Scalable Data Analytics and Machine Learning on the Cloud

To fulfil the requirements for the academic degree of Doctor of Engineering (Dr.-Ing.)
Accepted Dissertation written by Abdallah Salama from Palestine
Submission Date: 2nd of November 2020, Examination Date: 15th of February 2021
Darmstadt — D 17

1. Referee: Prof. Carsten Binnig
2. Referee: Prof. Ansgar Scherp

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Computer Science
Scalable Data Management

Scalable Data Analytics and Machine Learning on the Cloud

Accepted Dissertation written by Abdallah Salama from Palestine

Darmstadt, Technische Universitaet Darmstadt

1. Referee: Prof. Carsten Binnig
2. Referee: Prof. Ansgar Scherp

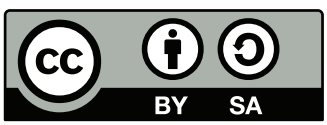
Submission Date: 2nd of November 2020
Examination Date: 15th of February 2021

Year thesis published in TUprints: 2021

Darmstadt — D 17

Please cite this document as:
URN: urn:nbn:de:tuda-tuprints-176251
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/17625

This document is being provided by TUprints,
E-Publishing-Service of TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



Thesis Statement pursuant to §22 paragraph 7 and §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Abdallah Salama, have written the submitted thesis independently pursuant to §22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are pursuant to §23 paragraph 7 of APB identical in content. For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Darmstadt, 2nd of November 2020

(Abdallah Salama)

I dedicate my thesis work to my late father, who unfortunately didn't stay in this world long enough to see his son receive a doctorate.

Acknowledgement

In the first place, I would like to express my sincere gratitude to my advisor Prof. Carsten Binnig for his invaluable and dependable support throughout my Ph.D. studies. During my Ph.D. journey, Prof. Binnig was always there with his motivation, patience, and immense knowledge. Words are not enough to thank him for all that he did for me during the six years of my Ph.D. studies.

Besides my advisor, I want to thank Prof. Ansgar Scherp. Prof. Scherp guided me in various periods throughout my studies with not only insightful comments and encouragement, but he also introduced me to a wide scope of research different from the focus of my thesis. My sincere thanks also go to my research colleagues, co-authors, and every single person who has helped me to work out each and every challenge I have met during my Ph.D. journey.

I want to thank also my research mates in the data management lab in TU Darmstadt. Muhammad Al-Hindi, Tobias Ziegler, Benjamin Hilprecht, Benjamin Haetasch, and Lasse Thostrup. Their feedback and reviews of my papers, as well as my thesis, profoundly improved this thesis work. I must also not forget Mona Hajdu; she has performed many administration tasks that benefited this work, and as she is the only one who can do them in the right way.

Last but not least, I would like to thank my family person by person: my parents, sister, and my brothers. I would not be at this position in life today without their motivation during my Ph.D. journey and life in general.

Abstract

In recent years, cloud computing has become an alternative to on-premise solutions for enterprises to host their IT-stack. The main idea behind cloud computing is to offer remote, bundled IT resources, especially data storage and computing power, without the need for active management by the user. Due to the economy of scale, cloud computing not only often comes with much lower costs than on-premise solutions but also provides users with the ability to scale-up and -down their resources based on their needs. A major building block of enterprise applications today are database management systems (DBMSs), which are used to store and query data about customers, orders, etc.

However, at the time of starting this research work in 2013, bringing the DBMSs to the cloud for online analytical processing (OLAP) and online transactional processing (OLTP) workloads was an open issue that needed to be tackled. The main reason was that the DBMS architecture, which was designed in the 1980s, was not able to optimally support the new challenges, such as elasticity and fault-tolerance that were arising when moving DBMSs to the cloud. In the first part of this thesis, we present *XDB*, which is a new parallel database architecture to support scalable data analytics in the cloud. *XDB* hence not only implements a new partitioning scheme that supports elastic scalability, but it also implements a fine-grained fault-tolerance cost model that minimizes the total runtime of a query in case of failures.

In addition to classical database workloads, deep learning workloads get more and more important in the cloud. Deep learning (DL) on deep neural networks (DNNs) has proven that it can solve complex problems such as image understanding and pattern recognition. However, training DL models can take a considerable amount of time. To speed-up the training process, several modern machine learning frameworks support distributed DL. However, users of these frameworks need manually to define the distribution strategy (i.e. number of workers and parameter servers), which is a long and complex process. This user intensive involvement makes these machine learning frameworks not ready for the cloud yet.

In the second part of this thesis, we present *XAI* as a middleware for scalable machine learning, which runs on top of existing machine learning frameworks. In *XAI*, we wanted to provide scalable support for artificial intelligence (AI) in the cloud similar to what we did in the first part for DBMS. *XAI* implements a new approach to automate the distributed deployment of a deep training job, which can optimally choose the number of parameter servers and workers to achieve scalable training in the cloud. In this case, the user of machine learning frameworks no longer spends considerable time to manually set the training and the distribution strategy of the DL job.

Contents

List of Figures

List of Tables

Part I.
Preliminaries

1 Introduction

1.1 Motivation

In recent years, cloud computing has become an alternative to on-premise solutions for enterprises to host their IT-stack. The main idea behind cloud computing is to offer remote, bundled IT resources, especially data storage and computing power, without the need for active management by the user. Due to the economy of scale, cloud computing not only comes with much lower costs than on-premise solutions, but it also provides users with the ability to scale their resources up and down their resources based on their needs. Figure 1.1 shows the results from a survey conducted by Kim Weins [44], which demonstrates that today, a high fraction of companies either already moved their applications into the cloud or are planning to do so.

A major building block of enterprise applications today are Database Management Systems (DBMSs), which are used to store and query data about customers, orders, etc. However, at the time of starting this research work in 2013, bringing the DBMSs to the cloud for Online Analytical Processing (OLAP) and Online Transactional Processing (OLTP) workloads was an open issue that needed to be tackled. The main reason was that the DBMS architecture, which was designed in the 1980s, was not able to optimally support the new challenges that were arising when moving DBMSs to the cloud.

Some of the main issues that DBMS are confronted with when deployed in a cloud setup include:

1. First, the main promise of cloud computing is that users can scale their applications on-demand, meaning that they can scale-up and -down the resources anytime, depending on the load. This feature of cloud computing is known as elasticity. However, DBMSs have been designed for static setups rather than elasticity.

2. Second, DBMSs have been designed for reliable hardware, while cloud data centers often run on commodity hardware with much higher failure rates. Classical recovery schemes, e.g., restarting a query from scratch, will increase the runtime dramatically. Thus, an efficient fault-tolerance scheme is required to tackle this issue when running a DBMS in a cloud.

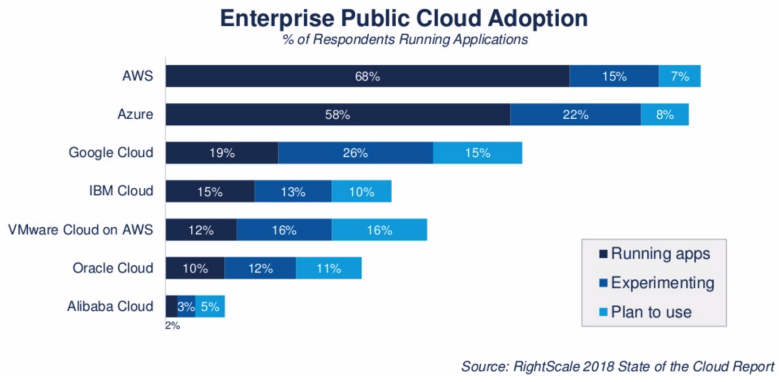


Figure 1.1.: Cloud Adoption by Enterprises in 2018

3. Finally, cloud computing enables new deployment models, such as Spot Instances, with a dynamic pricing scheme based on market demands. DBMSs or any other applications that need to make use of these new deployment models must have the capabilities to take advantage of the cost-availability trade-off that comes with these new deployment models.

The motivation for the first part of this thesis is thus to tackle these issues by implementing a new scalable and fault-tolerant database architecture that can be used in typical cloud setups. The main focus of this work was on a database architecture for analytics workloads (OLAP). In the next section, we will explain in more detail the contributions of this thesis to address the challenges mentioned above. While there exist many other issues when moving DBMSs to the cloud, such as multi-tenancy and privacy, these issues are beyond the scope of this thesis.

In addition to classic OLAP workloads, machine learning and in particular deep learning (DL) are becoming more and more important and are increasingly finding their way into enterprises in different areas. Deep learning has been shown to deliver remarkable improvements for many complex tasks from image classification to machine translation [34, 40, 68]. However, the quality of the deep learning increases with the amount of the data that need to be trained, and thus, scalable and distributed training procedures for deep neural networks (DNNs) are becoming more and more important.

The need to process large data sets makes deep learning workloads another ideal candidate to be supported by the cloud providers. Although recent machine learning frameworks support distributed deep learning, it is still a difficult task to set up a scalable, distributed learning system. The main reason is that users must decide on a multitude of low-level parameters to distribute the training, which affects the overall scalability of training DNNs. For example, they must select how many parameter servers to use. This often leads to a long and tedious trial-and-error process before the desired performance advantages of distributing the training materialize are achieved (if at all). Moreover, this trial-and-error process may need to be repeated when new GPU hardware generations, network, or even a new stack is available.

The motivation for the second part of this thesis is, thus, how to design a middleware that can leverage existing deep learning frameworks such as TensorFlow [15] or MXNet [21] and automatically scale them for a given cluster size in the cloud. In the next section, we will also explain our approach and contributions regarding this new middleware.

1.2 Contributions

This thesis makes the following contributions to the fields of scalable data analytics and machine learning in the cloud.

(1) *XDB*: We first present *XDB*, a parallel shared-nothing database system. The main goal of *XDB* is to offer SQL-like services for data analytics in the cloud. At its core, *XDB* implements a middleware approach on top of an existing single-node database system. In the middleware layer, *XDB* implements novel concepts for data partitioning and fault-tolerance that enables scalable data analytics in the cloud:

- **Locality-aware Partitioning Scheme:** modern parallel databases, as well as other data processing platforms such as Hadoop [17], horizontally partition data to provide parallel data processing capabilities for analytical workloads. The drawback of this approach is that it does not achieve data-locality. Data-locality reduces excessive data transfer when executing complex analytical

queries, which can significantly reduce the performance. In this thesis, we present a locality-aware partitioning scheme called *predicate-based reference partitioning* (PREF). PREF, which is at the opposite end of the classical reference partitioning schemes, co-partitions tables based on join predicates instead of foreign keys (which are not necessary to be join-predicates). Additionally, our partitioning scheme supports elasticity by partitioning tables into multiple partitions independent of the given resources. Thus, the redistribution of partitions becomes more flexible when the resources shrink or increase resulting in elastic scalability.

- **Cost-based Fault-tolerance Scheme:** modern parallel databases, as well as other data processing platforms such as Hadoop [17], implement different strategies for fault-tolerance. While MapReduce-like systems, such as Hadoop, materialize each intermediate result to allow recovery from mid-query failure, parallel databases restart the query execution when a mid-query failure occurs. The problem of both approaches is that they can not efficiently handle analytical workloads, which consist of queries that vary in their execution time from seconds to multiple hours. In this thesis, we present a novel, cost-based fault-tolerance scheme. Our scheme selects a subset of intermediate results to be materialized (we call it *materialization configuration*) such that the query runtime is minimized under the presence of mid-query failures. To select which intermediate results should be materialized, we present a cost model that helps to find the optimal materialization configuration. Our cost model relies on statistics about the query, such as the execution and materialization time of the operators, as well as statistics about the cluster, such as the cluster size and the mean time between failures (MTBF).

(2) *Spotgres*: As a related contribution to *XDB*, we present *Spotgres*, a variant of *XDB*. The idea of *Spotgres* is that we extend *XDB* to be deployable by using new cloud deployment models such as Spot Instances. Spot Instances apply a dynamic pricing scheme based on market demands. Thus, when deploying a parallel data processing engine on a cluster of Spot Instances, the major challenge here is to find a bidding strategy that is optimal for a given workload and satisfies user constraints, such as the maximal budget. At the time we developed *Spotgres*, there was no support from cloud vendors such as Amazon to create a set of bids for a cluster of Spot Instances. Therefore, the user had to manually find the best set of bids and continuously adapt the set of bids on Spot Instances if instances fail. In *Spotgres*, we tackled this issue by presenting a constraint-based bid advisor that finds an optimal set of bids based on a given set of user constraints, e.g., a maximal budget or minimal availability, for a user workload that needed to be satisfied.

(3) *XAI*: The last contribution of this thesis is *XAI*, which is a middleware employed on top of existing deep learning frameworks and which enables the data scientists to easily scale-out distributed training of DNNs. The aim of *XAI* is that data scientists can use a simple interface to specify the model that needs to be trained as well as the resources available (e.g., number of machines, number of GPUs per machine). Based on this input, *XAI* automatically deploys the model on the available resources optimally. To enable scalable deep learning, *XAI* implements the following novel concepts:

- **Distributed Optimizer:** A distributed optimizer, which optimizes for the distribution parameters (number of parameter servers and workers) to maximize the overall throughput of a training process (maximizes the training speed on a given cluster).

- **Automatic Deployment:** Additionally, *XAI* implements automatic deployment, which generates training scripts based on the given cluster manager and machine learning frameworks, and auto-

matically distributes the training scripts to cluster nodes based on the cluster manager used such as Kubernetes or Slurm.

1.3 Publications

The following is a list of papers that have been published as part of this thesis. These papers form the core of this thesis and include the details for all the contributions described above. In all of these publications, Abdallah Salama, the author of this thesis, was either the main author or a co-author, as mentioned below:

1. (Main author) *XDB*: A Novel Database Architecture for Data Analytics as a Service [28].
   We presented this paper in the International Congress on Big Data organized by the IEEE Computer Society. The conference has taken place in 2014 in Anchorage, USA. The paper was awarded with the **best paper award**.

2. (Co-Author) Locality-aware Partitioning in Parallel Database Systems [121].
   The main author of this paper is E. Zamanian. My contribution was to develop the concepts with the main author. Additionally. I implemented parts of the paper as well as conducted experiments. We presented this paper in ACM SIGMPD/PODS Conference. The conference has taken place in 2015 in Melbourne, Australia.

3. (Main author) Cost-based Fault-tolerance for Parallel Data Processing [97].
   We presented this paper in ACM SIGMPD/PODS Conference. The conference has taken place in 2015 in Melbourne, Australia.

4. (Co-Author) *Spotgres*: Parallel Data Analytics on Spot Instances [24].
   The main author of this paper is C. Binnig. My contribution was to extend the fault-tolerance model of *XDB* to deploy in *Spotgres*. We presented this paper at a workshop on cloud databases at the 31st IEEE International Conference on Data Engineering (ICDE) in 2015. The conference has taken place in Seoul, South Korea.

5. (Main author) *XAI*: A Middleware for Scalable AI [98].
   We presented this paper at the Data Conference in 2019. The conference has taken place in Prague, Czech Republic. The paper was awarded with the **best paper award**.

1.4 Outline

The thesis is structured as follows: Chapter 2 presents an overview of the concepts and theoretical background used in this work. Chapter 3 gives an overview of *XDB* as a new database architecture for the cloud while Chapter 4 and Chapter 5 discuss the details regarding elasticity and fault-tolerance. Chapter 6 introduces *Spotgres* as a variant of *XDB* which can be deployed on Spot Instances. Then, in Chapter 7, we present *XAI*, our middleware for scalable AI, including our distributed optimizer, and finally we conclude in Chapter 8.

2 Background and Discussion

In this chapter, we focus on the basic concepts that are required to understand this thesis. We will discuss more recent related works later in each of the main chapters, where we discuss the core contributions of the thesis. We divided the background chapter into two main sections: a section on *Distributed Database Systems*, and a section on *Distributed Deep Learning Systems*. At the end of each section, there is a short discussion about the pitfalls of the available techniques to scale-out DBMSs and DL in the cloud.

2.1 Distributed Database Systems

A distributed database is a collection of multiple, logically-interrelated databases distributed over a computer network [86]. A distributed database management system (DDBMS) is then required to manage these distributed databases. A DDBMS is a software system that manages the distributed database and provides a transparent access mechanism to the users. In this thesis, we use distributed and parallel databases terms interchangeably.

In the next subsections, we will introduce two critical topics in distributed database systems. The first topic illustrates the different architectures of parallel databases, and the second topic highlights data replication and partitioning, their importance for elasticity and the reliability of the distributed database systems. Moreover, we will explain the query processing in parallel and distributed databases.

2.1.1 Parallel Database Architectures

In this subsection, we will elucidate system architectures for implementing parallel database systems. We will also discuss the strengths and weaknesses of each architecture. The critical difference between these architectures is the design choice between the main hardware elements that build the distributed systems, i.e., processors, maim memory, and disks, which are connected through an interconnection network. The architecture depends on how the hardware resources are shared [87]. In *XDB*, we used a shared-nothing architecture to support elastic scalability. Other architectures such as shared-disk or shared-memory will be presented as well.

Shared-Memory

Any processor in a shared-memory architecture has access to a global shared memory as shown in Figure 2.1. This access is achieved normally through an interconnect, for example, a cross-bar switch. The shared-memory architecture has two main advantages: simplicity and load balancing [87]. In the shared-memory architecture, both metadata and control information are shared between all the processors,

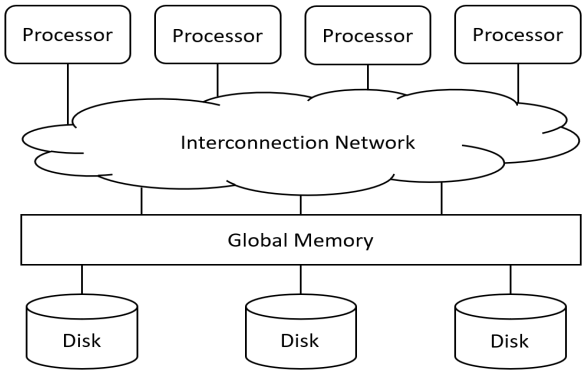


Figure 2.1.: Parallel Database Shared-memory Architecture

which makes data easily accessible to any processor. Thus, writing database software is similar to a single processor case. The load balancing can simply be achieved by allocating each new task to the least busy processor. Moreover, the processors can exchange messages efficiently.

However, according to M. Tamer Özsu et al. [87], the shared-memory architecture suffers from three problems: high cost, poor extensibility, and low availability. The cost stems from connecting each processor to each memory module via sophisticated hardware. Moreover, the waiting time of processors increases due to the conflicting access to the shared memory, which makes this approach non-extendable to add processors (increasing contention). Finally, since memory is shared among all processors, any memory fault can affect all processes. This problem can be solved by using duplex memory, which adds extra cost to the already high price of the interconnections of such systems. Thus, this architecture is not common in the implementation of database systems in cloud environments. However, Hewlett Packard Enterprise (HP) built a single-memory high-performance computer known as The Machine. The Machine converted to a "memory-driven architecture" to support big data systems [57].

Shared-Disk

Based on the problems of the shared-memory approach, which mainly result from the limited bandwidth of the shared memory, in addition to the high connection cost of the processor and the shared-memory, shared-disk [87] proposes another architecture where each processor has exclusive access to its own memory while keeping the disk access shared among all processors as shown in Figure 2.2.

By implementing this architecture, the cost introduced in the shared-memory architecture will be reduced since there is no longer a need to connect each processor to all memory modules. Moreover, the extendability of this architecture is improved since each processor has its own access to the memory. Thus, the conflicting accesses to the memory will be minimized. Isolating memory in the shared-disk architecture helps to isolate memory faults as well, and consequently, increases the availability. Although shared-disk has several advantages over shared-memory, it still has some complexity in the implementation, especially for OLTP workloads. For example, the shared-disk architecture requires a distributed database system protocols such as lock manager to achieve global cache consistency. Maintaining cache consistency among the node can introduce performance issues due to the high communication overhead between the processors. Cloudera Impala [9], Oracle Exadata [11], and many other cloud DBMSs today use shared-disk architecture.

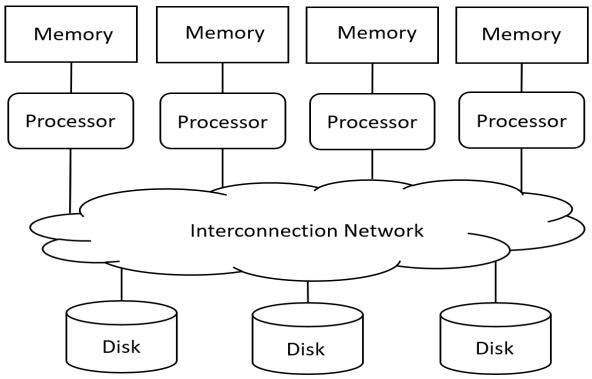


Figure 2.2.: Parallel Database Shared-disk Architecture

Shared-Nothing

In the shared-nothing architecture [54], each processor has exclusive access to its main memory and disk unit(s). Similar to shared-disk, each node is controlled by its own copy of the operating system. Thus, each node can be viewed as a local site (with its own database and software) in a distributed database system. Figure 2.3 shows an example of a shared-nothing architecture. In a shared-nothing architecture, the nodes are entirely independent and there is no way for any node to access the disk or memory of any other node.

Shared-nothing systems provide no hardware-sharing abstractions [54], and as such, the coordination between the nodes is left entirely to the DBMS. The most common technique employed in DBMSs, to support shared-nothing parallel databases, is to run the standard process model on each node. Thus, each node is capable of processing SQL queries the same way a single node database system does. However, each node stores a portion of the data such that each node executes in parallel against the data it stores. The tables in the database are partitioned and distributed over the nodes using *horizontal data partitioning*. In the next sections, we provide a deeper overview on data partitioning and parallel query execution strategies.

The contentious issue associated with shared-memory and disk architectures, namely the sharing of resources, is addressed in shared-nothing architecture, revealing a more scalable architecture. However, the shared-nothing architecture is harder to implement and manage [54]. For example, to achieve good performance, proper partitioning of the data is required to reduce the expensive transfer of data between nodes. This places a significant burden on the Database Administrator (DBA) to locate the table efficiently [54]. Moreover, the coordination between processors (nodes) is required to handle transaction completion, load balancing, and to support certain maintenance tasks. The coordination includes exchanging control messages between the processors for issues like distributed deadlock and two-phase commit [52]. This requires additional implementation that can influence the performance if it is not done carefully [54]. In this thesis, we focus on shared-nothing architecture. According to M. Hellerstein et al. [54], shared-nothing architecture has beneficial scalability and cost characteristics such as no-single-point of failure and simple dynamic cluster sizing.

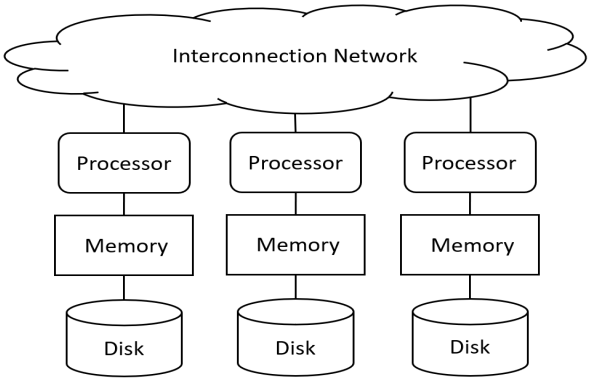


Figure 2.3.: Parallel Database Shared-nothing Architecture

2.1.2 Data Partitioning and Replication

In shared-nothing databases, a crucial question that arises is how the database should be partitioned and placed across nodes. M. Tamer Özsu et al. [87] states two primary techniques for placing data: partitioned (or non-replicated) and replicated. By partitioning, data is divided using different functions to several disjoint partitions, and then each partition resides on a different node. In the replication scheme, replication can be either full or partial. In the full replication, the entire database is fully replicated and placed on each node, while in the partial replication scheme, each partition of the database is stored on one or more than one node, but not necessarily all nodes.

Data partitioning and replications are crucial techniques to improve scalability, performance, and availability in large-scale solutions such as parallel databases [10]. Data partitioning helps the database system to scale-out by dividing data across multiple nodes, and to avoid the system reaching a physical hardware limit e.g., storage space. Accordingly, data access operations on each partition take place over a smaller volume of data, and additionally, non-conflicting operations can be performed in parallel, making the system performing efficiently. Data partitioning and replications are essential for improving the availability; by partitioning data across several nodes, a single point of failure can be avoided. If a node fails, only the data on that node is made unavailable, while operations on other nodes can continue. Using data replication, the operation on one failed node can continue on another node(s) that has the same data partition.

Partitioning Schemes

In this subsection, we discuss different types of fragmentation (partitioning) techniques in parallel databases [10].

- **Horizontal Partitioning:** with this technique, data (e.g., a table in a database) is split horizontally into partitions and distributed across nodes. Each partition is known as a shard. Each shard holds a disjoint subset of data (e.g., a subset of rows from a table), however, all partitions have the same schema.

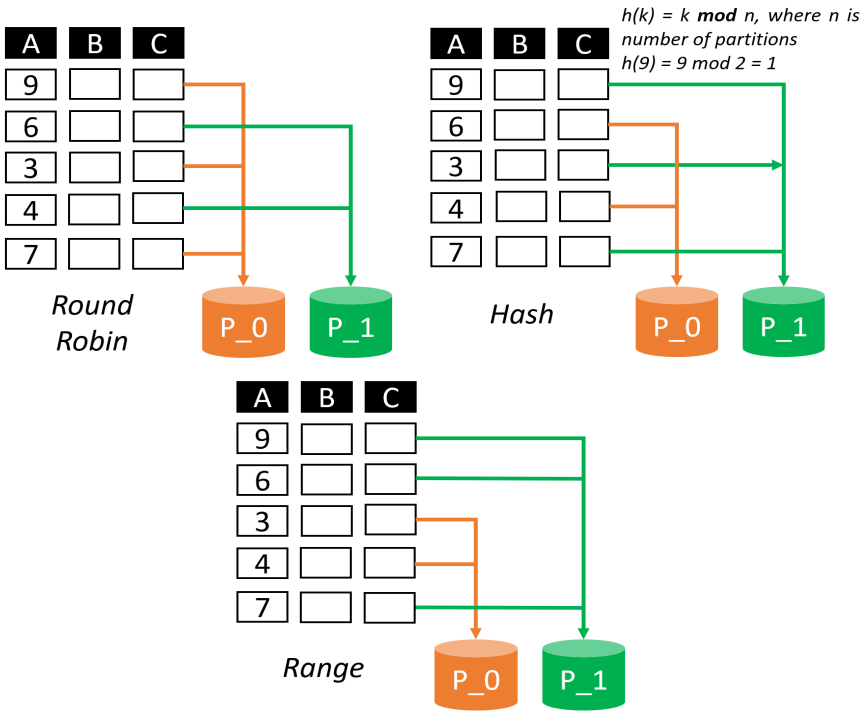


Figure 2.4.: Horizontal Partitioning Strategies

- **Vertical Partitioning:** with this technique, data is split vertically into partitions such that each partition holds data of certain fields (i.e., columns of a table). The fields are divided based on their pattern of use. For example, frequently accessed fields can be placed on the same node, while the less frequently accessed field on a different node.

In the following, we show different strategies that are used commonly in horizontal data partitioning [87]:

- **Round-robin partitioning:** with n partitions, the i^{th} tuple in insertion order is assigned to partition $(i\%n)$. This strategy enables sequential access to a relation (i.e., a table in a database) to be done in parallel.
- **Hash partitioning:** applies a hash function to some attributes, the result of the hash function is a number denotes to the node as shown in Figure 2.4. This partitioning strategy allows exact-match queries in the selected attributes to be executed on exactly one node.
- **Range partitioning:** distributes tuples based on the value intervals (ranges) of some attributes. This kind of partitioning is well-suited for range queries. Figure 2.4 shows an example of range partitioning. The values of field A which are ≤ 5 are stored in P_0 while those greater are stored in P_1.

2.1.3 Distributed Query Processing

According to K. Sattler [99], distributed query processing is the process of answering queries in a distributed environment where data is distributed and managed at different sites (nodes) connected via a

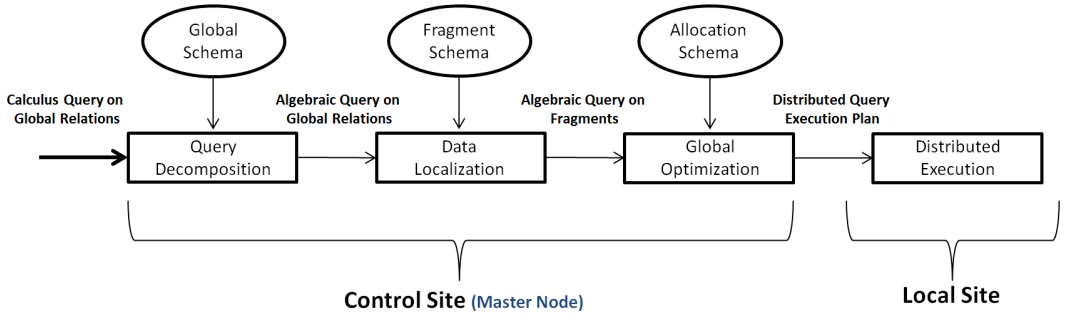


Figure 2.5.: Phases of Distributed Query Processing

computer network. Query processing, in general, consists of different phases or stages that transform the high-level query (e.g., SQL) into a query execution plan (lower-level query operators in some variation of relational algebra) as well as the execution of this physical plan. In a distributed environment, the issue arises on how to find the optimal execution strategy that guarantees a cost-effective query execution. According to M. Tamer Özsu et al. [87], several factors play a role in the execution strategy. The distribution of the data, the degree of the replication, and the communication cost.

Figure 2.5 shows the layers of distributed query processing. In the first layer, the query decomposition takes place. In this phase, the query is normalized and simplified by removing redundant predicates. The result is an enhanced algebraic query. The same step is also performed on a single DBMS. The next phase is the data localization phase, in which it is determined which fragment queries are to be involved and where the global query is transformed into fragment queries. In the global optimization phase, the optimal ordering of the fragment queries from the previous step is found in addition to specifying the communication operations. At the last stage, which is performed locally at each site, a local optimization is conducted at each site to determine access methods for the local fragment queries using the local schema.

Query Parallelism

Parallelism is used to provide speed-up and scale-up by adding more resources (processors) to handle the increasing workload without increasing the response time [77]. The main challenge to overcome is how to design an architecture that will allow the parallel execution of multiple queries, or will divide a query into parts that are executed in parallel such that the resultant overhead will not affect performance. In the previous sub-section, we have presented different parallel database architectures. In this thesis, we focus on the shared-nothing architecture. However, we will introduce different forms of query parallelism that are implemented in different architectures: inter-query parallelism and intra-query parallelism (inter-operator and intra-operator). In the following text, we introduce the two forms and show an example of a parallel execution of a query in Figure 2.6.

Inter-query Parallelism: In inter-query parallelism, several queries are executed concurrently on multiple processors or nodes. This form of parallelism is common in OLTP where queries are normally light-weight [77]. The advantage of this form is that it increases transaction throughput. However, it does not increase the speed of the of the queries themselves. Thus, this scheme is not enough to implement in parallel databases for OLAP workloads, since OLAP queries are heavy-weight, i.e., they are complex

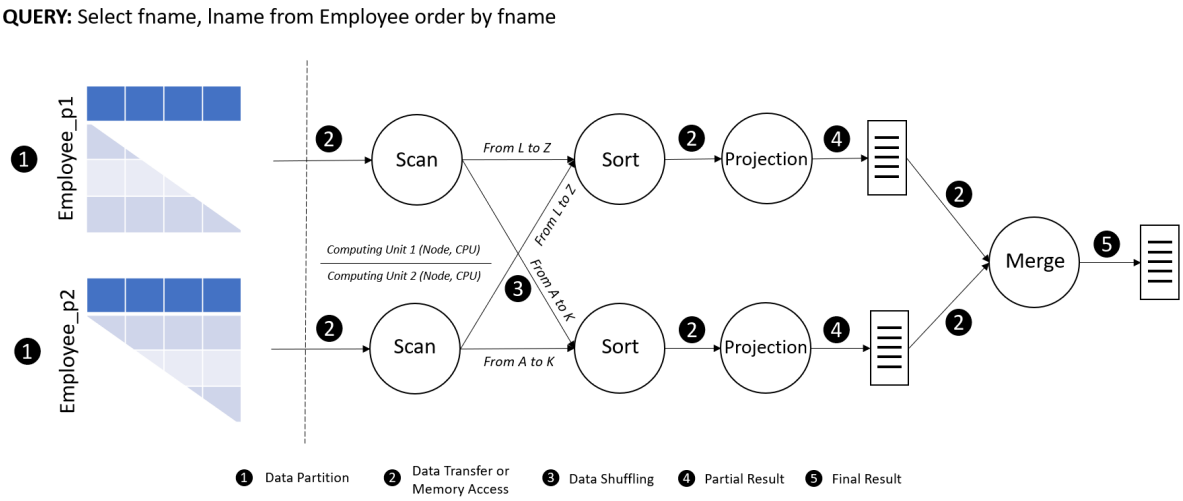


Figure 2.6.: Parallel Execution of a Query (Intra-operator Parallelism)

queries that access huge amounts of data and have expensive operations. An example database system that supports inter-query Parallelism is Oracle Rdb [84].

Intra-query Parallelism: In intra-query parallelism, a single query is decomposed into smaller tasks that execute concurrently on multiple processors, and consequently, the query's response time improves [77]. There exist two forms of intra-query parallelism: *inter-operator parallelism* and *intra-operator parallelism*. In inter-operator parallelism, operators of the same query are executed concurrently such that an operator runs on one processor. For example, a query that involves joining 4 tables, can be executed in parallel on two processors. Each processor can join two tables and the result of both joins can be joined further to produce the final result.

Intra-operator parallelism enables an operator that accesses some data to be executed by multiple nodes, each working on a different partition of the data. With intra-operator parallelism, the same operator is applied to multiple partitions, thereby dividing the response time by the number of nodes. Intra-operator parallelism exploits the various forms of data placement and dynamic partitioning by using specific algorithms for the different relational operators [77]. Figure 2.6 shows an example of executing a simple select query using intra-operator parallelism. The scan operator for example, is executed on the two partitions P_1 and P_2 of the *Employee* table. Assuming that the *Employee* table is not range-partitioned, step 3 is then required to shuffle data in order to enable the scan operator to be executed on the two partitions to support parallelism.

2.1.4 Early Parallel Database Architectures

In this subsection, we introduce two early parallel database architectures, namely Gamma [41] and Volcano [50]. We highlight these architectures since they show the classical design of parallel databases. In this section, we will discuss the pitfalls of these architectures for cloud computing.

