memory requirements have dropped by such a significant amount that the SAE is truly feasible for using inside an IDE. That is, even for large projects – such as the JDK – the amount dropped below 100 MB that are permanently blocked for the incremental maintenance.

For the lightweight analyses and the metrics a large amount of irrelevant data is already omitted from materialization by basic filtering. The lightweight analyses require approx. 10% of the total memory for all data and the metrics closer to 5%. The local increment scope optimization further reduces the memory. In the lightweight analyses a considerable reduction by 50% of the memory was shown. The metrics had only one query where the optimization was applicable, yet this already amounted to approx. 30% of the required memory. The intra-procedural dataflow analyses are different in the sense that they absolutely required the local increment scope to be feasible with low memory when they are specified in a declarative manner.

The memory savings are provided automatic – or with the help of one annotation for the local increment scope – and the analyses as well as the integration into the IDE were written with the full data sets in mind. That means no manual filtering of the data to suit exactly the provided analyses was necessary. As can be seen from the lightweight analyses, such a filtering would be very tedious due to the diversity of the written queries.

Furthermore, the materialized memory of the SAE includes heavy indexing and is still comparatively low. Many of the lightweight queries rely on joins, or use existential quantification that relies on joins. Hence, these queries would be executed much slower without our use of indexing.

### 6.6.6 Class Scope vs. Method Scope

Given the data provided in this chapter we can better judge the trade-offs between using the class scope (lower memory, higher runtime) and the method scope (higher memory, lower runtime).

In our sampled analyses the memory savings of the class scope only applied to three analyses, whereas the method scope provided optimizations for 8 analyses. Nevertheless the savings contributed were quite considerable. In the lightweight analysis the optimization at the class scope for the *UR* and *UG* analyses together contributed a large amount of the savings. Likewise, the only metric where the optimization was applicable required the class scope. Note that we can of course expect to find other metrics that are also optimizable at the method scope, e.g., cyclomatic complexity comes immediately to mind. Nevertheless, the class scope is worth having if the accompanying runtime is acceptable.

In general, the runtime for either scope stayed in the same order of magnitude for our measured analyses. The dataflow analyses had the most distinguished difference between the two scopes. Yet even here, the majority of incremental changes for individual classes stayed within the time frame of 10 milliseconds. Nevertheless, the decision is really affected by measure of how complex the analyses are. The beauty of the approach is that the scopes are declarative and easily changed. Hence, if longer running analyses are encountered that do not benefit from the class scope, they can easily be toned down to the method scope.

# 7 Incrementalized and Modular Architecture Conformance Checking

In this chapter we use the SAE to check code against architectural specifications that provide (i) architectural units for grouping source code elements, e.g., groups of classes, and (ii) constraints for modeling valid and invalid dependencies. The dependencies at the code level are then checked against valid and invalid dependencies in the specification. The incrementalization is used for two distinct purposes in this chapter. First, changes to the code are checked incrementally to quickly determine whether the updated code still conforms to the architectural specification. Second, changes to the specification are incrementally maintained to quickly determine whether the same code base still conforms to the updated architectural specification. In other words the first incrementalization uses a fixed set of expectations about the dependencies in the code and allows us to continually enforce these expectations w.r.t. changes in the code. The second incrementalization allows us to rephrase our expectations and quickly determine whether the new expectations are sensible.

The second incrementalization – for maintaining the specification – enables us to formulate a new approach to architecture conformance checking, that enables architects to define intended architectural dependencies in modular units termed *slices*. Existing approaches in architecture conformance checking require the whole architecture and its intended dependencies to be specified in one single document. Based on the experience gained when modeling the architecture of real systems (e.g., Hibernate [BK04]), such approaches do not scale for large architectures in the sense that the architecture specification is overloaded with a large amount of information. In other words the specification features so many architectural elements and dependencies, that architects can hardly focus on individual architectural design decisions and the relevant parts of the system.

Incrementalization is an enabling factor for a modular architecture conformance checking approach, since the modular units (slices) must be independently checked. Hence, modularizing a non-incremental conformance check is not efficient, since an analysis must run multiple times for each slice. The inefficiency is mainly due to the fact that architecture analyses are whole program analyses that consider a large amount of data and, hence, are expensive. Furthermore, the automated incrementalization made the development of such modularized analyses very easy. The reason being, that slices can overlap in terms of the architectural units whose dependencies are described. Hence, each different slices must be notified when architectural units change. Using the automated incrementalization this notification is performed by the database, but otherwise requires an extra implementation of the update logic.

The remainder of the chapter is organized as follows. Section 7.1 gives an overview of architecture conformance checking and related approaches and discusses the issue of modularization in more detail. In Section 7.2, we briefly introduce the Hibernate framework [BK04], which we use to illustrate concepts of the proposed approach and to evaluate its effectiveness. Section 7.3 introduces the tool Vespucci that implements the modularized specification language. In Section 7.4 we present an in-depth evaluation of the modularization based on the Hibernate cases study performed using Vespucci. After that, we discuss related work in Section 7.6. Finally, we give a summary and discuss future work.

## 7.1 Modularized Architecture Conformance Checking

A documented software architecture is an acknowledged success factor for the development of large, complex systems [SG96]. Traditionally, architecture description languages (ADLs) have been used to specify the architecture and verify its properties. Generally, this process has been detached from coding and the architecture specification has been considered as a means to prescribe the structure of the code resulting from programming or eventually to generate a first skeleton of that code. However, as systems evolve over time, due to new requirements or corrections, the implemented architecture starts to diverge from the intended architecture [EGK$^+$01, GL00, MRB06] — resulting in *architecture erosion* [PW92].

To combat architecture erosion, several approaches have emerged that focus on structural dependencies [EKKM08, MNS95, SJSJ05, TV09] and whose proponents argue for automated checking of architecture specifications w.r.t. the static structure of the source code. These approaches generally allow to group[1] source code elements into *building blocks* — cohesive units of functionality in the software system — and to specify in which way a building block is allowed to statically depend on which other building block. The specification formalisms in these approaches vary and can be summarized as: (i) a flat graph with building blocks as nodes and allowed dependencies as edges [MNS95]; (ii) a matrix notation with building blocks in rows/columns and their dependencies in the cells [SJSJ05]; (iii) a graph with hierarchical nodes and component-connector style ports to manage internal/external dependencies [EKKM08]; (iv) a textual specification of access restrictions on target building blocks [TV09].

Such specifications are used either analytically [MNS95] — to analyze already written code for conformance with an intended static structure — or constructively

[1] using, e.g., regular expressions over classes or source files

[EKKM08, TV09] to enforce the code's compliance with the specification of the static structure continuously during development. Constructive approaches were proven to help developers in realizing the intended architecture. Several case studies [Her10, KMHM08, KMNL06] show that constructive approaches can prevent structural erosion [RLGBAB08, WCKD11].

Though current approaches have proven to be valuable, they all share the property that a single monolithic specification is used and – as in case of a monolithic software system — a monolithic description of the structure does not scale and becomes unmaintainable once the software reaches a certain complexity. Sangal et al. [SJSJ05] explicitly try to solve the maintainability and scalability issues using a special notation called dependency structure matrices (DSMs). However, we believe that the problem is not so much the notation. The root of the problem is the monolithic nature of the specifications. Based on some preliminary experience with modeling the architecture of real systems, such as Hibernate [BK04], we doubt that any such approach can scale, even with compact notations such as dependency structure matrices. As a result, typically only the highest level of components and/or libraries is considered [TV09, RLGBAB08]; requiring different notations and tools for different levels of the design. This precludes a seamless design at various granularity levels.

In this chapter, we argue that modeling a software's static structure should consist of multiple views, that focus on different parts and on different levels of detail. We take the position that like programming languages, architecture modeling languages in general should support modularity and scoping mechanisms to support modular reasoning about different architectural concerns and information hiding to facilitate evolution.

Accordingly, we propose a novel modeling approach and tool, called Vespucci, that allows to separate the specification of a software's static structure into multiple complementary views, called *slices* throughout this chapter. Each slice can be reasoned over in separation. Multiple slices can express different views on the same part of the software and each slice can be evolved individually. Hence, evolution of large scale specifications consisting of several slices is facilitated by distributing work to systematically update the architecture in a modular fashion. Contrary to a monolithic specification, our approach also has the benefit that individual concerns can remain stable. Stable parts can be modularized into different slices to be separated from architectural "hot-spots", i.e., slices that require frequent changes during the evolution.

The contributions of this chapter are:

- A first approach towards the specification of a software's structural dependencies that supports a modularized specification by means of individual slices.
- A new approach for modeling a software's structural dependencies that combines the advantages of hierarchical and graph-based modeling approaches to enable reasoning over a software's static structure at different abstraction levels.
- Discussion of an implementation of the proposed approach that enables the specification and checking of a software's structural dependencies.

## 7.2 Architecture of Hibernate

As part of the development of Vespucci, we did a comprehensive analysis of the architecture of the object-relational mapping framework Hibernate [BK04]. We provide a short overview of Hibernate and its architecture in this section since we will refer to it to discuss and motivate various features of Vespucci.

We chose Hibernate as it is a large, mature, widely-adopted software system, which has been continually updated and enhanced. We reengineered the architecture of the core of Hibernate in version 1.0.1 (July 2002) and played back its evolution until version 3.6.6 (July 2011)[2]. During this time the core grew from 2 000 methods in over 255 classes organized in 18 packages to 17 700 methods in over 1 954 classes in 100 packages.

In the following, the major building blocks of Hibernate's architecture are presented. A *building block* is a logical grouping of source code elements that provide a cohesive functionality, independent of the program's structuring, e.g., in packages or classes. The scope of a building block depends on the considered abstraction level and ranges from a few source code elements up to several hundreds. For example, Hibernate's support for different SQL dialects is represented by one top-level building block with many source code elements, but further structured into smaller building blocks for elements that abstract over the support for concrete dialects and those that actually implement the support.

The architectural model of Hibernate 1.0 consists of the 22 top-level building blocks shown in Table 7.1. Of these 22 top-level building blocks, 16 were further structured. In total, we identified 73 second-level building blocks. Given the size of Hibernate 1.0, we did not analyze lower levels. On average each top-level building block already only contains 11 classes and the 2nd level building blocks consist of

[2] Hibernate 4.0 was released after the case study.

| Top-level Building Block | 2L Building Blocks | Classes contained | Elements contained | Relation to Packages |
|---|---|---|---|---|
| Cache | 4 | 6 | 60 | ≡ |
| CodeGeneratorTool | 0 | 9 | 68 | ⊂ |
| ConnectionProvider | 3 | 5 | 51 | ≡ |
| DatabaseActions | 3 | 9 | 59 | ⊂ |
| DataTypes | 10 | 37 | 410 | ≡ |
| DeprecatedLegacy | 2 | 2 | 6 | ⊂ |
| EJBSupport | 0 | 1 | 22 | ≡ |
| HibernateORConfiguration | 2 | 2 | 39 | |
| HibernateORMapping | 12 | 33 | 389 | ≡ |
| HQL (Hibernate Query Lang.) | 3 | 9 | 130 | ≡ |
| IdentifierGenerators | 4 | 12 | 92 | ≡ |
| MappingGeneratorTool | 0 | 19 | 233 | ⊂ |
| PersistenceManagement | 6 | 35 | 674 | |
| PropertySettings | 0 | 1 | 43 | |
| Proxies | 0 | 3 | 23 | ⊂ |
| SchemaTool | 2 | 5 | 34 | ⊂ |
| SessionManagement | 6 | 10 | 312 | |
| SQLDialects | 3 | 12 | 119 | ≡ |
| Transactions | 2 | 4 | 37 | ≡ |
| UserAPI | 9 | 9 | 63 | |
| UtilitiesAndExceptions | 2 | 33 | 235 | |
| XMLDatabinder | 0 | 2 | 21 | |

**Table 7.1:** Overview of Hibernate 1.0

even fewer classes. The key figures of the architecture are given in Table 7.1. In the following, we discuss those elements of the architecture that are most relevant when considering the modeling of architectures. The complete architecture can be downloaded from the project's website [Ves12].

For Hibernate 1.0 nine of the building blocks have a one-to-one mapping to a package (cf. Table 7.1 – Relation to Packages ≡). Six building blocks map to a subset (⊂) of the code of some non-cohesive package. For example, the package `cirrus.hibernate.impl` contains classes for creating proxies as well as classes related to database actions. These sets of classes have no interdependencies and belong to different building blocks. The source code elements of the remaining building blocks are spread across several packages. For example, the code related to session handling is spread across two packages in version 1.0.

Overall, the architecture features several well modularized building blocks, such as the Cache, HQL or Transactions building blocks, which are only coupled with at most three other building blocks. The number of well modularized building blocks with few dependencies is, however, small. The majority of Hibernate's functionality belongs to building blocks that exhibit high coupling, such as PersistenceManagement, SessionManagement and DataTypes.

## 7.3 The Vespucci Approach

In this section, we first describe the three major parts Vespucci [Ves12] consists of: (1) a declarative source code query language to overlay high-level abstractions over the source code, (2) an approach that enables the modular, evolvable, and scalable modeling of an application's structural dependencies, and (3) a runtime for checking the consistency between the modeled and the implemented dependencies. After that, we present in Section 7.3.4 the different modeling approaches supported by Vespucci. Finally, we discuss in Section 7.3.5 how the proposed approach facilitates the evolution of the specification and the underlying software and how it supports large(r) scale software systems.

### 7.3.1 High-level abstractions over source code

Vespucci is concerned with modeling and controlling structural dependencies at the code level. But, it does so at a high-level of abstraction.
**Ensembles** are Vespucci's representation of high-level building blocks of an application, whose structural dependencies are modeled and checked. Specifically, Vespucci's ensembles are groups of source code elements, namely type, method, and field declarations. The definition of an ensemble involves the specification of source code elements that belong to it by means of source code queries. We refer to the set of source code elements that belong to an ensemble as the *ensemble's extension*.