Gamma

Gamma [41] is a parallel database system that is running on several processors as a backend to a UNIX host machine. For implementation, Gamma uses a shared-nothing architecture consisting of processors interconnected via a communication network such as a hypercube or a ring, and with a disk connected to each processor. Gamma employs three key techniques to support scalability to 100s of processors. First, it employs horizontal data partitioning e.g., hash partitioning, that enables processing of a large relation on multiple processors concurrently without introducing communication overhead. Second, Gamma uses hash-based parallel algorithms such that each operator is executed on several or all processors (*intra-operator parallelism*), and the input stream for each operator is partitioned into disjoint sets according to a hash function. Third, Gamma coordinates multi-operator queries by using a dataflow scheduling technique. The scheduler needs to send *Initiate* control message to an operator. An operator, on the other hand, sends the ID i.e., a port to communicate with, in addition to a *Done* message once the execution finishes. Thus, by using this technique, minimal coordination is required to control complex queries.

Gamma has several missing issues that are not resolved in the system, e.g., load balancing (inter-query with intra-query parallelism), query optimization (sub-queries rewriting with plan optimization), the limited data model and extensibility [50, 53]. The aforementioned issues led to the search for a more flexible and powerful query processing system. In the next subsection, we highlight Volcano [50], another parallel database architecture that extends the operator design in Gamma and comprises a more flexible and extendable query processing model.

Volcano

In Gamma, a query is constructed as an operator tree where each node is assigned one or more operator processes at each site. The operator design used in Gamma gives each operator control within its own process. The synchronization between the operators is performed in producer-consumer relationships through simple control messages as explained in the previous sub-section. According to G. Graefe [50], this design works well in Gamma. However, it can not be adapted to a single-process query evaluation due to the multiple locations of control that can not be realized in one process without multi-processing mechanisms such as threads. Thus, Volcano presents a parallel database system with an extended operator and data transfer concept that are more intended for both sequential and parallel query execution. Volcano [50] is a parallel and extensible database architecture. The main contribution of Volcano that it provides a mechanism to separate query processing from the operator and data model. To do so, Volcano provides a uniform interface between operators, e.g, scan, select, or join, makes Volcano extensible to new operators. Different from Gamma, Volcano supports different schemes of parallelism such as intra-operators parallelism, inter-operator parallelism, and pipelined parallelism (known as vertical parallelism in [50]) through an operator called the exchange operator. In *XDB*, we implemented a similar operator.

In Volcano, all operators parallelized using the exchange operator. Volcano realizes parallelism simply by inserting the exchange operator in the desired place within the query tree. For example, to support inter-parallelism in the case of a merge-join between two sorted inputs, an exchange operator can be inserted between the sort operator of one or two inputs and the merge-join. Then the parent process forks

the child process that will produce the first input sorted, and immediately after forking, the parent process turns to the second sort operation. This way, both sort operations can run in parallel.

Although Gamma and Volcano provided key solutions for parallel databases, however, they have pitfalls that prevent them to be deployed efficiently on the cloud. Gamma, for example, lacks an elastic and efficient partitioning scheme that can reduce the data shipment between the nodes. Gamma supports different horizontal partitioning strategies: *round robin*, *hash*, and *range partitioned*. These classical partition strategies do not achieve data locality. Volcano, on the other hand, is implemented based on a shared-memory architecture which is the least scalable and cloud-friendly architecture among the three parallel database architectures that we introduced in Section 2.1.1.

2.1.5 Discussion

In 2013, the year when we started our research work *XDB*, DBMS had little support for being deployed in cloud environments. DBMS has several design principles that make it unready to be delivered by a cloud platform, such as supporting static setups and adopting classical recovery schemes. Thus, in *XDB*, we implemented solutions for the aforementioned issues to bring DBMS to the cloud.

As shown in Section 2.1.1, the shared-disk and the shared-nothing are the most commonly used architectures of parallel databases for OLAP as well as OLTP. In the shared-disk architecture, all connected nodes share the same disk(s). Each processor has its own memory, but all processors access directly the same disk(s). Moreover, locking is implemented to manage global cache consistency. The contention and the overhead of locking limit the scalability of the shared-disk architecture. By contrast, the shared-nothing architecture has gained a lot of popularity over other architectures (shared-disk and shared-memory). In this architecture, each node runs a database software, and each node has its own processor, private memory, and a local disk(s). The processors communicate with each other through messaging over the network. As nothing is shared, processors do not interface. That makes the shared-nothing architecture theoretically the most scalable architecture, and thus, the ideal choice for parallel database systems that are designed for intensive analytical processing i.e., OLAP. For these considerations, we implemented *XDB* based on the shared-nothing architecture.

However, according to R. Klopp [6], there is a mismatch between cloud computing and the shared-nothing architecture. In the cloud, processors and memory scale independently of storage disk and I/O bandwidth, more compute node (processor and memory) can be dynamically added with full access to a shared storage system, results in elasticity which makes the cloud a compelling choice. However, in the shared-nothing architecture, the compute and the storage scale together. This tight connection leads to the fact that I/O bandwidth, is the key to read the performance. The scalability is more about scaling I/O than about scaling compute [6]. This conclusion comes from the imbalance Moore's Law that influences computer architecture. The compute performance outstrips I/O performance over the years resulting in that imbalance. To solve this imbalance, a smart data partitioning and distribution scheme are required to avoid that I/O is becoming a bottleneck, and to minimize data shipping between nodes at the same time.

Our target in *XDB* is to provide a shared-nothing parallel database architecture for scalable data analytics that runs on the cloud, by providing a flexible partitioning scheme that minimizes data shipments and guarantees distribution of partitions dynamically among nodes in a cluster, and a fine-grained fault-tolerance scheme that allows the system to re-execute only the failed operator on a replica of the same data partition, averting the system from restarting the query entirely in case of failure.

2.2 Distributed Deep Learning

In this section, we will give an overview of distributed deep learning. In the first subsection, we will discuss deep learning and deep neural networks. In the next subsection, we highlight the common distribution schemes which are applied when training deep learning models in parallel.

2.2.1 Deep Learning

To remove the confusion of differentiating between artificial intelligence (AI), machine learning (ML), and deep learning (DL), we first explain these three terminologies that are often used interchangeably. Artificial intelligence is a branch of computer science that aims to build and program computers to perform tasks that require human intelligence [83]. In recent times, AI has made practical achievements in several fields, such as automated reasoning [103], robotics [70], and natural language processing [119]. Machine learning is a core part of AI that includes statistical techniques and algorithms to empower machines to learn from user input data independently, and then use what they have learned to make informed decisions [18].

Going a step narrower, DL is a specific approach of ML where algorithms permit machines to learn and extract features from vast amounts of data by themselves [18]. Moreover, deep learning algorithms can adapt, through repetitive training, to discover hidden features in data. To do so, deep learning predominantly relies on artificial neural networks (ANNs). ANNs progressively learn using neurons arranged in many layers, similar to how a human brain learns. In the next subsections, we highlight the architecture of the neural network model and deep learning phases. Additionally, we will look to schemes that are followed in distributing deep learning.

Deep Neural Networks

Deep neural networks (DNNs) represent a class of machine learning models that have been rapidly evolving over the last couple of years. DNNs have proven that they can solve complex problems such as image understanding and machine translation. A DNN is an artificial neural network (ANN) with multiple layers of interconnected neurons between the input and the output layers. Figure 2.7 shows the architecture of a deep neural network. The DNN in the figure has an input layer of n neurons, and an output layer of m neurons with layers in between. The input layer represents the first layer that receives training data, e.g., pixel values of an image, while the output layer represents the results that the trained DNN is intended to give, e.g., the category of the given image.

A layer in a DNN is shown in Figure 2.8. Each neuron in this layer represents a mathematical function that transforms sets of inputs to sets of outputs. The neural network is organized such that neuron's output in each layer provides the input to the neurons in the following layer. Via this layering, the overall network represents a function $f : x \mapsto y$, that maps an input x that goes into the input layer to an output y that leaves the output layer. The purpose of f is to provide an estimate function to a target function f', e.g, a classifier that can map an image (as input) to a category (as output).

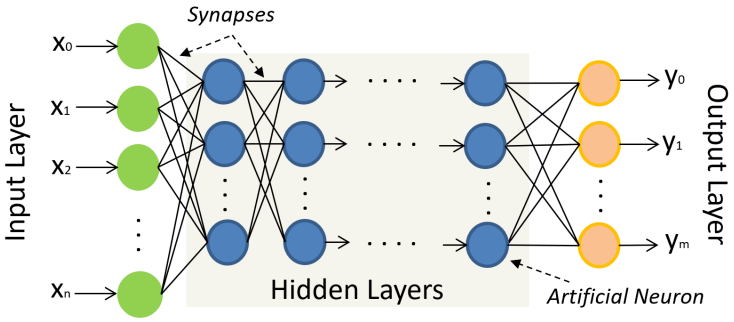


Figure 2.7.: DNN Architecture

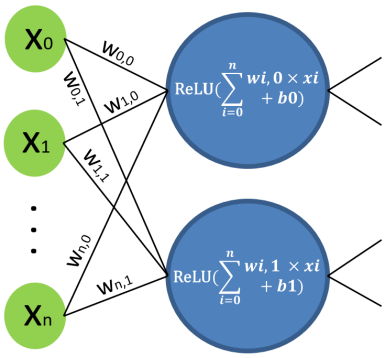


Figure 2.8.: DNN Operator

To train a DNN, the set of parameters known as model parameters, i.e., the weights, the biases, and the thresholds of every artificial neuron are adjusted such that function f approximates the target function f' with the best accuracy. The training process consists of two main steps: the first step is the ***forward propagation*** [74], where at the beginning of training process, a set of weights and biases are randomly initialized, and then used to calculate the output signal of each neuron for each example in a training batch (a batch is a subset from the whole dataset that are used for training). In each neuron, the weighted sum is calculated, and summed up with the bias as shown in Figure 2.8, before passing the summed value to an activation function e.g., Rectified Linear Units (*ReLU*). *ReLU* is used as a common activation function for deep learning model. If the summed value is negative, the activation function returns 0, otherwise it returns the value itself, serving by that two issues in the learning process: (1) interactions effect between the variables (inputs) i.e., when an input's value affects the predicted value differently based on another input's value, and (2) non-linear effects which result from the different values of the biases that are coming from the previous layer (nodes). The *forward propagation* step will produce an approximated value in the output neurons for each input example. This approximated value is then used to evaluate the performance of the network by calculating the difference between the function f' output and the predicted output of the target function f. The error function is defined as follows:

$$Error = \frac{1}{2}(f - f')^2 \tag{2.1}$$

The second step is the ***back-propagation*** [96]. The goal of the back-propagation is to reduce the error in the estimate i.e., the difference between the estimate f and the actual output f'. Since the actual output is constant, the only way to reduce the error is to change the predicted value. The predicted value depends on weights and biases, and thus, in the back-propagation, gradient descent algorithms are used to calculate the gradient of the error function 2.1 with respect to the neural network's weights and biases [95], and then, the gradients are propagated back in the network layer by layer to update weights and biases. The adjusted weights and biases are then used in the forward propagation step of the next training iteration (batch).

2.2.2 Distribution Schemes

Training DNNs with vast amounts of training data is a resource and time expensive task. Thus, it is often performed in a distributed infrastructure, which consists of several compute nodes, where each might be

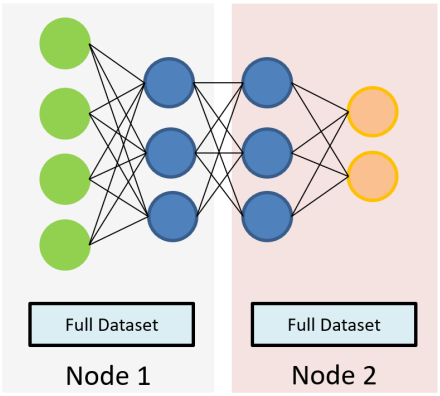


Figure 2.9.: Model Parallelism

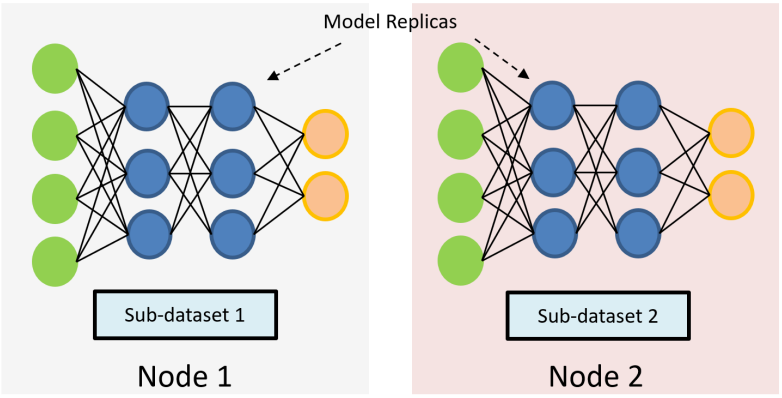


Figure 2.10.: Data Parallelism

provided with multiple GPUs to reduce the runtime of the training process [74]. DL comes with many possibilities for parallelization, and accordingly, several machine learning frameworks such as TensorFlow [15] and MXNet [21] support distributed DL by implementing several parallelization schemes. In this thesis, we mainly focus on the predominant schemes, model and data parallelism. Other parallelization schemes such as hybrid or pipeline parallelism are out of the scope of this thesis.

Model Parallelism

Model parallelism refers to a model being logically split into several parts (i.e., some layers in one part and some in another), then placing them on different nodes as shown in Figure 2.9. In model parallelism, each worker node trains a part of the DL model on the full training dataset, i.e., no shard [74]. The worker, which comprises the input layer of the DL model, is fed with the training data. As explained in 2.2.1, in the forward propagation step, the output signal in each neuron is calculated and forwarded to the next layer. In model parallelism, the output signals in each layer are forwarded to the worker that comprises the next layer of the DL model. In the back-propagation, gradients are calculated at the worker that holds the output layer, and propagated back to the workers that hold the hidden and input layers.

As mentioned above, in model parallelism, the DL model is split. Thus, less memory is needed for each worker to store the parameters. That makes the model parallelism the preferable choice when the complete DL model is too large to fit in a single node. However, model parallelism has disadvantages associated with the heavy communication between the workers. Additionally and according to A. Mirhoseini et al. [78], an ineffective split of the DL model between workers can lead to the stalling of workers due to communication overhead and synchronization delays. Consequently, increasing model parallelism might not result in training speedup.

Data Parallelism

On the contrary to model parallelism, each worker node in a data parallelism scheme has a replica of the DL model [74]. The training dataset is divided into distinct shards and fed into the model replicas of the

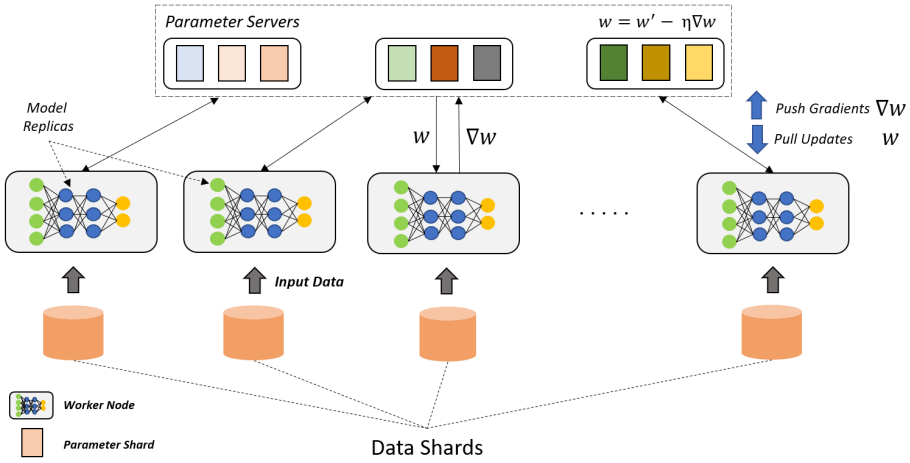


Figure 2.11.: Parameter Server Architecture

workers as shown in Figure 2.10. In data parallelism, each worker trains the model on its shard of data, and accordingly, each worker will have its own updates of the model parameters.

As in data parallelism, each worker trains the model parameters, which mandates a parameter synchronization between all workers. Parameter synchronization poses several challenges in how and when the parameters should be synchronised. According to R. Mayer et al. [74], the parameter server architecture is the most prominent architecture of parallel DL systems, that is implemented to manage the parameters update process. The system roots of the parameter server architecture date back to the blackboard architecture [102] and MapReduce [35] as reported by Alex Smola [20]. In the parameter server architecture, there are two types of entities, i.e., workers and servers [59]. As shown in Figure 2.11, the model parameters are divided into shards and distributed to the parameter server(s), which then can be updated in parallel. Among the systems that use the parameter server architecture are TensorFlow [15], Apache MXNet [21], DistBelief [37], and SparkNet [80].

The main advantage of data parallelism is that it can be applied to any DL model without further domain knowledge of the model, as required in model parallelism. Additionally, data parallelism scales well for DL models which are compute-intensive but which have fewer parameters. However, data parallelism can be limited for these DL models that have many parameters, as the parameter synchronization becomes a bottleneck [64, 67].

2.2.3 Discussion

There are several machine learning frameworks such as Google TensorFlow or Apache MXNet that support distributed deep learning. However, they are hard to use. Performing a scalable training job is hard even for a data scientist. This is different for (D)DBMSs, which implement a distributed optimizer that finds an optimal execution strategy for each query [30]. In distributed DL, all users including data scientists and inexperienced users need to manually define the distributed execution strategy. Many users can not do so since it is required that they go through a long and complex process to find the optimal servers to workers ratio. A user needs to take into consideration several aspects such as the model, the dataset size, and the used infrastructure.

This user intense involvement makes these machine learning frameworks not ready for the cloud yet. Thus, we wanted in the scope of our thesis work to provide scalable support for AI in the cloud similar to DBMS. In *XAI*, we introduced optimizers that perform similar jobs as the optimizers in DBMS do. In this case, the user of the machine learning framework no longer spends a considerable time to manually set the training and the distribution strategies of the deep learning job; *XAI* solves all these issues and delivers to the DL-framework users not only the optimal distribution strategy, but also the optimal hyperparameters required for model accuracy and have an influence on the efficiency of the training process.

Part II.

Scalable Data Analytics in the Cloud

3 XDB: An Architecture for Scalable Data Analytics in the Cloud

3.1 Introduction

Motivation

Parallel shared-nothing databases such as Greenplum [112] and Terradata [87] are major platforms for analyzing large amounts of structured data efficiently in classical data warehousing scenarios. Most existing parallel shared-nothing database systems have been designed to run on clusters with highly available hardware components. Moreover, these systems are typically deployed on clusters where the number of nodes is rarely changing in order to adjust to changes in the workload.

In order to offer SQL-like services for data analytics in the cloud, providers such as Amazon and Google do not use existing parallel shared-nothing database systems as a basis. Instead, current cloud services for data analytics are based on more recent developments such as Hadoop and Hive or other homegrown data management systems such as Google's BigQuery to offer SQL-like analytics on large data sets. A major reason for this trend is that existing parallel shared-nothing database systems are expensive and that they do not fulfill many of the requirements needed for providing a service for data analytics in the cloud: (1) Different from classical data warehousing scenarios, the data size as well as the number and complexity of queries might change quickly over time in a cloud service for data analytics. Thus, elasticity with regard to data and queries is an important aspect to cope with unpredictable workloads. (2) Second, a fine grained fault-tolerance scheme that provides the ability to recover from node failures is another essential property of a data management system. Typically, databases handle node failures by restarting the complete query on a replica of the data. This scheme is good for clusters where the probability of a node failure is low. However, when running on clusters of commodity hardware or on IaaS offerings (such as Amazon's Spot Instances as an extreme case) node failures are much more likely. In this case, a cost-based fault-tolerance scheme that supports recovery from mid-query faults is essential to save computation costs and to deliver a decent performance. (3)Third, query optimization and parallelization must be able to handle complex analytics including user-defined operators as well as iteration and recursion. Typically, databases create a static plan before execution, which requires exact cardinality estimates upfront to produce efficient plans. However, exact cardinality estimates are hard to achieve for complex analytical queries [82]. Thus, an adaptive optimization and parallelization scheme that adopts an execution plan based on the actual cardinalities is important in this case to produce efficient parallel plans.

To address the aforementioned problems, we built an open source parallel shared-nothing database system, called *XDB*[1]. *XDB* is implemented using a middleware approach on top of an existing single node database system (MySQL in our case). An *XDB* cluster consists of a Master Node, which accepts analytical queries and compiles them using its catalog. The Master Node then selects a Query Coordinator, which is responsible for executing and monitoring the query in an elastic and fault-tolerant way. Having a flexible number of Query Coordinators is important to scale with the number of queries in the system. In order to execute a query, the selected Query Coordinator splits the query into multiple query-fragments (i.e., sub-queries), which are parallelized incrementally and then send to the Compute Nodes (each of which hosts an instance of the single-node database system) for optimization and execution.

[1] https://xdb.googlecode.com

Contributions

In the middleware layer, *XDB* implements the following novel concepts (which are the contributions of this thesis work) to address the problems mentioned before: (1)*Elastic Partitioning Scheme: XDB* implements a partitioning scheme that supports elasticity. The main idea of the partitioning scheme is, that a table is partitioned into many small sized partitions. Moreover, partitions of different tables that are frequently used together (e.g., for joining) are co-located on the same Compute Node to minimize network costs of query processing. Compared to traditional partitioning schemes used in distributed databases where the number of partitions is determined as a function of the number of nodes and available disks in each node, in our novel elastic partitioning scheme the number of partitions is determined by a given threshold for the size of partitions. This efficiently helps to re-distribute partitions when Compute Nodes join or leave an *XDB* cluster. (2) *Cost-based Fault-Tolerance Scheme: XDB* offers the ability to restart queries from mid-query faults. The idea is that each query-fragment in a Compute Node materializes its intermediate result locally in its database instance such that subsequent query-fragments are able to recover from that intermediate result. Moreover, each partition of a table is also replicated by a given replication factor to other Compute Nodes such that query-fragments reading data from a table can be recovered as well. (3) *Adaptive Query Optimization and Parallelization:* Finally, *XDB* implements an adaptive query optimization and parallelization scheme to better support complex analytical queries. Therefore, an analytical query is optimized and parallelized incrementally in *XDB* to minimize negative effects of errors in cardinality estimation.

Outline

The goal of this chapter is to give an overview of the novel concepts implemented in *XDB*. The remainder of this chapter is organized as follows: Section 3.2 shows the architecture of *XDB* and presents an overview of the programming model and query execution. Section 3.3 to 3.5 then present details about each of the new concepts. Section 3.6 discusses the experimental results, which demonstrate the viability of our middleware approach including promising results for the elasticity and fault-tolerance properties of the system as well as the benefits of adaptive optimization and parallelization. Section 3.7 describes related work. Finally, we conclude and discuss possible future work in Section 3.8.

3.2 *XDB* Overview

XDB is a parallel database built using a middleware approach that leverages an existing single-node database for query processing (i.e., MySQL in the current prototype). In the remainder, we first show the architecture of *XDB* cluster. Afterward, we then present how complex analytical queries can be formulated using *XDB* and how these queries are compiled, optimized, and executed.

3.2.1 System Architecture

Figure 3.1 shows the system architecture of *XDB*. *XDB* cluster consists of one *Master Node*, which accepts analytical SQL queries or complex functions from clients, several *Query Coordinators*, which are responsible for coordinating and monitoring the query execution and finally a huge number of *Compute*

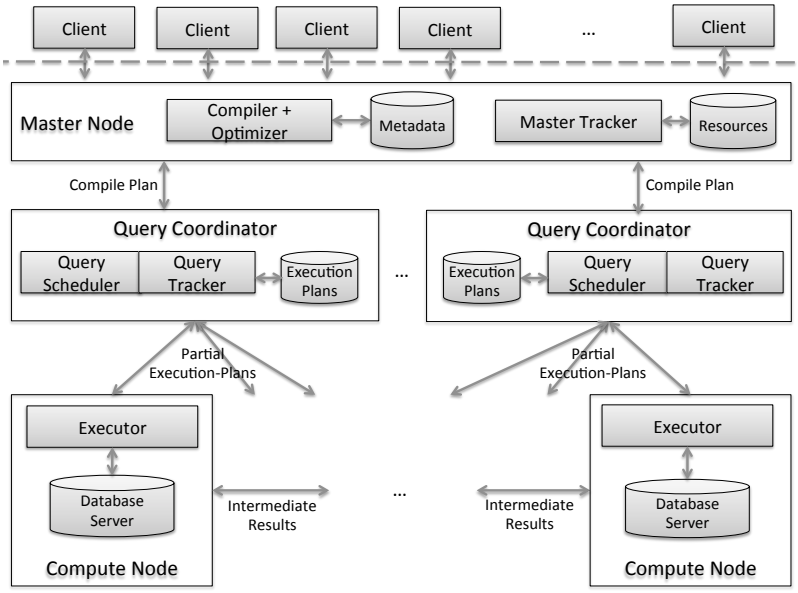


Figure 3.1.: System Architecture of XDB

Nodes, which actually execute sub-queries of the given query in parallel over the partitioned data. In the following, we briefly discuss the tasks of each node type.

Master Node: The Master Node hosts two components, a *Compiler-and-Optimizer* and a *Master Tracker*. The *Compiler-and-Optimizer* component takes a SQL query or a function as input and produces a compile plan using the catalog (which holds metadata and statistics about the tables). The compile plan is then optimized using rule-based and cost-based optimizations for join-reordering and predicate push-down. The second component, the *Master Tracker*, has different tasks: First, it monitors the cluster resources (Query Coordinators and Compute Nodes) and restarts them if these nodes fail. Second, the Master Tracker assigns a compile plan to a Query Coordinator, which is responsible for coordinating and monitoring its execution.

Query Coordinator: A Query Coordinator hosts two components: a *Query Scheduler* and a *Query Tracker*. The *Query Scheduler* takes a compile plan from the Master Node and splits it into multiple partial execution plans and sends those plans to different Compute Nodes for parallel execution. The result of each partial execution plan is materialized locally by each Compute Node. Materialization is used for different reasons: implementing fault-tolerance, adaptive query parallelization as well as distributed query processing over the individual MySQL database instances. The second component, the *Query Tracker*, is responsible for monitoring and redeploying a partial execution plan in case of a Compute Node failure. We separate the Query Coordinator from the Master Node to be able to scale the number of Query Coordinators independently from the load in the system.

Compute Node: As discussed before, a Compute Node receives a partial execution plan (represented as SQL-queries) from the Query Coordinator and the *Executor* component and then executes these plans. It then signals the Query Coordinator once a partial execution plan has finished.

```
CREATE FUNCTION basketAnalysis(OUT frequentItems TABLE)
BEGIN
  VAR t1 =
 SELECT
  l1.l_partkey, l2.l_partkey,
  p1.p_type, p2.p_type, count(*) as frequency
 FROM
  lineitem l1, lineitem l2, part p1, part p2
 WHERE l1.l_orderkey = l2.l_orderkey
  AND l1.l_partkey = p1.p_partkey
  AND l2.l_partkey = p2.p_partkey
  AND l1.l_partkey != l2.l_partkey
  AND l1.l_shipdate >= date '1998-03-01'
  AND l2.l_shipdate >= date '1998-03-01'
  AND l1.l_shipdate <= date '1998-03-31'
  AND l2.l_shipdate <= date '1998-03-31'
 GROUP BY
  l1.l_partkey, l2.l_partkey, p1.p_type, p2.p_type
 HAVING frequency>=2;

 :frequentItems =
  CALL similarity(:t1, p1.p_type, p2.p_type, 0.3);
END;
```

Figure 3.2.: Basket Analysis Function (TPC-H Schema)

3.2.2 Query Language and Execution

XDB supports analytical queries over arbitrary database schemata, which is partitioned over different Compute Nodes using our elastic partitioning scheme. An analytical query in *XDB* can either be implemented as a pure SQL statement[2], or it can be defined using a function in FunSQL [25]. Different from a pure SQL statement, a FunSQL function supports the definition of a data flow graph (such as other languages like Scope [122] or Pig [47] do). A FunSQL function in *XDB* typically consists of assignments of SQL statements to variables and calls to other functions.

An example of a FunSQL function is given in Figure 3.2, which implements a basket analysis on top of the TPC-H schema to retrieve frequent product combinations with similar product names placed together in the same order. The FunSQL function uses a SQL statement and a call to another function which is implemented using a user-defined operator (UdOp): While the first SQL statement returns products (parts) bought together in a given time frame, the user-defined code extracts those combinations, which have similar product descriptions (column `p_type`) using a string similarity metric.

Analytical SQL queries and FunSQL functions are compiled by the Master Node into data flow graphs (i.e., compile plans), which consist of relational operators as well as user-defined operators

[2] Currently, *XDB* supports only a subset of SQL'92 but *XDB* is conceptually not limited to this subset.

(UdOps) representing calls to functions implemented by user-defined code. Rule-based optimizations like selection-pushdown are then applied to the compile plan.

The compile plan is then handed over to one Query Coordinator. The Query Scheduler of the Query Coordinator first parallelizes the given compile plan using the attached information from the catalog about partitioning schemes and the locations of partitions. For parallelization, we re-use the framework presented in [122]. As a next step, the Query Scheduler decides, which intermediate results to materialize for fault-tolerance. Details about the fault-tolerance model implemented in *XDB* are discussed in Section 3.4.

Finally, before executing a compile plan, the Query Scheduler decides for each partial compile plan on which Compute Node to execute it. In order to execute the partial compile plans, the Query Scheduler generates a partial execution plan for each partial compile plan and sends it to the given location. For code generation, the relational operators in each partial compile plan are turned into query-fragments (SQL queries) over the given partitions. If a query-fragment accesses data from a remote Compute Node, *XDB* uses federated tables in MySQL to read table data from a different database instance. The complete execution plan thus forms a dependency graph of partial execution plans, which defines the order of their execution.

3.3 Elastic Partitioning Scheme

The elastic partitioning scheme presented in this section is designed to fulfill the following requirements: (1) tables should be partitioned into small partitions such that table data can easily be redistributed without re-partitioning the data, (2) tuples, which are used by joins, should be co-located on the same node, and (3) if a new bulk of data is loaded, existing data does not need to be re-partitioned.

For implementing the elastic partitioning scheme, we first define reverse reference partitioning. Reverse reference partitioning (RREF) works similar to reference partitioning (REF) [42] but in the opposite direction of a foreign-key constraint: For example, assume we have two tables `customer` and `orders`. For reference partitioning, we first partition table `customer` (e.g., by hash) and then co-partition table `orders` based on its foreign-key to table `customer`. For reverse reference partitioning, it is exactly the other way around where table `orders` is first partitioned. Reverse reference partitioning the table `customer` by the partitioned table `orders` means that a customer tuple c is added to an output partition `customer(p)` (where p is the number of the partition), if the partition `orders(p)` contains a tuple which references c.

Next, we describe a heuristically-based design algorithm that defines a partitioning scheme for each given table in a given schema. Small tables that are below a user-given threshold (t) are replicated to all nodes in the cluster and removed from the schema before applying the algorithm: (1) We select the biggest table in a given schema as a seed table that is hash-partitioned on its primary key into n partitions such that each partition is smaller than a user-given threshold (e.g., $t = 1m$ tuples). (2) We then reference partition or reverse reference partition all tables connected by a foreign key to the seed table. Reverse reference partition might introduce some redundancy. Therefore, we further split partitions if they exceed the user-given threshold (t). (3) We recursively apply the second step to all tables partitioned in the last step (ordered ascending by their size). That way, we can co-partition tables in a given schema by foreign keys, which represent potential join predicates. If a cycle exists in the schema, the algorithm breaks it by ignoring tables for which we already have assigned a partitioning scheme. Figure 3.3 shows the resulting partitioning scheme if we apply the procedure above using the threshold of $t = 1m$.

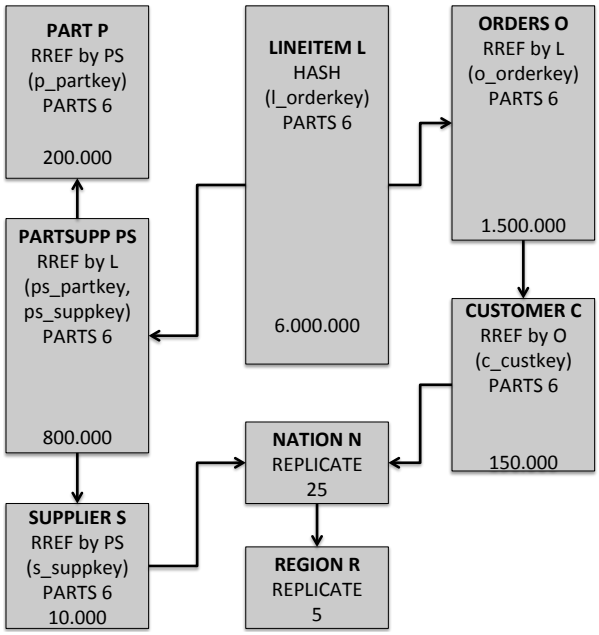


Figure 3.3.: Partitioned TPC-H schema

The elastic partitioning scheme enables efficient query processing since data items, which are joined using a foreign-key relationship can be co-located on the same nodes. However, as discussed before, reverse reference partitioning might produce duplicates in the different output partitions of a table. Therefore, *XDB* must rewrite queries using a distinct union over different partitions in certain cases, which can be efficiently implemented by adding a flag to eliminate duplicates locally. However, for most queries, no rewrite for duplicate elimination is necessary. More precisely, if at least one hash-partitioned table is in the join path of a query and the query uses only join paths that are used for partitioning, then no rewrite is necessary. In our experiments with the TPC-H benchmark, we see that duplicates have no adverse effect on the query performance of *XDB*.

If subsequent bulks are loaded into *XDB*, no tuples are added to existing partitions. Instead, *XDB* produces new partitions using the same partitioning scheme that we determined for the first bulk. Before partitioning a subsequent bulk, *XDB* must make sure that the bulk contains all referenced tuples. For example, if we load new `lineitem` tuples, we need to make sure that all referenced `orders` and `customer` tuples are contained in the bulk. This can be achieved by ETL-processes. Again, this might lead to duplicates, which *XDB* needs to remove by rewriting the analytical queries accordingly.

3.4 Cost-based Fault-Tolerance

In this section, we first present the strategy of how *XDB* enumerates different materialization configurations for a given analytical SQL query. *XDB* uses a two-step approach for enumerating different materialization configurations. It first compiles an optimized parallel compile plan for a given analytical SQL query and then enumerates all possible materialization configurations for the compile plan.

Figure 3.4 represents an overview of the compilation process: (1) A given analytical SQL-query is first compiled and optimized. The result is a *Compile Plan*, which consists of relational operators. (2) For

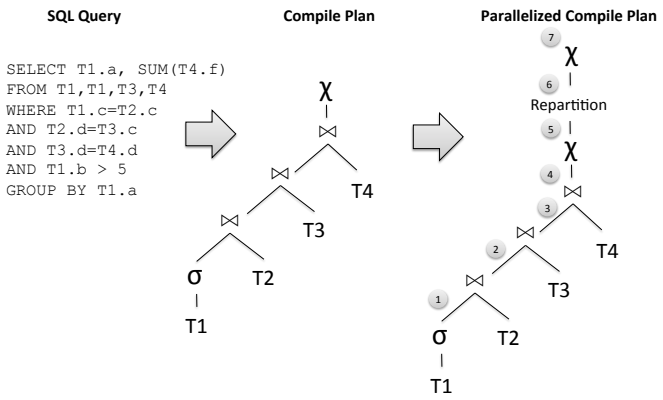


Figure 3.4.: Compilation Process in XDB

parallelization, the compile plan is annotated with additional repartition operations that are required when executing the query over a horizontally partitioned database. The result is called a *Parallelized Compile Plan*.