The visual notation of an ensemble is a box with a name label. For example, Figure 7.1 shows two ensembles, one called SessionManagement and one called HQL. Vespucci explicitly predefines the so-called *empty ensemble* that never matches any source elements and is depicted using a simple gray box (■). The empty ensemble supports some common modeling tasks, e.g., to express that a utility package should not have any dependencies on the rest of the application's code.

**The source code query language** is introduced – mostly example-driven – in the following paragraphs. The language is not the primary focus of this chapter, which is rather on modularity mechanisms for modeling structural dependencies. In fact, the approach as a whole is parameterized by the query language, in the sense that the modularization mechanisms can be reused with other more expressive query languages and more sophisticated query engines. For a more systematic definition of the current query language, the interested reader is referred to the website of the project [Ves12].

The query language provides a set of basic predicates that can select individual fields or methods, entire classes, packages, or source files. Predicates take quoted parameters, which filter respective code elements by their signature, e.g., the predicate **package**('cirrus.hibernate.helpers') selects code elements in Hibernate's helpers package, using the package name as the filter. The query defines the Utilities ensemble, which we have used in modeling Hibernate's structural dependencies.

In the above example, source code elements are precisely specified by their fully qualified signature. Furthermore, wildcards ("*") can be used to abstract over individual predicate parameters. For example, the field predicate below selects field declarations in class Hibernate with any name (the second parameter is "*"), of a type that ends with the suffix Type. We have used the query to define an ensemble called TypeFactory which serves as a factory for Hibernate's built-in types.

```
field('*.Hibernate','*','*Type')
```

Queries can be composed using the standard set theoretic operations (union, intersection, difference), or by passing a query as an argument to a type parameter of another query. This form of composition is useful to reason over inheritance for selecting all sub-/supertypes of a given type. For example, consider the query:

```
class_with_members(subtype+('Dialect'))
```

It uses the basic predicate class_with_members, which selects a class and all its members. Since the predicate expects a type to be selected, we can instead pass a sub-query. The subtype+ query returns the transitive closure of all subtypes of the class Dialect. Hence, the example query selects all classes (and their members) that are a subtype of the class Dialect. In Hibernate these represent all supported SQL dialects – the shown query actually defines the ensemble ConcreteDialects.

As already mentioned, the query language is interchangeable. What is interesting about the use of the source query language as an ingredient of our approach is that it enables modeling structural dependencies at a high-level of abstraction. Furthermore, it supports the definition of ensembles that cut across the modular structure of the code, e.g., TypeFactory cuts across the class-based decomposition of code. This enables feature-based control of structural dependencies.

Vespucci provides an **ensemble repository** that stores the definitions of all ensembles. It serves as a project-wide repository and provides the starting point for modeling an application's intended structural dependencies. Capturing all ensemble definitions in a single repository serves two purposes. First, it enables a model of intended structural dependencies to be modularized with the guarantee that all modules refer to the same extension for a particular ensemble. Second, it allows modules to pose global constraints quantifying over all defined ensembles (see the discussion about global constraints in the following section).

## 7.3.2 Modeling Structural Dependencies

**Dependency slices** are Vespucci's mechanism to support the modularized specification of an application's structural dependencies. A slice captures one or more specific design decisions, by expressing one or more constraints over ensemble inter-dependencies, e.g., which ensemble(s) is/are allowed to use a certain other ensemble.

For illustration, Figure 7.1 shows an an exemplary slice, which governs dependencies to source code elements that implement the Hibernate query language, represented by the HQL ensemble. Specifically, it states that elements pertaining to HQL may be used ONLY by those pertaining to the SessionManagement ensemble. The cycle attached to the arrow pointing to HQL states that globally, i.e., for all ensembles in the ensemble repository, this is the only dependency on HQL's elements that is allowed.

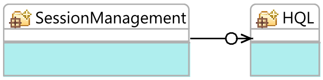


**Figure 7.1:** Dependency rule for Hibernate Query Language

Figure 7.2 shows another example slice, which states that source code elements pertaining to SQLDialects are only allowed to be used by PersistenceManagement's or SessionsManagement's source code elements.

There can be an arbitrary number of slices in a model of structural dependencies; the set of ensembles referred to in different slices may overlap. Deciding about the number/kind of slices, in which one may want to break down the specification of an application's structural dependencies is a matter of modeling methodology, as we elaborate on in section 7.3.4. Yet, we envision the default strategy to be one in which each slice is used to express allowed and expected dependencies from the

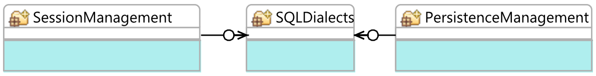


**Figure 7.2:** Users of SQL Dialects

perspective of a single ensemble; this strategy was used in the case study and in the examples shown in this chapter. For this purpose, the visual notation features arrow symbols that are shown next to the ensemble that is constrained[3]. For example, by looking at Figure 7.1 we can reason about *all* dependencies that are allowed for HQL and looking at Figure 7.2 we can reason about *all* dependencies that are allowed for SQLDialects.

Ensembles that participate in a slice but which have no arrow symbols next to their box are not constrained. For example, both slices refer to SessionManagement, but make no statement w.r.t. the total of its allowed dependencies. From these two slices we can see that SessionManagement's source code elements are allowed to depend on both SQLDialect's and HQL's source code elements. However, SessionManagement and PersistenceManagement are not constrained.

**Constraint types** are classified into two basic categories: constraints that are defined w.r.t. the allowed and those w.r.t. the not-allowed dependencies. Constraints on allowed dependencies are further classified as *Outgoing and Incoming Constraints* and *Local and Global Constraints*. The rationale for distinguishing between the above types of constraints relates to enabling modular reasoning about individual architectural concerns. Modular reasoning fosters scalability by allowing each slice to be understood as a single unit of comprehension, and also fosters evolvability as each slice can be adapted without the need to refer to other slices. We elaborate on the role that different constraint types play with these respects in the following section. Here, we exclusively focus on explaining the meaning of these different constraints.

An *incoming constraint* restricts the set of source code elements that may use the elements of a particular ensemble (target ensemble). Incoming constraints are denoted by the symbol "⇾" shown next to the target ensemble (cf. Figure 7.1, Figure 7.2). For example, the constraint in Figure 7.1 restricts source code dependencies, of which the target element belongs to HQL: the source of the dependency must belong to SessionManagement; source code dependencies from and to the

[3] For this chapter the visual models were compressed to save space. Hence the distinction may not be as obvious as it is when you use the Vespucci tool.

source code elements belonging to SessionManagement are — w.r.t. that slice — unrestricted.

An *outgoing constraint* restricts the set of source code elements on which code elements of a specific ensemble (source ensemble) may depend. Outgoing constraints are visually denoted by the symbol "⋗" shown next to the source ensemble. For

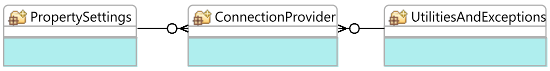


**Figure 7.3:** Constraints on the Connection Provider

example, the slice in Figure 7.3 features two outgoing constraints; from ConnectionProvider to PropertySettings, respectively to UtilitiesAndExceptions. Outgoing constraints only affect code elements of their source ensemble. Hence, the slice in Figure 7.3 governs the dependencies of code elements involved in providing connections (captured by the ConnectionProvider ensemble). They may only use generic functionality (captured by the UtilitiesAndExceptions ensemble), or functionality for getting and setting properties (captured by PropertySettings). The targets of the constraint (PropertySettings and UtilititiesAndExceptions) can — w.r.t. the slice in Figure 7.3 — depend on any other ensemble.

*Global constraints* quantify over all defined ensembles. Visually they are denoted by a "○" attached to a constraint. All constraints considered in the examples so far were global. For example, the constraint shown in Figure 7.1 affects code elements that belong to any ensemble defined in the repository of Hibernate, even if not referred to by the slice, e.g., ConnectionProvider or PropertySettings in Figure 7.3. Code elements of the latter ensembles are not allowed to depend on elements in HQL.

Global constraints are hard constraints w.r.t. the addition of new ensembles into the architecture. Whenever new ensembles are defined in the ensemble repository, they are included when checking a global constraint. The purpose is to provide tight control over the evolution of the architecture. If a new ensemble has dependencies that violate a global constraint, then architects can asses whether the violation needs to be removed from the code or, whether the currently defined architectural rules are too narrow. The essential point is that an architect has assessed the situation and no uncontrolled erosion of the software's structure has occurred.

*Local constraints* quantify only over ensembles that are referenced in one particular slice. Visually, they are characterized by the lack of the "○" symbol. Figure 7.4 depicts local constraints on the implementation of Hibernate's support for different SQL dialects (e.g., "Oracle SQL","DB2 SQL"). Each dialect is realized by implementing a common interface. Elements of this interface are captured by the AbstractDialect ensemble. Support for specific dialects is captured by ConcreteDialects. The TypeNameMap ensemble captures code elements involved in implementing a specialized dictionary for mapping database type names to a common set of names. The defined constraints specify that only code pertaining to ConcreteDialects is allowed to depend on code pertaining to AbstractDialect and code in the latter is only allowed to depend on TypeNameMap's code. Furthermore, neither source code elements of AbstractDialect nor TypeNameMap are allowed to depend on elements of ConcreteDialects due to the incoming constraint between the empty ensemble and ConcreteDialects. However, the constraints of the slice in Figure 7.4 do not restrict in any way code elements belonging to ensembles that are not referenced by this slice, e.g., code pertaining to HQL (slice in Figure 7.1) could use code pertaining to ConcreteDialects.

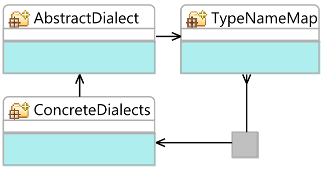


**Figure 7.4:** Supporting multiple SQL Dialects

Local constraints provide tight control over the evolution of source code w.r.t. the scope of the ensembles referenced in a slice. Their purpose is to capture localized rules that reason only over a part of all the dependencies in the architecture, e.g., as in Figure 7.4 where only dependencies pertaining to the implementation of multiple SQL dialects are considered. The implementation details of involved ensembles can change (and respectively their extensions), but the changes are guaranteed to adhere to the specified allowed/expected dependencies. The rest of the architecture can evolve independently, i.e., new ensembles and dependencies can be introduced as long as they do not violate the localized rules.

**Different kinds of dependencies** can be constrained individually by annotating constraints. The kinds of dependencies are those that can be found in Java code (e.g., Field Read Access, Field Write Access, Inherits, Calls, Creates,...; a complete reference is available online [Ves12]); by default, all kinds of dependencies are constrained and no further annotation of a constraint is necessary. Dependency kinds are important when documenting detailed design choices.

For example, Figure 7.5 restricts only dependencies of the create kind (i.e., object creations) to the ConcreteConnectionProviders; only the ConnectionProviderFactory

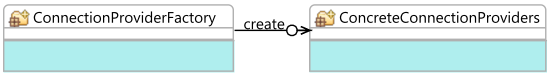


**Figure 7.5:** Restricting connection provider creation to a factory

is allowed to create new connection providers. All other dependencies are allowed for all ensembles, hence clients may use the created provider, e.g., by calling its methods. The range of possible applications is broad, e.g., one can also disallow classes from throwing particular exceptions, while allowing their methods to catch them.

**Nesting of ensembles** is also enabled in Vespucci to reflect part-whole relationships. The information about child/parent relationships between ensembles is stored in the global repository. For illustration consider that the slice shown in Figure 7.4 actually models the internal architecture of Hibernate's support for SQL dialects. One can express this relation by making the ensembles referred to in Figure 7.4 children of the SQLDialects ensemble, as shown in Figure 7.6.

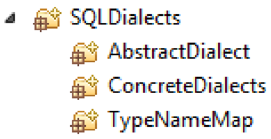


**Figure 7.6:** (Sub-)Ensembles of SQL Dialects

The extension of an ensemble that has inner ensembles is the union of the extension of its inner ensembles; i.e., an ensemble with inner ensembles does not define its own query to match source elements, but instead reuses the queries of its inner ensembles. Hence, the semantics of nesting is that constraints defined for parent ensembles implicitly apply to source code elements of all their children, e.g., constraints defined for SQLDialects in the slice in Figure 7.2 apply to all its children ensembles.

Constraints that cross an ensemble's border are disabled in Vespucci for keeping the semantics simple. Due to slices, this can be done without loss of expressivity. If an architect needs to define a constraint between two ensembles that do not have the same ancestor ensemble, it is always possible to specify the constraint in a new slice that just refers to the directly relevant ensembles.

With hierarchical modeling, architects can distinguish between ensembles that are involved in the architectural-level modeling of dependencies (SQLDialects) and those involved in modeling decisions at lower design levels (ensembles in Figure 7.4). In the following sections, we discuss how the combination of slices and hierarchies facilitates the incremental refinement of a software's architecture and is advantageous in case of software evolution.

### 7.3.3 Constraint Enforcement and Tooling

Conceptually, checking the implementation against the modeled dependencies is done as described next.

First, for each ensemble its extension along with the set of source code dependencies related to it (those that have the ensemble as source or target) is calculated; self-dependencies, i.e., source code dependencies, where the source and target elements belong to the same ensemble are filtered out. Furthermore, dependencies from and to source code elements that do not belong to any ensemble are ignored.

Second, each slice is checked on its own. To do so, Vespucci iterates over all ensembles of each slice and checks that none of the dependencies between respective source code elements violates a defined constraint. For example, to check the compliance of an application's source code with the slice in Figure 7.3, Vespucci effectively checks that the target of all code dependencies, starting at a code element in ConnectionProvider, either belongs to PropertySettings or to UtilitiesAndExceptions.

The implementation of Vespucci's dependency checker is integrated into the Eclipse IDE. Checking is done as part of the incremental build process and incremental checking ([EKS+07]) is efficient enough for (at least) mid-sized projects such as Hibernate. Likewise the modeling side is incrementalized; hence, changes to queries are immediately reflected in the IDE.