For the resulting *Parallelized Compile Plan*, all the different materialization configurations are enumerated. A materialization configuration defines for each intermediate result whether it should be materialized or not. For example, in Figure 3.4, it defines for each of the intermediate results $1-7$, whether it should be materialized. *XDB* currently uses an exhaustive search to enumerate all possible materialization combinations. In general, this results in 2^n different combinations for n intermediate results.

Moreover, some configurations can be pruned. For example, the repartitioning operator of *XDB* always materializes its result before the data is shipped over the network to the consuming operators. Thus, in our example, we only need to enumerate 2^6 different materialization configurations since the intermediate result 6 is materialized in all cases. For each materialization configuration, *XDB* invokes its cost function to estimate the runtime for the given materialization configuration and selects the configuration with the minimal estimated runtime. Details about the cost function can be found in [29].

3.5 Adaptive Parallelization

In *XDB* an analytical query is optimized and parallelized incrementally . Incremental optimization has already shown to be beneficial for complex analytics in [82], since cardinality estimates for complicated compile plans are typically error-prone.

XDB piggybacks the incremental execution on top of the materialized intermediate results of the fault-tolerant query execution strategy of *XDB* as described before. As an avenue for future work, we also plan to integrate a cost model as in [82] to select which intermediate results to materialize for adaptive parallelization, in order to minimize adverse effects of errors in cardinality estimation.

The main ideas are that (1) *XDB* executes a given compile plan incrementally level by level (i.e., levels are determined by operators which materialize their output) and (2) materialized intermediate results are stored using partitioned in-memory tables in order to adaptively adjust the degree of parallelism for the subsequent level. Our experiments show that the additional costs for using partitioned in-memory tables are negligible. The number of partitions used to store the intermediate results is determined based on the cardinality estimate of an intermediate result and a given threshold for the size of each partition.

3.6 Experimental Evaluation

The main goal of the experimental evaluation is to show that the architecture of *XDB* is viable (i.e., how its performance compares to other middleware approaches). Moreover, for each novel concept (elastic partitioning, cost-based fault-tolerance, and adaptive parallelization), we compare *XDB* to the standard behavior in databases to see the benefits of our novel concepts. Showing the detailed trade-offs for each concept is an essential avenue of future work.

3.6.1 Experimental Setup

System Description: In all experiments, we used MySQL 5.6.11 as the underlying database for *XDB*. For MySQL, we used the following configuration: InnoDB as the default storage engine and read committed as the default transaction isolation level. Our initial prototype of *XDB* was built using Java 1.7 and was executed on different cluster setups as described below (i.e., one private cluster and one public cluster running on Amazon EC2).

Data and Queries: For all our experiments, we used the TPC-H benchmark schema and the data generator to generate a data set with $SF = 64$, and we created an index on each selection attribute used in any of the TPC-H queries. We partitioned the data as shown in Figure 3.3 using different thresholds to result in 8, 16, 32 or 64 partitions. For each experiment, we distributed the set of partitions equally to all Compute Nodes in the cluster. Moreover, for each experiment, we also used different workloads (e.g., for experiment 1, we used all 22 TPC-H queries). Details about the data distribution and the workloads are discussed for each experiment separately.

Cluster Setup: For our experiments, we deployed *XDB* on two different cluster setups: (A) an internal cluster and (B) a cluster runing on Amazon EC2:

- *Cluster A (internal)*: This cluster consists of 8 virtualized cluster nodes. We configured *XDB* to run one Master Tracker and one Query coordinator. Moreover, *XDB* was using 8 Compute Nodes, each running on a different virtualized cluster node. Each of the virtualized cluster nodes used the following configuration: Intel Xeon X5650 with 2.67 GHz (4 virtual CPU Cores), 16 GB RAM, and a virtualized 160 GB HDD. Each virtualized cluster node was running the following software stack: Xen as a VM monitor, CentOS 6, Cloudera CDH 4.6 Free Edition including Hive, MySQL 5.6.11 as well as Java 1.7.
- *Cluster B (Amazon EC2)*: This cluster consists of $8-64$ Amazon EC2 machines (m1.large) using local instance storage or EBS for the data directory of MySQL. We configured *XDB* to run one Master Tracker and one Query coordinator and one Compute Node per virtualized cluster node. Each m1.large EC2 node has two virtual CPUs, 7.5 GB of RAM, and 2 times $420GB$ as local instance storage. Each virtualized cluster node was running the following software stack: Amazon Linux, MySQL 5.6.11 as well as Java 1.7.

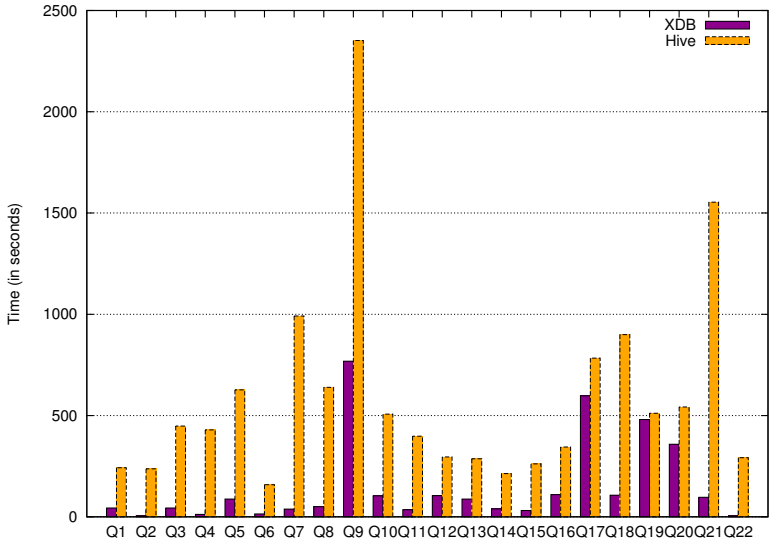


Figure 3.5.: XDB vs. Hive (TPC-H SF = 64)

3.6.2 Experiment 1: XDB Performance

The first experiment evaluates the performance of *XDB* in a static cluster setup (on Cluster A) without any node failures. We compare the runtime of *XDB* using all 22 queries of the TPC-H benchmark and compare the runtime to Hive installed on the same cluster. The data was partitioned using the 32 partitions variant of our elastic partitioning scheme, as described before. Thus, each Compute Node in the cluster was holding 4 partitions. We executed each query ten times on Hive and *XDB* and reported the average runtime in this experiment.

The results of this experiment show that the speedup factor of *XDB* is comparable to other middleware approaches, which use a database for processing large volumes of structured data. Examples for this include HadoopDB (Hadapt), for which an average speedup factor ranging from $6-42$ for TPC-H is reported in [22] depending on the underlying database.

Figure 3.5 shows the results of this experiment. We can see that *XDB* shows speedup factors from 1.5 to 200 for the individual queries when compared to Hive. On average, *XDB* is approximately 22 times faster than Hive, whereas almost 100% of the runtime is spent in the underlying MySQL instance. For all queries, the *XDB* cluster was I/O bound since the cluster nodes used virtualized disks. Thus, using cluster nodes that are optimized for I/O heavy loads and a more advanced analytical database will result in even better runtime for *XDB*.

3.6.3 Experiment 2: Elastic Partitioning Scheme

Compared to traditional partitioning schemes, which partition a database based on the number of available I/O devices, the elastic partitioning scheme of *XDB* tends to create a high number of small partitions since it uses a fixed threshold for the size of each partition. Having a large number of small partitions enables *XDB* to easily re-distribute partitions (if a new node joins the cluster or if a node leaves the cluster).

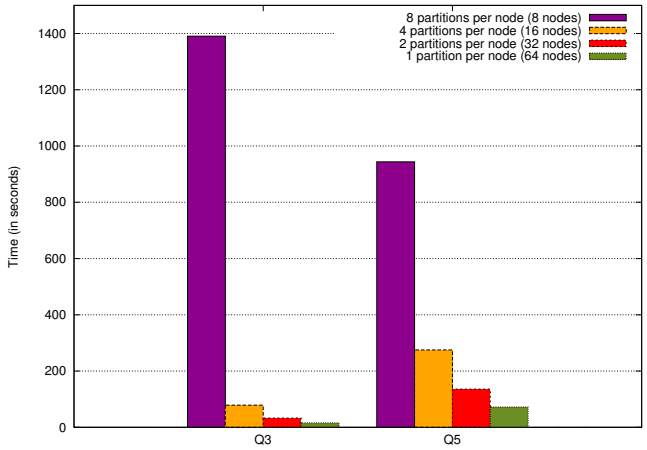


Figure 3.6.: Benefits of Elastic Partitioning

However, storing multiple partitions on the same node might also result in higher random I/O per node, which could harm the overall query performance.

The goal of this experiment is to show the effects when re-distributing the small partitions to a cluster with a different number of nodes in order to implement elasticity. To this end, we analyze the query performance for different distribution scenarios: We used the TPC-H $SF = 64$ database, which was split into the 64 partitions using our elastic partitioning scheme as described before. Moreover, we distributed the 64 partitions equally to 4, 8, 16, 32, and 64 EC2 machines and executed two TPC-H queries (queries 3 and 5). We report the average runtime over ten runs.

For this experiment, we used cluster B, whereas each cluster node was configured to use one dedicated EBS-storage volume (with 600 IOPS). The result of this experiment can be seen in Figure 3.6. Being able to re-distribute the data to a higher number of nodes results in a superlinear speedup of queries in *XDB* due to less random I/O per node. This effectively helps to increase the throughput of queries in *XDB*. Consequently, if the load is increasing, the elastic partitioning scheme efficiently helps to re-distribute the same data to more nodes. Moreover, another consequence which results from this experiment is: if the total data size increases (i.e., the number of partitions grows), *XDB* can achieve constant time query processing by adding new nodes and re-distributing the data.

3.6.4 Experiment 3: Cost-based Fault-tolerance

The goal of this experiment is to show the behavior of *XDB* under different stress levels s (i.e., a higher level s represents an increasing number of Compute Nodes failing during query execution).

Therefore, we execute a query which results in an execution plan with four levels (i.e., a level hosts multiple operators running in parallel whereas each operator checkpoints its output to enable mid-query recovery): in this experiment, levels 2 and 4 execute only one operator (i.e., a union over the partial execution plans of levels 1 and 3). Levels 1 and 3 execute 32 operators, which read data from the underlying database (in parallel on different partitions).

In this experiment, we used the TPC-H database $SF = 64$ partitioned into 32 partitions and run the execution plan above on Cluster A (i.e., we distributed four partitions to each Compute Node).

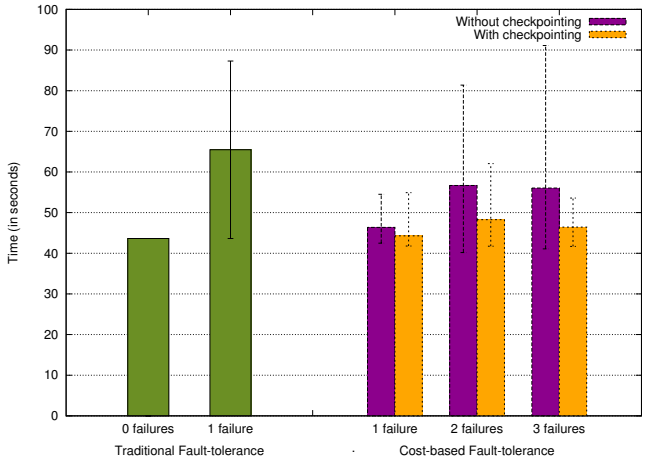


Figure 3.7.: Benefits of Cost-based Fault-tolerance

Consequently, levels 1 and 3 have each $8 \cdot 4 = 32$ partial execution plans, whereas four partial execution plans are executed on the same Compute Node.

In order to simulate different stress levels, we inject a failure at different points in time randomly distributed over the complete runtime of the query and thus force the query to recover (from stored intermediate results). For the injection of a failure, we vary the number of nodes that fail from $s = 1$ to $s = 3$, whereas each node has the same probability of failing.

In a second variant, we introduce safe checkpoints (which are not lost if a node fails) for the union operations in levels 2 and 4, such that the query can always restart from this intermediate results. We compare the results of *XDB* to the traditional fault-tolerance scheme of databases, which complete restart a query from its base tables once an error occurs (called naive restart).

Figure 3.7 shows the results of this experiment reporting the average runtime over 10 runs with a given stress level for traditional and cost-based fault-tolerance. On the left-hand side, we see the runtime without any node failure ($44s$), whereas with a traditional naive restart scheme, the query takes $66s$ on average and $88s$ maximally. On the right-hand side, we show the runtime of the cost-based fault-tolerance scheme for the stress levels 1 to 3 (i.e., 1 to 3 node failures are injected) : the average runtime for each stress level s is better than the average runtime of the naive restart with one failure.

Moreover, for the cost-based fault-tolerance scheme, we analyzed two variants: with and without safe-checkpoints. With safe-checkpoints enabled, *XDB* guarantees that once an intermediate result is materialized, it can always be used to recover (even if a node fails that produced the result). This can be achieved by storing the intermediate result to a fault-tolerant storage medium (e.g., a fault-tolerant disk such as Amazon's EBS or systems such as RAMCloud [85]). The experiment shows that with safe-checkpoints enabled, the average runtime almost remains stable over the different stress levels and has a much smaller variance as without enabling safe-checkpoints.

3.6.5 Experiment 4: Adaptive Parallelization

The final experiments show the benefits of adaptive parallelization for complex analytics. In this experiment, we are running a variant of the analytical function shown in Figure 3.2, which executes a

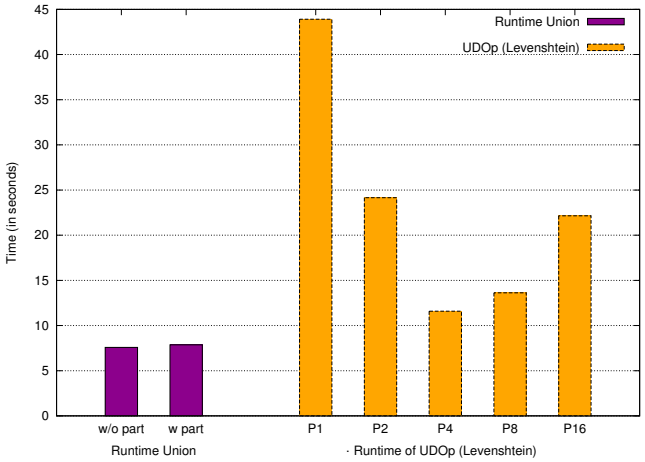


Figure 3.8.: Benefits and Costs of Adaptive Parallelization

basket analysis. For this experiment, we vary the degree parallelism of the last function call (i.e., the `similarity` function). This function is implemented by a user-defined operation in *XDB* (UdOp), which implements a Levenshtein similarity function and a filter based on the resulting similarity in order to find related products by name.

While the cardinality estimation of MySQL returns $2,867,264$ tuples for the input of the UdOp, the actual cardinality is only $514,672$. Thus, this overestimation of the cardinality might result in a too large number of parallel partial execution plans, which might not be executed efficiently.

The right bar group of Figure 3.8 shows the runtime of the UdOp under different degrees of parallelism (P1-P16). The runtime decreases up to a degree P4 (i.e., four parallel partial execution plans) and then increases again. On the left-hand side of Figure 3.8, we show the additional costs of preparing adaptive parallelization by storing an intermediate result (i.e., the input of UdOp) in partitioned in-memory tables (w parts). This overhead is negligible when compared to writing the intermediate result to a non-partitioned in-memory table (w/o parts).

3.7 Related Work

Middleware approaches to build parallel databases

There are many existing approaches for large scale data analytics (AsterData [45], Greenplum [113], HadoopdDB/ Hadapt [22] to name a few), which use a middleware approach to build a parallel database system on top of a single node database. While all of these approaches support SQL-like workloads on large data sets, their architecture does not focus on providing a SQL-like data analytics service in the cloud. Compared to *XDB*, none of these systems provides features like an elastic partitioning scheme which co-locates data, a cost-based fault-tolerance mechanism to re-start from mid-query faults and adaptive parallelization combined in one system.

Elasticity in databases

Currently, many novel database systems that address elasticity for transactional (OLTP) workloads. One important line of work to support elasticity OLTP workloads are NoSQL databases (e.g., Amazon

Dynamo [38]). These systems typically build on the ideas of distributed hash tables (DHTs) to assign tuples to cluster nodes. For analytical workloads, existing parallel database systems typically implement approaches that need to re-partition data before re-distributing it [87]. Different from these approaches, *XDB* uses a novel elastic partitioning scheme, which allows re-distribution without re-partitioning. To the best of our knowledge, such a partitioning scheme has not been implemented in a parallel database before.

Fault-tolerance in databases

Typically, fault-tolerance in databases is handled by re-starting a query completely since a node failure should be a rare case. However, when running on commodity hardware or IaaS offerings such as Amazon's Spot Instances, a node failure is the normal case. This requires a more fine-granular fault-tolerance scheme. Two novel approaches, which tackle this problem are FTOps [111] and Osprey [116]. FTOps presents approaches for intra-query fault-tolerance without blocking and presents an optimizer to find the best physical implementation for an operator to recover from mid-query faults. Osprey splits analytical queries over a star schema into sub-queries on partitions and executes a final merge of all sub-queries. If a sub-query fails, it is re-started on a different replica. *XDB* extends the ideas presented in Osprey to arbitrary schemata and queries. Moreover, *XDB* also presents a cost model to find the best deployment under different failure rates.

Adaptive Parallelization

Finally, there is a huge number of research publications about adaptive query processing for single-node database systems, as shown in [63]. However, not many approaches have been discussed to achieve adaptively parallelize for complex analytical queries in parallel databases. One approach is presented in [48] for adaptively optimizing recursive queries in Teradata. [115] implements another approach for dynamic mid-query re-planning to effectively execute complex SQL queries and machine learning tasks. Compared to *XDB*, none of these approaches uses partitioned intermediate results, which almost add no additional overhead if intermediate results are already materialized (e.g., to implement a cost-based fault-tolerance scheme).

3.8 Conclusions and Outlook

In this chapter, we presented an elastic and fault-tolerant database system called *XDB*, which supports complex analytics. *XDB* integrates different novel concepts: elastic partitioning, cost-based fault-tolerance, and adaptive parallelization. *XDB* is implemented using a middleware approach over an existing single-node database system using its efficient query processing techniques. The elastic partitioning scheme splits tables into small partitions using a threshold to determine the number of partitions to create. Compared to traditional partitioning schemes, elastic partitioning allows to re-distribute partitions without re-partitioning. For execution, *XDB* splits a compile plan into multiple partial compile plans, whereas intermediate results are materialized. On top of materialization, *XDB* piggy packs its concepts for cost-based fault-tolerance and adaptive parallelization.

As the main avenue for future work, we plan to evaluate each of the concepts of *XDB* in more detail. Moreover, we plan to extend *XDB* to support more complex workloads (e.g., machine learning tasks) as well as various data formats and compare it with the various existing systems for big data analytics.

4 Locality-aware Partitioning in Parallel Database Systems

4.1 Introduction

Motivation

Modern parallel database systems (such as SAP HANA [26], Greenplum [112] or Terradata [87]) and other parallel data processing platforms (such as Hadoop [114], Impala [9] or Shark [115]) horizontally partition large amounts of data in order to provide parallel data processing capabilities for analytical queries (e.g., OLAP workloads). One major challenge when horizontally partitioning data is to achieve a high data-locality when executing analytical queries since excessive data transfer can significantly slow down the query execution on commodity hardware [93].

A common technique to reduce the network costs in analytical workloads, which was already introduced in the 1990s by the first parallel database systems, is to co-partition tables on their join keys in order to avoid expensive remote join operations [41, 46]. However, in complex schemata with many tables, this technique is limited to only subsets of tables, which share the same join key. Moreover, fully replicating tables is only desirable for small tables. Consequently, with existing partitioning schemes, remote joins are typically unavoidable for complex analytical queries with join paths over multiple tables using different join keys.

Reference partitioning [42] (or REF partitioning for short) is a more recent partitioning scheme that co-partitions a table by another table that is referenced by an outgoing foreign key (i.e., referential constraint). For example, as shown in Figure 4.1, if table CUSTOMER is hash partitioned on its primary key `custkey`, then table ORDERS can be co-partitioned using the outgoing foreign key (fk) to the table CUSTOMER table. Thus, using REF partitioning, chains of tables linked via foreign keys can be co-partitioned. For example, table LINEITEM can also be REF partitioned by table ORDERS. However, other join predicates different from the foreign key, or even incoming foreign keys are not supported by REF partitioning. For example, table SUPPLIER in Figure 4.1 can not be REF partitioned by the table LINEITEM.

Contributions

In this chapter, we present a novel partitioning scheme called *predicate-based reference partitioning* (or PREF for short). PREF is designed for analytical workloads where data is loaded in bulks. The PREF partitioning scheme generalizes the REF partitioning scheme such that a table can be co-partitioned by a given join predicate that refers to another table (called partitioning predicate). In Figure 4.1,

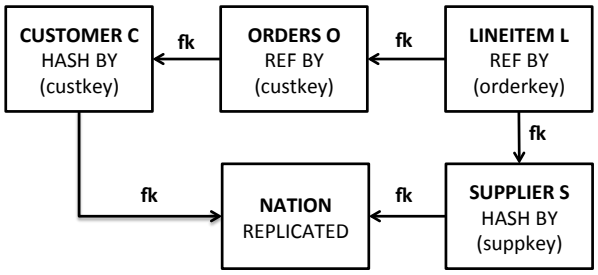


Figure 4.1.: Partitioned TPC-H Schema (simplified)

table SUPPLIER can thus be PREF partitioned by table LINEITEM using an equi-join predicate on the attribute suppkey as partitioning predicate. In order to achieve full data-locality with regard to the partitioning predicate, PREF might introduce duplicate tuples in different partitions. For example, when PREF partitioning the table SUPPLIER as described before and the same value for the suppkey attribute appears in multiple partitions of the table LINEITEM, then the referencing tuple in table SUPPLIER will be duplicated to all corresponding partitions of SUPPLIER. That way, joins which use the partitioning predicate as join predicate can be executed locally per node. However, in the worst case, the PREF partitioning scheme might lead to full replication of a table. Our experiments show that this is only a rare case for complex schemata with a massive number of tables and can be avoided by our automatic partitioning design algorithms.

Furthermore, it is a hard problem to manually find the best partitioning scheme for a given database schema that maximizes data-locality using our PREF partitioning scheme. Existing automated design algorithms [81, 91, 105] are not aware of our PREF partitioning scheme. Thus, as a second contribution, we present two *partitioning design algorithms* that are aware of PREF. Our first algorithm is *schema-driven* and assumes that foreign keys in the schema represent potential join paths of a workload. Our second algorithm is *workload-driven* and additionally takes a set of queries into account. The main idea is to first find an optimal partitioning configuration separately for subsets of queries that share similar sets of tables and then incrementally merge that partitioning configuration. In our experiments, we show that by using the PREF partitioning scheme, our partitioning design algorithms outperform existing automated design algorithms, which rely on tight integration with the database optimizer (i.e., to get the estimated costs for a given workload).

Outline

In Section 4.2, we present the details of the PREF partitioning scheme and discuss details about query processing and bulk loading. Afterward, in Section 4.3, our schema-driven automatic partitioning design algorithm is presented. Section 4.4 then describes the workload-driven algorithm and discusses potential optimizations to reduce the search space. Our comprehensive experimental evaluation with the TPC-H [2] and the TPC-DS benchmark [1] is discussed in Section 4.5. We have chosen these two benchmarks as we wanted to show how our algorithms work for a simple schema with uniformly distributed data (TPC-H) and a complex schema with skewed data (TPC-DS). Finally, we conclude with related work in Section 4.6 and a summary in Section 4.7.

4.2 Predicate-based Reference Partitioning

In the following, we first present the details of our predicate-based reference partitioning scheme (or PREF for short) and then discuss important details of executing queries over PREF partitioned tables as well as bulk-loading those tables. In terms of notation, we use capital letters for tables (e.g., table T) and small letters for individual tuples (e.g., tuple $t \in T$). Moreover, if a table T is partitioned into n partitions, the individual partitions are identified by $P_i(T)$ (with $1 \leq i \leq n$).

4.2.1 Definition and Terminology

The PREF partitioning scheme is defined as follows:

Definition 1 (PREF Partitioning Scheme)**.** *If a table S is partitioned into n partitions using an arbitrary horizontal partitioning-scheme, then table R is* PREF *partitioned by that table S and a given partitioning predicate p, iff (1) for all $1 \leq i \leq n$, $P_i(R) = \{r | r \in R \wedge (\exists s \in P_i(S) | p(r,s))\}$ holds and (2) $\forall r \in R | (\exists r \in P_i(R), 1 \leq i \leq n)$. In* PREF*, we call S the referenced table and R the referencing table. The referenced table could be again* PREF *partitioned. The seed table of a* PREF *partitioned table R is the first table T in the path of the partitioning predicates that is not* PREF *partitioned.*

Condition (1) in the definition above means that a tuple $r \in R$ is in a partition $P_i(R)$ if there exists at least one tuple $s \in P_i(S)$ that satisfies the given partition predicate p (i.e., $p(r,s)$ evaluates to true) for the given i. A tuple s that satisfies $p(r,s)$ is called partitioning partner. Therefore, if p is satisfied for tuples in different partitions of S, then a copy of r will be inserted to all these partitions, which leads to duplicates (i.e., redundancy). Moreover, condition (2) means that each tuple $r \in R$ must be assigned to at least one partition (even if there exists no tuple $s \in P_i(S)$ in any partition of S that satisfies $p(r,s)$). In order to satisfy condition (2), we assign all tuples $r \in R$ that do not have a partitioning partner in S in a round-robin fashion to the different partitions $P_i(R)$ of R.

As mentioned before, any partitioning scheme can be used (e.g., hash, range, round-robin, or even PREF) for the *referenced table*. For simplicity but without loss of generality, we use only the HASH and PREF partitioning scheme in the remainder of the chapter. Moreover, only simple equi-join predicates (as well as conjunctions of simple equi-join predicates) are supported as partitioning predicates p since other join predicates typically result in full redundancy of the PREF partitioned table (i.e., a tuple is then likely to be assigned to each partition of R).

Example: Figure 4.2 shows an example of a database before partitioning (upper part) and after partitioning (lower part). In the example, the table LINEITEM is hash partitioned and thus has no duplicates after partitioning. The table ORDERS (o) is PREF partitioned by table LINEITEM (l) using a partitioning predicate on the join key (orderkey); i.e., ORDERS is the referencing table and LINEITEM the referenced table as well as the seed table. For the table ORDERS, the PREF partitioning scheme introduces duplicates to achieve full data-locality for a potential equi-join over the join key (orderkey). Furthermore, the table CUSTOMER (c) is PREF partitioned by ORDERS using the partitioning predicate on the join key (custkey); i.e., CUSTOMER is the referencing table and ORDERS the referenced table whereas LINEITEM is the seed table of the CUSTOMER table. Again, PREF partitioning the CUSTOMER table results in duplicates. Moreover, we can see that the customer (custkey=3), who has no order, is also added to the partitioned table CUSTOMER (in the first partition).

Thus, by using the PREF partitioning scheme, all tables in a given join path of a query can be co-partitioned as long as there is no cycle in the query graph. Finding the best partitioning scheme for all tables in a given schema and workload that maximizes data-locality under the PREF scheme, however, is a complex task and will be discussed in Sections 4.3 and 4.4.

For query processing, we create two additional bitmap indexes when PREF partitioning a table R by S: The first bitmap index dup indicates for each tuple $r \in R$ if it is the first occurrence (indicated by a 0 in the bitmap index) or if r is a duplicate (indicated by a 1 in the bitmap index). That way, duplicates that result from PREF partitioning can be easily eliminated during query processing, as will be explained in more detail in Section 4.2.2. The second index hasS indicates for each tuple $r \in R$ if there exists a tuple $s \in S$ which satisfies $p(r,s)$. That way, anti-joins, and semi-joins can be optimized. The example in Figure 4.2

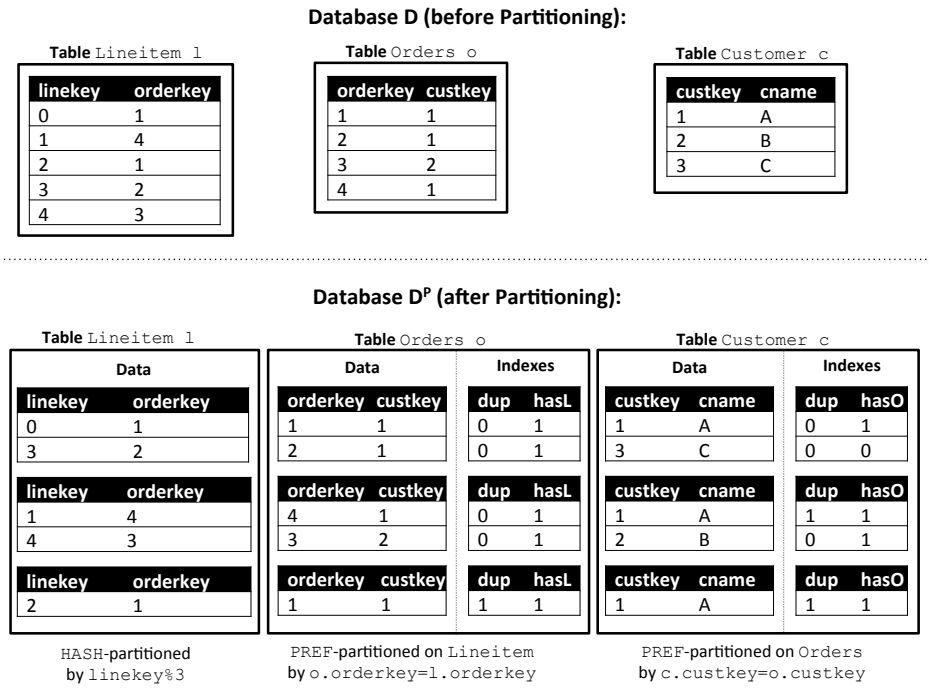


Figure 4.2.: A PREF partitioned Database

shows these indexes for the two PREF partitioned tables. Details about how these indices are used for query processing is discussed in the following Section 4.2.2.

4.2.2 Query Processing

In the following, we discuss how queries need to be rewritten for correctness if PREF partitioned tables are included in a given SQL query Q. This includes adding operations for eliminating duplicates resulting from PREF partitioned tables and adding re-partitioning operations for correct parallel query execution. Furthermore, we also discuss rewrites for optimizing SQL queries (e.g., to optimize queries with anti-joins or outer joins). All these rewrite rules are applied to a compile plan P of query Q. Currently, our rewrite rules only support SPJA queries (Selection, Projection, Join, and Aggregation), while nested SPJA queries are supported by rewriting each SPJA query block individually.

Rewrite Process: The rewrite process is a bottom-up process, which decides for each operator $o \in P$ if a distinct operation or a re-partitioning operation (i.e., a shuffle operation) must be applied to its input(s) before executing the operator o. Note that our distinct operator is not the same as the SQL DISTINCT operator. Our distinct operator eliminates only those duplicates which are generated by our PREF scheme. Duplicates from PREF partitioning can be eliminated using a disjunctive filter predicate that uses the condition `dup=0` for each `dup` attribute of a tuple in an (intermediate) result. A normal SQL DISTINCT operator, however, can still be executed using the attributes of a tuple to find duplicates with the same values. In the rest of the chapter, we always refer to the semantics of our distinct operator. Moreover, the re-partitioning operator also eliminates duplicates resulting from PREF partitioning before shuffling tuples over the network.

In order to decide if a distinct operation or a re-partitioning operation must be applied, the rewrite process annotates two properties to each intermediate result of an operator $o \in P$. In the following, we also use the variable o to refer to the intermediate result produced by the operator o.

- $Dup(o)$: defines if the (intermediate) result o is free of duplicates resulting from PREF partitioning ($Dup(o) = 0$) or not ($Dup(o) = 1$). For tables, we use the same notation; i.e., $Dup(T)$ defines whether table T contains duplicates due to PREF partitioning or not. For a hash partitioned table T, we get $Dup(T) = 0$.
- $Part(o)$: defines a partitioning scheme for the (intermediate) result o including the partitioning method $Part(o).m$ (HASH, PREF, or NONE if neither of the other schemes holds), the list of partitioning attributes $Part(o).A$ and the number of partitions $Part(o).c$. Again, for tables we use the same notation; i.e., $Part(T)$ defines the partitioning scheme of table T.

In the following, we discuss the rewrite rules for each type of operator of an SPJA query individually. Moreover, we assume that the last operator of a plan P is always a projection operator that can be used to eliminate duplicates resulting from PREF partitioning. We do not discuss the selection operator since neither additional duplicate eliminations nor re-partitioning operators need to be added to its input (i.e., the selection operator can be executed without applying any of these rewrites).

Inner equi-join $o = (o_{in1} \bowtie_{o_{in1}.a_1 = o_{in2}.a_2} o_{in2})$: The only re-write rule, we apply to an inner equi-join, is to add additional re-partitioning operators over its inputs o_{in1} and o_{in2}. In the following, we discuss three cases when no re-partitioning operator needs to be added.

(1) The first case holds, if both inputs are hash partitioned and they use the same number of partitions (i.e., $Part(o_{in1}).c = Part(o_{in2}).c$ holds). Moreover, $Part(o_{in1}).A = [a_1]$ and $Part(o_{in2}).A = [a_2]$ must hold as well (i.e., the join keys are used as partitioning attributes).

(2) The second case holds, if $Part(o_{in1}).m =$ HASH and $Part(o_{in2}).m =$ PREF whereas the join predicate $a_1 = a_2$ must be the partitioning predicate of the PREF scheme. Moreover, the partitioning scheme $Part(o_{in1})$ must be the one used for the seed table of the PREF scheme $Part(o_{in2})$.

(3) The third case holds, if $Part(o_{in1}).m =$ PREF and $Part(o_{in2}).m =$ PREF whereas the join predicate $a_1 = a_2$ must be the partitioning predicate of the PREF schema of one input (called referencing input). The other input is called referenced input. Moreover, both PREF schemes must reference the same seed table.

For example, case (2) above holds for a join operation $(l \bowtie_{l.linekey = o.linekey} o)$ over the partitioned database in Figure 4.2. Moreover, case (3) holds for a join operation $(o \bowtie_{o.custkey = c.custkey} c)$ over the same schema.