The rationale behind the decision to ignore dependencies to source code elements that do not belong to any ensemble is that dependencies to an application's essential libraries and frameworks are most often not of architectural relevance and should not clutter the overall specification. Nevertheless, it is always possible to create an ensemble that covers some fragment of a fundamental library to restrict its usage. E.g., while it generally does not make sense to restrict the usage of the JDK, it may still be useful to restrict the usage of the `java.util.logging` API, because the project as a whole uses a different API for logging and it has to be made sure that no one accidentally uses the default logging API. One possibility to model such a decision is to create a global incoming constraint from an empty ensemble to the ensemble representing the `java.util.logging` API. However, Vespucci provides a specialized view that lists source code elements that do not belong to any ensemble to make it easily possible to find unintended holes in the specification.

### 7.3.4 On Modeling Methodology

Figure 7.7 schematically shows four principal ways to model the architecture of a hypothetical system consisting of four ensembles (boxes labeled 1 to 4) with Vespucci. In (A), all constraints are modeled in a single model. In (B), the model makes use of hierarchical structuring – specifically, ensembles 1 and 2 are nested into an ensemble 1&2. In (C), the model makes use of slicing; specifically, per ensemble one slice is defined, modeling only decisions related to that ensemble, but slicing at other granularity levels is conceivable (see below). In (D), the model makes use of both slices and hierarchies, which is the expected typical usage of Vespucci.

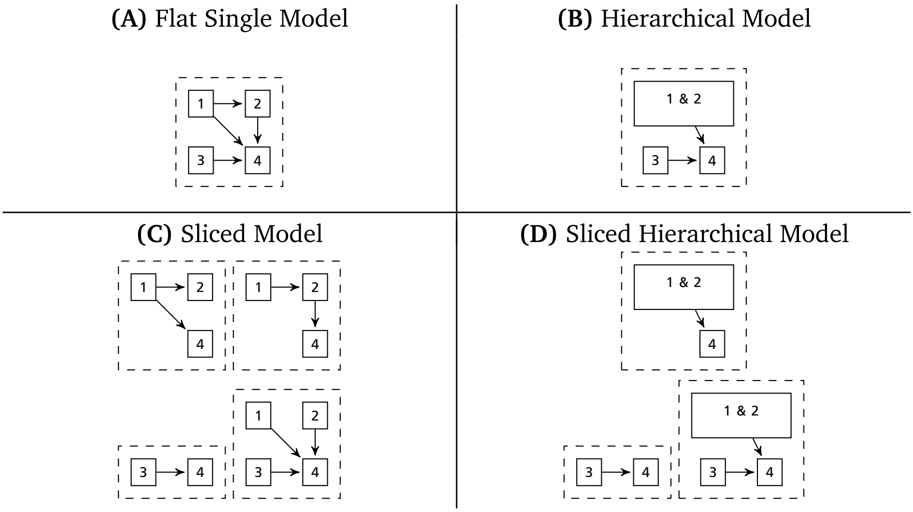


**Figure 7.7:** Alternative architectural models of dependencies

In general, the structural dependency model of a system in Vespucci consists of an arbitrary number of slices. It is a matter of modeling decisions – taken by the architect – in how many slices she breaks down the overall architectural specification. As part of this process, a trade-off is to be made between (i) creating (large(r)) slices that capture several architectural rules related to multiple ensembles that conceptually belong together and (ii) creating one slice per ensemble that just captures the architectural rules related to that ensemble. In the former case cohesiveness is fostered while in the latter case (local) comprehensibility of the architecture and evolvability of the specification is fostered.

In the Hibernate case study, as a rule of thumb, each high-level slice focused on design decisions concerning one ensemble. For instance, the slices in Figure 7.1 and Figure 7.3 focus on specific design decisions related exclusively to allowed *incoming dependencies* to HQL, respectively allowed *outgoing dependencies* of ConnectionProvider. Internal dependencies for ensembles with nested sub-ensembles were in general related to a small set of ensembles and hence captured in a single slice, as e.g., in Figure 7.4, where the internals of SQLDialects were captured.

The one-slice-per-high-level-ensemble strategy for breaking down specifications is just a first approximation. For reasons of better managing complexity and evolvability as well as understandability, it may make sense to choose more fine-grained or coarse-grained strategies. One such strategy is to split the specification of incoming and outgoing dependencies of an ensemble, if those are too complex or evolve in different ways. On the other hand, slices of related ensembles may be merged, when their separated specifications are too simple to justify separate slices or hard to understand in isolation.

One may criticize that a specification becomes complex with an increasing number of slices. However, a single specification that controls the dependencies to the same degree is no less complex and includes all information that is captured in the slices. For example, if internal dependencies are controlled, they need to be specified and maintained in a single specification as well. The focus here is to make a case for enabling the architects to break down specifications of structural dependencies in several modules that are more manageable w.r.t. scalability and evolvability and can be reasoned over in isolation. Hence, slices also facilitate distribution of work, such that large architectures can be maintained by a team rather than a single architect.

Per ensemble slicing of the dependency model may also impair understandability of dependencies pertaining to several modules. A view of the dependencies for multiple ensembles (in contrast to their individual constraints) can be advantageous for the exploration of the architecture, e.g., if one wants to follow transitive dependencies such as the path of communication from ensemble A to B. Note that if such a path is relevant to the architect, it can also be encoded as a slice. A second scenario for global comprehension is to find all slices in which an ensemble participates. This can be supported by a simple analysis over the defined slices.

All the above said, systematically deriving guidelines for structuring architectural decisions into slices and distributing the work is a matter of performing comprehensive studies and is out of the scope of this thesis.

## 7.3.5 Scalability and Evolvability

Vespucci enables architects to reason about architectural decisions concerning structural dependencies of a set of ensembles in isolation, while treating the rest of the system as a black-box, and to do so in a top-down manner. This is due to (1) Vespucci's support for breaking down the specification into slices, (2) mechanisms for expressing structural rules via a constraint system, (3) a scoping mechanism that enables to quantify locally or globally over the set of affected ensembles, and (4) Vespucci's support for enabling the hierarchical organization of specifications. The latter is a traditional mechanism to govern complexity [Sim62] and will for this reason not be further considered in the following discussion.

**Support for modular reasoning**

Slices enable the architect to focus on constraints that concern individual ensembles or a set of strongly related ensembles. This makes it possible to isolate a small set of related architectural decisions from the rest for the purpose of modularly reasoning about them, while treating the rest as a black-box.

This fosters scalability by reducing the number of ensembles and constraints that need to be considered at once: Each slice in Figure 7.7 (C) contains less ensembles and constraints than the model in Figure 7.7 (A). One may argue that slicing actually increases the overall number of elements (ensembles/constraints) — since some of them are mentioned in multiple slices. However, as they represent the same abstractions in all slices, the overall number of elements that need to be understood remains the same as in the model A.

Consider for illustration the slice depicted in Figure 7.2. It expresses that only SessionManagement and PersistenceManagement may use SQLDialects with the minimum amount of explicitly mentioned ensembles and constraints. No rules governing dependencies between SessionManagement and PersistenceManagement, respectively between those and other ensembles, are specified. The slice in Figure 7.2 models architectural constraints from the perspective of SQLDialects. Dependencies between SessionManagement and PersistenceManagement or between those and other ensembles are irrelevant from this perspective and are, thus, left unspecified. Further, we do not explicitly enumerate all ensembles that are not allowed to depend on SQLDialects.

Vespucci's constraint system for modeling dependencies and the way checking for architecture compliance operates (see previous section) is key to the conciseness of specifications. Slices are checked in isolation. The constraint system interprets the lack of a constraint in a slice as "don't care" in the sense that the presence or absence of code dependencies is ignored. E.g., potential dependencies between SessionManagement and PersistenceManagement are ignored when checking compliance with rules in slice in Figure 7.2. They may well be the subject of specification in other slices to be reasoned on separately.

The role played in this respect by our distinction of incoming and outgoing constraints needs to be highlighted here. It is the use of the incoming constraints in Figure 7.2 that enables us to talk about constraints from the perspective of SQL-Dialects – excluding from consideration any further dependencies in which, e.g., SessionManagement may engage. Incoming/outgoing constraints are "unilateral" – they belong to one ensemble. Without this distinction, we would be left with "bilateral" constraints; mentioning one such constraint that affects SessionManagement would require to mention all other constraints affecting SessionManagement; hence, making it impossible to slice specifications.

The ability to abstract over any dependencies that are not explicitly constrained comes in also very handy when handling ensembles that are expected to be ubiquitously used, e.g., Hibernate‘s Utilities ensemble. Such ensembles would typically contribute a significant amount of complexity to architectural specifications, if the specification approach requires to explicitly mention allowed dependencies. By using a constraint system this complexity can be avoided. The specification would make no mention of dependencies to Utilities, in order to leave it unconstrained.

The ability to state a constraint that affects arbitrary many ensembles without having to enumerate those explicitly is due to the ability to make global statements. Ensembles that are not explicitly mentioned in a slice are reasoned over by global constraints, e.g., the slice shown in Figure 7.2 implicitly states that all other ensembles mentioned in Figure 7.1, 7.2, 7.3, and many more, are not allowed to use SQL Dialects. This specification is much smaller compared to enumerating this fact for all other ensembles constituting the rest of Hibernate. The latter would be necessary, if Vespucci only had allowed and not-allowed constraints and no distinction between local and global scopes.

#### Support for evolution

Due to slicing, architectural models also become easier to extend. First, slices remain stable in case of extensions that do not affect their ensembles/constraints. Second, affected slices are easier to identify. Finally, existing global constraints automatically apply to new ensembles.

Consider for illustration the following scenario that occurred during the evolution of Hibernate from version 1.0 to version 1.2.3. In this step, a new ensemble — called Metadata — to represent Hibernate's new support for metadata was introduced. This change was accommodated mostly incrementally. First, the specification as a whole was extended incrementally by introducing a new slice, referring to the ensembles that Metadata is allowed to use and be used from. Second, the set of existing slices that eventually required revision was restricted to those modeling the dependencies of ensembles referred to in the new Metadata slice. For example, the slice that defined constraints for DataTypes was refined to enable the usage by Metadata. Slices that modeled unrelated architectural decisions, e.g., those governing dependencies of ConnectionProvider (cf. Figure 7.3), did not require any reviewing. Yet, previously stated global constraints carry over to the new ensemble, ensuring e.g., that it does not unintentionally use SQLDialects (slice in Figure 7.2); the usage of non-constrained ensembles, e.g., Utilities, is also granted automatically.

The way the mapping between ensembles and source code is modeled has an effect on the stability of the model in face of evolution of the system. Here we hit a variant of the well-known "fragile pointcut problem". One way to mitigate this problem is by using stable abstractions in the source code in the queries. However, this is not always feasible; in the case study queries had to be adapted as the system evolved. Here the tool support provided by Vespucci offered some help to identify changes in the source code by: a) showing elements that do not belong to an ensemble, b) showing (sub-)queries with empty results; c) specifying that a list of ensembles should be non-overlapping (i.e., to prevent accidental matches). Even so we are aware that better source code query technology and tool support for it is needed; in this thesis, we focus on the modularity mechanisms on top of the query language.

## 7.4 Evaluation of the Modularity Mechanisms

In this section, we evaluate quantitatively the effectiveness of Vespucci's mechanisms to modularize the specification of a software's intended structure. This evaluation is performed from two complementary perspectives: (a) reduction of complexity, which is measured as the number of ensembles and constraints, and (b) facilitating architecture maintainability during system evolution. As a basis we use the re-engineered architecture of Hibernate (c.f. Sec. 7.2), which allows us to study an architecture of a size that is representative for mid- to large-scale projects. We also give a critical discussion of the broader applicability of our results and of threats to the validity of our study at the end of the section.

The goal of our evaluation is to assess the modularization mechanisms of Vespucci and not the accuracy of architectural violation control. Therefore, even though Vespucci is targeted at continuous architecture conformance checking, the identification of violations to the architecture is not the purpose of our quantitative evaluation. Nevertheless, it is important to highlight that in terms of enforcing conformance

Vespucci is able to control violations in the source code similar to related approaches [MNS95, KS03, SJSJ05, EKKM08, TV09, AAA09, dSB12].

### 7.4.1 Scalability

We first analyze the reduction in complexity when reasoning about an architecture specification. This analysis was performed by comparing the architecture of Hibernate 1.0 modeled in the four principal ways schematically depicted in Figure 7.7 and outlined in the previous section. The model with both slices and hierarchies (Figure 7.7, D) was the primary model produced during our study of Hibernate. The other three models were produced to measure the complexity reduction for the different mechanisms (hierarchies, slices, combination of both).

**Scalability with regard to the number of ensembles**

We first compare different mechanisms w.r.t. the number of ensembles referenced by isolated dependency rules. The baseline is a single monolithic specification with a total of 79 ensembles, modeled by following Figure 7.7 (A). The other three models Figure 7.7 (B-D) are quantified in the diagrams in Figure 7.8. The y-axis of all three diagrams denominates the number of ensembles referenced per architectural model.

The diagram on the upper left shows reduction in complexity for hierarchical structuring only. The model is a single specification, but high-level ensembles may be collapsed to reduce the overall number of ensembles to consider at once. The x-axis denominates the number of collapsed ensembles ordered by the number of their sub-ensembles. The values on the y-axis show how many ensembles are referenced after collapsing an enclosing ensemble, i.e., the enclosing ensemble is referenced instead of all its children. The values are accumulated, since multiple ensembles can be collapsed together. For example, in a model with the top five most complex high-level ensembles collapsed, the architect has to consider 41 ensembles at once. When collapsing all ensembles in the hierarchy, we are left with 22 top level ensembles, hence hierarchical structuring reduces the number of ensembles to approx. 27% of the total (22 of 79).