Otherwise, if none of these three cases holds, the rewrite procedure applies re-partitioning operators to make sure that both inputs use a hash partitioning scheme where the join key is the partitioning attribute, and both schemes use the same number of partitions. The re-partitioning operator also eliminates duplicates resulting from a PREF scheme, as discussed before. If one input is already hash partitioned (using the join key as partitioning attribute), then we only need to re-partition the other input accordingly.

After discussing the rewrite rules, we now present how the properties $Dup(o)$ and $Part(o)$ are set by the rewrite procedure. If we add a re-partitioning operation as discussed before, then we use the hash partitioning scheme of the re-partitioning operator to initialize $Part(o)$ and set $Dup(o) = 0$ since we eliminate duplicates. In case that we do not add a re-partitioning operation (i.e., in the cases 1-3 before), we initialize $Part(o)$ as follows: In case (1), we set $Part(o)$ to be the hash partitioning scheme of one of the inputs (remember, that both inputs use the same partitioning scheme) and $Dup(o) = 0$ since hash partitioned tables never contain duplicates. In cases (2) and (3), we use the PREF scheme of the referenced

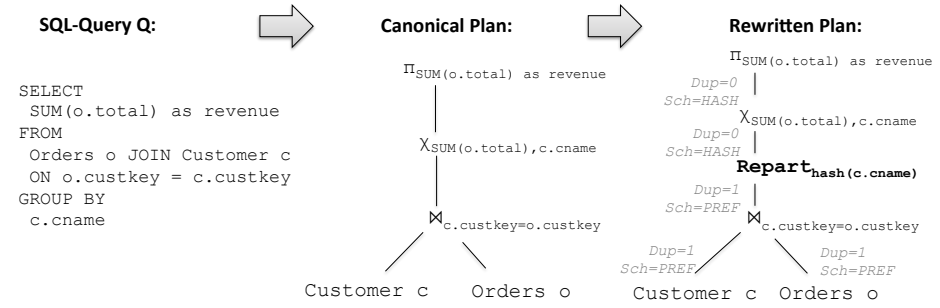


Figure 4.3.: Rewrite Process for Plan P

input to initialize $Part(o)$. Moreover, in the case (2), we always set $Dup(o) = 0$. In case (3), we set $Dup(o) = 0$ if the referenced input has no duplicates. Otherwise, we set $Dup(o) = 1$.

For example, for the intermediate result of the join $(c \bowtie_{c.custkey=o.custkey} o)$ where case (3) holds, the rewrite procedure initializes *Part* to be the PREF scheme of the ORDERS table and sets *Dup* to 1.

Other joins: While equi-joins can be executed on partitioned inputs (as discussed before), other joins such as cross products $o = (o_{in1} \times o_{in2})$ and theta joins $o = (o_{in1} \bowtie_p o_{in2})$ with arbitrary join predicates p need to be executed as remote joins that ships the entire smaller relation to all cluster nodes. For these joins, we set $Part(o).m =$ NONE . Moreover, we also eliminate duplicates in both inputs and set $Dup(o) = 0$. Furthermore, an outer join can be computed as the union of an inner equi-join and an anti-join. An efficient implementation of anti-joins is presented at the end of this section.

Aggregation $o = \chi_{GrpAtts,AggFuncs}(o_{in})$**:** If the partition scheme of the input operator o_{in} is hash partitioned and if the condition $GrpAtts.startWith(Part(o_{in}).A)$ (i.e., the list of group-by attributes starts with or is the same as the list of partitioning attributes), then we do not need to re-partition the input. Otherwise, a re-partitioning operator is added that hash partitions the input o_{in} by the $GrpAtts$. Moreover, if $Dup(o_{in}) = 1$ holds, the re-partitioning operator also eliminates duplicates (as described before). Finally, the rewrite process sets $Part(o)$ to $Part(o_{in})$ if no re-partitioning operator is added. Otherwise, it sets $Part(o)$ to the hash partitioning scheme used for re-partitioning. Moreover, in any case, it sets $Dup(o) = 0$. Figure 4.3 shows an example of an aggregation query over the partitioned database shown in Figure 4.2. In that example, the output of the join is PREF partitioned and contains duplicates (as already discussed before). Thus, the input of the aggregation must be re-partitioned using a hash partitioning scheme on the group-by attribute *c.cname* (which is used as the partitioning scheme of its output). Moreover, the re-partitioning operators eliminate the duplicates resulting from PREF.

Projection $o = \pi_{Atts}(o_{in})$**:** For the projection operator the input o_{in} is never re-partitioned. However, if $Dup(o_{in}) = 1$ we add a distinct operation on input o_{in} that eliminates duplicates using the dup indexes. For this operator, we set $Part(o) = o_{in}$ and $Dup(o) = 0$.

Further Rewrites for Query Optimization: Further rewrite rules can be applied for query optimization when joining a PREF partitioned table R with a referenced table S on p using the index hasS: (1) An anti-join over R and S can be rewritten by using a selection operation with the filter predicate $hasS = 0$ on R without actually joining S. (2) A semi-join over R and S can be rewritten by using a selection operation with the filter predicate $hasS = 1$ on R without actually joining S.

4.2.3 Bulk Loading

As discussed before, the PREF scheme is designed for data warehousing scenarios where new data is loaded in bulks. In the following, we discuss how inserts can be executed over a PREF partitioned table R that references a table S. We assume that the referenced table S has already been bulk loaded.

In order to insert a new tuple r into table R, we have to identify those partitions $P_i(R)$ into which a copy of a tuple r must be inserted. Therefore, we need to identify those partitions $P_i(S)$ of the referenced table S that contains a partitioning partner (i.e., a tuple s, which satisfies the partitioning predicate p for the given tuple r). For example, in Figure 4.2 table CUSTOMER is PREF partitioned referencing table ORDERS. When inserting a customer tuple with $custkey = 1$ into table CUSTOMER, a copy must be inserted into all three partitions since all partitions of ORDERS have a tuple with $custkey = 1$.

For efficiently implementing the insert operation of new tuples without executing a join of R with S, we create a partition index on the referenced attribute of table S. The partition index is a hash-based index that maps unique attribute values to partition numbers i. For example, for the table ORDERS schema in Figure 4.2, we create a partition index on the attribute $custkey$ that maps e.g. $custkey = 1$ to partitions 1 to 3. We show in our experiments in Section 6.5 that partition indexes help to efficiently execute bulk loading of new tuples.

Finally, updates and deletes over a PREF partitioned table are applied to all partitions. However, we do not allow that updates modify those attributes used in a partitioning predicate of a PREF scheme (neither in the referenced nor in the referencing table). Since join keys are typically not updated in data warehousing scenarios, this restriction does not limit the applicability of PREF.

4.3 Schema-driven Automated Partitioning Design

In this section, we present our schema-driven algorithm for automated partitioning design. We first discuss the problem statement and then give a brief overview of our solution. Afterward, we present important details on how we maximize data-locality while minimizing data-redundancy.

4.3.1 Problem Statement and Overview

The *Problem Statement* can be formulated as the following optimization problem: Given a schema S (including referential constraints) and the non-partitioned database D, define a partitioning scheme (HASH or PREF) for each table $T \in S$ (called partitioning configuration) such that data-locality in the resulting partitioned database D^P is maximized with regard to equi-join operations over the referential constraints, while data-redundancy is minimized. In other words, while the main optimization goal is maximizing data-locality under the given partitioning schemes, among those materialization configurations with the same highest data-locality, the one with the minimum data-redundancy should be chosen.

Note that in the above problem statement, we do not consider full replication as a possible choice for a table. The reason is that full replication is only desirable for small tables, while PREF can find a middle ground between partitioning and full replication for other tables that can not be fully replicated.

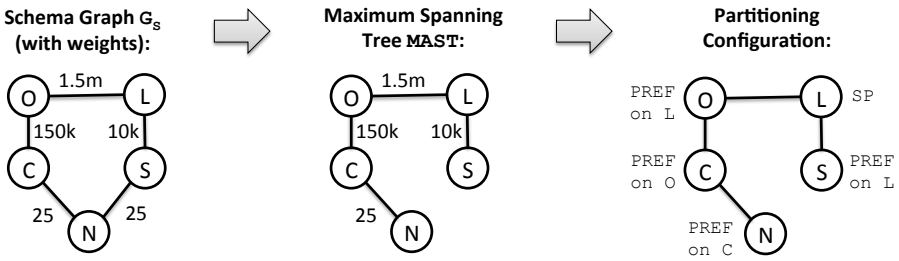


Figure 4.4.: Schema-driven Partitioning Design

Furthermore, small tables that are candidates for full replication can be excluded from the given database schema before applying our design algorithm. In order to solve the before mentioned optimization problem, our algorithm executes the following three steps.

The *first step* is to create an undirected labeled and weighted graph $G_S = (N, E, l(e \in E), w(e \in E))$ for the given schema S (called *schema graph*). While a node $n \in N$ represents a table, an edge $e \in E$ represents a referential constraint in S. Moreover, the labeling function $l(e \in E)$ defines the equi-join predicate for each edge (which is derived from the referential constraint) and the weighting function $w(e \in E)$ defines the network costs if a remote join needs to be executed over that edge. The weight $w(e \in E)$ of an edge is defined to be the size of the smaller table connected to the edge e. The intuition behind this is that the network costs of a potential remote join over an edge e depend on the size of the smaller table since this table is typically shipped over the network. It is clear that we ignore the selectivity of more complex queries (with selection operators and multiple joins), and thus $w(e \in E)$ only represents an upper bound. However, our experiments show that $w(e \in E)$ is a good proxy to represent the total workloads costs even for workloads with complex queries. Figure 4.4 (left-hand side) shows the schema graph resulting from our simplified version of the TPC-H schema for scaling factor $SF = 1$.

As a *second step*, we extract a subset of edges E_{co} from G_S that can be used to co-partition all tables in G_S such that data-locality is maximized. For a given connected G_S, the desired set of edges E_{co} is the maximum spanning tree (or MAST for short). The reason for using the MAST is that by discarding edges with minimal weights from the G_S, the network costs of potential remote joins (i.e., over edges not in the MAST) are minimized and thus data-locality as defined above is maximized. A potential result of this step is shown in Figure 4.4 (center).

Typically, there exists more than one MAST with the same total weight for a connected G_S. For example, in Figure 4.4, instead of discarding the edge between SUPPLIER and NATION, one could also discard the edge between CUSTOMER and NATION since this edge has the same weight. If different MASTs with the same total weight exist, then the following step must be applied for each MAST individually.

Finally, in the *last step*, we enumerate all possible partitioning configurations that can be applied for the MAST to find out which partitioning configuration introduces the minimum data-redundancy. Minimizing data-redundancy is essential since this has a direct effect on the runtime of queries (even if we can achieve maximal data-locality). The partitioning configurations, which we enumerate in our algorithm, all follow the same pattern: one table in the MAST is selected to be the seed table that uses a hash partitioning scheme. In general, it could use any of the existing partitioning schemes such as hash, round-robin, or range partitioning. As partitioning attribute, we use the join attribute in the label $l(e)$ of the edge $e \in E$, which is connected to the node representing the seed table and has the highest weight $w(e)$. All other tables are recursively PREF partitioned on the seed table using the labels of the edges in the MAST as

partitioning predicates. Figure 4.4 (right-hand side) shows one potential partitioning configuration for the MAST, which uses the LINEITEM table as the seed table.

4.3.2 Maximizing Data-Locality

Data-locality (DL) for a given schema graph G_S and the subset of edges E_{co} used for co-partitioning is defined as follows:

$$DL = \frac{\sum_{e \in E_{co}} w(e)}{\sum_{e \in E} w(e)}$$

While $DL = 1$ means that E_{co} contains all edges in G_S (i.e., no remote join is needed), $DL = 0$ means that E_{co} is empty (i.e., no table is co-partitioned by any other table). For example, if we hash partition all tables of a schema on their primary keys, then data-locality will be 0 (as long as the tables do not share the same primary key attributes).

In order to maximize data-locality for a given schema graph G_S that has only one connected component, we extract the maximum spanning tree MAST based on the given weights $w(e \in E)$. The set of edges in the MAST represents the desired set E_{co} since adding one more edge to a MAST will result in a cycle, which means that not all edges can be used for co-partitioning. If G_S has multiple connected components, we extract the MAST for each connected component. In this case, E_{co} represents the union over the edges of all maximum spanning trees.

One other solution (instead of extracting the MAST) is to duplicate tables (i.e., nodes) in the G_S in order to remove cycles and allow one table to use different partitioning schemes. However, join queries could still potentially require remote joins. For example, if we duplicate table NATION in the G_S of Figure 4.4 (left-hand side), we can co-partition one copy of NATION by CUSTOMER and one copy of NATION by SUPPLIER. However, a query using the join path $\mathtt{C}-\mathtt{N}-\mathtt{S}$ then still needs a remote join either from over the edge $C-N$ or the edge $N-S$. Therefore, in our schema-driven algorithm, we do not duplicate nodes at all.

4.3.3 Minimizing Data-Redundancy

The next step after maximizing data-locality is to find a partitioning configuration for all tables in the schema S, which minimizes data-redundancy in the partitioned database D^P. Therefore, we first define data-redundancy (DR) as follows:

$$DR = \frac{|D^P|}{|D|} - 1 = \frac{\sum_{T \in S} |T^P|}{\sum_{T \in S} |T|} - 1$$

While $|D^P|$ represents the size of the database after partitioning, $|D|$ represents the original size of the database before partitioning. $|D^P|$ is defined to be the sum of sizes of all tables $T \in S$ after partitioning (denoted by T^P). Consequently, $DR = 0$ means that no data-redundancy was added to any table after partitioning, while $DR = 1$ means that 100% data-redundancy was added after partitioning (i.e., each tuple in D exists in average twice in D^P). Fully replicating each table to all n nodes of a cluster, thus results in data-redundancy $n-1$.

Listing 4.1: Enumerating Partitioning Configurations

```
function findOptimalPC(MAST mast, Database D){                          1
  PartitionConfig optimalPC;                                            2
  optimalPC.estimatedSize = MAX_INT;                                    3
                                                                        4
  for(each node n_ST in N(mast)){                                       5
    //build new PC based on seed table                                  6
    PartitionConfig newPC;                                              7
    newPC.addScheme(n_ST, SP);                                          8
    addPREF(mast, n_ST, newPC);                                         9
                                                                        10
    //estimate size of newPC                                            11
    estimateSize(newPC, mast, D);                                       12
    if(newPC.estimatedSize<optimalPC.estimatedSize)                     13
      optimalPC = newPC;                                                14
  }                                                                     15
  return optimalPC;                                                     16
}                                                                       17
                                                                        18
//recursively PREF partition tables                                     19
function addPREF(MAST mast, Node referring,                             20
  PartitionConfig pc){                                                  21
  for(each node ref connected                                           22
    to referring by edge e in mast){                                    23
    if(pc.containsScheme(ref))                                          24
      continue;                                                         25
    newPC.addScheme(ref, PREF on referring by l(e));                    26
    addPREF(mast, ref, pc);                                             27
  }                                                                     28
}                                                                       29
```

In Listing 4.1, we show the basic version of our algorithm to enumerate different partitioning configurations (PCs) for a given MAST. For simplicity (but without loss of generality), we assume that the schema graph G_S has only one connected component with only one MAST. Otherwise, we can apply the enumeration algorithm for each MAST individually.

The enumeration algorithm (function findOptimalPC in Listing 4.1) gets a MAST and a non-partitioned database D as input and returns the optimal partitioning configuration for all tables in D. The algorithm, therefore, analyzes as many partitioning configurations as we have nodes in the MAST (line 5-15). Therefore, we construct partitioning configurations (line 7-9) that follow the same pattern: one table is used as the seed table that is partitioned by one of the seed partitioning schemes (or SP for short) such as hash partitioning and all other tables are recursively PREF partitioned on the edges of the MAST (see function addPREF). For each partitioning configuration *newPC*, we finally estimate the size of the partitioned database when applying *newPC* and compare it to the optimal partitioning configuration so far (line 12-14). While seed tables in our partitioning design algorithms never contain duplicate tuples, PREF

partitioned tables do. In order to estimate the size of a database after partitioning, the expected redundancy in all tables, which are PREF partitioned, must be estimated. Redundancy is cumulative, meaning that if a referenced table in the PREF scheme contains duplicates, the referencing table will inherit those duplicates as well. For example, in Figure 4.2 the duplicate orders tuple with $orderkey = 1$ in the ORDERS table results in a duplicate customer tuple with $custrkey = 1$ in the referencing CUSTOMER table. Therefore, in order to estimate the size of a given table, all referenced tables up to the seed (redundancy-free) table must be considered. The details of the size estimation of tables after partitioning are explained in Appendix A.1.

4.3.4 Redundancy-free Tables

As described in Section 4.3.3, the final size of a given table after partitioning is determined by the redundancy factors of all the edges from the seed table to the analyzed table. In complex schemata with many tables, this might result in full or near full redundancy for PREF partitioned tables. This is because only one table is determined by the algorithm to be the seed table, while all other tables are PREF partitioned. In order to remedy this problem, our enumeration algorithm can additionally take user-given constraints as input, which disallows data-redundancy for individual tables. Therefore, we adopt the enumeration algorithm described in Section 4.3.3 as follows: (1) We also enumerate partitioning configurations that can use more than one seed table. We start with configurations with one seed table and increase the number up to $|S|$ seed tables until we satisfy the user-given constraints. Since the maximal data-locality for a MAST monotonically decreases with an increasing number of seed tables, we can stop the enumeration early once we find a partitioning configuration that satisfies the user-given constraints. This scheme will be the partitioning scheme with the maximal data-locality that also satisfies the user-given constraints. (2) We prune partitioning configurations early that add data-redundancy for tables where a user-given constraint disallows data-redundancy. That means for tables where we disallow data-redundancy, we can either use a seed partitioning scheme or a PREF partitioning scheme whose partition predicate refers to the primary key of a table that has no data-redundancy.

4.4 Workload-driven Automated Partitioning Design

In this section, we discuss our workload-driven automated partitioning design algorithm. Again, we start with the problem statement and then give a brief overview of our solution. Afterward, we discuss the details of our algorithm.

4.4.1 Problem Statement and Overview

The *Problem Statement* can be formulated as the following optimization problem: Given a schema S, a workload $W = \{Q_1, Q_2, ..., Q_n\}$ and the non-partitioned database D, define a partitioning scheme (HASH or

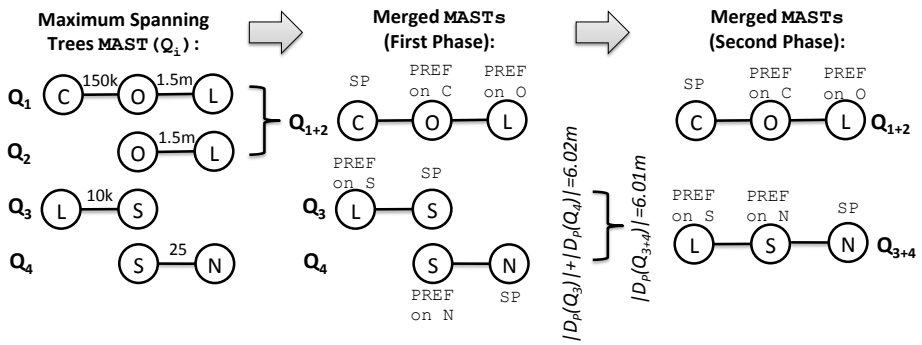


Figure 4.5.: Workload-driven Partitioning Design

PREF) for the tables used by each query $Q_i \in W$ (called partitioning configuration) such that data-locality is maximized for each query Q_i individually, while data-redundancy is globally minimized for all $Q_i \in W$. Like schema-driven partitioning, the main optimization goal is to maximize data-locality under the given partitioning schemes; data-redundancy is only subordinate. In order to solve this optimization problem, our algorithm executes the following three steps.

In the *first step*, our algorithm creates a separate schema graph $G_S(Q_i)$ for each query $Q_i \in W$ where edges represent the join predicates in a query. Afterwards, we compute the $\texttt{MAST}(Q_i)$ for each $G_S(Q_i)$. That way, data-locality for each query $Q_i \in W$ is maximized since one optimally partitioned minimal database $D^P(Q_i)$ could be generated for each query individually. However, this would result in a very high data-redundancy since individual tables will most probably exist several times (using different partitioning schemes for different queries). For example, Figure 4.5 (left hand side) shows the MASTs resulting from four different queries in a workload $W = \{Q_1, Q_2, Q_3, Q_4\}$. Again, if different MASTs with the same total weight exist for one query, we can keep them all to find the optimal solution in the following steps.

In the *second step*, we merge MASTs of individual queries in order to reduce the search space of the algorithm. Given the MASTs, the merge function creates the union of nodes and edges in the individual MASTs.In this phase, we merge a $\texttt{MAST}(Q_j)$ into a $\texttt{MAST}(Q_i)$ if the MAST of Q_j is entirely contained in the MAST of Q_i (i.e., $\texttt{MAST}(Q_i)$ contains all nodes and edges with the same labels and weights of $\texttt{MAST}(Q_j)$). Thus, no cycles can occur in this merge phase. The merged MAST is denoted by $\texttt{MAST}(Q_{i+j})$ If $\texttt{MAST}(Q_j)$ is fully contained in different MASTs, we merge it into one of these MASTs. Moreover, at the end of the first merging phase, we determine the optimal partitioning configuration and estimate the total size of the partitioned database for each merged MAST (using function `findOptimalPC` in Listing 4.1). Figure 4.5 (center) shows a potential result of the first merging phase. This step effectively reduces the search space for the subsequent merging phase.

In the *last step* (i.e., a second merge phase), we use a cost-based approach to merge MASTs further. In this step, we only merge $\texttt{MAST}(Q_j)$ into $\texttt{MAST}(Q_i)$ if the result is acyclic and if we do not sacrifice data-locality while reducing data-redundancy (i.e., if $|D^P(Q_{i+j})| < |D^P(Q_i)| + |D^P(Q_j)|$ holds). Figure 4.5 (right-hand side) shows a potential result of the second merging phase. In this example, MAST of Q_3 and Q_4 are merged since the size of the resulting database $D^P(Q_{3+4})$ after merging is smaller than the sum of sizes of the individual partitioned databases $D^P(Q_3) + D^P(Q_4)$. For query execution, a query can be routed to the MAST, which contains the query and which has minimal data-redundancy for all tables read by that query.

4.4.2 Maximizing Data-Locality

In order to maximize data-locality for a given workload W, we first create a separate schema graph $G_S(Q_i)$ for each query $Q_i \in W$ as described before. The schema graph for the workload-driven algorithm is defined the same way as described in Section 4.3 as $G_S = (N, E, l(e \in E), w(e \in E))$ and can be derived from the query graph of a query Q_i: A query graph is defined in the literature as an undirected labeled graph $G_Q = (N, E, l(e \in E))$ where each node $n \in N$ represents a table (used by the query). An edge $e \in E$ represents a join predicate between two tables while the labeling function $l(e \in E)$ that returns the join predicate for each edge.

Currently, when transforming a query graph $G_Q(Q_i)$ into an equivalent schema graph $G_S(Q_i)$, we only consider those edges which use an equi-join predicate as a label. Note that this does not mean that queries in workload W can only have equi-join predicates. It only means that edges with non-equi join predicates are not added to the schema graph since these predicates already result in high data-redundancy when used for co-partitioning tables by PREF as discussed in Section 4.2. Moreover, for creating the schema graph G_S, a weighting function $w(e \in E)$ needs to be defined for the G_S. This is trivial since the table sizes are given by the non-partitioned database D that is an input to the workload-driven algorithm as well. Note that in addition to table sizes, edge weights G_S could also reflect the costs of a query optimizer to execute the join, if these information items are provided. However, then the merging function would need to be more complex (i.e., a simple union of nodes and edges is not enough since the same edge could have different weights). In the following, we assume that edge weights represent table sizes.

Once the schema graph $G_S(Q_i)$ is created for each query $Q_i \in W$, we can derive the maximum spanning tree $\texttt{MAST}(Q_i)$ for each $G_S(Q_i)$. The $\texttt{MAST}(Q_i)$ represents the set of edges $E_{co}(Q_i)$ that can be used for co-partitioning tables in Q_i. All edges that are in the query graph of Q_i but not in the $\texttt{MAST}(Q_i)$ will result in remote joins. Data-locality DL for a query is thus defined in the same way as before in Section 4.3.2 as the fraction of the sum of weights in $E_{co}(Q_i)$ and the sum of weights for all edges in $G_S(Q_i)$. As shown in Section 4.3 using the edges of a MAST for co-partitioning maximizes data-locality unless we additionally allow to duplicate tables (i.e., nodes) in order to remove cycles in $G_S(Q_i)$. Moreover, in contrast to the schema-driven algorithm, if a connected G_S has different MASTs with the same total weight, our algorithm additionally finds the optimal partitioning configuration for each of the MASTs. It estimates the size of the partitioned database, as shown in Listing 4.1. For the subsequent merging phase, we only keep that MAST, which results in a partitioned database with minimal estimated size.

4.4.3 Minimizing Data-Redundancy

Merging the MASTs of different queries is implemented in two steps as described before: using heuristics in the first merge phase to effectively reduce the search space for the second cost-based merge phase to reduce data-redundancy further. The result after both merging phases is a set of MASTs and an optimal partitioning configuration for each MAST. If a table appears in different MASTs using different partitioning schemes in the partitioning configuration, we duplicate the table in the final partitioned database D^P that we create for all MASTs. However, if a table appears in different MASTs and uses the same partitioning scheme, we do not duplicate this table in D^P. Data-redundancy for a set of MASTs is thus defined as a

fraction of the sum of all partitioned tables and the size of the non-partitioned D. In the following, we only discuss the cost-based merging in detail since the first merge step is trivial.

For cost-based merging, we first define the term *merge configuration*. A merge configuration is a set of merge expressions, which defines for each query Q_i in a given set of queries if the MASTs are merged or not: Q_{i+j} is a merge expression which states that the MASTs of Q_i and Q_j are merged, while $\{Q_i, Q_j\}$ is a set of two merge expressions which state that the MASTs are not merged. Thus, the most simple merge expression is a single query Q_i. For example, for a set of individual queries is $\{Q_1, Q_2, Q_3\}$, $\{Q_{1+2}, Q_3\}$ is one potential merge configuration which holds two merge expressions where Q_1 and Q_2 are merged into one MAST. The problem statement for cost-based merging can thus be re-formulated to find the merge configuration, which results in minimal data-redundancy for all queries in the workload W without sacrificing data-locality.

The search space for all merge configuration for n queries in a given workload W is the same as counting the number of non-empty partitions of a set which is defined by the Bell number $B(n)$ as follows [51]:

$$B_n - B_0 = \sum_{k=1}^{n} S(n,k)$$

$S(n,k)$ is the Stirling number of the second kind [51], which counts the number of ways to partition a set of n elements into k nonempty subsets. Our first merge phase reduces the search space, since queries that are contained in other queries are merged (i.e., removed from the workload), which reduces n in the formula above. For example, for TPC-DS, we can reduce the MASTs for 99 queries to 17 MASTs (i.e., connected components) after the first merging phase. However, for huge workloads, the search space is typically huge after the first merging phase.

Therefore, we use dynamic programming for efficiently finding the optimal materialization configuration for a given workload W. We can use dynamic programming since the optimality principle holds for merge configurations: Let M be an optimal merge configuration for queries $\{Q_1, Q_2, ..., Q_n\}$. Then, every subset M_S of M must be an optimal merge configuration for the queries it contains. To see why this holds, assume that the merge configuration M contains a subset M_S, which is not optimal. That is, there exists another merge configuration $M_{S'}$ for the queries contained in M_S with strictly lower data-redundancy. Denote by M' the merge configuration derived by replacing M_S in M by $M_{S'}$. Since M' contains the same queries as M, the data-redundancy of M' is lower than the data-redundancy of M. This contradicts the optimality of M.

We execute dynamic programming to find the optimal merge configuration with n queries. In our dynamic programming algorithm, to find the optimal merge configuration for level l (i.e., with l queries), we execute a binary *merge step* of an optimal merge configuration of level $l-1$ with one individual query. Thus, in total dynamic programming must analyze 2^n different merge configurations. Moreover, a binary merge step must enumerate all possible merge configurations of size l, which can be constructed from both inputs. Each binary merge step for level l has to analyze maximally l merge configurations. For example, if we want to enumerate all merge configurations of level $l = 4$ which result from merging one merge configuration of level $l = 3$ having two merge expressions $\{Q_{1+2}, Q_3\}$ and a query Q_4, we have to enumerate three resulting merge configurations $\{Q_{1+2}, Q_3, Q_4\}$, $\{Q_{1+2+4}, Q_3\}$, and $\{Q_{1+2}, Q_{3+4}\}$ but not for example $\{Q_{1+2+3+4}\}$. Moreover, memoizing analyzed merge configurations also helps to prune the search space because the same merge configuration might be enumerated by different binary merge steps. For all merge configurations, the binary merge step has to check if the merge configuration is valid (i.e.,

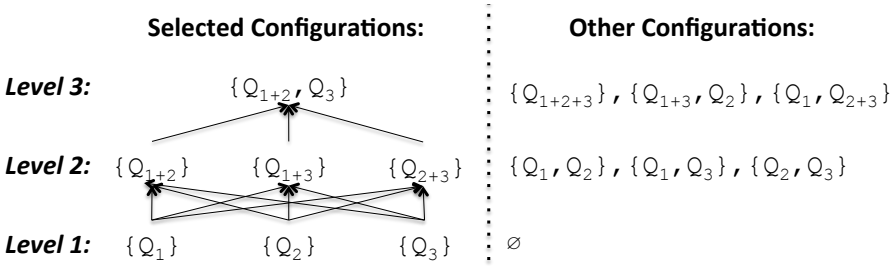


Figure 4.6.: Enumerating Merge Configurations

no cycle occurs in the MAST). Finally, estimating the size for a merge configuration is done by estimating the size for each MAST separately (see Section 4.3) and then summing up the individual estimated sizes.

Example: Figure 4.6 shows an example of our dynamic programming algorithm for enumerating merge configurations for three queries. The left-hand side shows the selected merge configurations, whereas the right-hand side shows the other enumerated merge configurations per level. In this example, the optimal merge configuration of the third level $\{Q_{1+2}, Q_3\}$ builds on the optimal merge configuration $\{Q_{1+2}\}$ of the second level.

4.5 Experimental Evaluation

In this section, we report our experimental evaluation of the techniques presented in our thesis. In our experiments, we used the TPC-H benchmark (8 tables without a skew and 22 queries) as well as the TPC-DS benchmark (24 tables with skew and 99 queries). The goal of the experimental evaluation is to show: (1) the efficiency of parallel query processing over a PREF partitioned database (Section 4.5.1), (2) the costs of bulk loading a PREF partitioned database (Section 4.5.2), (3) the effectiveness of our two automatic partitioning design algorithms: schema-driven (*SD*) and workload-driven (*WD*) (Section 4.5.3), and (4) the accuracy of our redundancy estimates and the runtime needed by these algorithms under different sampling rates (Section 4.5.4).

For actually running queries in parallel, we implemented the PREF partitioning scheme and query processing capabilities over PREF partitioned tables in an open-source parallel database called *XDB* [27]. XDB is built as a middleware over single-node database instances running MySQL. The middleware of XDB provides the query compiler, which parallelizes SQL queries and then uses a coordinator to execute sub-plans in parallel over multiple MySQL nodes. Thus, the complete query execution is pushed down into MySQL.

4.5.1 Efficiency of Query Processing

Setup

In this experiment, we have been executing all 22 TPC-H queries on a database with $SF = 10$. We did not use a higher SF since this SF can already show the effects of varying data-locality and data-redundancy. For the experiment, we deployed XDB on an Amazon AWS cluster with 10 EC2 nodes

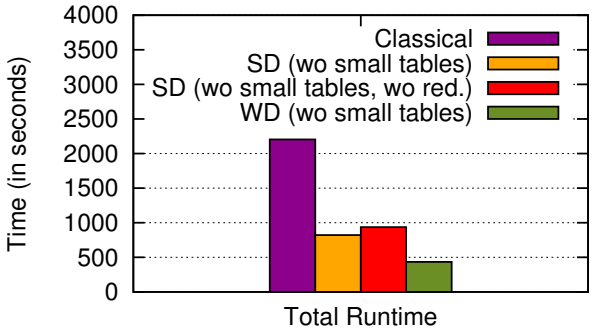


Figure 4.7.: Total Runtime of All TPC-H Queries

(m1.medium), which represent commodity machines with low computing power. Each m1.medium EC2 node has one virtual CPU (2 ECUs), 3.75 GB of RAM, and 420 GB of local instance storage. Each node was running the following software stack: Linux, MySQL 5.6.16, and XDB using Java 8.

Results

For partitioning the TPC-H database, we compare the following variants where Table 4.1 shows the resulting data-locality *DL* and data-redundancy *DR* for all variants:

- *Classical Partitioning (CP):* This represents the classical partition design in data warehousing [55], where one manually selects the biggest table `LINEITEM` and co-partitions the biggest connected table `ORDERS` to hash partitioned them on their join key. Moreover, all other tables are replicated to all nodes.
- *SD (wo small tables):* This represents our *SD* algorithm where we remove small tables (i.e., `NATION`, `REGION` and `SUPPLIER`) from the schema before applying the design algorithm and replicate those tables to all 10 nodes (as discussed in Section 4.3.1). The *SD* design algorithm then suggests using the `LINEITEM` table as the seed table.
- *SD (wo data-redundancy, wo small tables):* Compared to the variant before, we additionally disallow data-redundancy for all non-replicated tables. For this variant, the *SD* design algorithm suggests to use two seed tables (`PART` and `CUSTOMER`) where `LINEITEM` is `PREF` partitioned by `ORDERS`, and `ORDERS` by `CUSTOMER`, while `PARTSUPP` is `PREF` partitioned by `PART`.
- *WD (wo small tables):* Our workload-driven partition design merges all 22 queries into 4 connected components in the first merge phase and then it is reduced to 2 connected components by our second cost-based merge phase: one connected component has 4 tables where `CUSTOMER` is the seed table (while `ORDERS`, `LINEITEM`, and `PART` are `PREF` partitioned) and the other connected component has also 4 tables where `PART` is the seed table (while `PARTSUPP`, `LINEITEM`, and `ORDERS` are `PREF` partitioned).

Figure 4.7 shows the total runtime of all TPC-H queries. For all variants, we excluded the runtime of queries 13 and 22 since these queries did not finish within 1 hour in MySQL using any of the partitioning configurations (due to expensive remote operations). In fact, when using our optimizations that we presented in section 4.2.2, we can rewrite query 13, which uses a left outer join. After rewriting, this query finishes in approximately 40s. The total runtime shows that the partitioning configuration suggested by *WD* (wo small tables) outperforms all other variants. Moreover, both *SD* variants also outperform *CP*.