The diagram in the upper right of Figure 7.8 shows the number of ensembles per slice when using only slices (no hierarchies). The x-axis denominates the modeled slices in the decreasing complexity order (decreasing number of referenced ensembles). Almost all slices refer to less than 27% of the ensembles (12% on average). The exemption are the three first slices that capture rules for the following building blocks (of central importance) (i) persisting classes, (ii) persisting collections, and (iii) the interface to Hibernate's internal data types. The combination of both mechanisms (diagram on the lower left side of Figure 7.8), yields a much smaller number

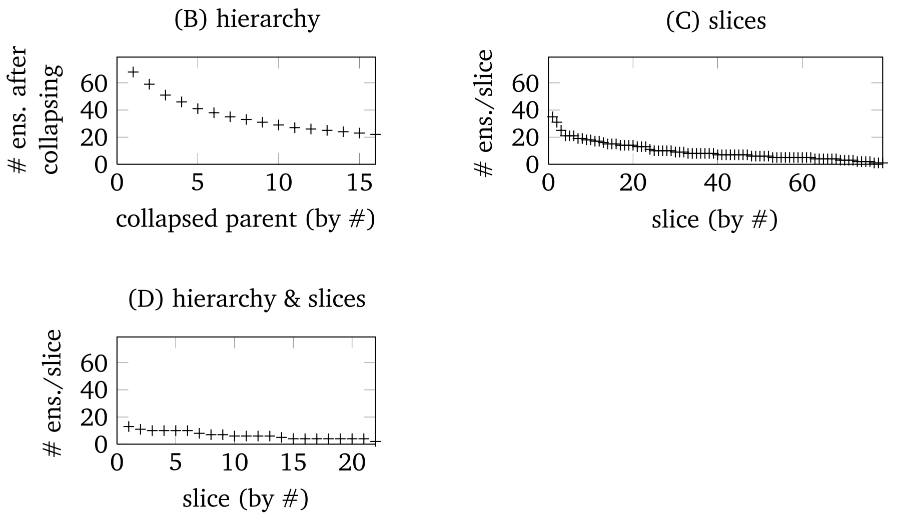


**Figure 7.8:** Comparison of ensemble reduction w.r.t. hierarchies and architectural slices (Hibernate 1.0)

of slices (x-axis), since it focuses on the top-level building blocks. In addition, the combined approach features slightly smaller slices; on average each slice references only 9% of the total number of ensembles.

**Scalability with regard to the number of constraints**

In the following, we compare how much each mechanism reduces the number of constraints used in isolated dependency rules. The comparison is similar to the comparison regarding the number of ensembles and the numbers are shown in Figure 7.9. The x-axis is organized in the same manner as in Figure 7.8. The y-axis denominates the number of constraints that are referenced in each architectural model.

The y-axis for hierarchical structuring (upper left diagram in Figure 7.9) shows the total number of constraints after collapsing an enclosing ensemble. The number includes (i) constraints that are abstracted away, since they are internal to the enclosing ensemble (cf. Figure 7.7 B; 1&2) and (ii) constraints that are abstracted away, since several constraints at the low level are subsumed by a single constraint at the high level (cf. Figure 7.7 B; 1&2 to 4). Both internal and external constraints contribute approx. half of the reduction in constraints (external slightly outweighs

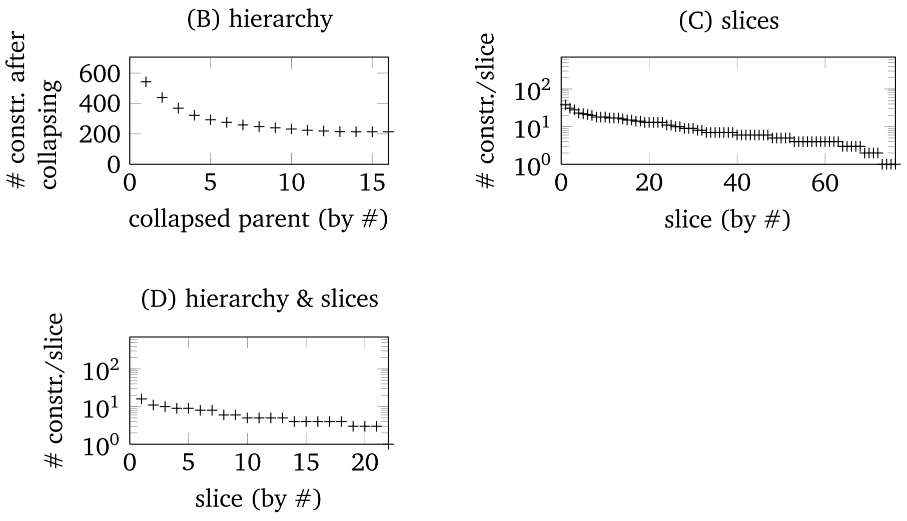


**Figure 7.9:** Comparison of constraint reduction w.r.t. hierarchies and architectural slices (Hibernate 1.0)

internal). As in the evaluation for ensembles, the y-values for the hierarchical composition (B) are accumulated, since we can use several hierarchical groupings together. For the architectural models using slices (C,D) the number of constraints is simply the number of constraints modeled in one slice.

The baseline (A) consists of 705 constraints in a single specification. If we consider the hierarchical model and collapse all enclosing ensembles, approx. 2/3 of the constraints are removed (down to 214, last value in the upper left diagram in Figure 7.9)). In comparison, slices (diagram in the upper right of Figure 7.9) show less than 5% of the total number of constraints and 1,3% on average (9 of 705) per slice. The combination of slices and hierarchical structuring (lower left diagram in Figure 7.9) features slightly smaller slices; on average 0.9% (6.5 constraints) of the total of 705 constraints modeled.

**Scalability with regard to the number of slices**

To control the architecture of Hibernate we have modeled top-level slices comparable to Figure 7.7 (D) and slices for the internal constraints of the 16 ensembles that are further structured; totaling to 35 slices. Thus, the overall number of slices is smaller than the overall number of ensembles (79) and remains manageable. Note that in these models we do not use the total of the 705 constraints. First, three ensembles at the top level (SessionManagement, PersistenceManagement, and User-API) have no slice (and no constraints), for the reason of being used by and using almost all other ensembles. This is an inherent problem of the modularization of the software system and should be treated by refactoring the code base. Second, the modeled top-level constraints subsume several constraints on the internal ensembles. We found the control provided by the top-level constraints mostly sufficient during the evolution of Hibernate. Hence, we modeled detailed constraints only in few cases to further our understanding of the dependencies between selected ensembles.

**Summary**

In this study the hierarchical structuring included 22 ensembles and 215 constraints (both approx. 1/3 of the total). Slices are much smaller; we have to collapse the first seven enclosing ensembles of the hierarchy to reduce the number of ensembles to 35, the number referenced in the most complex slice (persisting classes). Collapsing all ensembles still references 5 times more constraints than the number referenced in the slice for persisting classes. Hence, the modeling approach based on slices scales much better by reducing each slice to 9.5 ensembles and 9 constraints on average. The combination of both mechanisms produces the best results by reducing each slice to 7.1 ensembles and 6.5 constraints on average, which means that a typical slice in the Hibernate model had about 7 ensembles and 6 to 7 constraints. Thus especially the number of constraints that need to be reasoned over at once remains manageable and includes on average only 3% of the constraints of the model using hierarchical structuring, with a maximum of 16 constraints, or 7% of the constraints in the single hierarchical model.

### 7.4.2 Evolvability

To evaluate the effectiveness of Vespucci in supporting architecture evolution we have compared a single model with hierarchies (Figure 7.7 B) to slices with hierarchies (Figure 7.7 D). The results are summarized in Table 7.2. The first three Columns show the analyzed version, its release year, and the number of LoC as an estimate for the size. Columns four and five characterize the architectural evolution in terms of ensembles and their queries. Overall, the number of ensembles has doubled. Column six shows the total number of slices in each version. We followed the methodology of one slice per ensemble – hence, the number of slices roughly follows the number of ensembles, with the exception of those ensembles that were not constrained (c.f. Sec. 7.4.1). Column seven shows that on average 33% of all slices (1/3 of the architecture specification) remained stable w.r.t. the previous version. The least stable revisions were the first and the last one. In the first revision,

| Vers. | Release Year | LoC | # Ens. (Top-Level) | Added/Removed Ensembles | # Slices (Top-Level) | Stable Slices | Dependencies reviewed using: Slices (Avg.) | Slices (Max.) | Single Model |
|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 2002 | 14703 | 22 | n/a | 19 | n/a | n/a | n/a | n/a |
| 1.2.3 | 2003 | 27020 | 26 | +5 / -1 | 23 | 4 (21%) | 2.4 | 6 | 61 |
| 2.0 | 2003 | 22876 | 28 | +5 / -3 | 25 | 12 (52%) | 1.9 | 6 | 40 |
| 2.1.6 | 2004 | 44404 | 30 | +2 / -0 | 27 | 9 (36%) | 2.6 | 6 | 38 |
| 3.0 | 2005 | 79248 | 36 | +9 / -3 | 33 | 8 (30%) | 4.5 | 8 | 118 |
| 3.6.6 | 2011 | 106133 | 39 | +3 / -0 | 36 | 9 (27%) | 5.0 | 11 | 87 |

**Table 7.2:** Analysis of the evolution of Hibernate's architecture

Hibernate was close to its inception phase, hence requiring more adaptations to its features. The last revision was the most extensive in terms of the timespan covered. The last three columns compare the complexity involved in performing the required updates of the architecture specifications. Columns eight and nine show the average, resp. maximal number of ensembles per slice, whose dependencies were updated, in the approach using slicing. The last column shows how many dependencies were updated in the single hierarchical model. On average only 4% to 6% of the number of dependencies updated in the single model were reviewed per slice (the maximum ranging between 7% and 15%). This reduction in complexity of the updates per slice is comparable with the reduction of the number of constraints between (B) and (D) in Figure 7.9.

The numbers indicate that the maintenance of individual slices is much easier than the evolution of the single architecture model and confirm what is qualitatively discussed in the previous section.

### 7.4.3 Threats to Validity

We identify two threats to the *construct validity* of our study. First, the reverse engineering of Hibernate's architecture was not performed by the original Hibernate developers. Hence, the resulting architecture design may not accurately reflect

Hibernate's real/intended architecture, which may lead to inconsistencies in the results. To mitigate this threat, the architectural model was created by three people — one student, one PhD candidate and one post-doctoral researcher — that together have many years of experience on object-relational mapping frameworks. Further, we extensively studied the available documentation to make sure that the model is true to Hibernate's architecture. Yet, it is likely that a different group would reverse engineer a different architectural model. But, it is unlikely that the architecture would be such different that our evaluation would become invalid. A second threat to construct validity is that other architects may modularize the architecture specification differently, resulting in a different number and scope of slices. However, the approach that we followed — roughly creating one slice per top-level ensemble — has proven to be useful and can at least be considered as one reasonable approach.

Threats to *conclusion validity* in our study could be related to the number of ensembles and architectural constraints involved in our analysis. We tried to mitigate this threat by considering an architectural model of a significant complexity. Our analysis concerned an architectural model that involved 79 ensembles, more than 700 architectural constraints and 35 architectural slices for Hibernate 1.0.

The main issue that threatens the *external validity* of our study is that it involved a single software system. To mitigate this threat we have used a well-known medium-size framework, which has been designed by taking into consideration guidelines and good practices. These characteristics allow us to analyze the benefits of Vespucci when modeling architecture designs of well-modularized software systems. In addition, we have discussed the properties of Hibernate's architecture that influence the results and compared them to other studies. However, we are aware that more studies involving other systems should be performed in the future. All our findings should be further tested in repetitions or more controlled replications of our study.

## 7.5 Evaluation of Incremental Performance

To evaluate the incremental maintenance of Vespucci slices, we have measured (i) the materialized memory required to maintain the architectural violations for slices, (ii) the runtime for maintaining the violations when performing incremental modifications to the Hibernate code base and (iii) the runtime for maintaining the violations when performing modifications to the architectural specification, i.e., ensemble definitions and constraints.

The evaluation follows a similar pattern as found in the last chapter. That is we measure incremental runtime and memory materialization. We do not compare to a non-incremental implementation for Vespucci. The reason is that, if we follow a strictly non-incremental approach, each slice must be checked in isolation from ground up, which means computing ensemble extents, computing dependencies in the code and correlating them to the ensembles and finally check them against the constraints. Even when pre-computing the ensemble extents and the correlations between code and ensemble dependencies, each slice still requires to filter the pre-computed data to see what is relevant for the constraints formulated in the slice. Hence, such an approach is – without further evaluation – unlikely to perform much better than the incremental approach.

In the following we first provide an overview of the datasets used during the evaluation. Then the performance of incremental maintenance is discussed in the order of (i) materialization of memory, (ii) maintenance of code modifications and (iii) maintenance of architectural specification changes.

### 7.5.1 Overview of measured Datasets

In the following we use excerpts of the Hibernate case study for the evaluation. The main focus lies on the later versions of Hibernate, i.e., 3.0 and 3.6, since these provide larger volumes of code and the Vespucci slices of these versions have reached a complexity that allows us make a evaluation of the performance of architecture checking for a real mid- to large-sized project.

#### Memory Materialization

To measure the memory materialization we use the code bases of Hibernate 3.0 and 3.6.6. Table 7.3 characterizes both versions in terms of the number of elements found in the code base ($2^{nd} - 5^{th}$ columns), as well as the time required for parsing ($6^{th}$ column). As in the last chapter we measure how in-memory databases deal with materialized base relations by materializing all base relations in the SAE ($7^{th}$ column) and in the deductive logic engine SWI ($8^{th}$ column).

| Dataset | # elements | | | | parse time | mat. mem. (MB) | |
|---|---|---|---|---|---|---|---|
| | classes | fields | methods | instr. | (ms) | SAE | SWI |
| 3.0 | 940 | 3 280 | 10 220 | 169 024 | 93.5 | 17,3 | 62,9 |
| 3.6.0 | 1 972 | 5 563 | 17 997 | 323 220 | 249.0 | 33,1 | 117,2 |

**Table 7.3:** Overview of the benchmarked dataset

As can be seen the amount of elements in the code base has nearly doubled between Hibernate 3.0 and Hibernate 3.6. As a consequence the amount of memory for the materialization has also doubled.