For the TPC-H schema, we found that *CP* represents the best partitioning configuration with minimal total runtime for all queries when not using `PREF`. Thus, *CP* in this experiment can be seen as a lower

Variant#	DL	DR
Classical	1.0	1.21
SD (wo small tables)	1.0	0.5
SD (wo small tables,wo data-red.)	0.7	0.19
WD (wo small tables)	1.0	1.5

Table 4.1.: Details of TPC-H Queries

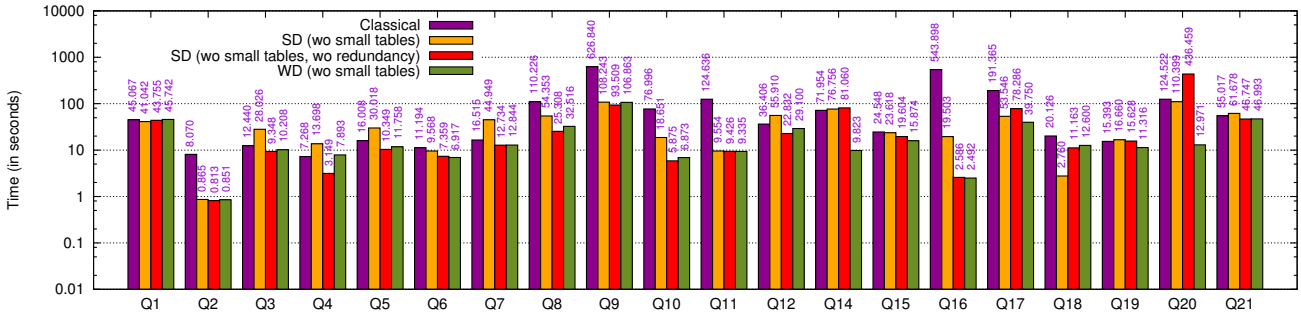


Figure 4.8.: Runtime for Individual TPC-H Queries

bound for existing design algorithms (such as [81, 91]) that are not aware of PREF. Consequently, by comparing with *CP*, we indirectly compare our design algorithms to those algorithms.

Figure 4.8 shows the runtime of each individual TPC-H query. The results show that whenever a query involves a remote operation, the runtime is higher (e.g., the runtime for query 17 and 20 for *SD* wo redundancy is worse than for *SD* or *WD*). Furthermore, when no remote operation is needed, but data-redundancy is high in *CP*, then the query performance also decreases significantly. This can be seen when we compare the runtime of queries 9, 11, 16, 17 for *CP* with all other schemes. For example, query 9 joins in total 6 tables where 4 are fully replicated, and PARTSUPP with $8m$ tuples is one of them.

However, when compared to *WD*, which has even a higher total data-redundancy, we see that this has no negative influence on the runtime of queries at all (which seems contradictory to the result for *CP*). The explanation is that for *WD*, each query has a separate database (i.e., only the tables needed by that query), which results in a minimal redundancy per query. In fact, the average data-redundancy of all individual databases is even a little lower than for *SD* (wo small tables). However, when taking the union of all individual databases (of all queries), the data-redundancy is even higher as for *CP*, as shown in Table 4.1.

Finally, Figure 4.9 shows the effectiveness of our optimizations that we presented in Section 4.2.2. Therefore, we execute different queries with (w) and without (wo) activating these optimizations. As a database, we use the TPC-H database $SF = 10$ partitioned using *SD* (wo small tables). Figure 4.9 shows the execution time for the following three queries: (1) the first query (left-hand side) counts distinct tuples in CUSTOMER (which has duplicates), (2) the second query (center) executes a semi-join of CUSTOMER and ORDERS (and counts all customers with orders), and (3) the third query (right-hand side) executes an anti join of CUSTOMER and ORDERS (and counts all customers without orders). The execution times show that with our optimizations, the runtime gets efficiently reduced by approximately two orders of magnitude for query (1) and (2). Moreover, query (3) did not finish within 1 hour without optimization, while it only took 0.497 seconds to complete with optimization.

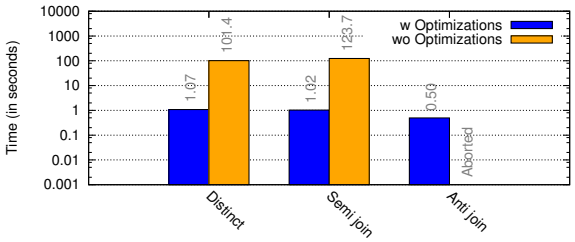


Figure 4.9.: Effectiveness of Optimizations

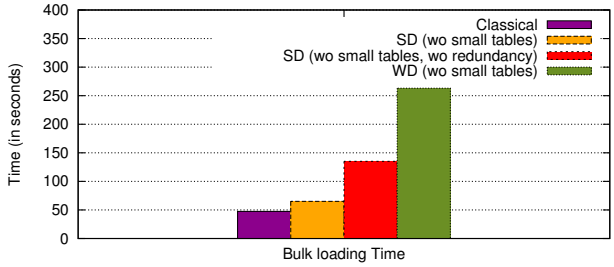


Figure 4.10.: Costs of Bulk Loading

4.5.2 Costs of Bulk Loading

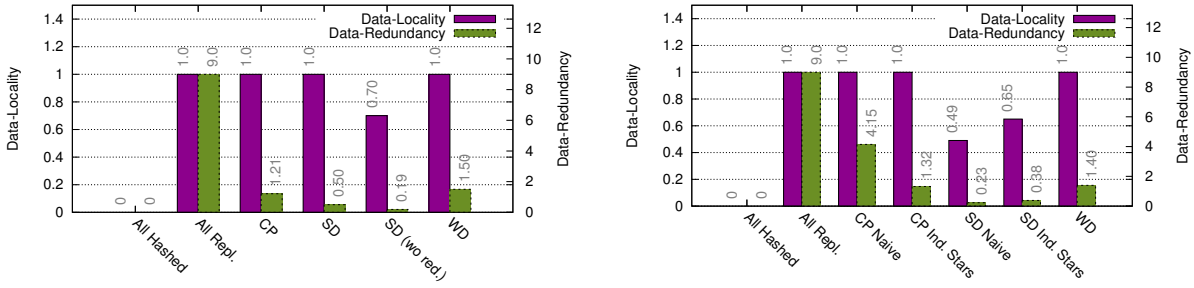
Setup

In this experiment, we bulk loaded the TPC-H database with $SF = 10$ into *XDB*.For the cluster setup, we use the same 10 machines as in the experiment before (see Section 4.5.1).

Results

We report the elapsed time of bulk loading a partitioned TPC-H database for all partitioning schemes discussed in Section 4.5.1. While the Classical Partitioning (*CP*) scheme uses only hash partitioning and replication, all other schemes also use PREF partitioned tables that are bulk loaded using the procedure described in Section 4.2.3. Thus, our schemes (*SD* and *WD*) have to pay higher costs when inserting a tuple into a PREF partitioned table since this requires a look-up operation on the referenced table. However, *CP* has a much higher data-redundancy (as shown already before) and therefore has higher I/O costs.

The results in Figure 4.10 show that the total costs of *SD* (wo small tables) are only a little higher when compared to *CP*. In *SD* (wo small tables, wo redundancy), the costs are a factor $2\times$ higher compared to *SD* (wo small tables). The reason is that the biggest table LINEITEM is PREF partitioned, where each tuple needs a look-up operation. When disallowing redundancy in *SD*, it is a common pattern that the biggest table is PREF partitioned. The reason is that the most prominent table is likely to have outgoing foreign keys that can be leveraged as partitioning predicates without adding redundancy. Finally, *WD* has the highest bulk loading costs since it pays the costs for higher redundancy and look-ups for bulk loading PREF tables. When comparing Figure 4.7 (Execution Costs) and Figure 4.10 (Loading Costs), we see that the better query performance is often paid by higher bulk loading costs, which is worthwhile in data warehousing scenarios.



(a) TPC-H (b) TPC-DS

Figure 4.11.: Locality vs. Redundancy

4.5.3 Effectiveness of Partition Design

Setup

In this experiment, we use an Amazon EC2 machine of type m2.4xlarge with eight virtual CPUs (26 ECUs), 68.4 GB of RAM and $2 \cdot 840$ GB of local instance storage to run our partitioning algorithms. The partitioning design algorithms were implemented in Java 8, and we did not parallelize its computation. Compared to the experiments before, we also use the TPC-DS database in this experiment to show the effects of skew.

Results

We first report the actual data-locality and data-redundancy were resulting from partitioning a TPC-H and a TPC-DS database of scaling factor $SF = 10$ into 10 partitions (i.e., for 10 nodes). We did not use a higher SF since the results for data-locality and data-redundancy would be very similar for our design algorithms with a higher SF. Afterward, we show how data-redundancy evolves, if we scale the number of nodes and partitions from 1 to 100 for both databases (using $SF = 10$ for all experiments). This shows how well scale-out scenarios are supported.

TPC-H (10 partitions): For partitioning the TPC-H database, we use all variants shown for the first experiment in Section 4.5.1. Figure 4.11(a) shows the data-locality and the actual data-redundancy, which results in the different variants shown before. Additionally, we added to two baselines: *All Replicated* (i.e., all tables are replicated) and *All Hashed* (i.e., all tables are hash partitioned on their primary key). While *All Replicated (AR)* achieves perfect data-locality ($DL = 1$) by full data-redundancy ($DR = 9 = n - 1$) where $n = 10$ is the number of nodes, *All Hashed (AH)* has no data-redundancy ($DR = 0$) but at the same time achieves no data-locality ($DL = 0$). Same as *All Replicated*, *CP* also achieves perfect data-locality ($DL = 1$) with less but still a high data-redundancy. Our design algorithms also achieve high data-locality, however, with much less data-redundancy. For example, *SD* (wo small tables) achieves perfect data-locality ($DL = 1$) with very little data-redundancy ($DR = 0.5$) while *WD* has a slightly higher data-redundancy ($DR = 1.5$). Moreover, when reducing data-redundancy to $DR = 0.19$ by *SD* (wo small tables, wo data-redundancy), we still achieve a reasonable data-locality of $DL = 0.7$.

TPC-DS (10 partitions): For partitioning the TPC-DS database, we compare the following variants:

- *CP (Naive and Individual Stars):* This represents the classical partition design as described before. For TPC-DS we applied it in two variants: (Naive) where we only co-partition the biggest table by

its connected biggest table and replicate all other tables, and (Individual Stars) where we manually split the TPC-DS schema into individual star schemata by separating each fact table and all its dimension tables into an individual schemata (resulting in duplicate dimension tables at the cut) and then apply *CP* for each star.

- *SD (Naive and Individual Stars, wo small tables):* For *SD* we removed 5 small tables (each with less than 1000 tuples) and applied the *SD* algorithm in the two variants described before: (Naive) where we apply the *SD* algorithm to all tables, and (Individual Stars) where we apply *SD* to each individual star.
- *WD (wo small tables):* We applied our *WD* algorithm, which merged all 99 queries representing 165 individual connected components (after separating SPJA sub-queries) into 17 connected components (i.e., MASTs) in the first merge phase and then by dynamic programming we reduced them to 7 connected components (i.e., the number of fact tables).

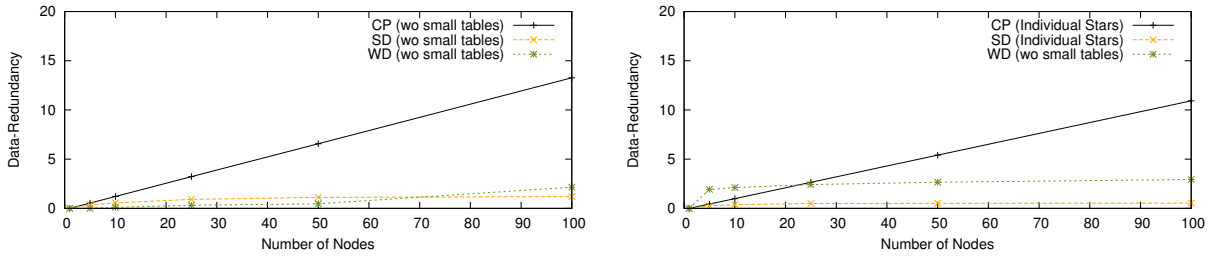
Figure 4.11(b) shows the actual data-locality and data-redundancy, which results for the different variants shown before, as well as for the two baselines (*All Replicated* and *All Hashed*). Notable is that *CP* has a higher data-redundancy $DR = 4.15$ to achieve perfect data-locality as for TPC-H. This is due to replicating more tables of the TPC-DS schema. *CP* (individual stars) involves manual effort but therefore has a much lower data-redundancy $DR = 1.32$. Moreover, while *SD* introduces even less data-redundancy ($DR = 0.23$), it also achieves a much lower data-locality ($DL = 0.49$) in its naive variant. *SD* individual stars mitigate this with almost the same *DR* and $DL = 0.65$. Finally, our *WD* algorithm results in perfect data-locality (without any manual effort) by adding a little more data-redundancy ($DR = 1.4$) compared to *CP* (individual stars).

TPC-H and TPC-DS (1-100 partitions): The goal of this experiment is to show the effect of scale-out on the data-locality and data-redundancy of all schemes discussed before. In this experiment, we partition the TPC-H and TPC-DS database of $SF = 10$ into $1 - 100$ partitions. For partitioning, we compare the best *SD* and the *WD* variant to the best *CP* variant of our previous experiments. We do not show both baselines *All Replicated* and *All Hashed*. While for *All Replicated DR* would be linearly growing (i.e., $DR = n$), *All Hashed* always has $DR = 0$. Figure 4.12 shows the resulting data-redundancy (*DR*) for TPC-H and TPC-DS: The best *CP* scheme has a *DR*, which is growing slower than *All Replicated* but has still a linear growth rate. *WD* and *SD* have a sub-linear growth rate, which is much lower for big numbers of nodes. Consequently, this means for *CP* that each node has to store more data as for the other schemes when scaling-out. Thus, scaling-out scenarios are not well supported in *CP* since the performance of query processing will decrease. Note that here we only show data-redundancy, since one can easily reason that data-locality will not change with varying number of nodes for all schemes. Therefore, since the data-redundancy of our approach grows much slower compared to *CP*, and their data-locality remains unchanged, it means that an increasing number of nodes will have a more positive effect on query processing in our approach compared to the *CP* scheme.

4.5.4 Accuracy vs. Efficiency of Partitioning

Setup

We use the same setup as in the experiment before (see Section 4.5.3).



(a) TPC-H (b) TPC-DS

Figure 4.12.: Varying # of Partitions and Nodes

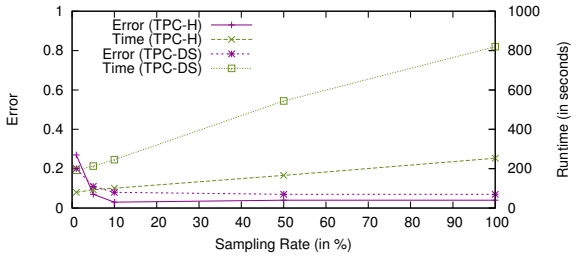


Figure 4.13.: Accuracy vs. Runtime (SD)

Results

In this experiment, we show the accuracy of our data-redundancy (DR) estimates when partitioning a TPC-H data"=base ($SF = 10$, wo skew) and a TPC-DS database ($SF = 10$, w skew) for varying sampling rates (i.e., $1 - 100\%$). For showing accuracy, we calculate the approximation error by $|Estimated(DR) - Actual(DR)|/Actual(DR)$. Moreover, we also analyze the runtime effort under different sampling rates (which includes the runtime to build histograms from the database). Figure 4.13 shows the results of this experiment for the SD (wo small tables) variant. We can see that a small sampling rate of 10% results in a deficient approximation error of about 3% for TPC-H and 8% for TPC-DS while the runtime effort is acceptable since it only needs to be executed once (101s for TPC-H and 246s for TPC-DS). The difference in approximation error between TPC-H and TPC-DS can be accounted for by the difference in the data distribution of these two benchmarks. While TPC-H is uniformly distributed, TPC-DS is highly skewed, which results in higher approximation error. The results of WD are not shown in Figure 4.13) since it has the same approximation error as SD. Moreover, the runtime of WD is dominated by the merge phase, which leads to approximately a factor of $10\times$ increase compared to SD.

4.6 Related Work

Horizontal Partitioning Schemes for Parallel Database Systems

Horizontally co-partition large tables on their join keys was already introduced in the 1990s by parallel database systems such as Gamma [41] and Grace [46] in order to avoid remote join operations. Today co-partitioning is getting even more critical for modern parallel data management platforms such as Shark [46] in order to avoid expensive shuffle operations in MapReduce-based execution engines since CPU

performance has grown much faster than network bandwidth [93]. However, in complex schemata with many tables, co-partitioning on the join key is limited since only subsets of tables can be co-partitioned, which share the same join key. Reference partitioning [42] (or REF partitioning for short) is an existing partitioning scheme to co-partition a table by another table referenced by an outgoing foreign key (i.e., referential constraint). By using REF partitioning, chains of tables can be co-partitioned based on outgoing foreign keys. However, REF partitioning does not support incoming foreign keys. Our PREF partitioning generalizes REF to use an arbitrary equi-join predicate as partitioning predicate. Another option to achieve a high-data-locality for joins is to replicate tables to all nodes in a cluster fully. However, when fully replicating tables data parallelism typically decreases, since the complete query is routed to one copy and executed locally. Simple Virtual Partitioning (SVP) [19] and Adaptive Virtual Partitioning (AVP) [71] are two techniques that achieve data parallelism for fully replicated databases by splitting a query into sub-queries which read-only subsets (i.e., virtual partitions) of the data by adding filter predicates. Compared to PREF, for multi-way joins, SVP and AVP can only add a predicate to at most two co-partitioned tables, which results in expensive full table scans for the other tables in the join path. Moreover, for modern data management platforms with a huge number of (commodity) nodes and large data sets, full replication is also not desirable.

Automatic Design Tuning for Parallel Database Systems

While there exists much work in the area of physical design tuning for a single node database system, much less work exists for tuning parallel database systems [33, 81, 89, 90, 91]. Especially, for automatically finding optimal partitioning schemes for OLAP workloads, we are only aware of a few approaches (e.g., those described in [81, 91, 105]). Compared to our automated design algorithms that build on PREF, these approaches rely only on existing partitioning schemes (such as hash, range-based, round-robin) as well as replication and decide which tables to co-partition and which to replicate. Moreover, the two approaches in [81, 91] are tightly coupled with the database optimizer. However, in this thesis, we show that our partitioning design algorithms, which are independent of any database optimizer, can give a much better query performance by efficiently using on our novel PREF scheme (even when not knowing the workload in advance). Recently, different automatic design partitioning algorithms have been suggested for OLTP workloads [33, 89, 90]. However, the goal of these approaches is to cluster all data used by individual transactions on a single node in order to avoid distributed transactions. For OLAP, however, it is desirable to distribute the data needed for one transaction (i.e., an analytical query) evenly to different nodes to allow parallel data processing. Thus, many of the automatic design partitioning algorithms for OLTP are not applicable for OLAP workloads.

4.7 Conclusions and Outlook

In this chapter, we presented PREF, a novel horizontal partitioning scheme, that allows to co-partition a set of tables by a given set of join predicates by introducing duplicates. Furthermore, based on PREF, we also discussed two automatic partitioning design algorithms that maximize data-locality while minimizing data-redundancy. While our schema-driven design algorithm uses only a schema as input and derives potential join predicates from the schema, the workload-driven algorithm additionally uses a set of queries as input. Our experiments show that while the schema-driven algorithm works reasonably well for small

schemata, the workload-driven design algorithm is more efficient for complex schemata with a bigger number of tables.

One potential avenue of future work is to adopt our automatic partitioning design algorithms to consider data-locality also for other operations than joins only (e.g., aggregations). Moreover, it would also be interesting to adopt our partitioning design algorithms to dynamic data (i.e., updates) and for mixed workloads (OLTP and OLAP) as well as for pure OLTP workloads. We believe that our partitioning design algorithms can also be used to partition schemata for OLTP workloads (when we disallow data-redundancy for all tables) since tuples that are used by a transaction can typically be described by a set of join predicates. Finally, partition pruning for PREF is another exciting avenue of future work.

5 Cost-based fault-tolerance for Parallel Data Processing

5.1 Introduction

Motivation

Parallel Data Processing Engines (PDEs) such as parallel databases (e.g., SAP HANA [26], Greenplum [112] or Teradata [87]) or other modern parallel data management platforms (e.g., Hadoop [114], Scope [122], Spark [120]) are used today to analyze large amounts of data in clusters of shared-nothing machines. While traditional parallel database systems have been designed to run in small clusters with highly available hardware components, modern PDEs are often deployed on large clusters of commodity machines where mid-query failures are a more common case.

In order to deal with mid-query failures, the before mentioned systems implement different strategies: (1) fault-tolerance in parallel databases is implemented in a coarse-grained manner by restarting a query completely when a mid-query failure occurs, whereas (2) modern MapReduce-style PDEs implement a more fine-grained fault-tolerance scheme, which either materializes each intermediate result (e.g., Hadoop) or uses lineage information to be able to recover sub-plans from mid-query failures (e.g., Spark).

However, none of the before mentioned fault-tolerance schemes can efficiently handle analytical workloads, which consist of a mix of queries with a strongly varying runtime ranging from seconds to multiple hours as commonly found in real deployments [92]. For example, while short running queries in Hadoop typically suffer from high materialization costs, long-running queries in parallel databases need to pay high recovery costs when a mid-query failure occurs right before the query is about to finish its execution. Moreover, other parameters such as the size of the cluster (or more precisely the number of nodes typically participating in executing a query) and the mean time between failures (MTBF) also have a strong influence on the efficiency of a fault-tolerance scheme which is not reflected by existing schemes.

Figure 5.1 shows the relationship between the query runtime and the probability that no mid-query failure is occurring while executing the query in different cluster setups varying in the cluster size (i.e., number of nodes) and the MTBF per cluster node.[1]

Cluster 1 in this figure represents a cluster setup with a large number of nodes and a low MTBF per node (which can typically be found in IaaS offerings such as Amazon's Spot Instances). For this type of cluster setup, we can see that even short-running queries already have a very low probability of succeeding. Thus, materializing intermediates to recover from mid-query failures would be beneficial in this setup. Cluster 4 represents a cluster setup with only a few nodes and a much higher MTBF than cluster 1. In this cluster setup materializing intermediates for queries results in unnecessary execution costs since the probability that queries in this setup succeed is always very high. However, for the other two cluster setups (i.e., cluster 2 and 3) the probability that a query succeeds in one attempt strongly depends on the query runtime.

[1] In this chapter, we assume exponential arrival times between failures and accordingly model the probability of having n-failures in time interval t as a Poisson process. That is, assuming that a query is executed on n nodes with independent failure rates, the likelihood of at least one failure within the cluster is given as $P(N_t^n > 0) = 1 - P(N_t^1 = 0)^n = 1 - e^{\frac{-tn}{MTBF}}$ where N_t^n is the number of failures in time interval t on n servers.

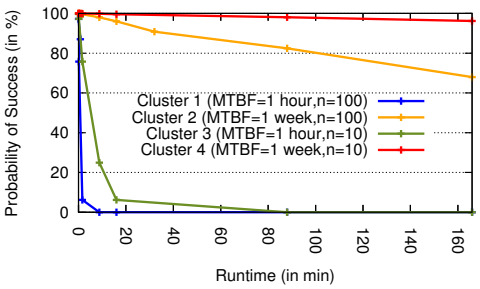


Figure 5.1.: Probability of Success of a Query

Contributions

We present a novel cost-based fault-tolerance scheme. Compared to the existing fault-tolerance schemes discussed before, our scheme selects a subset of intermediates to be materialized (further referred to as *materialization configuration*) such that the query runtime is minimized under the presence of mid-query failures. In order to select which intermediates should be materialized, we present a cost model, which helps to find an optimal materialization configuration for a given query using statistics about the query and the cluster (such as the cluster size and the MTBF). Moreover, we present efficient pruning techniques to reduce the search space of potential materialization configurations enumerated by our cost model. We integrated our approach as well as the existing fault-tolerance schemes in an existing open-source parallel database called *XDB* [28] and executed a comprehensive experimental evaluation. In our evaluation, we show that our cost-based fault-tolerance scheme can effectively deal with different kinds of queries (i.e., varying runtime and costs for materialization) as well as with different cluster setups and thus outperforms all existing strategies.

Outline

First, in Section 5.2 we discuss the assumptions of our cost-based fault-tolerance scheme. In Section 5.3, we present our cost-based fault-tolerance scheme in detail: We show how our cost-based fault-tolerance scheme enumerates different materialization configurations for a given query and how this enumeration can be integrated into existing cost-based optimizers. Moreover, we discuss details of how our cost model estimates the runtime costs of a query under mid-query failures in order to find the optimal materialization configuration. Afterward, in Section 5.4, we discuss pruning techniques to reduce the search space of the enumeration process. Our comprehensive experimental evaluation with the TPC-H benchmark is then discussed in Section 5.5. Finally, we conclude with related work in Section 5.6 and a summary in Section 5.7.

5.2 Assumptions

In this section, we first discuss our assumptions regarding the parallel execution model. Secondly, we discuss which types of failures our cost-based fault-tolerance scheme can handle effectively.

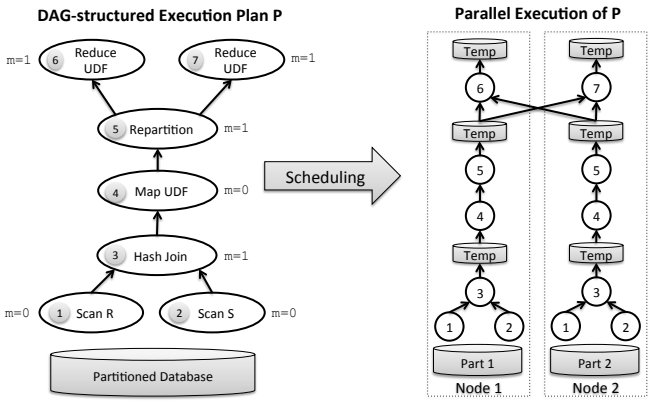


Figure 5.2.: Parallel Execution Model

5.2.1 Parallel Execution Model

Based on the typical design of Parallel Data Processing Engines (PDEs), we assume that the parallel execution plan P is represented as a directed acyclic graph (DAG) and that the data is horizontally partitioned over multiple nodes of a shared-nothing cluster. We further assume that one or more operators $o \in P$ are executed in parallel over partitions on individual nodes Figure 5.2 shows an example of a plan and its partition-parallel execution using two nodes. In order to apply our cost-based fault-tolerance scheme in a PDE, we require that for each operator $o \in P$ there exists a property $m(o)$ which defines whether the output of o should be materialized (i.e., $m(o) = 1$) or not (i.e., $m(o) = 0$) during execution. The set of all properties $m(o)$ for each $o \in P$ is called materialization configuration M_P for plan P. For example, for the plan in Figure 5.2, the output of the operators 3 and 5, 6 7, is set to be materialized. For operators $o \in P$ with $m(o) = 1$, we assume that these are blocking operators (i.e., these operators materialize their output completely before the consumer starts its execution). For operators $o \in P$ where $m(o) = 0$, we assume that the PDE uses its standard input/output behavior such as pipelining data to the consuming operator. That way, our cost-based fault-tolerance scheme can be easily integrated into many existing PDEs by simply materializing the output of sub-plans.

Moreover, our cost-based fault-tolerance scheme supports DAG-structured execution plans which contain arbitrary operators $o \in P$ (i.e., UDFs as well as standard relational operators) as long as the following estimates can be provided for each operator as input to our cost model: the runtime cost $t_r(o)$ for executing an operator o and the materialization cost $t_m(o)$ for materializing its output to a given storage medium. Both $t_r(o)$ and $t_m(o)$ are given for partition parallel execution (i.e., an operator is executed in parallel over all partitions) Typically, these estimates are calculated based on input/output cardinalities of each operator [79].

Finally, in order to support particular platform-specific properties of individual PDEs, operators can be marked as non-materializable (i.e., $m(o)$ is set continuously to 0) or it can be marked as always-materialized (i.e., $m(o)$ is continuously set to 1) before our cost model is applied. For example, some PDEs always materialize the output of a repartitioning-operator. In this case, these operators will be marked as always-materialized. Operators that are either marked as non-materializable or as always-materialized are called bound operators, denoted as $f(o) = 0$. All other operators are called free operators, denoted as $f(o) = 1$. All bound operators are excluded by the enumeration process of our cost-based fault-tolerance scheme.

Notation	Description
P	DAG-structured execution plan P.
M_P	Materialization configuration for execution plan P.
o	Operator $o \in P$.
$m(o)$	Indicates if an operator $o \in P$ should be materialized (i.e., $m(o) = 1$) or not (i.e., $m(o) = 0$).
$f(o)$	Indicates if an operator $o \in P$ is free and optimizable ($f(o) = 1$) or bound ($f(o) = 0$).
$t_r(o)$	The estimated accumulated execution cost of an operator $o \in P$.
$t_m(o)$	The estimated accumulated materialization cost of an operator $o \in P$.
$t(o)$	The total accumulated runtime cost of an operator $o \in P$; i.e. $t(o) = t_r(o) + t_m(o) \cdot m(o)$.
P^c	Collapsed plan resulting from P and M_P.
c	Collapsed operator $c \in P^c$.
Pt	Path from a source to a sink in P^c.
R_{Pt}	Total execution cost of a path Pt without recovery costs for mid-query failures.
T_{Pt}	Total execution cost of a path Pt with recovery costs for mid-query failures.
$CONST_x$	An internal constant x used by our cost model.

Table 5.1.: Terminology and Description

5.2.2 Failure Model

As other work [110], we assume exponential arrival times between failures and that failures are independent. Furthermore, our cost-based scheme is designed to handle process and node failures in clusters of shared-nothing machines. In order to give accurate estimates for the expected total runtime in the presence of mid-query failures, our cost model depends on the assumption that intermediates are not lost if a mid-query failure occurs; i.e., our cost-based scheme assumes that we can restart a query always from the last successfully materialized intermediate result after a given mean time to repair (MTTR). Consequently, if all intermediates are materialized to a separate fault-tolerant storage medium (e.g., Hadoop materializes its results to HDFS), our cost model will be accurate since intermediates are not lost in case of a mid-query failure. In case intermediate results can be lost due to a mid-query failure, our cost model, which estimates the runtime of a query under failure, will be too optimistic. For example, if intermediate results are stored locally per node to the main memory, our cost model will not be accurate since failures lead to a loss of intermediates. As a future avenue of work, we plan to extend our cost model to also be accurate in these cases.

It is important to note that the assumption that intermediates are not lost by mid-query failures does not limit the applicability of our cost-based strategy in PDEs. For example, lineage information [120] makes it possible to reconstruct lost intermediate results of individual nodes without re-executing the whole query plan. Since typically only one or two nodes fail at the same time, our cost-based fault-tolerance scheme is still applicable though slightly optimistic. For the rest of the chapter, we assume that intermediate results are not lost due to a mid-query failure.

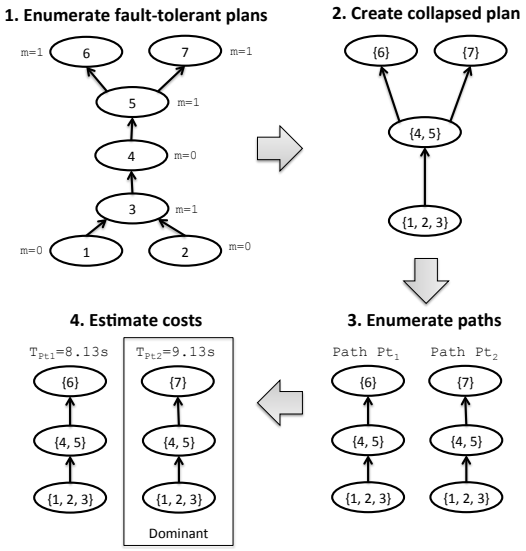


Figure 5.3.: Steps of our Procedure

5.3 Cost-based fault-tolerance

In this section, we first give an overview of our cost-based fault-tolerance scheme and then explain the individual steps in detail.

5.3.1 Overview of the Cost Model

The main goal of our cost-based fault-tolerance scheme is to find an execution plan P and a materialization configuration M_P for a given query such that the total runtime of that query under mid-query failures is minimized. Moreover, our cost-based fault-tolerance scheme is a fine-grained strategy which restarts only sub-plans on nodes that failed.

In order to find an optimal plan, cost-based optimizers of PDEs typically enumerate different equivalent execution plans P and apply a cost function to find the best plan with the minimal runtime cost. However, neither do they enumerate different materialization configurations for fault-tolerance, nor do they consider the costs for recovering from mid-query failures. Therefore, we propose to change the cost-based optimizer to use an enumeration procedure that finds the best combination of a plan P and a materialization configuration M_P to minimize the overall run-time under the previously described failure model. We call this combination $[P, M_P]$ a fault-tolerant plan.

In the following, we give a high-level description of our procedure $findBestFTPlan(Q)$, which finds the best fault-tolerant plan for a given query (see Figure 5.3): (1) First, our procedure enumerates different fault-tolerant plans $[P, M_P]$ for the given query Q. (2) Second, for each enumerated fault-tolerant plan, our procedure creates a collapsed plan P^c where all operators in M_P that do not materialize their output are collapsed into the next materializing operator(s). The idea of the collapsed plan is to represent the granularity of re-execution using these collapsed operators (i.e., if a collapsed operator has finished successfully, it does not need to be re-executed again). (3) Third, for a collapsed plan, all execution paths $Pt \in P^c$ (i.e., paths from each source to each sink in P^c) are enumerated and (4) the total cost T_{Pt} is

estimated for each execution path Pt using a cost function that takes statistics about the operators and the cluster (e.g., the mean time between failures) into account. The path Pt with the maximal estimated total cost for a given materialization configuration is marked as a dominant path of the fault-tolerant plan $[P, M_P]$. The dominant path is a good representative for the total runtime of the fault-tolerant plan under failures.[2] In Figure 5.3, for example, path Pt_2 is marked as the dominant path, and thus the estimated runtime for the given fault-tolerant plan $[P, M_P]$ is the runtime of this path under mid-query failures. Finally, our procedure selects that fault-tolerant plan, which has the shortest dominant path among all enumerated fault-tolerant plans.

Listing 5.1 shows the pseudo-code of our procedure to implement the before mentioned steps. The input is a given query Q and the output is a fault-tolerant plan $[P, M_P]$. In order to estimate the total cost of a path Pt under mid-query failures, the cost function $estimateCost$ requires that the following statistics are given: The runtime costs $t_r(o)$ to execute each operator $o \in P$ and the costs $t_m(o)$ to materialize the output of each operator $o \in P$. Both cost values can be derived from cardinality estimates that are calculated by a cost-based optimizer. Moreover, other parameters that are needed for the cost estimation are the following cluster statistics: the mean time between failures (MTBF) and the mean time to repair (MTTR) for one cluster node. In this chapter, we assume that all these parameters are given by the function call $getCostStats$ in line 6 of Listing 5.1.

In the following sections, we present the details for each of the before-mentioned steps (1-4) and explain the pseudo-code in detail. Table 5.1 summarizes the most important terminology used in the remainder of this chapter.

5.3.2 Step 1: Enumerating Fault-tolerant Plans

Our procedure in Listing 5.1 enumerates potential fault-tolerant plans $[P, M_P]$ for a given query Q using the function $enumFTPlans$ (line 5 in Listing 5.1). A naive implementation of this function would first enumerate all plan and then for each plan use exhaustive search to explore the 2^n variations n being the number of free operators in P to find the optimal plan.

The problem complexity of enumerating all join orders in DAG-structured plans is already $\mathcal{NP}$-hard [79]. Therefore, we use an approximate algorithm to implement the enumeration function $enumFTPlans$. In the first phase, this function uses dynamic programming to find the top-k plans (produced by the last iteration) ordered ascending by their cost without mid-query failures. In the second phase, function $enumFTPlans$ then enumerates all potential materialization configurations for these plans. The intuition to analyze the top-k plans is that a plan P that has slightly higher costs than a plan P' in the first phase, can have lower costs when including the costs to recover from mid-query failures. For example, a plan P that has an operator o with low materialization costs (i.e., $t_m(o)$ is small) at a position in a plan right before a failure is likely to happen (based on the given MTBF) will "waste" much less time to recover than a plan P' which does not have this property.

However, enumerating all materialization configurations for the top-k plans can still be very expensive since the search space for potential materialization configurations is growing exponentially with the number of free operators in a plan. An interesting observation is that the top-k plans might have many

[2] To estimate the cost we do not use the average expected cost but a more pessimistic estimate using percentiles as described in Section 5.3.5, which simplifies the model and allows to avoid complex models involving the maximum over stochastic variables.

Listing 5.1: Find Best Fault-tolerant Plan

```
function findBestFTPlan(Query Q){                                          1
  Plan bestP = null;                                                       2
  Mat.Conf. bestM = null;                                                  3
  int bestT = MAX_INTEGER;                                                 4
  for(each [P,M_P] in enumFTPlans(Q)){                                     5
    cost stats = getCostStats(P);                                          6
    int domT_Pt = 0;                                                       7
    Plan P^c = collapsePlan(P, M_P, stats);                                8
    for(each path Pt in P^c){                                              9
      int T_Pt = estimateCost(Pt, stats);                                  10
      if(T_Pt > domT_Pt) //dominant path                                   11
        domT_Pt = T_Pt;                                                    12
    }                                                                      13
    if(domT_Pt<bestT){ //store if dom. path is shorter                     14
      bestT = domT_Pt;                                                     15
      bestP = P                                                            16
      bestM = M_P                                                          17
    }                                                                      18
  }                                                                        19
  return [bestP,bestM];                                                    20
}                                                                          21
```

paths in common. Based on that observation, we present techniques to prune the search space in Section 5.4.

5.3.3 Step 2: Creating a Collapsed Plan

In its second step, our procedure creates a collapsed plan P^c for the given fault-tolerant plan $[P,M_P]$ by calling the function *collapsePlan* (line 8 in Listing 5.1). As mentioned before, the collapsed plan is the basis for estimating the total cost of a given fault-tolerant plan $[P,M_P]$, including the cost to recover from mid-query failures.

In order to construct the collapsed plan P^c from the given plan P and the materialization configuration M_P, all operators $o \in P$ that are defined by M_P to be not materialized are collapsed into the next consuming operators in the DAG-structured plan, which materialize their output.

In Figure 5.3, we show the collapsed plan P^c (step 2) for the given fault-tolerant plan $[P,M_P]$ (step 1). To put it differently, a collapsed operator $c \in P^c$ represents a sub-plan of P that, once it has materialized its output, does not need to be re-executed again if a mid-query failure occurs. The set of operators of P collapsed into one operator $c \in P^c$ is denoted by $coll(c)$.

Moreover, for each collapsed operator $c \in P^c$, function *collapsePlan* also calculates the runtime (without costs for mid-query failures). The runtime of a collapsed operator c is defined as $t(c) = t_r(c) + t_m(c)$ (i.e., runtime costs plus materialization costs). This runtime is used in step 4 (see Section

5.3.5) to estimate the runtime of the collapsed plan P^c under mid-query failures. In the following, we show how each component is calculated.

The runtime costs $t_r(c)$ of a collapsed operator are determined by the longest execution path in $coll(c)$ called the dominant path $dom(c)$ of a collapsed operator c. For example, in Figure 5.3 (step 2) the dominant path $dom(\{1,2,3\})$ of the collapsed operator $\{1,2,3\}$ is represented by the two operators $\{2,3\}$ if $t_r(2) \geq t_r(1)$ holds. The runtime costs $t_r(c)$ are thus calculated as shown by Equation 5.1. For example, the execution cost of the collapsed operator $\{1,2,3\}$ in Figure 5.3 (step 2) is $t_r(\{1,2,3\}) = (t_r(2) + t_r(3)) \cdot CONST_{pipe}$ if $t_r(2) \geq t_r(1)$ holds. The constant $CONST_{pipe}$ with a value in the interval $(0,1]$ is used to reflect the effects of pipeline parallelism in the sub-plan represented by a collapsed operator c.[3]

$$t_r(c) = \left(\sum_{o \in dom(c)} t_r(o) \right) \cdot CONST_{pipe} \tag{5.1}$$

The materialization costs $t_m(c)$ of a collapsed operator $c \in P^c$ are the materialization costs of the final operator in the longest path. For example, in Figure 5.3 (step 2) $t_m(\{1,2,3\}) = t_m(3)$.

5.3.4 Step 3: Enumerating Paths

Once the collapsed plan P^c is derived from P (as described before), our procedure enumerates all potential execution paths $Pt \in P^c$ and estimates the costs for each path (line 9-13 in Listing 5.1). An execution path Pt is defined as a path from a source operator (i.e., operators with no incoming edges) to a sink operator (i.e., operators with no outgoing edges) in the collapsed plan P^c.

For each enumerated path Pt, we estimate the total runtime cost T_{Pt} using the function call *estimateCost* (line 9), which we describe in the next section in detail. From all enumerated execution paths, the dominant execution path Pt' is selected. The dominant execution path is the path that has the maximal estimated runtime cost under mid-query failures. The intuition is that in a PDE with inter-operator parallelism, the dominant path is a good candidate to represent the total runtime of the complete collapsed plan P^c.

5.3.5 Step 4: Cost Estimation

In this section, we discuss how we estimate the total runtime cost of a given execution path Pt under mid-query failures (line 9 in Listing 5.1). The cost can be generally split down into three components: (1) the runtime cost without failures, (2) the expected runtime that is lost/wasted for each failure, and (3) the number of attempts required to finish a query. While we already know (1), we focus here on (2) and (3).

Wasted Runtime Cost: If we assume that queries start at time t_0, the runtime wasted because of failure for an operator in path Pt is a linear function of time t from the start (t_0) until the operator finishes

[3] Since the constant $CONST_{pipe}$ strongly depends on the execution strategy implemented by the PDE and the underlying hardware, it must be derived individually by calibration experiments.

$(t_0 + t(c))$. Figure 5.4 illustrates it for our running example; the potentially wasted runtime increases linear with time t until the operator finishes, and the result successfully materializes, which resets it for the next operator. Consequently the average wasted runtime for a failure of an operator c is the likelihood of a failure f at time t given that the failure happens during the execution of the operator, times the current execution time $(t - t_0)$:

$$w(c) = \int_{t_0}^{t_0+t(c)} (t - t_0) \cdot P(f_t \mid f_{t_0<t<t_0+t(c)})\, dt \tag{5.2}$$

Here $P(f_t \mid f_{t_0<t<t_0+t(c)})$ corresponds to the likelihood of a failure f at time t given that a failure happens during the execution of the operator ($f_{t_0<t<t_0+t(c)}$). As outlined earlier we assume exponential arrival time between failures, and that failures are independent, which further allows us to simplify the average cost for a single machine to:[4]

$$\begin{aligned} w(c) &= \int_{t_0}^{t_0+t(c)} (t - t_0) \cdot \frac{P(f_t)}{P(f_{t_0<t<t_0+t(c)})}\, dt \\ &= \int_{t_0}^{t_0+t(c)} \frac{(t - t_0) \cdot e^{-\frac{t}{MTBF_{cost}}}}{MTBF_{cost} \cdot \left(-e^{-\frac{t_0+t(c)}{MTBF_{cost}}} + e^{-\frac{t_0}{MTBF_{cost}}}\right)}\, dt \\ &= MTBF_{cost} - \frac{t(c)}{e^{\frac{t(c)}{MTBF_{cost}}} - 1} \end{aligned} \tag{5.3}$$

First, it should be noted that $w(c)$ does no longer depend on t_0 because we have a stationary process (i.e., a Poisson process). Secondly, a limit analysis for $MTBF_{cost} \to \infty$ shows that:

$$w(c) \to \frac{1}{2} \cdot t(c) \tag{5.4}$$

In fact, already for $MTBF_{cost} > t(c)$ the average wasted time w_0 is close $t(c)/2$. While potentially surprising, the reason is simple: the higher the $MTBF_{cost}$ is compared to the execution time of the operator, the more evenly distributed is the failure rate during the execution, resulting in an average closer to the middle $(t(c)/2)$ of the execution time. As our primary goal is not a precise failure model, but a reasonable fast to calculate cost model, we use $t(c)/2$ as an approximation of $w(c)$ in the remainder of the chapter.

Number of Attempts: Given our estimate for the average wasted runtime for the failure of operator c, we now estimate the number of additional attempts we need because of failures to successfully run the operator.

The likelihood that we have a failure in time-interval t given the exponential arrival times is $F(t) = 1 - e^{\frac{-t}{MTBF}}$ [87, 110]. Accordingly the probability that an operator $c \in Pt$ fails is $\eta(c) = F(t(c)) = 1 - e^{\frac{-t(c)}{MTBF_{cost}}}$ and that it succeeds $\gamma(c) = 1 - \eta(c) = e^{\frac{-t(c)}{MTBF_{cost}}}$. As a result the likelihood that operator c succeeds in N attempts is given as:

[4] $MTBF_{cost} = MTBF \cdot CONST_{cost}$ represents the MTBF transformed into an internal cost value of the PDE.

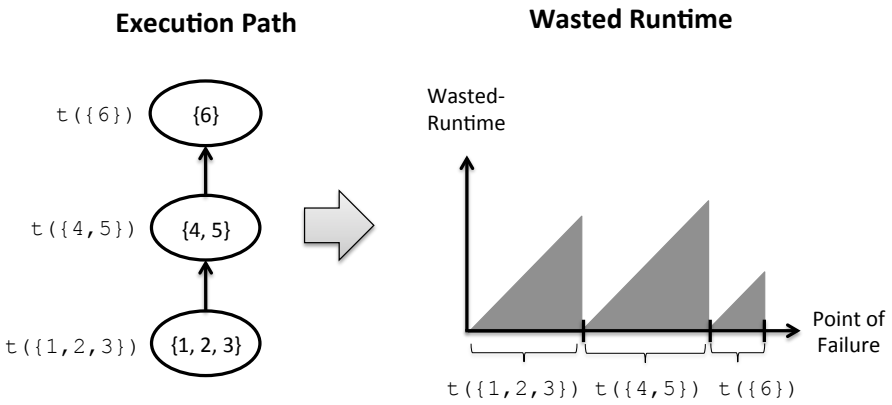


Figure 5.4.: Wasted Runtime Cost

$$S(A \leq N) = \underbrace{\gamma(c)}_{A=0} + \underbrace{\eta(c) \cdot \gamma(c)}_{A=1} + ... + \underbrace{\eta^N \cdot \gamma(c)}_{A=N} \tag{5.5}$$

Here A corresponds to the number of attempts for operator c, not counting the first attempt (to not count the case that we do not have any failures). As it can be noted, $S(A \leq N)$ is a geometric series. For a given finite N, the cumulative probability of success can be presented as the following closed-form expression: $S(A \leq N) = \gamma(c) \cdot (1 - \eta(c)^{(N+1)})/(1 - \eta(c)) = (1 - \eta(c)^{(N+1)})$. Moreover, for $N \to \infty$ the cumulative probability of success is $\gamma(c)/(1 - \eta(c)) = \gamma(c)/\eta(c) = 1$ (i.e., at some point every operator will succeed).

Using the closed-form expression, we now derive the number of attempts $a(c)$ that operator c needs to achieve a desired probability of success $\hat{S}$ (i.e. $S(A \leq N) \geq \hat{S}$) as shown by Equation 5.6.

$$a(c) = max\left(\left(\frac{ln(1-\hat{S})}{ln(\eta(c))} - 1\right), 0\right) \tag{5.6}$$

In all our experiments in Section 5.5, we use $\hat{S} = 0.95$ (i.e., the 95th percentile) that is often used in literature [110] to represent the worst case.

Total Runtime: In order to estimate the total runtime T_{Pt} of an execution path Pt under the presence of mid-query failures, the idea is to sum up the estimated total runtime $T(c)$ of each operator in path Pt as shown by Equation 5.7.

$$T_{Pt} = \sum_{c \in Pt} T(c) \tag{5.7}$$

Based on the given number of attempts $a(c)$ per operator $c \in Pt$, we can estimate the total runtime $T(c)$ of an operator c (under node failures) as follows where $MTTR_{cost}$ is the mean time to repair a failure represented as an internal cost value:[5]

$$T(c) = \underbrace{t(c)}_{(1)} + \underbrace{a(c) \cdot w(c)}_{(2)} + \underbrace{a(c) \cdot MTTR_{cost}}_{(3)} \tag{5.8}$$

[5] As for $MTBF_{cost}$, $MTTR_{cost} = MTTR \cdot CONST_{cost}$ represents the MTTR transformed into an internal cost value of the PDE.

c	{1,2,3}	{4,5}	{6}	{7}
t(c)	4	3	1	2
w(c)	2	1.5	0.5	1
γ(c)	0.94	0.95	0.98	0.96
a(c)	0.0648	0	0	0
T(c)	4.13	3	1	2

Table 5.2.: Example - Cost Estimation

The idea of $T(c)$ is that an operator c needs at least the time $t(c)$ shown as component (1) in Equation 5.8 to finish its execution (i.e., its pure execution time without mid-query failures). Moreover, the second component (2) represents the additional wasted runtime up to attempt $a(c)$ which is given as a sum of the average wasted runtime cost $w(c)$ of c and the number of attempts $a(c)$. The last component (3) of $T(c)$ represents the costs needed to redeploy an operator (i.e, $a(c) \cdot MTTR_{cost}$). Based on $T(c)$, we are now able to calculate the total estimated cost T_{Pt} of path Pt under failures using Equation 5.7 and thus determine the dominant path of a fault-tolerant plan $[P, M_P]$.[6]

Example: In the following we estimate the costs for the two execution paths Pt_1 and Pt_2 shown in Figure 5.3 to calculate T_{Pt1} and T_{Pt2}. For this example, we assume that $t(c)$ for each collapsed operator c is given (as shown in the following table). Moreover, we use $MTBF_{cost} = 60$ and $MTTR_{cost} = 0$. Based on this information, we can derive the average wasted time $w(c)$ and the probability of success using the Equations 5.4 and 5.5 as shown in Table 5.2. Based on these values and $\hat{S} = 0.95$, we can calculate the number of attempts $a(c)$ per operator and the total runtime $T(c)$ using Equations 5.6 and 5.8. Consequently, we get $T_{Pt1} = 8.13$ and $T_{Pt2} = 9.13$ for the two paths and thus the dominant path is Pt_2.

5.4 Pruning of Search Space

In this section, we discuss pruning rules to reduce the search space of potential fault-tolerant plans $[P, M_P]$. All pruning techniques are based on the previously described cost model.

5.4.1 Rule 1: High Materialization Costs

The first pruning rule is applied to a DAG-structured execution plan P that is returned by the first phase of function $enumFTPlans(Q)$ as described in Section 5.3.2 (i.e., before enumerating different materialization configurations).

The intuition of this pruning rule is to mark an operator $o \in P$ as non-materializable if materializing o is guaranteed to lead to a higher runtime cost under mid-query failures. When applying this pruning rule,

[6] It should be noted, that this model is an approximation and not an exact model, mainly for performance reasons. For instance, we assume that operations between machines are not blocking (e.g., one machine can always move ahead without waiting for another). Hence, we might underestimate the execution time. Furthermore, we do not model the different paths as stochastic variables, which they ultimately are. Overall, this leads to faster calculation times but also imprecision. However, as our experimental evaluation will show for many scenarios the estimates are close enough.

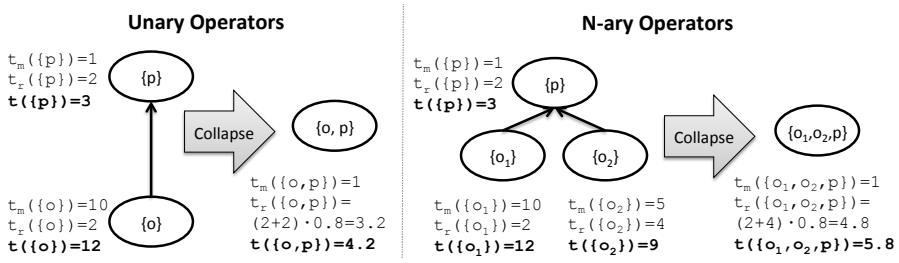


Figure 5.5.: Rule 1 - High Materialization Costs

we differentiate two cases: (1) the operator o is a child of a unary parent operator p and (2) the operator o is a child of an n-ary parent operator p (i.e., p has more than one child operator). In the following, we explain the pruning rule for these two cases more formally.

In the first case (1), we mark an operator o as non-materializable (i.e., we set $m(o) = 0$ and $f(o) = 0$), if $t(\{o,p\}) \leq t(\{o\})$ holds where $\{o,p\}$ and $\{o\}$ are collapsed operators. The runtime for collapsed operators without the additional costs for mid-query failures is calculated as discussed in Section 5.3.3. In the following we show that if $t(\{o,p\}) \leq t(\{o\})$ holds, we can guarantee that the runtime with recovery costs T_{Pt} of an arbitrary path Pt that contains $\{o,p\}$ will always be predicted to be less or equal the runtime of the same path where we replace $\{o,p\}$ by two separate operators $\{o\}$ and $\{p\}$ (where each output is materialized). Therefore, we mark such an operator o as non-materializable.

Following the equations in Section 5.3, if $t(\{o,p\}) \leq t(\{o\})$ holds, the wasted runtime $w(\{o,p\})$ is less or equal the wasted runtime $w(\{o\})$. Moreover, for the same reason the number of attempts $a(\{o,p\})$ is also less or equal $a(\{o\})$ since the probability of success $\gamma(\{o,p\})$ is greater equal than $\gamma(\{o\})$. To that end, the total estimated runtime (including the additional costs for mid-query failures) T_{Pt^1} of an arbitrary path Pt^1 that contains a collapsed operator $\{o,p\}$ is less or equal the total estimated runtime of a variant of Pt^1 called Pt^2 where we replace $\{o,p\}$ by only $\{o\}$. Adding operator $\{p\}$ to the path Pt^2 only increases the total runtime T_{Pt^2}. Thus, $T_{Pt^1} \leq T_{Pt^2}$ holds if $t(\{\text{o,p}\}) \leq t(\{\text{o}\})$ holds.

Figure 5.5 (left-hand side) shows an example with two operators, where the total runtime costs of the collapsed operator $\{o,p\}$ is less than the estimated total runtime costs of the child operator $\{o\}$. For calculating the execution costs $t_r(\{o,p\})$, we use $CONST_{pipe} = 0.8$ in this example. Since $t(\{o,p\}) < t(\{o\})$ holds, we mark operator o as non-materializable and thus exclude operator o from the enumeration of materialization configurations.

Case (2) is similar to case (1): In order to simplify the presentation, but without loss of generality, we assume that p has two child operators o_1 and o_2. In this case, we mark the two child operators o_1 and o_2 as non-materializable, if and only if it is guaranteed that the operator $\{o_1,o_2,p\}$, which results from collapsing o_1, o_2 and p, has a total estimated runtime costs less than o_1 and less than o_2. This is the case, if $t(\{o_1,o_2,p\}) \leq t(\{o_1\}) \wedge t(\{o_1,o_2,p\}) \leq t(\{o_2\})$ holds. In order to proof this pruning rule, we can use the same argument as before individually for $\{o_1\}$ and $\{o_2\}$. This proof can be extended to an arbitrary number of child operators of p.

Figure 5.5 (right hand side) shows an example for this case where the total runtime costs of the collapsed operator $\{o_1,o_2,p\}$ is less than the estimated total runtime costs of both operators $\{o_1\}$ and $\{o_2\}$. For calculating the execution costs $t_r(\{o_1,o_2,p\})$ we use $CONST_{pipe} = 0.8$ in this example. Since $t(\{o_1,o_2,p\}) \leq t(\{o_1\}) \wedge t(\{o_1,o_2,p\}) \leq t(\{o_2\})$ holds, we mark operator 1 and 2 as non-materializable and exclude those operators from the enumeration of materialization configurations.

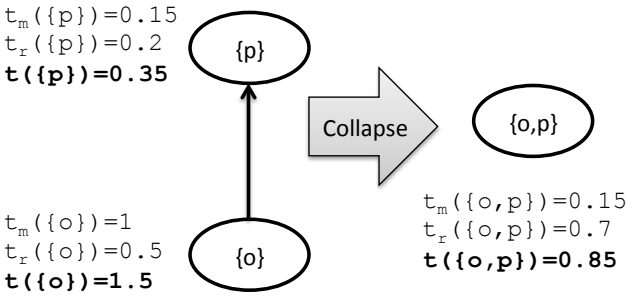


Figure 5.6.: Rule 2 - Short-Running Operators

5.4.2 Rule 2: High Probability of Success

The second pruning rule is applied to a DAG-structure execution plan P enumerated by the first phase of function $enumFTPlans(Q)$ described in Section 5.3.2 (i.e., before enumerating different materialization configurations). This rule is only applied to operators o that are a child of a unary parent operator p. The idea of this pruning rule is to mark an operator o as non-materializable if the collapsed operator $\{o,p\}$ has a probability of success higher than the desired probability of success.

More formally, we mark o as non-materializable (i.e., we set $m(o) = 0$ and $f(o) = 0$ for all materialization configurations), if and only if $\gamma(\{o,p\}) \geq \hat{S}$ whereas $\hat{S}$ is the desired probability of success as discussed in Section 5.3.5. In that case, we expect that the collapsed operator finishes without an additional attempt (i.e., no mid-query failure happens). Since we save the materialization cost of o when collapsing o into p, the total runtime for any possible execution path Pt^1 which contains $\{o,p\}$ is under our model guaranteed to be less or equal total runtime of a variant of Pt^1 called Pt^2 where we replace $\{o,p\}$ by $\{o\}$ and $\{p\}$. For a high MTBF (which leads to higher success probabilities), this rule marks operators with even high total execution costs as non-materializable.

Figure 5.6 shows an example for this pruning rule. For the collapsed operator $\{o,p\}$ we get a probability of success $\gamma(\{o,p\}) = 0.999$ when applying the cost equation in Section 5.3 using a mean-time-between-failures of $MTBF_{cost} = 3600$. Since this probability is higher than our $\hat{S} = 0.95$, we mark operator $o \in P$ as non-materializable and thus exclude operator o from the enumeration of materialization configurations in our cost model.

5.4.3 Rule 3: Long Execution Paths

The last pruning rule is applied during the enumeration of execution paths for a given fault-tolerant plan $[P,M_P]$ to stop the path enumeration early without analyzing all execution paths (i.e., the pruning can be applied after line 9 in Listing 5.1). The idea of this rule is that we stop the path enumeration early if we find an execution path Pt, which has a runtime that is longer than the best memoized dominant path of all previously enumerated fault-tolerant plans for a given query Q.

More formally, if the analyzed path Pt of the current fault-tolerant plan has guaranteed higher total execution cost than the best memoized dominant path of a previously analyzed fault-tolerant plan (i.e., if $T_{Pt} \geq bestT$ holds), then we do not need to analyze any other execution path of the current fault-tolerant

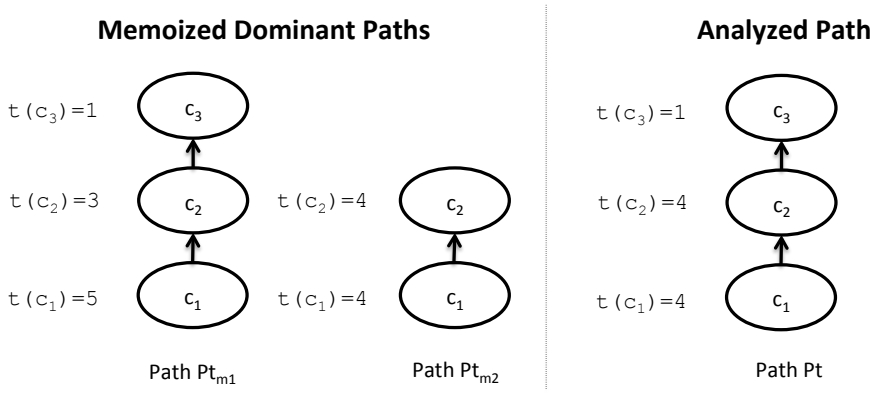


Figure 5.7.: Rule 3 - Memoizing best Dominant Paths

plan. Consequently, in this case, we can stop the path enumeration for the current fault-tolerant plan early and continue with the next fault-tolerant plan without analyzing all paths.

In order to find out that if we can stop the path enumeration, we analyze if one of the following conditions holds: (1) $R_{Pt} \geq bestT$: If for the runtime R_{Pt} of path Pt without any failure, $R_{Pt} \geq bestT$ holds, we can skip all remaining paths for the given fault-tolerant plan $[P, M_P]$. The reason is that the dominant path (called Pt' further on) has a runtime $T_{Pt'}$ under mid-query failures that is greater equal the runtime T_{Pt} of the current path Pt. Thus, since $T_{Pt'} \geq T_{Pt}$, $T_{Pt} \geq R_{Pt}$ and $R_{Pt} \geq bestT$ holds, we can conclude that $T_{Pt'} \geq bestT$ holds as well. Moreover, since the runtime of a path without mid-query failures can be computed as $R_{Pt} = \sum_{c \in Pt} t(c)$, we can calculate R_{Pt} easily without calling the *estimateCosts* function (in line 9 in Listing 5.1). (2) $T_{Pt} \geq bestT$: If for any path $T_{Pt} \geq bestT$ holds, we can skip all remaining paths for the given fault-tolerant plan $[P, M_P]$ since then the dominant path (called Pt') of the the given fault-tolerant plan $[P, M_P]$ will have a runtime $T_{Pt'}$ under mid-query failures that is guaranteed to be greater equal $bestT$. Thus, since $T_{Pt'} \geq T_{Pt}$ and $T_{Pt} \geq bestT$ holds, we can conclude that $T_{Pt'} \geq bestT$ holds as well. Compared to the rule before, however, we have to call the *estimateCosts* function to compute T_{Pt}.

Additionally, instead of memoizing only one best dominant path and its runtime $bestT$, we could memoize multiple best dominant paths with different numbers of collapsed operators for pruning even more aggressively. These memoized dominant paths can then be used to check for a given execution path Pt, if we find a memoized dominant path Pt_m with the same number of collapsed operators where the following equation holds:

$$\forall (i \in \{0...|Pt|\}) : sort(Pt).getT(i) \geq sort(Pt_m.getT(i)) \tag{5.9}$$

This means, we sort the operators in the paths Pt_m and Pt descending by their total execution costs (using the function *sortTDesc*) and pairwise compare the total costs of the operators with the same index i in both sorted lists. If the condition in Equation 5.9 holds, we can derive that $T_{Pt} \geq T_{Pt_m}$. If the condition holds, we say $Pt \geq Pt_m$ for short. The rationale why this holds is that the average wasted runtime $w(c)$, as well as the number of attempts $a(c)$, will be greater than or equal for each operator c in path Pt compared to path Pt_m and thus T_{Pt} will also be greater than or equal T_{Pt_m}. Moreover, we also can compare path Pt to any memoized dominant path Pt_m with a smaller number of collapsed operators, since we always can add further collapsed operators with total costs $t(\{o\}) = 0$ to Pt_m.

Figure 5.7 shows an example for this pruning rule. The left hand-side shows two memoized dominant paths: Pt_{m1} is the best dominant path with three collapsed operators (c'_1 to c'_3) and Pt_{m2} is the best

dominant path with two collapsed operators (c_1'' and c_2'') that are already sorted descending by runtime to simplify the presentation. For the given path Pt with three collapsed operators (c_1 to c_3), we see that $Pt \geq Pt_{m1}$ does not hold but $Pt \geq Pt_{m2}$ holds. Thus, we can skip the enumeration of the remaining paths of the given fault-tolerant plan $[P, M_P]$.

Finally, since multiple equivalent DAG-structured physical execution plans are enumerated for the same query (as described in Section 5.3) in cost-based enumeration, this rule will lead to an even better reduction of the search space if we store *bestT* as well as the memoized best dominant paths for the complete enumeration process of all equivalent execution plans for a given query. In our experiments in Section 5.5, we analyze the efficiency of this pruning rule when it is applied to all equivalent execution plans for a given query.

5.5 Experimental Evaluation

The goal of the experimental evaluation is to show: (2) the accuracy of our cost-based fault-tolerance scheme and its robustness (see Section 5.5.4), and (3) the effectiveness of our pruning techniques to reduce the search space of potential materialization configurations (see Section 5.5.5).

5.5.1 Setup and Workload

Cluster Setup

For our experiments, we used a cluster of 10 commodity machines each having 2 CPUs (Intel Xeon E5345, 2.33GHz, 4 cores), 8 GB RAM, and two local SCSI hard disks (10,000rpm) each with 73.4 GB storage capacity. As a fault-tolerant storage medium, we used an external iSCSI storage with 12 disks (5,400rpm), each having 4 TB storage capacity attached via 1 GB Ethernet. Each cluster node was running the following software stack: openSUSE 13.1, MySQL 5.6.16, and Java 8.

Workload

Moreover, for the workload in our experiments, we used the TPC-H benchmark schema and data. We replicated the two small tables `NATION` and `REGION` to all cluster nodes. All other tables were horizontally partitioned into 10 partitions and distributed to different nodes (i.e., one partition per node). We co-partitioned the two tables `LINEITEM` and `ORDERS` using `HASH`-partitioning on the `orderkey` attribute. For all other tables, we used the `RREF`-partitioning [28] that partially replicates individual tuples in order to minimize distributed joins as follows: `CUSTOMER` `RREF` by `ORDERS` (on `custkey`), `PARTSUPP` `RREF` by `LINEITEM` (on `suppkey` and `partkey`), as well as `SUPPLIER` `RREF` by `PARTSUPP` (on `suppkey`).

Implementation

For actually running queries in parallel, we implemented our cost-based fault-tolerance scheme as well as the other existing fault-tolerance schemes in the open source PDE called *XDB* [28], which provides DAG-structured execution plans.*XDB* is implemented as a middleware that executes queries over sharded single-node MySQL database instances. The middleware provides the fault-tolerant query optimizer, which determines the subset of intermediates that will be materialized during execution. For execution,

the query coordinator splits the execution plan according to the materialization configuration into sub-plans over different partitions, which are then executed on the corresponding cluster nodes. Sub-plans were configured to store their output to the external iSCSI storage. In order to achieve fault-tolerance concerning mid-query failures, a query coordinator monitors the execution of individual sub-plans and restarts them once a failure is detected.

Statistics

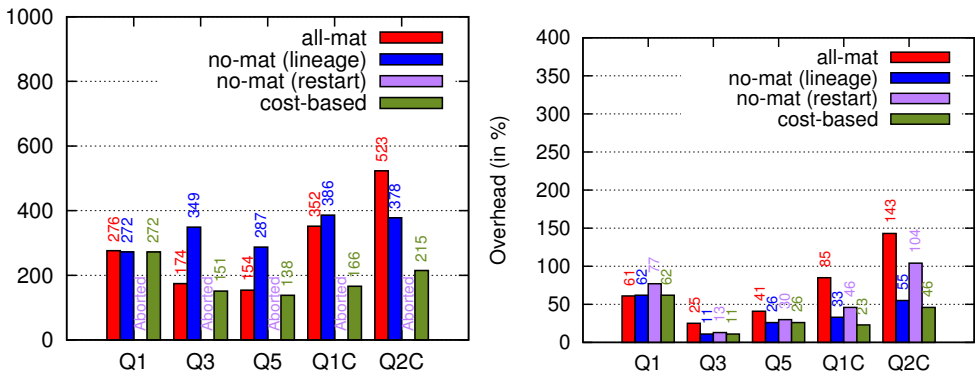
In order to show the effect of different failure rates, we injected failures using different MTBFs that we also use as input to our fault-tolerant query optimizer. In all experiments, we used a monitoring interval to $2s$ in *XDB*. Thus, on average, a failed operator was redeployed in $1s$, and we, therefore, use MTTR=$1s$ in all experiments. Moreover, we also use perfect cost estimates for $t_r(o)$ and $t_m(o)$ for our experiments in Section 5.5.2 and Section 5.5.3. In order to derive perfect query statistics, we executed all queries in *XDB* (w/o injecting failures) and measured $t_r(o)$ and $t_m(o)$ for each operator o. Additionally, in order to show the effects of non-exact estimates in Section 5.5.4, we introduced errors in these statistics. Finally, as constants, we use $CONST_{cost} = 1$ since the estimates represent the real-time as well as $CONST_{pipe} = 1$ that we derived using a calibration experiment in *XDB*.

5.5.2 Efficiency for Different Queries

In this experiment, we compare the overhead of different existing fault-tolerance schemes to our cost-based scheme when mid-query failures happen while executing queries over a partitioned TPC-H database of $SF = 100$. The reported overhead in this experiment represents the ratio of the runtime of a query under a given fault-tolerance scheme (i.e., including the additional materialization costs and recovery costs) over the baseline execution time. The baseline execution time for all schemes is the pure query runtime without additional costs (i.e., no extra materialization costs and no recovery costs due to mid-query failures). Thus, if we report that a scheme has 50% overhead, it means that the query execution under mid-query failures using that scheme took 50% more time than the baseline. The fault-tolerance schemes, which we compare in this experiment, are:

- **all-mat:** This represents the strategy of Hadoop, where all intermediates are materialized. Moreover, for recovering a fine-grained strategy is used (i.e., only sub-plans that fail are restarted).
- **no-mat (lineage):** This represents the strategy of Shark, where lineage information is used to re-compute failed sub-plans. Moreover, for recovering a fine-grained strategy is used.
- **no-mat (restart):** This represents the (coarse-grained) strategy of a parallel database, where the complete query plan is restarted once a sub-plan fails.
- **cost-based:** This represents our strategy that materializes intermediates based on a cost model. Moreover, for recovering a fine-grained strategy is used.