### Incremental Runtime for Code Changes

As a change set we replayed changes in the minor versions of Hibernate 3.6, that is the version 3.6.6 was replayed to version 3.6.10. Hibernate 3.6 was chosen, since it provides the highest workload for the incrementalization for both code size as well as size of the slices that define the architecture of this version and must be maintained w.r.t. the code changes. The replay is very coarse grained, since we just unpacked the release jar of version 3.6.10, computed changes in compiled files and grouped them by compilation time. The latter provides a small amount of granularity, yet is not comparable to the granularity at which developers changed the code. Nevertheless, this treatment allows us to measure the incremental runtime w.r.t. different workloads.

The incremental changes are characterized in detail in Table 7.4. The table follows the general layout discussed in the last chapter (cf. Sec. 6.2.2), hence we will not repeat all details. The table numbers each event in the sequence of changes by a value $x$ that is later used as the x-axis for the measured runtime values. Details of size of the changes at the class and method scope are provided in the $3^{rd} - 6^{th}$ and $7^{th} - 10^{th}$ respectively. The table omits a detailed ratio for the size of the class scope vs. the size of the method scope, since these are 100% in all cases except $x = 3$ - $x = 6$. In these four cases the ratio is around 20%.

Compared to the recorded developer changes of the last chapter, the events consider on average a larger amount of classes and methods. In addition, we find several events that only add new classes (e.g., $x = 7$ - $x = 13$). Furthermore, the cases of $x = 4$ - $x = 6$ are events that change a large amount of classes and must rather be seen as major rebuilds than as incremental changes by developers. Nevertheless, the events are fine-grained enough to obtain a a good measure of the work performed for incremental maintenance, although in the light of the data presented in the last chapter they can not be seen as representative of working with Vespucci in an IDE.

### Incremental Runtime for Specification Changes

To measure the effectiveness of the incremental maintenance w.r.t. changes in the architectural specification, we have replayed the adaptations made to the architecture description between Hibernate 3.0 and Hibernate 3.6.6.

The changes replayed for this evaluation reflect two different cases of adaptations: (i) changes in ensemble queries and (ii) changes of constraints in the slices. The replay is performed separately for both. First, for the changes in ensemble queries we simply updated the ensembles in Hibernate 3.0 with the final queries we use

| $x$ | $\Delta$ | class scope ($\circ$) | | | | method scope ($\blacktriangle$) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $\Delta_m^{add}$ | $\Delta_m^{del}$ | $\Delta_i^{add}$ | $\Delta_i^{del}$ | $\Delta_m^{add}$ | $\Delta_m^{del}$ | $\Delta_i^{add}$ | $\Delta_i^{del}$ |
| 1 | 54 | 332 | 100 | 3550 | 2276 | 332 | 100 | 3550 | 2276 |
| 2 | 7 | 56 | 3 | 338 | 45 | 56 | 3 | 338 | 45 |
| 3 | 97 | 830 | 791 | 9631 | 8908 | 71 | 32 | 2742 | 1478 |
| 4 | 371 | 2763 | 2698 | 37665 | 37248 | 203 | 138 | 7448 | 5932 |
| 5 | 445 | 4221 | 4192 | 112441 | 113123 | 102 | 73 | 19435 | 18318 |
| 6 | 1084 | 7981 | 7662 | 166747 | 161657 | 578 | 259 | 51473 | 41341 |
| 7 | 37 | 191 | 0 | 5021 | 0 | 191 | 0 | 5021 | 0 |
| 8 | 6 | 47 | 0 | 1084 | 0 | 47 | 0 | 1084 | 0 |
| 9 | 45 | 329 | 0 | 7879 | 0 | 329 | 0 | 7879 | 0 |
| 10 | 2 | 27 | 0 | 270 | 0 | 27 | 0 | 270 | 0 |
| 11 | 2 | 24 | 0 | 239 | 0 | 24 | 0 | 239 | 0 |
| 12 | 1 | 9 | 0 | 845 | 0 | 9 | 0 | 845 | 0 |
| 13 | 1 | 7 | 0 | 742 | 0 | 7 | 0 | 742 | 0 |
| 14 | 59 | 316 | 5 | 13539 | 518 | 316 | 5 | 13539 | 518 |
| 15 | 50 | 381 | 0 | 3692 | 0 | 381 | 0 | 3692 | 0 |
| 16 | 56 | 516 | 144 | 10987 | 2522 | 516 | 144 | 10987 | 2522 |
| 17 | 142 | 752 | 3 | 11233 | 36 | 752 | 3 | 11233 | 36 |
| 18 | 59 | 293 | 0 | 5731 | 0 | 293 | 0 | 5731 | 0 |
| 19 | 172 | 952 | 15 | 19318 | 59 | 952 | 15 | 19318 | 59 |
| 20 | 37 | 295 | 0 | 6030 | 0 | 295 | 0 | 6030 | 0 |

**Table 7.4:** Overview of the change set from Hibernate 3.6.6 to 3.6.10

in Hibernate 3.6.6. This treatment does not completely reflect the way Vespucci was used, since we adapted queries in a more fine-grained incremental manner, yet it provides a valid measurement of the work performed when updating queries. Second, for the constraints in the slices, we replayed every constraint change in each slice as a single event. This treatment very much reflects how Vespucci was used, since we typically added or removed single constraints in a single slice.

There is no need to further characterize the changes in the datasets. As we will see in the evaluation all change events elicit a very similar runtime. This is quite natural, since query changes (and changes in constraints) consider all current code elements (or their dependencies). For example, if a query is slightly changed to include a single additional class, the complete set of code elements is searched for the class. The same is true if the query is changed to include an additional package with many classes, yet the result set of the query is larger. Hence, some changes propagate small updates to the results and some propagate bigger updates, yet the propagation only amounts to a small portion of the performed work.

### 7.5.2 Memory Materialization

The memory materialization in Vespucci has two distinct properties. First Vespucci requires to materialize and maintain a large amount of data due to the fact that it must cope with changes in the specification. In essence all code elements and all dependencies must be materialized, even if some do not currently contribute to ensembles and their constraints. The reason is that Vespucci allows to formulate queries similar to ad-hoc queries in a database. For example, the user changes the definition of an ensemble query and the system should immediately compute the new result of the query. To make this efficient the potential elements that can be queried are kept in memory. The same is true for dependencies, i.e., the current specification of constraints between ensembles can cover only parts of the dependencies, but the specification is subject to change.

The second property of Vespucci is that for efficient maintenance all dependencies are doubly indexed, once for their source and once for their target. In short, each dependency has a source and target code element and Vespucci needs to determine what the source and target ensembles are and essentially raise the dependency to the architectural level, to see whether constraints are violated. To do this the dependencies must be joined to the ensemble elements, once on the source and once on the target. Hence, the amount of memory used for indexing is rather large.

The memory materialized when using the SAE as an engine for Vespucci is shown in Table 7.5. The first column lists which version of Hibernate was considered, the $2^{nd}$ column lists the memory materialized by the SAE for maintaining architectural violations in the Vespucci approach. Note, that the numbers indicate the total amount of memory when maintaining architectural violations for all slices. However, the largest amount of memory is consumed by the indexing discussed above and the materialized memory is very much independent of the number of slices that are maintained. The $3^{rd}$ and $4^{th}$ columns compare the materialized memory of Vespucci against materializations of base relations in the SAE and in XSB (cf. Table 7.3).

Compared to the materialization of the base relations Vespucci requires approx. 80% of the memory and only approx. 20% of the memory materialized in the SWI system. Here again we must stress that both compared values reflect unindexed data and, hence, Vespucci must be considered as far superior, since it materializes less memory with a a heavy indexing. As we can see the ratios between Vespucci and the base relations do not differ much across the different Hibernate versions. This is quite natural, since the code elements and the dependencies can be expected to contribute similar fractions to the entire code base. For example, if 25% of the instructions were dependencies in Hibernate 3.0, we can expect that the ratio in Hibernate 3.6.6 is also around 25%.

| Hibernate Ver. | Memory (MB) | | |
|---|---|---|---|
| | SAE | $\frac{\text{SAE}}{\text{Base}}$ | $\frac{\text{SAE}}{\text{SWI}}$ |
| 3.0 | 13.75 | x0.8 | x0.22 |
| 3.6.6 | 25.25 | x0.76 | x0.22 |

**Table 7.5:** Memory materialization for maintaining architectural violations

Overall the amount of materialized memory is such that we can consider the in-memory approach of the SAE scalable for continuous architectural checking of mid to large sized projects. In cases of larger projects, such as the JDK and the Scala compiler – used as data in the last chapter – more memory is required. However, if these projects do not have an overly different ratio of dependencies vs. total code elements, the values for materialized base relations (466,8 and 92,5 MB) are valid upper bounds for the required memory. Hence, in the case of the Scala compiler this is well acceptable to continuously check for architectural violations. In the case of the JDK it is still feasible, albeit ranges of 500+ MB can be considered too much on some developer systems, since the memory is permanently blocked for the architecture checking.

### 7.5.3 Incremental Runtime for Code Changes

The incremental runtime for the maintaining the violations when the code changes is depicted in Figure 7.10. The measured times for each event (cf. Sec.7.5.1) are plotted on a logarithmic scale y-axis that shows the time taken to compute the incremental update in milliseconds. The x-axis shows the change events in the order in which they appeared in our change set. For simplicity the events are numbered from 1 to 20 and the value at $x = 0$ denotes the initial computation performed on the entire code base (right after starting the IDE). For the majority of the changes the incremental runtime is one or two orders of magnitude faster than the initial computation. The notable exceptions are of course the events at $x = 4$ - $x = 6$ we have already classified as major rebuilds rather then incremental changes. From the data we can see that small changes can be computed in the course of 10-30 milliseconds. Larger changes requires times in the range of 200-500 milliseconds. Note again that the incremental changes are very coarse grained. The change set used in the last chapter contained many events that are comparable to $x = 2$ or $x = 10$ to $x = 13$ . Thus, when using Vespucci for continuous architecture checking, violations are highly likely to be recomputed in the course of 10-30 milliseconds for the majority of changes developers make in their IDE.

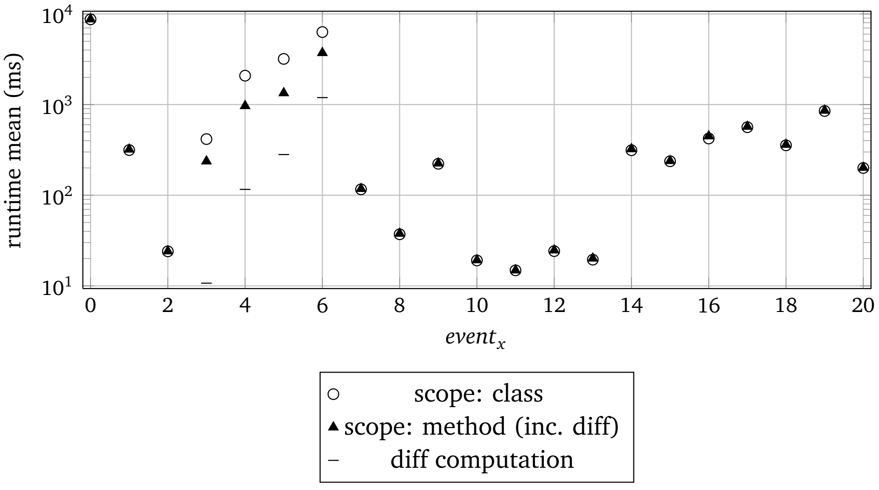


**Figure 7.10:** Incremental runtime for maintaining violations during code changes

## 7.5.4 Incremental Runtime for Specification Changes

In the following the incremental runtime for changes to the architectural specification is measured by (i) replaying changes to the ensemble queries made between Hibernate 3.0 and Hibernate 3.6.6. and (ii) replaying the changes made to the constraints between these two Hibernate versions. In both cases we use the code base of Hibernate 3.6.6 as the input over which the architecture is maintained. Thus, the replay reflects the work we performed when we evolved the specification from one version to the next, i.e., running the old specification on the new code base and then successively adapting queries and constraints to meet new expectations and architectural decisions not eminent in the old code base or categorizing new dependencies as violations.

### Changes to Ensemble Queries

In total we had 36 ensembles in Hibernate 3.6.6 (cf. Table 7.2), 33 old and 3 new. Figure 7.11 depicts the time to update the query of each ensemble. Note that we simply list all ensembles whether their query has changed or not, since all have comparable workloads. The reason is that Vespucci is currently not smart when it comes to query recompilation. It basically removes the old extent of an ensemble and then adds the new extent, without analyzing which elements are the same.

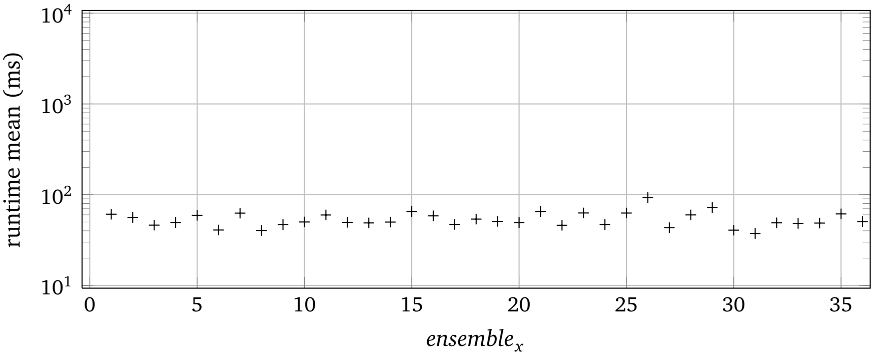


**Figure 7.11:** Incremental runtime for maintaining violations during ensemble query changes

Overall the changes can be computed in little time (40-100 milliseconds). None of the ensembles take exceedingly more or less time. The reason being that ensembles elements are recomputed quite fast, since the code elements (classes, methods, fields) are materialized. Hence, the computation in general can be performed fast. Furthermore, the number of code elements is comparatively small, i.e., Hibernate 3.6.6 has less than 25 000 elements, which means that traversing the elements to compute the new ensemble extent is quite fast when performed in-memory.