We compare the overhead of these schemes for different TPC-H queries with varying complexity: Q1 (no join), Q3 (3-way join), and Q5 (6-way join). Moreover, we run two complex queries: a variant of Q1(called Q1C) and a variant of Q2 (called Q2C). Q1C is a nested query that uses Q1 as an inner query and joins the result with the `LINEITEM` table to count the individual items with a given status that have a price above the calculated average. Thus, Q1C is a query that has an aggregation operator in the middle



(a) Low MTBF (b) High MTBF

Figure 5.8.: Varying Queries

of the plan. Q2C modifies Q2 (which is already nested) such that the inner aggregation query (4-way join) is used as a common-table-expression (CTE), which is consumed by two outer queries. For the two outer queries, we used the original outer query (5-way join) with different filter predicates on the PART table. This query represents a DAG-structured plan. Moreover, for injecting failures, we use the following two settings per query: (1) an MTBF per node which is 10% higher than the baseline runtime of each query to simulate high failure rates, and (2) an MTBF per node which is 10× the baseline runtime to simulate low failure rates. For measuring the actual runtime, we created 10 failure traces for each unique MTBF using an exponential distribution where $\lambda = 1/MTBF$ and used the same set of traces for injecting failures to compare the overhead of different fault-tolerance schemes. We used this method in all experiments.

The result of this experiment is shown in Figure 5.8. The **cost-based model always selects the sweet spots for materialization** for different queries and different failure rates. Thus, the cost-based scheme has the least or comparable overhead as the best existing fault-tolerance scheme. Query Q1 is an exception in this experiment since it has no free operator that can be selected for materialization. Thus, all schemes show almost the same overhead except for no-mat (restart).

Low MTBF

For all queries (except Q1), the not-mat (lineage) scheme has a higher overhead than the cost-based scheme. Moreover, the no-mat (restart) scheme does not finish any query (i.e., we aborted them after 100 restarts). Another interesting pattern is that both star-join queries (Q3 and Q5) have a similar overhead for the cost-based and the all-mat scheme. The reason is that the cost-based scheme materializes most intermediate results (except the most expensive ones), and therefore the resulting overhead is similar to the all-mat scheme. For more complex queries (Q1C and Q2C), the cost-based scheme has the best overhead. The reason is that these queries contain an aggregation operator in the middle of the plan, which has low materialization costs. The cost-based scheme selects this aggregation operator as a checkpoint that efficiently minimizes the overhead under mid-query failures. Moreover, for Q1C and Q2C, the all-mat scheme has a much higher overhead compared to the cost-based scheme. The reason is that the total materialization costs for many operators are relatively high, and the cost-based scheme thus does not materialize these operators.

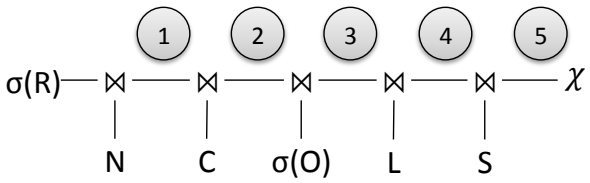


Figure 5.9.: TPC-H Query 5 (Free operators 1-5)

High MTBF

For low failure rates, the results are different. The no-mat (lineage) scheme and the cost-based scheme are the best schemes for Q3 and Q5. The reason is that the cost-based scheme only materializes a few small intermediates and thus is similar to the no-mat (lineage) scheme. For Q1C and Q2C, the no-mat (lineage) scheme is slightly worse since the cost-based scheme materializes the small aggregation operator in the middle of the plan and has thus a lower overhead if a failure occurs. The no-mat (restart) scheme also tends to have a slightly higher overhead than the cost-based scheme since it is a coarse-grained scheme. Interestingly, the all-mat scheme also has only a slightly higher overhead than the cost-based scheme for Q3 and Q5. The rationale is that these queries have moderate total materialization costs (approx. $20-30\%$ of the runtime costs) under all-mat. Q1C and Q2C, however, have much higher materialization costs (approx. $60-100\%$ of the runtime costs) for the all-mat scheme leading to much higher overhead.

5.5.3 Efficiency for Varying Statistics

In this experiment, we compare the overhead of the different fault-tolerance strategies (a) when running the same query with varying runtime for a fixed MTBF to show the effect of short- and long-running queries and (b) when running the same query under MTBFs to show the effect of different cluster setups.

Exp. 2a - Varying Query Runtime (Figure 5.10)

In this experiment, we executed TPC-H query 5 over different scaling factors ranging from $SF = 1$ to $SF = 1000$. This resulted in query execution times ranging from a few seconds up to multiple hours. We selected TPC-H query 5 in this experiment since this is a typical analytical query with multiple join operations and an aggregation operator on top (see Figure 5.9). For this experiment, the output of every join operator was defined to be a free operator (marked with the numbers 1-5 in Figure 5.9) and thus could be selected by our cost model to be materialized. Thus, for each enumerated plan, our procedure in Section 5.3 enumerated 2^5 materialization configurations when pruning was deactivated. Moreover, we injected mid-query failures using a MTBF of 1 day (1440 minutes) per node.

The result of this experiment is shown in Figure 5.10. The x-axis shows the baseline-runtime of the query (i.e., when no failure is happening), and the y-axis shows the overhead under mid-query failures. The **cost-based scheme has the lowest overhead for all queries**, starting with 0% for short-running queries and ending with 247% for long-running queries. Compared to our cost-based scheme, the other schemes impose a higher or comparable overhead depending on the query runtime. Both no-mat schemes also start with 0% overhead for short-running queries; however, for queries with a higher runtime, the overhead increases. As expected, for the restart-based no-mat scheme, queries with a high runtime tend not to finish since the complete query is restarted over and over. The lineage-based not-mat scheme

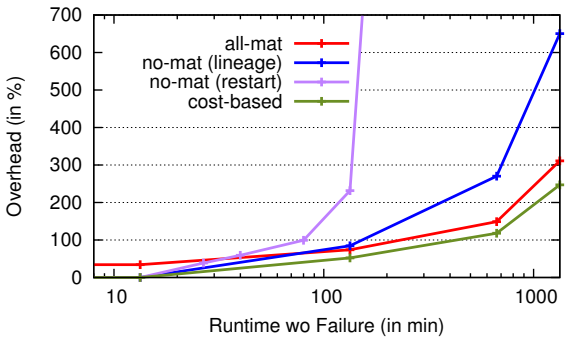


Figure 5.10.: Varying Runtime

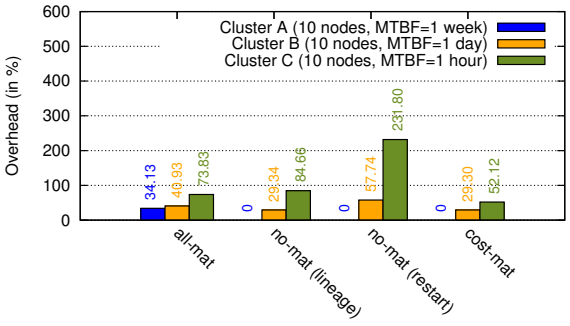


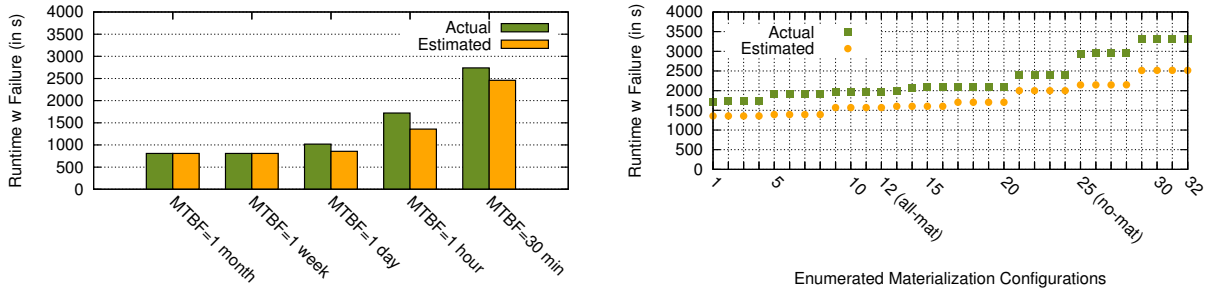
Figure 5.11.: Varying MTBF

degrades a more gracefully. However, it still has the second-highest overhead since sub-plans need to be restarted from scratch. The all-mat scheme behaves very similarly to the cost-based scheme for short- and long-running queries. The reason is that the total materialization costs of all operators (1-5 in Figure 5.9) represent only 34.13% of the total runtime costs. For long-running queries, the cost-base scheme materializes the intermediates 2 and 3. As a result, the cost-mat scheme has 63% less overhead than the all-mat scheme resulting from lower materialization costs and fewer attempts to finish the query under mid-query failures. For short-running queries, the overhead of all-mat is exactly 34% higher since the cost-based scheme does not materialize any intermediate.

Exp. 2b - Varying MTBF (Figure 5.11)

This experiment shows the overhead of the fault-tolerance schemes mentioned before when varying the MTBF. In this experiment, we executed TPC-H query 5 over $SF = 100$ using a low selectivity. This resulted in a query execution time of 905.33s (i.e., approx. 15 minutes) as a baseline-runtime when injecting no failures and adding no additional materializations in the plan. In order to show the overhead, we executed the same query using the following MTBFs per node: 1 week, 1 day, and 1 hour.

Figure 5.11 shows the overhead of the individual schemes under varying MTBFs. This figure shows the same trends as already reported before in Figure 5.10. The **cost-based scheme has the lowest overhead for all MTBFs** when compared to the other schemes using the same MTBF. Both not-mat schemes show a higher increase of the overhead under high failure rates (i.e., a low MTBF). The all-mat scheme again imposes unnecessary overhead for low failure rates and is the second-best for high failure rates since the materialization overhead for all operators of Q5 is only 30% of the query runtime.



(a) Varying MTBF

(b) Varying Mat. Confs.

Figure 5.12.: Accuracy of Cost Model

5.5.4 Accuracy and Robustness of Cost Model

In this experiment, we show the accuracy and robustness of our cost model: (a) For showing the accuracy of our cost model, we compare the actual runtime with the estimated runtime for different fault-tolerant plans (enumerated by our cost-based scheme) and for different MTBFs. (b) For showing the robustness of our cost model, we introduce errors in the statistics and analyze the effects on the plan selection.

Exp. 3a - Accuracy of Cost Model (Figure 5.12)

In this experiment, we executed TPC-H query 5 over $SF = 100$ using a low selectivity. This resulted in a query execution time of 905.33s (i.e., approx. 15 minutes) as a baseline-runtime when injecting no failures and adding no additional materializations in the plan. In order to cover a wide range of MTBFs, we added extreme MTBFs to cover a wide range from 30 minutes to 1 month (different from Experiment 1b in Section 5.5.4).

Figure 5.12(a) shows the accuracy results (i.e., actual vs. estimated runtime) for different MTBFs. While for high MTBFs (i.e., low failure rates), the error is 0%, we get an error of 30% for low MTBFs. In general, the cost model tends to underestimate the runtime when injecting failures. However, with an increasing estimated runtime, we see also an increase in the actual runtime. This behavior is crucial for a good cost model to select plans with minimal actual runtime.

Figure 5.12(b) shows the accuracy when enumerating different 2^5 materialization configurations for the plan of TPC-H query 5 shown in Figure 5.9 for a fixed MTBF of 1 hour. The x-axis shows the 2^5 enumerated materialization configurations sorted ascending by their estimated runtime. The y-axis shows the estimated/actual runtime for each of the enumerated plans. The plot shows that there is a **high correlation of the estimated and actual runtime** for all enumerated materialization configurations, which validates our cost model (i.e., a plan which has lower estimated cost also has a lower actual cost). As discussed before, this behavior is crucial for a good cost model to select plans with minimal actual runtime.

Exp. 3b - Robustness of Cost Model (Table 5.3)

In this experiment, we evaluate the sensitivity of our cost model to inaccurate statistics. We again use the plan of TPC-H query 5, as shown in Figure 5.9. The reason is that the runtime of this plan under mid-query failures strongly depends on the materialization configuration chosen by our cost-based scheme. As shown before in Figure 5.12(b), the runtime varies for all enumerated materialization configurations

Ranking w exact statistics	1	2	3	4	5
MTBF $\times 0.1$	3	4	1	2	5
MTBF $\times 0.5$	3	4	1	2	5
MTBF $\times 2$	8	7	5	6	2
MTBF $\times 10$	28	27	25	26	18
I/O costs $\times 0.1$	11	12	9	10	13
I/O costs $\times 0.5$	3	1	2	4	11
I/O costs $\times 2$	5	7	6	8	2
I/O costs $\times 10$	27	25	28	26	17
Compute & I/O costs $\times 0.1$	28	27	25	26	8
Compute & I/O costs $\times 0.5$	7	8	5	6	2
Compute & I/O costs $\times 2$	3	4	1	2	5
Compute & I/O costs $\times 10$	3	4	1	2	5

Table 5.3.: Robustness of the Cost Model

from 1358s to 2517s (for $SF = 100$ and an MTBF of 1 hour). To evaluate the sensitivity of our scheme, we vary the input statistics of our cost model and report how these changes affect the top-5 plans. When changing the I/O costs, we multiplied the materialization costs ($t_m(o)$) of each operator with a given perturbation factor, before applying our cost model. When changing all costs, we multiplied all operator costs ($t_r(o)$ and $t_m(o)$) with a given perturbation factor, before applying our cost model.

The baseline is the ranking of materialization configurations shown in Figure 5.12(b) that represents the case with exact statistics. Table 5.3 shows the results of this experiment. Each line in this table shows which materialization configuration of the baseline ranking moved to the top-5 positions when perturbing the statistics (i.e., the higher the number, the worse is the selected plan). In general, perturbations with small factors (i.e., $0.5\times$ and $2\times$) often **change the order within the top-5 materialization configurations only slightly**. This shows that our cost-based scheme is robust towards typical perturbations. In this case, our cost-based approach does not select the most optimal fault-tolerant plan, but it selects a fault-tolerant plan that is close to the optimal plan in terms of its runtime under mid-query failures as shown in Figure 5.12(b). However, for extreme perturbations (i.e., $0.1\times$ and $10\times$) our cost model is more sensitive. In the worst case, a materialization configuration that was on position 28 in the baseline ranking (out of 32) is placed on rank 1 after perturbation, resulting in a materialization configuration that has a $1.7\times$ higher runtime compared to the optimal materialization configuration. Moreover, perturbations in the I/O costs have a much stronger effect as perturbations of the other two categories. This is also clear since our cost-based scheme then favors configurations with fewer materializations when compared to the perfect ranking.

5.5.5 Effectiveness of Pruning Rules

In our final experiment, we show the effectiveness of our pruning rules presented in Section 5.4. Therefore, we enumerate all 1344 equivalent join orders of TPC-H query 5 (i.e., we do not enumerate plans with cartesian products) and apply our cost model with and without pruning rules enabled for $SF = 10$ and three different cluster setups with varying MTBFs: 1 week, 1 day, and 1 hour. We analyze the pruning efficiency for different cluster setups since the pruning rules 2 and 3 depend on the given MTBF.

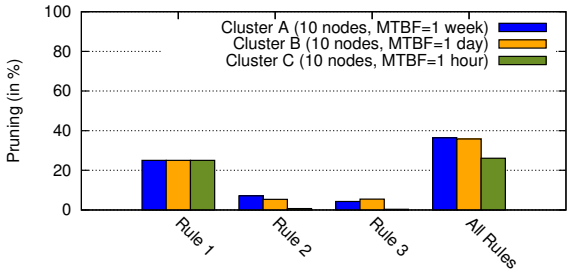


Figure 5.13.: Effectiveness of Pruning

Figure 5.13 shows the percentage of fault-tolerant plans that are pruned by the individual rules and the overall percentage of pruned fault-tolerant plans accumulated for all pruning rules 1-3. If we do not apply any pruning rule (no pruning), then 2^5 different materialization configurations need to be analyzed for each enumerated execution plan since the TPC-H query has 5 free operators. Thus, without pruning $43,008$ fault-tolerant plans must be enumerated when no pruning is activated. When activating all pruning rules, **in the best case 36% of the fault-tolerant plans can be pruned** for a MTBF of 1 week whereas in the worst case 26% can be pruned for a MTBF of 1 hour. In the following, we report the results when activating the pruning rules one after each other and explain the decreased effectiveness of the pruning rules for lower MTBFs.

The first rule (i.e., rule 1 in Section 5.4) is the most efficient rule and prunes constantly (i.e., independent of the MTBF) 25% of all fault-tolerant plans. The reason is that some of the join operators in TPC-H query 5 have a quite sizeable intermediate result (e.g., when joining `LINEITEM` and `SUPPLIER`). Thus, materializing the output of those joins is more expensive than running the subsequent operator, which means that we can set these join operators with large intermediate results to be not materialized.

The second rule (i.e., rule 2 in Section 5.4) is less efficient than rule 1 and prunes only $0.74\% - 7.15\%$ depending on the given MTBF. First, this rule generally prunes fewer materialization configurations since only the very last operators of a query tree (i.e., aggregation and projection) are typically short running and thus can be set to be not materialized. Moreover, for a higher MTBF, the probability of success of an operator increases even for longer running operators. In that case, more operators can be pruned (i.e., they are set to be not-materialized).

Compared to rules 1 and 2, rule 3 does not prune fault-tolerant plans eagerly before they are enumerated but it prunes them lazily during the enumeration of execution paths. More precisely, it prunes a fault-tolerant plan, once t finds an execution path Pt, which has a total runtime T_{Pt} longer than the best dominant path found so far. Thus, the efficiency of this pruning rule depends strongly on the enumeration order of execution paths. In this experiment, we count those fault-tolerant plans where this rule can be applied at all and at the end, regarding only half of the fault-tolerant plan as being pruned. The reason why we only count only the half of the fault-tolerant plans is, that pruning has two extreme cases: (1) the rule can be already applied for the first enumerated execution path (i.e., we skip all other execution paths of the same fault-tolerant plan), or (2) the rule is applied only for the very last enumerated execution path (i.e., we do not skip any other execution path of the same fault-tolerant plan). Thus, on average, half of the costs for analyzing the paths can be avoided by this rule.

Figure 5.13 shows that the pruning efficiency of rule 3 is also increasing for higher MTBF. The reason is that the best dominant path has a lower total runtime $bestT$ for an increasing MTBF. Thus, the pruning condition 1 of rule 3 (see Section 5.4.3), which compares the runtime R_{Pt} of a path Pt without mid-query failures to the memoized best dominant path, holds more often.

5.6 Related Work

Parallel databases [7, 12, 13, 14] typically can handle various types of failures. One of the main issues discussed in the literature is to maintain consistency in transactional workloads despite failures [87]. Moreover, another well-studied field in parallel databases is how to achieve high-availability of the database by replicating partitions to multiple nodes in a cluster [58]. When introducing replication in transactional workloads, modern parallel databases often relax consistency (e.g., by using eventual consistency) in favor of better supporting availability or network partitioning [31] in order to be able to scale in large clusters. However, to the best of our knowledge, there is no published work on how parallel databases handle mid-query failures in analytical workloads. The first approach in most parallel databases is to restart a query once a mid-query failure happens. However, this scheme is not efficient when running on large clusters of commodity machines or IaaS offerings such as Amazon's Spot Instances, where a mid-query failure is a much more common case. This requires a more fine-granular fault-tolerance scheme, which checkpoints intermediate results. Fine-granular fault-tolerance scheme are typically found in modern PDEs (such as Hadoop [8], Shark[115], Dryad [62]) as well as in many stream processing engines [61, 108]. While stream processing engines checkpoint the internal state of each operator for recovering continuous queries, MapReduce-based systems [35] such as Hadoop [8] typically materialize the output of each operator to handle mid-query failures. For being able to recover, they rely on the fact that the intermediate result is persistent even when a node in the cluster fails, which requires expensive replication and prevents support for latency-sensitive queries. Other systems, like Impala [9] and Shark [115], store their intermediates in main-memory in order to better support short running latency-sensitive queries. Moreover, Shark uses the idea of resilient distributed datasets [120], which store their lineage in order to enable re-computation instead of replicating intermediates. In contrast, in Dryad [62], operators can be configured to either pipeline their output to the next operator(s) or materialize the intermediate results. However, in Dryad, this must be configured manually; i.e., no optimizer decides which results should be materialized and which should be pipelined for an efficient execution under failures.

Two more recent approaches, which tackle the same problem as discussed in this chapter, are FTOps [111] and Osprey [116]. FTOps also presents an optimizer to find the best fault-tolerance strategy for each operator of a given plan. However, compared to our approach, FTOps only supports tree-structured plans and not DAG-structured plans. Another difference is that FTOps only supports plans with aggregations at the top, while we support arbitrary positions of aggregation operators in the plan. Moreover, FTOps runs as a post-processing step and uses a brute-force enumeration technique to analyze different fault-tolerance strategies for input plan while our approach analyses the top-k plans and efficiently prunes the search space of different fault-tolerant plans. Compared to FTOps, Osprey applies MapReduce-style fault- tolerance to parallel databases: Osprey splits analytical queries over a star schema into multiple sub-queries over individual partitions and executes a final merge of all sub-queries. If a sub-query fails it is re-started on a different replica. However, Osprey does not provide a cost-based optimizer to select an optimal fault-tolerant plan that includes additional materializations.

5.7 Conclusions and Outlook

In this chapter, we presented our novel cost-based fault-tolerance scheme for parallel data processing. Compared to the existing strategies which either materialize all intermediates or no intermediates, our scheme selects an optimal subset of intermediates to be materialized such that the query runtime is minimized under the presence of mid-query failures. Our experiments show, that our scheme is efficient for different queries as well different cluster setups whereas the existing schemes only have their sweet-spot for certain cluster setups and query workloads.

One primary avenue of future work is to integrate other fault-tolerance strategies (e.g., check-pointing of the operator state to also support mid-operator failures) into our cost-based fault-tolerance scheme. This could be helpful, especially for long-running operators, which otherwise are likely to fail often. Moreover, we also want to look into the more dynamic decision for cases where data is skewed or statistics are hard to estimate (e.g., for user-defined functions).

6 Spotgres: Parallel Data Analytics on Spot Instances

6.1 Introduction

Motivation

Market-based IaaS offers such as Amazon's EC2 Spot Instances represent a cost-efficient way to operate a cluster. Compared to On-demand and Reserved Instances, which follow a fixed pricing scheme, the per hour price of Spot instances changes dynamically, whereas the Spot price is often significantly less when compared to On-demand and even the Reserved Instances. In order to rent one or several Spot instances, the user places a bid, which includes the number of instances of one particular machine type and the maximal per hour price that the user is willing to pay (called bid price). If the bid price exceeds the Spot price for the given machine type, Amazon will launch the instance(s) for the user. In that case, Amazon charges the user the current Spot price and not the bid price. Moreover, Spot Instances are billed in the granularity of an hour (using the Spot price of the last full hour if an instance is running longer than an hour). However, if the Spot price exceeds the bid price, Amazon could and will most likely kill all instance(s) without signaling the user beforehand, whereas Amazon does not charge for the last non-complete hour of uptime. Consequently, the availability of Spot Instances and the actual cost of the complete cluster are directly linked to the bidding strategy (i.e., the cost the user is willing to spend per hour per node) and the actual price.

Figure 6.1 shows the Spot price history in the AWS management console for an EC2 machine of type $c1.medium$ for one availability region (eu-west-1c) over one month. Compared to the on-demand price of $c1.medium$ machines, which is approx. 13 cent per hour in that availability region, the Spot price is significantly less in most of the time-varying between 1 and 3 cent per hour. However, we can also see some individual spikes, which range up to 50 cent per hour. Thus, if the user follows a strategy to place a bid which is only a little above the current Spot price (e.g.4 cent per hour), she minimizes the risk of paying a high per hour price, but the user will (most likely) lose her instances when spikes occur. On the other hand, if the user bids a high price (e.g., above the highest spike), she minimizes the risk of losing an instance but increases the risk of paying a per hour price, which reflects one of the spikes.

Parallel Data-Processing Engines (PDEs) such as parallel databases (e.g., SAP HANA [26], Greenplum [112] or Terradata [87]) or other modern parallel data management platforms (e.g., Hadoop [114], Scope [122], Shark [115]) are used today to analyze large amounts of data in clusters of shared-nothing machines. When deploying a Parallel Data-Processing Engine (PDE) on a cluster of Spot Instances, a significant obstacle is to find a bidding strategy that is optimal for a given workload and satisfies user constraints such as the maximal budget. Currently, there is no support from Amazon to create a set of bids for a cluster of Spot Instances. Thus, the user has to manually find the best set of bids for her constraints and continually needs to adapt her set of bids (i.e., re-bid) on Spot Instances if individual instances fail.

Moreover, another obstacle is that existing PDEs implement rigid fault-tolerance schemes to deal with mid-query failures. Whereas traditional parallel databases restart a query from scratch (on a replica) if a mid-query failure occurs, modern PDEs materialize intermediates to implement a more fine-grained scheme which restarts a sub-query from the last checkpoint. However, no PDE adaptively adjusts its execution strategy to a varying mean time between failures (MTBF) and other characteristics of the workload (e.g., query runtime and materialization costs for intermediates).

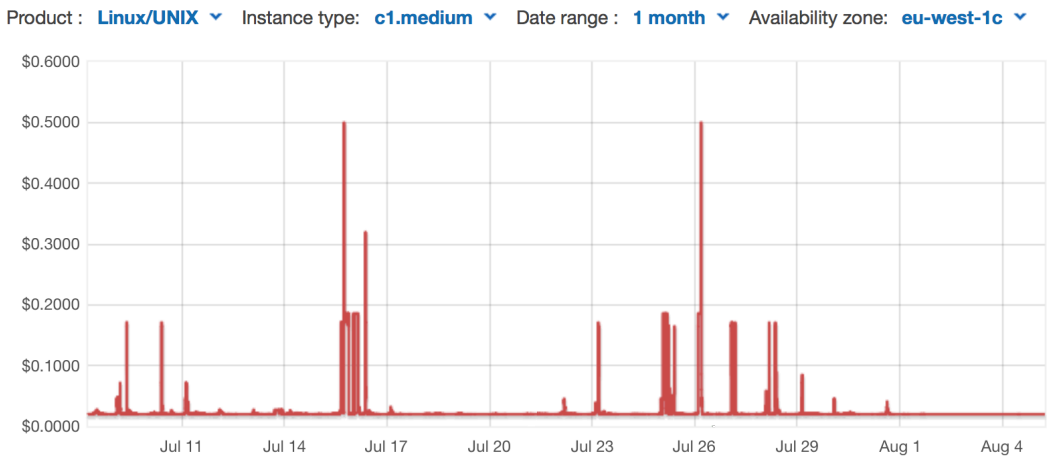


Figure 6.1.: Spot Price History (1 Month)

Contributions

In this chapter, we represent a novel PDE called *Spotgres* that tackles the before mentioned issues. *Spotgres* extends a typical PDE architecture by (1) a constraint-based bid advisor that finds an optimal cluster configuration (i.e., a set of bids) based on a given set of constraints and (2) a cost-based fault-tolerance scheme that takes various parameters (such as MTBF and query statistics) into account to efficiently execute analytical queries in parallel over the set of launched Spot Instances.

The constraint-based bid advisor of *Spotgres* supports the user to specify constraints on the resources in the cluster as well as a quality constraint on the maximal budget or the minimal availability. On the one hand, if the maximal budget is given as a constraint, the bid advisor will propose a set of bids that aims to maximize the availability. On the other hand, if the minimal availability per node is given as a constraint, the bid advisor will propose a set of bids that aims to minimize the total costs per hour for the given availability.

The cost-based fault-tolerance scheme of *Spotgres* implements an efficient execution strategy for the set of launched Spot Instances. Therefore, *Spotgres* regularly collects statistics (such as the MTBF) for the individual machine types in the cluster and finds the best materialization strategy for a given query that minimizes the runtime under mid-query failures.

Our experiments show that the bid advisor effectively meets the given constraints. For example, the bid advisor suggests a cluster of Spot Instances that has an availability of 98% for only 25% of the costs of the cheapest On-demand cluster. Moreover, our cost-based fault-tolerance scheme shows that even with high failure rates (e.g., low MTBFs) queries can be executed more efficiently compared to other existing fault-tolerance schemes.

Outline

In Section 6.2, we first present the architecture of *Spotgres* and discuss how the bid advisor and the cost-based fault-tolerance scheme are integrated into a typical PDE architecture. In Section 6.3, we discuss the details of the bid advisor, which uses constraint programming to solve the optimization problem discussed above. Moreover, in Section 6.4 we present the details of our cost-based fault-tolerance scheme. Finally, Section 6.5 shows our experimental evaluation and Section 6.6 summarizes related work.

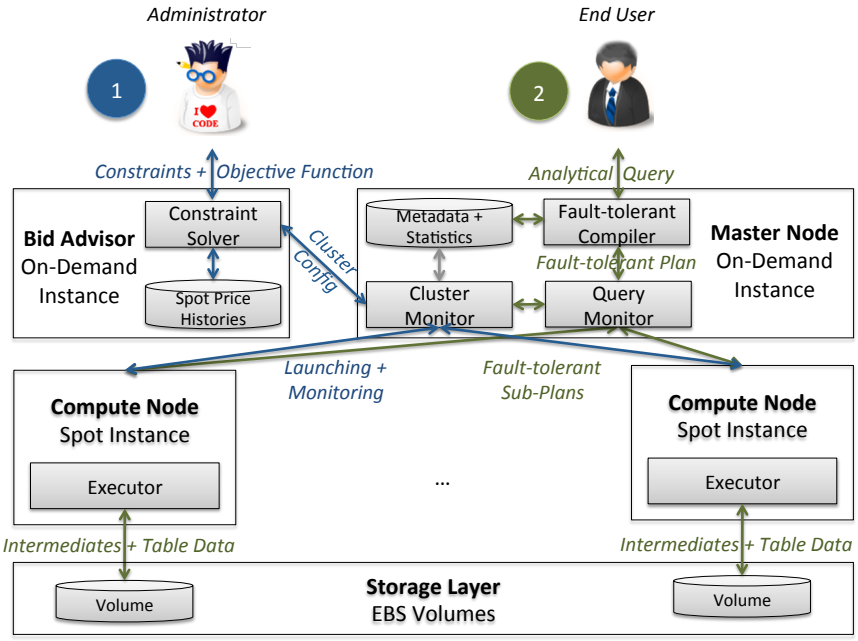


Figure 6.2.: *Spotgres* Architecture

6.2 Spotgres Architecture

Figure 6.2 shows the architecture of our *Spotgres* PDE along two important aspects: (1) the cluster configuration using constraints and (2) the cost-based fault-tolerant query execution. In Figure 6.2, the blue arrows represent the control-flow of the cluster configuration, whereas the green arrows represent the control-flow of the query execution. In the following, we explain the details of the individual components.

Bid Advisor: The most critical component for the cluster configuration is the bid advisor which takes a set of input constraints from the administrator (e.g., resource constraints) and derives an optimal cluster configuration represented as a set of bids on Spot Instances (whereas each bid includes the machine type, the number of instances, and the bid price). Moreover, the administrator must define an objective function to derive a cluster configuration. Currently, *Spotgres* supports only two objective functions: either maximizing the availability of each cluster node for a given total budget or minimizing the cost for the complete cluster for a given availability per node. The input constraints and the objective function are translated into a constraint program by the *Constraint solver*, which then finds an optimal set of bids. Therefore the solver uses the Spot price histories (of a given window) to solve the optimization problem. The suggested cluster configuration is used by the cluster monitor (which is a part of the master node) to launch the Spot Instances.

Master Node: The master node hosts components for compiling and executing queries as well as monitoring compute nodes (which execute sub-queries). The *Fault-tolerant Compiler* takes a query from the end-user and finds a plan that materializes a subset of intermediate results such that the runtime of the query is minimized under mid-query failures. For selecting a subset of intermediate results that should be materialized, *Spotgres* implements a cost model that uses query statistics (i.e., histograms of the data) and cluster statistics in order to find an optimal fault-tolerant plan that minimizes the runtime under mid-query failures. For query execution, the *Query Monitor* gets the fault-tolerant plan and splits it into sub-plans that are executed by the compute nodes in parallel over different partitions and materializes their output to

the storage layer (EBS in our case) for fault-tolerance. For monitoring the complete cluster, the *Cluster Monitor* pings all individual compute nodes. In case of a node failure, *Cluster Monitor* uses the bid advisor to start new machines, i.e., to re-bid for failed Spot Instances. Moreover, the cluster monitor also updates the cluster statistics such as mean time between failures (MTBF) that are used by the fault-tolerant query compiler to select the subset of intermediates that should be materialized.

Since the *Master Node* and the *Bid Advisor* are central components in *Spotgres*, they are deployed on an On-Demand Instance such that we do not need to deal with fault-tolerance in these nodes at all.

Compute Nodes: The compute nodes are executing sub-queries (i.e., sub-plans) over individual partitions and materializing the result of the sub-plans to the attached EBS volumes of the storage layer. All compute nodes are deployed on Spot Instances which means that these nodes can fail during query execution.

Storage Layer: *Spotgres* horizontally partitions a database D into n partitions P_i with $i = 1...n$ (i.e., $D = [P_1, ..., P_n]$ and replicates individual partitions $P_i \in D$ to k different nodes for fault-tolerance. Each copy of a partition is stored in a separate EBS volume. That way, if a node in the cluster fails, the cluster monitor can reattach the EBS volume(s) of the failed instance to a new (or an already running) instance. Afterward, a failed sub-plan can be restarted from the most recent intermediate result that was materialized to an EBS volume.

6.3 Constrained-based Bid Advisor

6.3.1 Overview

In this section, we give an overview of the bid advisor, which suggests a cluster configuration (i.e., a set of bids) based on the given input constraints. The input constraints of the user are shown in the following table:

Input	Description	Constr.Type
CU	Minimal number of Amazon Compute Units in the cluster	Resource
RAM_{cu}	Minimal RAM per CU in the cluster	Resource
B	Maximal budget for the complete cluster	Quality
A_{cu}	Availability per CU	Quality

The first two constraints are called *Resource* constraints, which define functional aspects of the cluster: i.e., the computational power of the complete cluster in compute units CU and the minimal main memory per CU. In order to specify the computational power per machine type, Amazon provides an abstract measure called *Compute Unit CU* (i.e., for each machine type the number of CUs is defined). That way, a CU can be seen as a virtual processor that provides a fixed computational power. Thus, in order to define the computational power of the complete cluster, the user gives the minimal number of CUs in the cluster

as an input constraint. The second input constraint is the minimal RAM per CU denoted as RAM_{cu}. These two constraints heavily depend on the expected workload.

The other two constraints in the table before are called *Quality* constraints, which define non-functional aspects of the cluster: i.e., the maximal budget B (i.e., an upper bound for the actual cluster cost) and the minimal availability per compute unit A_{cu}. Out of these two constraints, the user must select one, whereas the other variable is used as the objective function: i.e., if the maximal budget B is provided as an input constraint, the objective function is to maximize the availability A_{cu}, whereas if the availability A_{cu} per CU is provided as an input constraint, minimizing the maximal budget B is the objective function. In the following, we discuss both cases in detail.

6.3.2 Minimizing Cluster Cost

In the first case, the user provides minimal availability per compute unit A_{cu} as a quality constraint as well as the two resource constraints: CU and RAM_{cu}. Afterwards, the bid advisor executes the following steps: (1) filter all machine types $T = [t_1, \ldots, t_n]$ that qualify for the given RAM_{cu} constraint, (2) derive the bid prices $P = [p_1, \ldots, p_n]$ per machine type $t_i \in T$ to satisfy the given A_{cu} and calculate the average billed prices $A = [a_1, \ldots, a_n]$ per machine type $t_i \in T$, and finally (3) find the optimal cluster configuration using the constraint solver.