Note that Vespucci's current treatment of recomputing ensemble extents means the runtime depends on the size of the project. Future versions of Vespucci should alleviate this problem, so that large projects do not have an incremental runtime with an order of magnitude higher than we have seen here. However, this also means that the runtime shown in Figure 7.11 is essentially a kind of worst case scenario and that – given a better treatment of query recompilation – Vespucci can perform better.

## Changes to Ensemble Constraints

For this evaluation we focus on the constraints changed at the highest level of the architecture, i.e., between top-level ensembles. At this level we changed 42 constraint in Hibernate 3.6.6; 30 were added and 12 were removed. Figure 7.12 depicts the time required to maintain to the architectural violations for each constraint change.

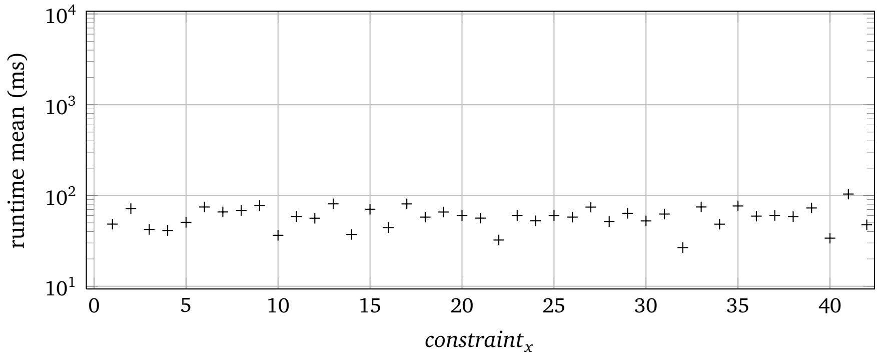


**Figure 7.12:** Incremental runtime for maintaining violations during ensemble constraint changes

Overall the changes can be computed in little time (20-100 milliseconds). The runtime for changing individual constraints also does not differ much, which deserves some discussion. The overall uniformity of the runtime seen in Figure 7.12 is essentially due to the fact that the number of dependencies between code elements in two ensembles does not differ much across ensembles. The dependencies are the determining factor for the computation, since – for a given constraint – all code level dependencies between the source and target ensembles are now either allowed or disallowed. Hence, if a constraint is changed all these dependencies between the source and target ensemble must be re-analyzed. Thus, only source and target ensembles with a high number of dependencies between them require more time for the incremental maintenance.

Note that we used global constraints between top-level ensembles in the architecture, which deserves some attention, since the above argument is between two ensembles. While global constraints – in essence – express one-to-many (or many-to-one) relations between ensembles, we used them together in the context of more global constraints on the same ensemble. Thus, adding (removing) a single global constraint to an ensemble boils down to re-analyzing dependencies between only two ensembles. Consider for example the global constraint between SessionManagement and HQL in Figure 7.1. Due to this constraint all other ensembles (except SessionManagement) are already disallowed to use HQL. Adding a second global incoming constraint (e.g., from SQLDialects) means that this fact has changed only for SQLDialects, all other ensembles are still disallowed to use HQL. Hence, the incremental maintenance only requires re-analysis of the dependencies between two ensembles. This was the case for all constraint changes in Figure 7.12.

## 7.6 Related Work

Closely related to Vespucci are approaches that support checking the conformance between code and architectural constraints on static dependencies [MNS95, KS03, SJSJ05, EKKM08, TV09, AAA09, dSB12]. The key difference is that none of the above approaches (nor other related work) offers the ability to modularize the architecture description into arbitrary many slices. They rather require a self-contained monolithic specification of the architecture, which does not support the kind of black-box reasoning enabled by slices (cf. Sec. 7.3.5). In the following, we discuss the above approaches separately; a summary of their support for the features elaborated in Sec. 7.3 is presented in Table 7.6.

Reflexion Models (RM) [MNS95] pioneered the idea of encoding the architecture via a declarative mapping to the source code. RM is an analytical approach that uses the modeled system architecture to generate deviations between source code and planned architecture, which is reviewed by the architect. The RM approach is not a constraint system, but rather requires the specification of the complete set of valid dependencies. Omission of dependencies is interpreted as "no dependency is allowed". Other approaches extend RM by (i) incorporating hierarchical organization [KS03], (ii) visual integration into the Eclipse IDE [KMNL06] and (iii) extending the process to continuously enforce compliance of structural dependencies between a planned architecture and the source code [RLGBAB08].

Sangal et al. [SJSJ05] discuss the scalability issue of architecture descriptions and propose a hierarchical visualization method called design structure matrices (DSMs), which originates from the analysis of manufacturing processes. The key advantage is the notation (matrices) that facilitates identification of architectural layers via a predominance of dependencies in the lower triangular half of the matrix. DSM features a very verbose constraint system. For example, exemptions on lower level ensembles are encoded by the order in which rules are declared, e.g., by first allowing PersistenceManagement to use SQLDialects and then disallowing the use of ConcreteDialects. While effective, this approach requires a carefully crafted sequences of constraints.

In previous work [EKKM08] we proposed an approach to continuous structural dependency checking; integrated into an incremental build process. As in Vespucci, we referred to conceptual building blocks as ensembles. However, the specification of architectural constraints has been completely revised for Vespucci. Previously we have defined LogEn; a first order logic DSL, that integrated query language and constraint specification. However, the meaning of a violation, i.e., a constraint, is defined by the end-user, which is complex in first order logic. Hence, we provided a visual notation (VisEn), which is less complex, but focuses on documenting the

| | Reflexion Models (RM) [MNS95] | Hierarchical RM [KS03] | DSM [SJSJ05] | LogEn/VisEn [EKKM08] | DCL [TV09] | Vespucci |
|---|---|---|---|---|---|---|
| Architectural slices | - | - | - | - | - | ✓ |
| Constraint system[1] | - | - | + | +++/-[2] | ++ | +++ |
| Hierarchies | - | ✓ | ✓ | ✓ | - | ✓ |
| Dependency Kinds | - | - | - | - | ✓ | ✓ |

[1] - (non existent) to +++ (very expressive)
[2] LogEn is very expressive; VisEn does not offer a constraint system

**Table 7.6:** Comparison with the state of the art

architecture and hence is not a constraint system, but requires explicit modeling of all dependencies. The focus of this work was on the efficient incrementalization of the checking process, hence slicing architecture specifications into manageable modular units was not supported.

Terra et al. [TV09] propose a dependency constraint language (DCL) that facilitates constructive checking of constraints on dependencies; discrimination of dependencies by kind is also supported. DCL offers a textual DSL for specifying constraints. DCL's constraint system is closest to Vespucci's, and can express the not-allowed, expected and incoming constraints. Yet, it lacks outgoing constraints and a scoping mechanism such as global/local constraints, which goes hand in hand with the lack of support for slicing specifications into modular units. The language supports no inherent hierarchical structure in the architecture.

A number of commercial tools have been documented (c.f. [dSB12]) for checking dependency among modules and classes using implementation artefacts, e.g., Hello2Morrow Sotograph [Sot12]. However, the scope of these tools is limited; they are only able to expose violations of "certain" architectural constraints such as inter-module communication rules in a layered architecture. That is, they do not provide means for expressing system constraints.

In [AAA09] the authors propose a technique for documenting a system's architecture in source code (based on annotations) and checking conformance of code with the intended architecture. The representation of the actual architecture in the source code is hierarchical, however, they do not support slicing of specifications in modular units and the modular architectural reasoning related to it.

Languages specialized on software constraints like SCL [HH06], LePUS3 [GNE08], Intensional Views [MKPW06], PDL [MDVW07] and Semmle .QL [dMSV$^+$08] can be used to check detailed design rules e.g., related to design patterns [GHJV95]. However, they are not expressive enough for formulating architectural constraints in a way that allows to abstract over irrelevant constraints, when reasoning about a part of the architecture in isolation.

In [SR99] authors introduce a technique to identify modules in a program called concept analysis. A concept refers to a set of objects that deal with the same information. The authors observed that, in certain cases, there is an overlap among concept partitions. The notion of slice in Vespucci could be considered as conceptually close to the notion of concept overlapping since Vespucci supports the grouping of ensembles that are ruled by the same design decisions. Other than that slices and concepts are different in the way they are defined and used. Concepts emerge while slices are explicitly modeled. Moreover, use case slices [JN04] are also related to our notion of slices, but focus on the modularization of the scattered and tangled implementation of use cases.

In [GBSC09] the authors discuss foundations and tool support for software architecture evolution by means of evolution styles. Basically, an evolution style is a common pattern how software architectures evolve. This case study complements our work by helping to identify evolution styles w.r.t. a software's structural architecture. The evolvability of a software that is developed in a commercial context is also discussed by Breivold et al. [BCE08]. They propose a model that — based on a software's architecture — evaluates the evolvability of the software. Based on our experience, the model also applies to open-source software, such as Hibernate. Aoyama [Aoy02] presents several metrics to analyze software architecture evolution. He made the general observation that discontinuous evolution emerges between certain periods of successive continuous evolution. Our case-study confirms this observation. We observed that some parts of Hibernate evolved continuously, while in other parts the evolution was disruptive. Using our modular architecture conformance checking approach architects can focus on continuous and disruptive slices individually.

## 7.7 Discussion

In this chapter, we proposed and evaluated Vespucci, an approach to modular architectural modeling and conformance checking. The key distinguishing feature of Vespucci is that it enables to break down specification and checking into an arbitrary number of models, called architectural slices, each focusing on rules that govern the structural dependencies of subsets of architectural building blocks, while treating the rest of the architecture as a black box. Vespucci features an expressive constraint system to express architectural rules and also supports hierarchical structuring of architectural building blocks.

To evaluate our approach, we conducted an extensive study of the Hibernate framework, which we used as a foundation for a qualitative evaluation, highlighting the impact of Vespucci's mechanisms on managing architectural scalability and evolvability. We also quantified the degree to which Vespucci can (a) reduce the number of ensembles and constraints that need to be considered at once, and (b) facilitates architecture maintainability during system evolution. For this purpose, we played back the evolution of Hibernate's structure. The results confirm that Vespucci's is indeed effective in managing complexity and evolution of large-scale architecture specifications. However, given that we have only done one extensive case study so far, we need to carry out further case studies before final conclusions on the scalability of the approach can be made. Furthermore, we quantified the materialized memory and the incremental maintenance time required by the SAE that was used as the engine for Vespucci. The results confirm that architecture conformance checking can be integrated into an IDE with low memory overhead at least for mid- to large-sized projects. The runtime for incremental maintenance is low enough to be considered real-time in most cases.

In future work, we will explore how IDE support can help to "virtually merge" slices into a virtual global architectural model and to automatically create "on-demand" slices to help architects to plan a software's evolution. Obviously, further empirical studies are needed to better understand the benefits and limitations of Vespucci. New studies need to be designed to asses the impact of the approach on architect's productivity and on software quality. In this respect it would be interesting to study the effect of modularization w.r.t. enlarged control, i.e., the modularization allows to efficiently maintain an architecture containing more ensembles, which provides tighter control over the source code.

# 8 Conclusions and Future Directions

This chapter presents the conclusions drawn from our work (Sec. 8.1) and discusses possible future directions for the work (Sec.8.2).

## 8.1 Conclusions

This thesis has proposed and evaluated an approach for automatic incrementalization of declaratively specified static analyses. The distinguishing feature of our approach is that only a minimum of main-memory is materialized, i.e., permanently blocked by the intermediate results of the incremental computation, which is an important property for a real-world integration into an IDE. Furthermore, the incremental computation is designed to be fast enough to be considered real-time, i.e., changes in results can be computed in a manner of a few tens of milliseconds. The static analyses are formulated in an SQL-style EDSL that is translated to the general formalism of relation algebra with extensions for nested collections and recursion. The EDSL allows to phrase the analysis with a full dataset in mind and incrementalization as well as optimizations to reduce the blocked memory are performed automatically. A key technique that enables declarative specifications to be memory efficient are incremental scopes, which enable the analyses to use transient memory for computations over data that is always changed together, hence, only small amounts of memory are permanently blocked. The static analyses and the accompanying incrementalizations and optimizations are formulated on top of a general purpose database-like system that can be reused in other context than in static analyses.

To evaluate our approach we conducted several case studies with static analyses that fall under different categories w.r.t. the characteristics of the analyses, i.e., computational complexity of the queries and materialized memory requirements. Bug finders, code smell detectors and metrics utilize simple heuristics and use structural queries or evaluate simple instruction sequences, without considering control and data flow. More concise bug finders utilize data flow analyses to detect malicious code. Architecture compliance checking requires a mapping from code to architectural units (ensembles) and a global correlation of all source code dependencies.

We have shown that there exists a wide range of static analyses where materialized memory in our approach stays well below the memory required to materialize the data of the entire program. More importantly the materialized memory in our approach included indexing data and, yet, materialization was still well below the materialization of the entire program. We have compared the approach to memory required in a deductive database and shown that in our approach memory stays within boundaries of 5% - 10% of the memory used by the deductive database (which was again unindexed). The exception were the architecture analysis, which still required only 20% - 25% of the memory required for an un-indexed materialization in the deductive database.

We have compared the non-incremental runtime (i.e., the initial computation prior to incrementalization) to a reference implementation. The majority of the analyses required comparable runtimes. The computationally more complex analyses are up to three times slower for computing the initial results of the incrementalization, which, however, quickly pays of (i.e., after three runs). The incremental runtime was in general fast enough to be considered real-time, i.e., developers are informed of changes in the results with out a perceived delay. The majority of changes was computed in less than 10 milliseconds and required only 20-30 milliseconds for larger changes. Larger changes in our case means changes that we removed and added 20-30 methods with a total of approx. 2 000 - 3 000 removed (and the same amount of added) instructions.

Finally, the incremental scoping mechanism provided significant improvements for the materialized memory. For the bug finders, code smell detectors we have shown that the reduction can be over 50% of the materialized memory. In our evaluation we have compared two scopes. The class scope offered the largest opportunities for saving memory at the cost of runtime, whereas the method scope could not optimized some queries, yet, has a better runtime. For these two scopes we have shown that the increase in runtime for the class scope still allows majority of incremental changes to be computed within a time frame of 10 milliseconds. Interestingly the method scope is a strict requirement for a purely declarative formulation of an efficient incremental in-memory computation of data flows. Without the method scope large real-world programs are not fit to to be indexed for an in-memory computation and, hence, either require larger runtime, or must be formulated non-declaratively.

Overall our results confirm that the in-memory approach for an incremental computation is feasible and produces results in real-time for a large variety of static analyses queries. We have included the JDK as a large code base and shown that the absolute values for materialized memory are within a bound of less than 100 megabyte for the sampled analyses. In terms of memory this yields a much better scalability compared to the overall requirements of deductive databases. Nevertheless, the feasibility is naturally tied to the program size and the performed analyses. With respect to the latter, the architecture analyses can be considered as one worst-case scenario for memory, since the analysis performs a global correlation of all dependencies. Dependencies are a large factor for non-localized data that can not be correlated within a class or method scope. In this respect we would require absolute values of up to 500 megabyte for large projects, which is still feasible given todays computers, yet comes into serious memory ranges that should not be blocked permanently. We discuss possible future extensions to overcome this problem and enable even more memory extensive analyses in the next section.

## 8.2 Future Directions

An immediate next step for the proposed language-integrated database is to provide tighter integration into the host programming language. As discussed in Sec. 4.3, tighter integration provides further automation for the query optimizer, which is currently not fully automated. While our current implementation was feasible enough to formulate the sampled static analyses, the integration is also the next step for automatically optimize multiple queries with common sub-expressions, of which we found a relevant portion in our sampled analyses to make such an optimization worthwhile.

In connection to a tighter integration into the host programming language it would be feasible to analyze imperative computations formulated in the host language and deduce incremental equivalents. As discussed in Section 4.3.5 such an analysis can be employed to integrate object-oriented data representations and rewrite them to a more relational style. This relational representation has the benefit that objects, which do not contribute to a query's results, yet are referenced by other materialized objects, are not permanently kept in-memory. Furthermore, the deduction of incremental equivalent computations can also be used to provide a front-end for query writing that does not use the declarative SQL-style. Yet, in our experience the declarative style has many benefits, e.g., conciseness of the queries. Nevertheless, derivations from imperative code can be used to incrementalize legacy code that was not formulated in our framework.

We will further explore the application of our approach to a larger set of static analyses. This includes more precise intra-procedural analyses as well as inter-procedural analyses. With respect to the latter we briefly considered the call graph construction in Chapter 6, which is an initial step for inter-procedural data flows. Yet, we will sample real inter-procedural data flow analyses, e.g., from the security domain, to measure the effectiveness for these analyses. We can expect a larger amount of consumed memory for these analyses, since inter-procedural data flow is typically based on the notion of analyzing methods w.r.t. different calling contexts. That is for each caller of a method the called method can use more detailed knowledge about concrete types of parameters etc. The number of contexts can become quite large, and such analyses – in general – require much memory for these contexts.

An interesting observation of the currently sampled analyses is that the vast majority of memory is required for indexing, even though we used filtered indices. Thus, an interesting extension of our approach is the persistence of the filtered indices. Persistence can easily be achieved by key-value stores such as Berkley DB[1]. Key-value stores can offer persistence without transactionality, hence, providing only a low overhead. The persistence will benefit from the fact that only a minimal necessary minimum of data is persisted, just as we currently materialize in-memory only a minimal amount of data. The extension enables analyses over even larger code bases and makes more complex analyses such as inter-procedural analyses feasible for real-world integration in an IDE. However, this extension requires careful analysis w.r.t. impeding the incremental runtime.

Finally, it will be of great interest to explore further applications of the database in contexts other than static analyses. Today many tools are integrated into IDEs that can benefit from incrementalizations, for example model-driven tools or language workbenches for external DSLs have similar qualities as static analyses over code. In fact, our application of the database to the Vespucci tool was already one such application. In this scenario we did not only use the database for incrementalization of code changes, but also for incrementalization of the specification, i.e., the architectural model. Yet, there are certainly other tools and approaches that can benefit from an automated incrementalization and will be easier to write using a declarative query language.

[1] http://www.oracle.com/technetwork/products/berkeleydb

## Bibliography

[AAA09] Marwan Abi-Antoun and Jonathan Aldrich. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. OOPSLA, 2009.

[AAD$^+$96] Sameet Agarwal, Rakesh Agrawal, Prasad M. Deshpande, Ashish Gupta, Je F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. 1996.

[AB93] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. Technical report, In Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects, 1993.

[AB01] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proc. Australian Software Engineering Conf*, pages 68–75, 2001.

[ABM09] Serge Abiteboul, Pierre Bourhis, and Bogdan Marinoiu. Satisfiability and relevance for queries over active documents. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS ’09, pages 87–96, New York, NY, USA, 2009. ACM.

[AE99] Reda Alhajj and Ashraf Elnagar. Incremental materialization of object-oriented views. *Data & Knowledge Engineering*, 29(2):121 – 145, 1999.

[AFP03] M. Akhtar Ali, Alvaro A.A. Fernandes, and Norman W. Paton. Movie: An incremental maintenance system for materialized object views. *Data &amp; Knowledge Engineering*, 47(2):131 – 166, 2003.

[AH87] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA ’87, pages 430–440, New York, NY, USA, 1987. ACM.

[AHM$^+$08] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. 25(5):22–29, 2008.

[AKKN12] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10):968–979, June 2012.

[AL80] Michel E. Adiba and Bruce G. Lindsay. Database snapshots. In *Proceedings of the sixth international conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 86–91. VLDB Endowment, 1980.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Aoy02] Mikio Aoyama. Metrics and analysis of software architecture evolution with discontinuity. IWPSE, 2002.

[ASSS09] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[BAT13] Bat, February 2013. URL: `https://github.com/Delors/BAT`.

[BB01] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[BCE08] H.P. Breivold, I. Crnkovic, and P.J. Eriksson. Analyzing software evolvability. COMPSAC, 2008.

[BCL89] José A. Blakeley, Neil Coburn, and Per-:1Vke Larson. Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, September 1989.

[BH95] Mikael Berndtsson and Jörgen Hansson. Issues in active real-time databases. Technical Report HS-IDA-TR-95-006, University of Skövde, School of Humanities and Informatics, 1995.

[BJ03] Frédéric Besson and Thomas Jensen. Modular class analysis with datalog. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 19–36. Springer Berlin Heidelberg, 2003.

[BK04] Christian Bauer and Gavin King. *Hibernate in Action*. Manning Publications Co., 2004.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[Blo08] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[BLS02] V. R. Basili, M. Lindvall, and F. Shull. A light-weight process for capturing and evolving defect reduction experience. In *Proc. Eighth IEEE Int Engineering of Complex Computer Systems Conf*, pages 129–132, 2002.

[BLT86] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[BM91] E. Bertino and L. Martino. Object-oriented database management systems: concepts and issues. *Computer*, 24(4):33 –47, april 1991.

[BR90] M.G. Burke and B.G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *Software Engineering, IEEE Transactions on*, 16(7):723 –728, jul 1990.

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.

[Cat00] Roderic Geoffrey Galton Cattell, editor. *The object data standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[CD92] Sophie Cluet and Claude Delobel. A general framework for the optimization of object-oriented queries. *SIGMOD Rec.*, 21(2):383–392, June 1992.

[CGL$^+$96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. *SIGMOD Rec.*, 25(2):469–480, June 1996.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989.

[Che13] Checkstyle 5.6, February 2013. URL: `http://checkstyle.sourceforge.net`.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 190 –200, mar 1995.

[CM04] B. Chess and G. McGraw. Static analysis for security. *Security Privacy, IEEE*, 2(6):76 – 79, nov.-dec. 2004.

[CMS07] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of java bytecode programs. *Software: Practice and Experience*, 37(6), 2007.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 577–589, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer Berlin Heidelberg, 1995.

[DGK82] Umeshwar Dayal, Nathan Goodman, and Randy H. Katz. An extended relational algebra with control over duplicate elimination. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, pages 117–123, New York, NY, USA, 1982. ACM.

[DLW96] Guozhu Dong, Leonid Libkin, and Limsoon Wong. On impossibility of decremental recomputation of recursive queries in relational calculus and sql. In *Proceedings of the Fifth International Workshop on Database Programming Languages*, DBLP-5, pages 7–, London, UK, UK, 1996. Springer-Verlag.

[DMP02] Giovanni Denaro, Sandro Morasca, and Mauro Pezzè. Deriving models of software fault-proneness. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 361–368, New York, NY, USA, 2002. ACM.

[dMSV$^+$08] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 78–133. Springer Berlin / Heidelberg, 2008.

[DP97] Guozhu Dong and Chaoyi Pang. Maintaining transitive closure in first order after node-set and edge-set deletions. *Information Processing Letters*, 62(4):193 – 199, 1997.

[DRS98] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). *SIGPLAN Not.*, 33(7):27–34, July 1998.

[DRW96] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems - a case study. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 117–126, New York, NY, USA, 1996. ACM.

[DS95] Guozhu Dong and Jianwen Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. Comput.*, 120(1):101–106, July 1995.

[DS00] Guozhu Dong and Jianwen Su. Incremental maintenance of recursive views using relational calculus/sql. *SIGMOD Rec.*, 29(1):44–51, March 2000.

[dSB12] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1), 2012.

[DT92] Guozhu Dong and Rodney W. Topor. Incremental evaluation of datalog queries. In *Proceedings of the 4th International Conference on Database Theory*, ICDT '92, pages 282–296, London, UK, UK, 1992. Springer-Verlag.

[EGK$^+$01] S. G Eick, T. L Graves, A. F Karr, J. S Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1), 2001.

[EKKM08] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. ICSE, 2008.

[EKS$^+$07] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. Automatic incrementalization of prolog based static analyses. PADL. 2007.

[Elk90] Charles Elkan. Independence of logic database queries and update. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '90, pages 154–160, New York, NY, USA, 1990. ACM.

[EM99] Andrew Eisenberg and Jim Melton. Sql: 1999, formerly known as sql3. *SIGMOD Rec.*, 28(1):131–138, March 1999.

[eNH07] Mario M éndez, Jorge Navas, and Manuel V. Hermenegildo. An efficient, parametric fixpoint algorithm for analysis of java bytecode. *Electronic Notes in Theoretical Computer Science*, 190(1):51 – 66, 2007.

[Eri99] Joakim Eriksson. Real-time and active databases: A survey. In StenF. Andler and Jörgen Hansson, editors, *Active, Real-Time, and Temporal Database Systems*, volume 1553 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 1999.

[Fin13] Findbugs 2.0.2, February 2013. URL: `http://findbugs.sourceforge.net`.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[FMZ01] Daniele Frigioni, Tobias Miller, and Christos Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *J. Exp. Algorithmics*, 6, December 2001.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[GBSC09] D. Garlan, J.M. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. WICSA/ECSA, 2009.

[GGMS97] Dieter Gluche, Torsten Grust, Christof Mainberger, and MarcH. Scholl. Incremental updates for materialized oql views. In François Bry, Raghu Ramakrishnan, and Kotagiri Ramamohanarao, editors, *Deductive and Object-Oriented Databases*, volume 1341 of *Lecture Notes in Computer Science*, pages 52–66. Springer Berlin Heidelberg, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GIS10] Miguel Garcia, Anastasia Izmaylova, and Sibylle Schupp. Extending scala with database query capability. *Journal of Object Technology*, 9(4):45–68, 2010.

[GIT11] ToddJ. Green, ZacharyG. Ives, and Val Tannen. Reconcilable differences. *Theory of Computing Systems*, 49:460–488, 2011.

[GJSM96] Ashish Gupta, H. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In Peter Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Advances in Database Technology - EDBT '96*, volume 1057 of *Lecture Notes in Computer Science*, pages 140–144. Springer Berlin / Heidelberg, 1996.

[GKKP10] Peter M. D. Gray, Larry Kerschberg, Peter J. H. King, and Alexandra Poulovassilis. *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[GL95] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 328–339, New York, NY, USA, 1995. ACM.

[GL00] Michael W Godfrey and Eric H. S Lee. Secrets from the monster: Extracting mozilla's software architecture. *COSET*, 2000.

[GLT97] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

[GM99a] A. Gupta and I.S. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[GM99b] Ashish Gupta and Inderpal Singh Mumick. Materialized views. chapter Maintenance of materialized views: problems, techniques, and applications, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[GMR95] Ashish Gupta, Inderpal S. Mumick, and Kenneth A. Ross. Adapting materialized views after redefinitions. *SIGMOD Rec.*, 24(2):211–222, May 1995.

[GMRS09] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 1063–1066, New York, NY, USA, 2009. ACM.

[GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509 –516, dec 1992.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM.

[GMUW08] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[GNE08] Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. Lepus3: An object-oriented design description language. Diagrams, 2008.

[GOE$^+$13] Paolo G. Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. Reify your collection queries for modularity and speed! *AOSD*, 2013.

[Gra92] Marc H. Graham. Issues in real-time data management. *Real-Time Systems*, 4:185–202, 1992.

[GYF06] Emmanuel Geay, Eran Yahav, and Stephen Fink. Continuous code-quality assurance with safe. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '06, pages 145–149, New York, NY, USA, 2006. ACM.

[Ham13] Hammurapi 5.7, February 2013. URL: `http://www.hammurapi.biz`.

[Han87] Eric N. Hanson. A performance analysis of view materialization strategies. *SIGMOD Rec.*, 16(3):440–453, December 1987.

[Hat12] Les Hatton. Defects, scientific computation and the scientific method. In AndrewM. Dienstfrey and RonaldF. Boisvert, editors, *Uncertainty Quantification in Scientific Computing*, volume 377 of *IFIP Advances in Information and Communication Technology*, pages 123–138. Springer Berlin Heidelberg, 2012.

[HBC02] E.N. Hanson, S. Bodagala, and U. Chadaga. Trigger condition testing and view maintenance using optimized discrimination networks. *Knowledge and Data Engineering, IEEE Transactions on*, 14(2):261 –280, mar/apr 2002.

[HD92] John V. Harrison and Suzanne W. Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICSLP*, pages 56–65, 1992.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[Her10] Sebastian Herold. Checking architectural compliance in component-based systems. SAC, 2010.

[HH06] Daqing Hou and H. James Hoover. Using scl to specify and check design intent in source code. *IEEE Trans. Softw. Eng.*, 32(6), 2006.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall Object-Oriented Series. Prentice Hall, 1996.

[Huy96] N. Huyn. Efficient self-maintenance of materialized views. Technical Report 1996-3, Stanford InfoLab, 1996.

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: scalable source code queries with datalog. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 2–27, Berlin, Heidelberg, 2006. Springer-Verlag.

[HYX11] Jinjin Hu, Hebiao Yang, and Qiuyan Xiong. Research of main memory database data organization. In *Multimedia Technology (ICMT), 2011 International Conference on*, pages 3187 –3191, july 2011.

[HZ90] S. Heiler and S. Zdonik. Object views: Extending the vision. In *Data Engineering, 1990. Proceedings. Sixth International Conference on*, pages 86 –93, feb 1990.

[HZ96] Richard Hull and Gang Zhou. A framework for supporting data integration using the materialized and virtual approaches. *SIGMOD Rec.*, 25(2):481–492, June 1996.

[ISO11] ISO/IEC. 9075-2: Information technology - Database languages - SQL - Part 2: Foundation, 2011.

[Jag90] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, December 1990.

[JAN87] H. V. Jagadish, Rakesh Agrawal, and Linda Ness. A study of transitive closure as a recursion mechanism. *SIGMOD Rec.*, 16(3):331–344, December 1987.

[JG12] Hemant Jain and Anjana Gosain. A comprehensive study of view maintenance approaches in data warehousing evolution. *SIGSOFT Softw. Eng. Notes*, 37(5):1–8, September 2012.

[JMS95] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 113–124, New York, NY, USA, 1995. ACM.

[JN04] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[Joh78] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, page 78–1273, 1978.

[JS82] G. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, pages 124–138, New York, NY, USA, 1982. ACM.

[KA11] George Konstantinidis and José Luis Ambite. Scalable query rewriting: a graph-based approach. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 97–108, New York, NY, USA, 2011. ACM.

[Ken83] William Kent. A simple guide to five normal forms in relational database theory. *Commun. ACM*, 26(2):120–125, February 1983.

[KLMR98] Akira Kawaguchi, Daniel Lieuwen, Inderpal Mumick, and Kenneth Ross. Implementing incremental view maintenance in nested data models. In Sophie Cluet and Rick Hull, editors, *Database Programming Languages*, volume 1369 of *Lecture Notes in Computer Science*, pages 202–221. Springer Berlin Heidelberg, 1998.

[KMB04] Cem Kaner, Senior Member, and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *In METRICS 2004. IEEE CS*. Press, 2004.

[KMHM08] J. Knodel, D. Muthig, U. Haury, and G. Meier. Architecture compliance checking - experiences from successful technology transfer to industry. CSMR, 2008.

[KMNL06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. CSMR, 2006.

[KR98] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in multiview: strategies and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):768–792, 1998.

[KS03] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. WCRE, 2003.

[Küc91] Volker Küchenhoff. On the efficient computation of the difference between consecutive database states. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 478–502. Springer Berlin Heidelberg, 1991.

[KZ08] Ioannis Krommidas and Christos Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *J. Exp. Algorithmics*, 12:16:1–16:22, June 2008.

[Lif96] Vladimir Lifschitz. Foundations of logic programming. 1996.

[Llo87] J. W. Lloyd. *Foundations of logic programming; (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM.

[LMS95] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 95–104, New York, NY, USA, 1995. ACM.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[LS93] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 171–181, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[LS03] Yanhong A. Liu and Scott D. Stoller. Dynamic programming via static incrementalization. *Higher Order Symbol. Comput.*, 16(1-2):37–62, March 2003.

[LSLR05] Yanhong A. Liu, Scott D. Stoller, Ning Li, and Tom Rothamel. Optimizing aggregate array computations in loops. *ACM Trans. Program. Lang. Syst.*, 27(1):91–125, January 2005.

[LT94] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. Technical report, Ithaca, NY, USA, 1994.

[MAB08] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4):22:1–22:50, December 2008.

[Mar03] R.C. Martin. *Agile software development: principles, patterns, and practices*. Alan Apt series. Prentice Hall, 2003.

[McC76] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308 – 320, dec. 1976.

[MD96] Mukesh Mohania and Guozhu Dong. Algorithms for adapting materialised views in data warehouses. 1996.

[MDVW07] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. AOSD, 2007.

[MKPW06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views. *Comput. Lang. Syst. Struct.*, 32(2-3), 2006.

[MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. *SIGPLAN Not.*, 24(10):397–406, September 1989.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. *SIGSOFT Softw. Eng. Notes*, 20, 1995.

[MRB06] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Manage. Sci.*, 52, 2006.

[Mye81] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 219–230, New York, NY, USA, 1981. ACM.

[Nak01] Hiroaki Nakamura. Incremental computation of complex object queries. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 156–165, New York, NY, USA, 2001. ACM.

[NB05] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 580–586, New York, NY, USA, 2005. ACM.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[OBBT89] Atsushi Ohori, Peter Buneman, and Val Breazu-Tannen. Database programming in machiavelli – a polymorphic language with static type inference. *SIGMOD Rec.*, 18(2):46–57, June 1989.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2nd edition, 2010.

[Pai82] Robert Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In *Advances in Data Base Theory*, pages 171–209, 1982.

[PD99] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, March 1999.

[PDR05] Chaoyi Pang, Guozhu Dong, and Kotagiri Ramamohanarao. Incremental maintenance of shortest distance and transitive closure in first-order logic and sql. *ACM Trans. Database Syst.*, 30(3):698–721, September 2005.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PMD13] Pmd 5.0.2, February 2013. URL: `http://pmd.sourceforge.net/`.

[PSCP02] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 802–813. VLDB Endowment, 2002.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4), October 1992.

[QGMW96] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*, pages 158 –169, dec 1996.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[RAF04] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proc. 15th Int. Symp. Software Reliability Engineering ISSRE 2004*, pages 245–256, 2004.

[RLGBAB08] Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. An industrial case study of architecture conformance. ESEM, 2008.

[Rou91] Nicholas Roussopoulos. An incremental access method for viewcache: concept, algorithms, and cost analysis. *ACM Trans. Database Syst.*, 16(3):535–563, September 1991.

[RU95] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23(2):125 – 149, 1995.

[Rus01] K. Rustan, M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 157–175. Springer Berlin Heidelberg, 2001.

[SBB$^+$02] Forrest Shull, Vic Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 249–, Washington, DC, USA, 2002. IEEE Computer Society.

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[Sem13] Semmle code, February 2013. URL: `http://semmle.com`.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[SI84] Oded Shmueli and Alon Itai. Maintenance of views. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 240–255, New York, NY, USA, 1984. ACM.

[Sim62] Herbert A. Simon. The architecture of complexity. In *Proceedings of the American Philosophical Society*, 1962.

[SJ96] Martin Staudt and Matthias Jarke. Incremental maintenance of externally materialized views. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 75–86, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. OOPSLA, 2005.

[SMA$^+$07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[Sot12] Hello2Morrow Sotograph, October 2012. URL: `http://www.hello2morrow.com/products/sotograph`.

[SR99] M. Siff and T. Reps. Identifying modules via concept analysis. 25(6):749–768, 1999.

[SR03] Diptikalyan Saha and C.R. Ramakrishnan. Incremental evaluation of tabled logic programs. In Catuscia Palamidessi, editor, *Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 392–406. Springer Berlin Heidelberg, 2003.

[SR05a] Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005. ACM.

[SR05b] Diptikalyan Saha and C.R. Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evaluation. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming*, volume 3668 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin Heidelberg, 2005.

[SR06] Diptikalyan Saha and C.R. Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs. In Pascal Hentenryck, editor, *Practical Aspects of Declarative Languages*, volume 3819 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin Heidelberg, 2006.

[SSH02] JohnA. Stankovic, SangH. Son, and Jörgen Hansson. Misconceptions about real-time databases. In Kam-Yiu Lam and Tei-Wei Kuo, editors, *Real-Time Database Systems*, volume 593 of *The International Series in Engineering and Computer Science*, pages 9–16. Springer US, 2002.

[SSW94] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. *SIGMOD Rec.*, 23(2):442–453, May 1994.

[Sto75] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1975 ACM SIGMOD international conference on Management of data*, SIGMOD '75, pages 65–78, New York, NY, USA, 1975. ACM.

[Sto10] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, April 2010.

[SY86] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.

[SZ90] G. M. Shaw and S. B. Zdonik. A query algebra for object-oriented databases. In *Proc. Sixth Int Data Engineering Conf*, pages 154–162, 1990.

[TSDNP02] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering java semantics via bytecode manipulation. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin Heidelberg, 2002.

[TV09] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12), 2009.

[UGMW01] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[UO92] Toni Urpí and Antoni Olivé. A method for change computation in deductive databases. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 225–237, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[Ves12] Vespucci, October 2012. URL: `www.opal-project.de/vespucci_project`.

[VMK97] Millist Vincent, Mukesh Mohania, and Yahiko Kambayashi. A self-maintainable view maintenance technique for data warehouses. 1997.

[VR01] Frédéric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. *SIGPLAN Not.*, 36(5):35–46, May 2001.

[VRCG$^+$10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: a java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224, Riverton, NJ, USA, 2010. IBM Corp.

[VSM05] Raphael Volz, Steffen Staab, and Boris Motik. Incrementally maintaining materializations of ontologies stored in logic databases. In Stefano Spaccapietra, Elisa Bertino, Sushil Jajodia, Roger King, Dennis McLeod, MariaE. Orlowska, and Leon Strous, editors, *Journal on Data Semantics II*, volume 3360 of *Lecture Notes in Computer Science*, pages 1–34. Springer Berlin Heidelberg, 2005.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[War92] David S. Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, March 1992.

[WCKD11] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. ICSE, 2011.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[Won00] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10:19–56, 0 2000.

[WPN06] Darren Willis, David J. Pearce, and James Noble. Efficient object querying for java. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 28–49, Berlin, Heidelberg, 2006. Springer-Verlag.

[WPN08] Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the java query language. *SIGPLAN Not.*, 43(10):1–18, October 2008.

[YRL99] Jyh-shiarn Yur, Barbara G. Ryder, and William A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 442–451, New York, NY, USA, 1999. ACM.

[YTB05] Hong Yul Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *Proc. Australian Software Engineering Conf*, pages 212–221, 2005.

[ZCC95] Mansour Zand, Val Collins, and Dale Caviness. A survey of current object-oriented databases. *SIGMIS Database*, 26(1):14–29, February 1995.

[ZWN$^+$06] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.

## Appendix

## A Original Findbugs Analyses Identifiers

The following tables list the original identifiers of the static analyses that were sampled from the Findbugs tool and presented in Chapter 6. Table A.1 lists the manually selected lightweight analyses, Table A.2 lists the randomly sampled lightweight analyses. And finally Table A.3 lists the dataflow analyses.

| Id | Findbugs Id |
|---|---|
| *CI* | CI_CONFUSED_INHERITANCE |
| $CN_1$ | CN_IDIOM |
| $CN_2$ | CN_IDIOM_NO_SUPER_CALL |
| $CN_3$ | CN_IMPLEMENTS_CLONE_BUT_NOT_CLONEABLE |
| $CO_1$ | CO_ABSTRACT_SELF |
| $CO_2$ | CO_SELF_NO_OBJECT |
| $DM_1$ | DM_GC |
| $DM_2$ | DM_RUN_FINALIZERS_ON_EXIT |
| *EQ* | EQ_ABSTRACT_SELF |
| $FI_1$ | FI_PUBLIC_SHOULD_BE_PROTECTED |
| *IMSE* | IMSE_DONT_CATCH_IMSE |
| $SE_1$ | SE_NO_SUITABLE_CONSTRUCTOR |
| *UUF* | UUF_UNUSED_FIELD |

**Table A.1:** Original identifiers of *manually* selected Findbugs lightweight analyses

| Id | Findbugs Id |
|---|---|
| *BX* | BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION |
| $DMI_1$ | DMI_LONG_BITS_TO_DOUBLE_INVOKED_ON_INT |
| *DP* | DP_DO_INSIDE_DO_PRIVILEGED |
| $FI_2$ | FI_USELESS |
| *ITA* | ITA_INEFFICIENT_TO_ARRAY |
| $MS_1$ | MS_PKGPROTECT |
| $MS_2$ | MS_SHOULD_BE_FINAL |
| $SE_2$ | SE_BAD_FIELD_INNER_CLASS |
| *SIC* | SIC_INNER_SHOULD_BE_STATIC_ANON |
| *SW* | SW_SWING_METHODS_INVOKED_IN_SWING_THREAD |
| *UG* | UG_SYNC_SET_UNSYNC_GET |
| *UR* | UR_UNINIT_READ_CALLED_FROM_SUPER_CONSTRUCTOR |

**Table A.2:** Original identifiers of *randomly* selected Findbugs lightweight analyses

| Id | Findbugs Id |
|---|---|
| *DL* | DL_SYNCHRONIZATION |
| $DMI_2$ | DMI_INVOKING_TOSTRING_ON_ARRAY |
| *RC* | RC_REF_COMPARISON |
| *RV* | RV_RETURN_VALUE_IGNORED |
| $SA_1$ | SA_FIELD_SELF_COMPARISON |
| $SA_2$ | SA_LOCAL_SELF_ASSIGNMENT |
| *SQL* | SQL_BAD_PREPARED_STATEMENT_ACCESS |

**Table A.3:** Original identifiers of selected Findbugs dataflow analyses