The first step is trivial since Amazon provides the number of $CU = [cu_1, \ldots, cu_n]$s and the RAM per machine type t_i. The result is the vector $T = [t_1, \ldots, t_n]$ of machine types that qualify for the given RAM_{cu} constraint. For the second step, the Spot price history per machine type $t_i \in T$ is used. Figure 6.3 shows the price history for one machine type over the last month (whereas the window we use for learning the bid price is configurable). Based on the history and the given A_{cu}, we can derive a bid price b_i that will (based on the history) give us the desired availability for that machine type. For example, in Figure 6.3 we set $b_i \geq \$0.03$ to guarantee an availability of $A_{cu} = 98\%$, whereas if we set $b_i \geq \$0.17$ we can achieve an availability of $A_{cu} = 99\%$. Moreover, we also use the Spot price history to calculate the average prices $A = [a_1, \ldots, a_n]$ per machine type $t_i \in T$ for the learning windows by dividing the history into intervals which have the same Spot price. The average billed prices instead of the bid prices are used to calculate the actual costs of a cluster configuration.

In order to find the optimal cluster configuration $C = [b_1, \ldots, b_n]$ which is a vector of bids (one for each machine type), *Spotgres* evaluates the constraint program as shown in Listing 6.1. A bid $b_i = [x_i, p_i]$ on a machine type i is a tuple which defines the number of instances x_i and the bid price p_i. The constraint program in step (3) returns the vector $X = [x_1, \ldots, x_n]$, whereas the bid prices $P = [p_1, \ldots, p_n]$ for each machine type are returned already by step (2).

In the following, we explain the details of the constraint program shown in Listing 6.1. The objective function obj calculates the total cluster costs based on the average prices per machine type. The variable b_i is a special variable which is set to $b_i = 1$ in order to indicate that machine type i must be included in the cluster configuration by adding the optional constraint $activeInstance\_i$ which is needed for re-bidding if some nodes in the cluster do not fail (see Section 6.3.4). The first constraint (i.e., *cuConstraint*) defines that there must be in total at least CU compute units in the cluster where cu_i is the number of compute units for machine type i. The other constraints (i.e., $lowerBound\_i$ and $upperBound\_i$) represent bounds on the numbers of instances of machine type i. These constraints can be used e.g., for re-bidding to make sure that instances that did not fail must be included in the new cluster configuration again.

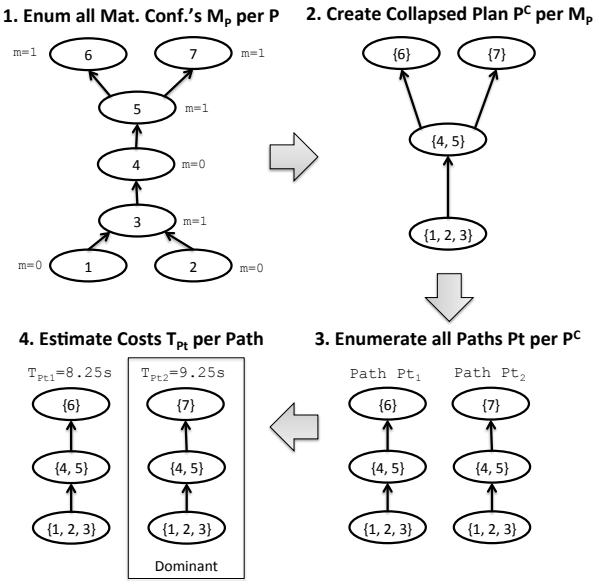


Figure 6.3.: Derive the Bid Price for a Machine Type

6.3.3 Maximizing Availability

Compared to the first case (in Section 6.3.2), where we minimize the total cluster costs, the second case has a maximal budget as an input constraint, and the objective function is to maximize the availability per compute unit A_{cu}. Since there is no simple function that maps the average cost per machine type to its actual availability, we apply a binary search over the procedure shown in Section 6.3.2. Thus, we start with $A_{cu} = 50\%$, and depending on the total cluster costs, we either increase or decrease the availability to $A_{cu} = 75\%$ or $A_{cu} = 25\%$, etc. until we find the highest value for A_{cu} that still satisfies the maximal budget B.

6.3.4 Optimizations and Variants

Diversity Optimization

One variant of the procedure in Listing 6.1 is that we add a diversity constraint as shown below:

$$\mathtt{diffTypesConstraint:}\ \ b_1 + \cdots + b_n = k$$

This constraint can be used to force the solver to choose k different machine types for the cluster configuration. Together with the *lowerBoundi* and *upperBoundi* constraint, the total number for each of the k machine types can be defined. The diversity constraint can be used for *Spotgres* cluster configurations where partitions should be replicated. The reason is that typically all instances of the same machine type fail at the same time when using Spot Instances (i.e., when the Spot price exceeds the bid price). Thus, replication only helps if data is copied to different machine types or the same machine type with a higher bid price. Currently, setting $k \geq 1$ only returns k different machine types in the cluster configuration. Returning the same machine type with a higher bid price is an avenue for future work.

Listing 6.1: Constraint Program

```
Minimize:
 obj: a_1·x_1 + ... + a_n·x_n + 0·b_1 + ... + 0·b_n

Subject To:
 cuConstraint: cu_1·x_1 + ... + cu_n·x_n >= CU
 lowerBound_1: r_1·b_1 − x_1 <= 0
 upperBound_1: MinInt·b_1 + x_1 <= 0
 ...
 lowerBound_n: r_n·b_n − x_n <= 0
 upperBound_n: MinInt·b_n + x_n <= 0

 (activeInstance_1:  b_1 = 1
 ...
 activeInstance_n:  b_n = 1)
```

Uptime Optimization

Another variant is an optimization that leverages the fact that a user is not charged for instances that have an uptime of less than an hour. Therefore, we use the cluster configuration we get from one of the procedures described in Section 6.3.2 and 6.3.3 and add/subtract a cost-tolerance to/from the given total cluster costs. We use that cost tolerance as a new constraint for the constraint program shown in Listing 6.1. Moreover, we change the objective function to maximize the number of intervals where the uptime is less than an hour. Thus, the main idea is to reduce the total cluster cost by adding cost tolerance (i.e., in the worst case, we are willing to pay a little more).

6.4 Cost-based Fault-tolerance

In this section, we present a high-level overview of our cost-based fault-tolerance scheme. Compared to the existing fault-tolerance schemes, our scheme selects a subset of intermediates (called materialization configuration) to be materialized such that the query runtime is minimized under the presence of mid-query failures taking the mean time between failures (MTBF) per cluster node into account since the MTBF can strongly vary over time.

The main goal of our cost-based fault-tolerance scheme is for a given query to find an execution plan P and a materialization configuration M_P that minimizes the total runtime of that query under mid-query failures. Moreover, our cost-based fault-tolerance scheme is a fine-grained strategy which restarts only sub-plans on nodes that failed.

In order to find an optimal plan, cost-based optimizers typically enumerate different equivalent execution plans P and apply a cost function to find the best plan with the minimal runtime cost. However, they neither enumerate different materialization configurations for each plan, nor do they consider the costs for recovering from mid-query failures. Therefore, we propose to change the cost-based optimizer to use an enumeration procedure that finds the best combination of a plan P and a materialization configuration M_P with minimal runtime under mid-query failures. We call this combination $[P, M_P]$ a fault-tolerant plan.

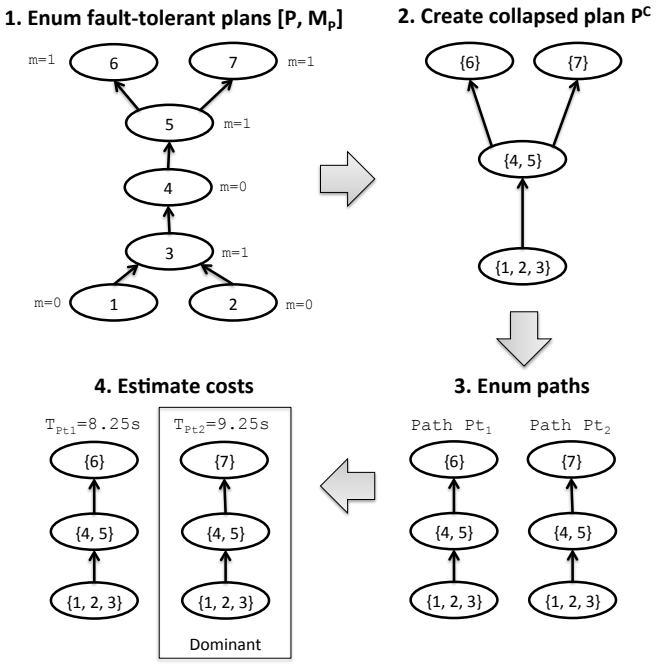


Figure 6.4.: Finding the best Fault-tolerant Plan

In the following, we give a high-level description of our procedure $findBestFTPlan(Q)$, which finds the best fault-tolerant plan for a given query (see Figure 6.4): (1) First, our procedure enumerates different fault-tolerant plans $[P, M_P]$ for the given query Q. (2) Second, for each enumerated fault-tolerant plan, our procedure creates a collapsed plan P^c where all operators in M_P that do not materialize their output are collapsed into the next materializing operator. The idea of the collapsed plan is to represent the granularity of re-execution using these collapsed operators (i.e., if a collapsed operator has finished successfully, it does not need to be re-executed again). (3) Third, for a collapsed plan, all execution paths (i.e., paths from each source to each sink in P^c) are enumerated and (4) the total cost T_{Pt} is estimated for each execution path Pt using a cost function that takes statistics about the operators and the cluster (e.g., the mean time between failures) into account. The path Pt with the maximal estimated total cost for a given materialization configuration (called the dominant path of that configuration) is selected. The total cost of the dominant path represents the total runtime of the fault-tolerant plan. In Figure 6.4, for example, path Pt_2 is the dominant path and thus the estimated runtime for the given fault-tolerant plan $[P, M_P]$ would be $9.25s$. To that end, our procedure finds an execution plan P and a corresponding materialization configuration M_P, which has minimal total costs under the presence of mid-query failures.

In order to estimate the total cost of a path Pt under mid-query failures, the cost function $estimateCost$ requires that the following statistics are given: The runtime costs $t_r(o)$ to execute each operator $o \in P$ and the costs $t_m(o)$ to materialize the output of each operator $o \in P$. Both cost values can be derived from cardinality estimates that are calculated by a cost-based optimizer. Moreover, other parameters that are needed for the cost estimation are the following cluster statistics: the mean time between failures (MTBF) and the mean time to repair (MTTR) for one cluster node, as well as the cluster size (i.e., the number of nodes). In this chapter, we assume that all these parameters are given.

6.5 Experimental Evaluation

In this section, we report the results of our experimental evaluation which shows (1) the effectiveness of our bid advisor for different input constraints and different objective functions (Section 6.5.1), and (2) the effectiveness of our cost-based fault-tolerance scheme by comparing the runtime of queries over the TPC-H schema for different cluster configurations (Section 6.5.2).

In both experiments, we use the Spot price history of the last 12 months ranging from 1st of September 2013 until the 8th of August 2014 for all 18 machine types where the price history was available in that time frame. Moreover, we use the following resource constraints as an input to the bid advisor: $CU \geq 10$ and $RAM_{cu} \geq 1.5GB$. Adding more CUs would not change the results of the bid advisor since this does not affect the optimization problem.

6.5.1 Effectiveness of Bid Advisor

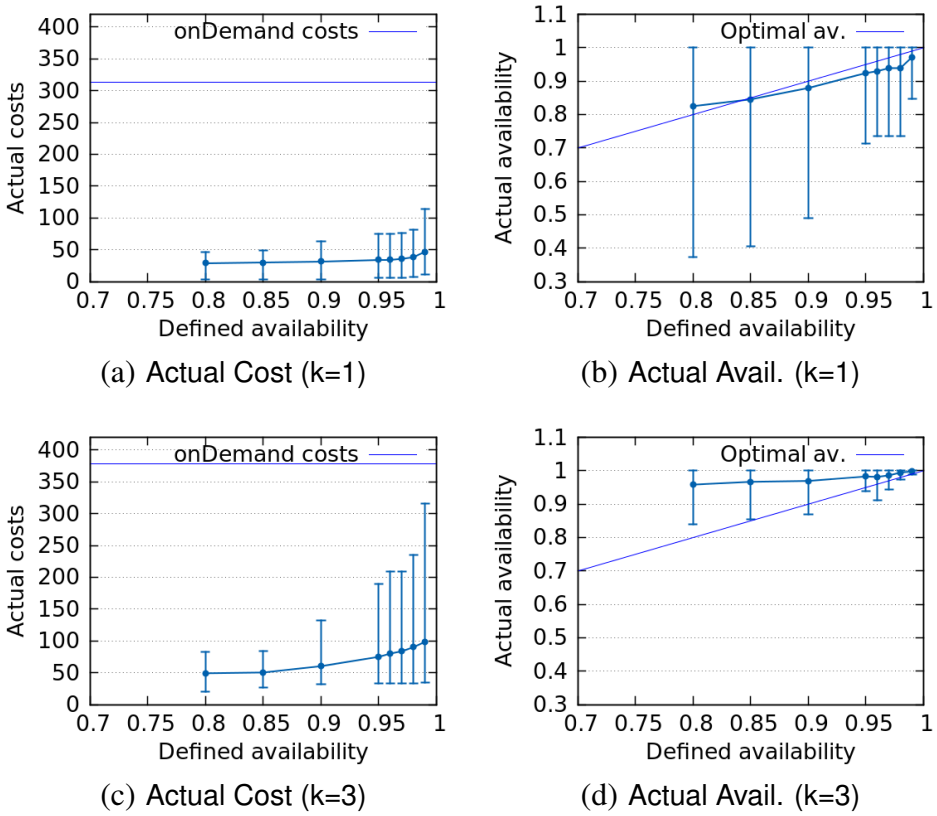
In this experiment, we show the effectiveness of our bid advisor for two scenarios: Exp. 1a minimizes cluster costs for a given availability (Figure 6.5) and Exp. 1b maximizes the availability for a given budget (Figure 6.6). For finding an optimal cluster configuration, we use one month of the price history for learning and use the subsequent month for testing. We repeat each experiment multiple times by shifting the learning and testing window forward for one week until the testing window reaches the end of the Spot price history (mentioned before).

For Exp. 1a, we use the given availabilities of 0.8, 0.85, 0.9, and from 0.95 to 0.99 every 1% step. For Exp. 1b, we use the given prices of 1%, 5%, 25%, 40%, 70%, 90%, and 100% of the cheapest On-demand cluster that satisfies the given resource constraints (i.e., $CU \geq 10$ and $RAM_{cu} \geq 1.5GB$). Moreover, we execute each experiment (1a and 1b) using the diversity variant of our algorithm mentioned in Section 6.3.4; using the diversities of $k = 1$ and $k = 3$ (i.e., a database without replication and one with replication factor 3). Moreover, for $k = 3$ we repeat Exp. 1a with the uptime optimization enabled to analyze for potential cost savings (Figure 6.7).

Exp. 1a - Minimizing Costs (Figure 6.5)

Figure 6.5(a) and 6.5(b) compare the defined availability (i.e., the input constraint) and the actual availability per node as well as the defined availability and the actual total costs of the cluster (which was the objective function in this experiment) for $k = 1$. Both figures show the average as well as the lower and upper quartile for the Spot cluster. For the actual total costs of the Spot cluster that we minimize in this experiment, we see that the average costs are below $50 (compared to more than $300 for the cheapest On-demand cluster). For the actual availability of the Spot cluster, we see that the average is very close to the defined availability (i.e., the input constraint). However, the lower and upper quartile show a high interquartile range (IQR).

Figure 6.5(c) and 6.5(d) show the results for a diversity of $k = 3$. The positive effect of $k = 3$ is the average actual availability gains when comparing to the average actual availability of $k = 1$. The average of the actual availability is always higher than the defined availability and has a much smaller IQR. However, the average total costs for the Spot Cluster and the cheapest On-demand cluster are rising both about $50. This is clear since, for $k = 3$, the bid advisor must pick 3 different machine types and can



(a) Actual Cost (k=1) (b) Actual Avail. (k=1)

(c) Actual Cost (k=3) (d) Actual Avail. (k=3)

Figure 6.5.: Exp. 1a - Minimizing Cluster Costs (Avgs and Quartiles)

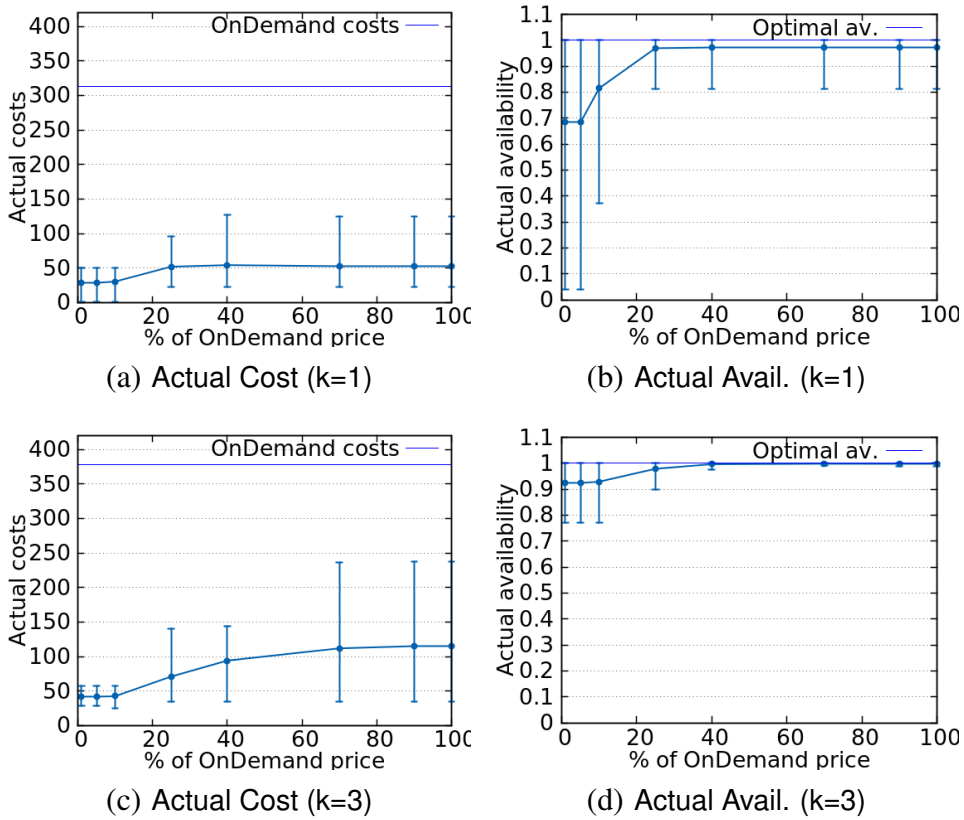
not compose the complete cluster only using the cheapest machine type. In the extreme case (i.e., for a defined availability of $A_{cu} = 0.99$) the total costs thus almost double for the Spot cluster (i.e., \$100 for $k = 3$ instead of \$50 for $k = 1$).

Exp. 1b - Maximizing Availability (Figure 6.6)

In this experiment, we set the maximal budget of the user for the entire cluster as a constraint. Moreover, we analyze the actual costs of the Spot cluster and the actual availability of each node (which is the objective function in this experiment). If we set the budget to 25% of the cheapest On-demand cluster, we can see that we already get an actual average availability per node of more than 0.96 for $k = 1$ (see Figure 6.6(b)) and even 0.99 for $k = 3$ (see Figure 6.6(d)). Moreover, the actual costs do not rise higher than 20% (i.e., \$50) of the cheapest On-demand cluster for $k = 1$ respectively 30% (i.e., \$120) for $k = 3$ on average even if we set the maximal budget to 100% of the cheapest On-demand cluster. The reason is that the Spot price is typically much lower than the bid price if we set the maximal budget to 100% of the cheapest On-demand cluster. For the IQR, we see similar effects as for Exp. 1a.

Exp. 1c - Uptime Optimization (Figure 6.7)

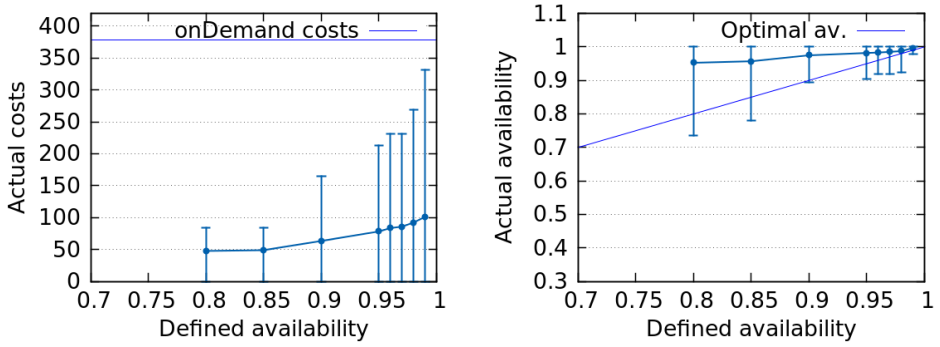
Finally, in the last experiment we re-execute Exp. 1a with the uptime optimization enabled for k=3 (whereas $k = 1$ shows similar results). The results are shown in Figure 6.7. Compared to Figure 6.5(c) and 6.5(d) (i.e., the same experiment without the uptime optimization), the most important difference is that the lower quartile of the actual costs in this experiment is at \$0 (i.e., in the best case we do not pay anything for our cluster).



(a) Actual Cost (k=1) (b) Actual Avail. (k=1)

(c) Actual Cost (k=3) (d) Actual Avail. (k=3)

Figure 6.6.: Exp. 1b - Maximizing Availability (Avgs and Quartiles)



(a) Actual Cost (k=3) (b) Actual Avail. (k=3)

Figure 6.7.: Exp. 1a with Uptime Optimization (Avgs and Quartiles)

In this experiment, we compare the overhead of different existing fault-tolerance schemes to our cost-based scheme when mid-query failures happen while executing queries. The reported overhead in this experiment represents the ratio of the runtime of a query under a given fault-tolerance scheme (i.e., including the additional materialization costs and recovery costs) over the baseline execution time. The baseline execution time for all schemes is the pure query runtime without additional costs (i.e., no extra materialization costs and no recovery costs due do mid-query failures). Thus, if we report that a scheme has 50% overhead, it means that the query execution under mid-query failures using that scheme took 50% more time than the baseline.

The fault-tolerance schemes, which we compared in this experiment, are:

- **all-mat:** This represents the strategy of Hadoop, where all intermediates are materialized to a fault-tolerant storage medium. Moreover, for recovering a fine-grained strategy is used (i.e., only sub-plans that fail are restarted).
- **n**o-mat (lineage): This represents the strategy of Shark, where intermediates are not used to recover. Moreover, lineage information is used to re-compute failed sub-plans.
- **n**o-mat (restart): This represents the strategy of a parallel database, where intermediates are not used to recover. Moreover, for recovering a coarse-grained strategy is used (i.e., the complete query plan is restarted once a sub-plan fails).
- **c**ost-based: This represents our cost-based strategy that materializes intermediates based on a cost-model. Moreover, for recovering a fine-grained strategy is used (i.e., only sub-plans that fail are restarted).

In the following, we report the overhead of all these strategies: (a) when running queries with varying runtime for a fixed MTBF to show the effect of the fault-tolerance strategies short- and long-running queries and (b) when running the same query for different MTBFs to show the effect of different cluster setups. For measuring the actual runtime of queries under mid-query failures, we recorded 10 traces to inject failure using a Poisson distribution per machine type with a given MTBF and used the same set of traces for all fault-tolerance schemes. Moreover, failures for nodes of the same machine type were correlated (i.e., all nodes of the same type failed together). In all experiments, we used a diversity constraint of $k = 3$. We used this method for all other experiments in this chapter to inject failures.

Exp. 2a - Varying Query Runtime (Figure 6.9)

In this experiment, we executed TPC-H query 5 over different scaling factors using parameters. This resulted in query execution times ranging from a few seconds up to multiple hours. We selected TPC-H query 5 in this experiment since this is a typical analytical query with multiple join operations and an aggregation operator on top (see Figure 6.8). We also used other queries of the TPC-H benchmark, but they showed similar results when varying their runtime.

For this experiment, the output of every join operator was defined to be a free operator (marked with the numbers 1-5 in Figure 6.8) and thus could be selected by our cost-model to be materialized. Thus, for each enumerated plan, our procedure enumerated 2^5 materialization configurations when pruning was deactivated. Moreover, we injected mid-query failures using a MTBF of 1 day per node.

The result of this experiment is shown in Figure 6.9(a). The x-axis shows the baseline-runtime of the query (i.e., when no failure is happening), and the y-axis shows the overhead as discussed before. The

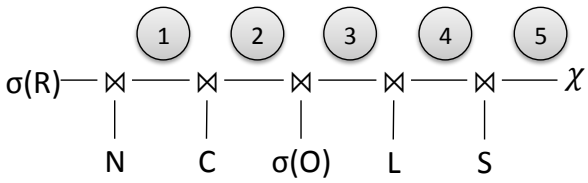
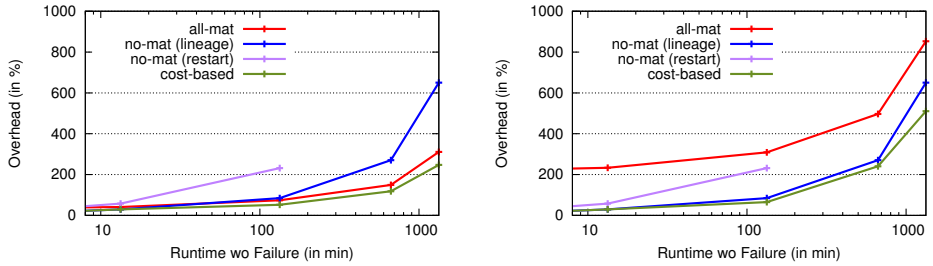


Figure 6.8.: TPC-H query 5 (Free operators 1-5)



(a) Normal I/O Cost (b) High I/O Cost

Figure 6.9.: Varying Runtime

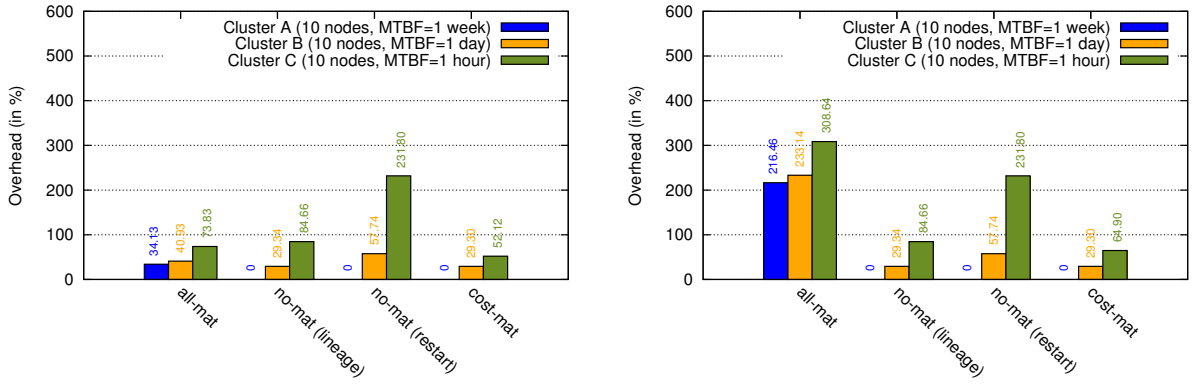
cost-based scheme has the lowest overhead for all queries, starting with 0% for short-running queries and ending with 247% for long-running queries. Thus, our scheme effectively finds the best materialization configuration for different queries of different lengths. Compared to our cost-based scheme, the other schemes impose a higher overhead depending on the query runtime. Both no-mat schemes pay 0% overhead for short-running queries. However, for queries with a higher runtime, the overhead increases for both no-mat schemes, whereas the restart-based scheme shows a much stronger increase. In fact, queries with a runtime higher than $7,820$s never finished in our experiment. The all-mat scheme pays 30% overhead for short-running queries, which is pure materialization overhead (i.e., no failure occurred). However, for a higher baseline-runtime, the overhead of this scheme does not increase as strong as for the no-mat schemes but still is 30% higher compared to the overhead of the cost-based scheme.

Figure 6.9(b) shows the same experiment as before with increased I/O costs (i.e., materialization costs were approx. $7\times$ higher than those used in Figure 6.9(a))[1] The rationale of this experiment is to show the effect of slower external storage systems (such as HDFS) on the overhead of all schemes. As before, the cost-based scheme has the lowest overhead for all queries (i.e., from short- to long-running queries). However, the overhead for queries with a higher runtime is increasing stronger compared to Figure 6.9(a) since fewer intermediates are materialized, and thus higher recovery costs need to be paid. Moreover, another difference is that the all-mat scheme has to pay the higher materialization overhead for all queries making this scheme unattractive in this setting. Finally, the two no-mat schemes (lineage and restart) are not affected by the increased I/O costs since they do not materialize additional intermediates.

Exp. 2b - Varying MTBF (Figure 6.10)

This experiment shows the overhead of the fault-tolerance schemes mentioned before when varying the MTBF. In this experiment, we executed TPC-H query 5 over $SF = 100$ using a low selectivity. This resulted in a query execution time of 905.33s (i.e., approx. 15 minutes) as a baseline-runtime when injecting no failures and adding no additional materializations in the plan. In order to show the overhead,

[1] We use different I/O rates of the EBS volumes.



(a) Normal I/O Cost

(b) High I/O Cost

Figure 6.10.: Varying MTBF

we executed the same query when applying different fault-tolerance strategies and injected mid-query failures using the following MTBFs per node to cover a range of different failure rates: 1 week, 1 day, and 1 hour.

Figure 6.10 shows the overhead of the individual schemes under varying MTBFs. We executed this experiment for normal I/O costs (see Figure 6.10(a)) and high I/O costs where materialization costs were approx. 7× higher as in the normal I/O case (see Figure 6.10(b)). Both figures show the same trends as already reported before in Figure 6.9(a) and Figure 6.9(b): The cost-based scheme has the lowest overhead for all MTBFs when compared to the other schemes using the same MTBF. Both not-mat schemes are again independent of the I/O cost. Moreover, they do show a substantial increase in the overhead for lower MTBFs (i.e., higher failure rates) compared to our cost-based scheme. As to be expected, the efficiency of the all-mat scheme depends mainly on the I/O costs. When having high I/O costs (Figure 6.10(b)), this scheme again has the highest overhead compared to other schemes since it pays the materialization costs for all operators independent of the MTBF.

6.6 Related Work

Amazon EC2 Spot Instances

There exist different approaches on (cost-efficient) checkpoint techniques for Spot Instances [66, 117, 118]. Moreover, different approaches for bidding strategies have been proposed and evaluated [106, 107]. However, to the best of our knowledge, no work is mainly targeting PDEs on Spot Instances, nor has there been work on automatically finding the best cluster configuration for given user constraints on the budget and the availability as described in this chapter.

Fault-tolerance Schemes

Fine-granular fault-tolerance scheme are typically found in modern PDEs (such as Hadoop [8], Shark[115], Dryad [62]) as well as in many stream processing engines [61, 108]. While stream processing engines checkpoint the internal state of each operator for recovering continuous queries,

MapReduce-based systems [35] such as Hadoop [8] typically materialize the output of each operator to handle mid-query failures. For being able to recover, they rely on the fact that the intermediate results are persistent even when a node in the cluster fails, which requires expensive replication and prevents support for latency-sensitive queries. Other systems, like Impala [9] and Shark [115], store their intermediates in main-memory in order to better support short running latency-sensitive queries. Moreover, Shark uses the idea of resilient distributed datasets [120], which store their lineage in order to enable re-computation instead of replicating intermediates. However, all of these systems still materialize the output of each operator.

6.7 Conclusions and Outlook

In this chapter, we present a novel PDE called *Spotgres* that can be deployed in a cost-efficient and reliable manner on Amazon's Spot Instances. *Spotgres* has two essential features: (1) a constraint-based bid advisor which finds an optimal cluster configuration (i.e., a set of bids) based on a given set of constraints and (2) a cost-based fault-tolerance scheme that takes various parameters (such as MTBF and query statistics) into account to efficiently execute analytical queries under mid-query failures. Our experiments show that *Spotgres* can deliver the same query performance as a traditional PDE on an On-demand cluster with only 1% of the total costs.

Part III.

Scalable Machine Learning in the Cloud

7 XAI: A Middleware for Scalable AI

7.1 Introduction

Motivation

Deep Neural Networks (DNNs) have recently seen a significant adoption and are today driving the adoption of Machine Learning (ML) and Artificial Intelligence (AI) across a wide range of application domains. The expressiveness of DNNs provides accurate solutions for many complex tasks such as speech recognition, machine translation or image understanding previously thought to be unsolvable by machines, simply by observing large amounts of data. A major obstacle for the adoption of DNNs, however, is that the training of deep networks can take multiple hours or days even with modern GPUs. Furthermore, sizes of datasets and complexity of DNNs continuously grow to solve even more complex tasks with higher accuracy. This has the effect that the computational intensity and memory demands of deep learning increase further.

In order to speed-up training of modern DNNs on large data sets, most of the deep learning frameworks (such as Tensorflow [16], MXNet [21], or CNTK [75]) support the distribution of the training process across multiple machines in a cluster of nodes. However, even if existing well-established models (such as AlexNet, GoogleNet, or ResNet) are being used it is still a challenging task for data scientists to implement scale-out distributed deep learning in their environments.

The main reason is that data scientists must decide on a multitude of low-level details (e.g., selecting how many parameter servers to use amongst many other parameters) in order to distribute the training, which have an effect on the overall scalability of training DNNs. This often leads to a long and tedious trial-and-error process before the desired performance advantages of distributing the training actually materialize (if at all).According to our own experiences when using Tensorflow and MXNet as well as based on discussions with other users of these frameworks and experience reports [88, 101], setting up a distributed learning process for a new model architecture can take days or weeks even for machine learning experts.

Furthermore, there are many aspects, which make the situation even worse. One major aspect is that the trial-and-error procedure of finding an optimal setup for scalable distributed training has to be repeated over and over not only for every new model architecture but also when a new generation of hardware (GPU, network, etc.) or even if a software version becomes available due to missing higher-level abstractions in those frameworks. This additionally turns the maintenance of model training at scale into a technical dept that needs to be payed constantly [100].

Contribution

In this chapter, we present *XAI*, a middleware on top of existing deep learning frameworks that enables data scientists to easily scale-out distributed training of DNNs. The aim of *XAI* is that data scientists can use a simple interface to specify the model that needs to be trained as well as the resources available (e.g., number of machines, number of GPUs per machine, etc.). Based on this input, *XAI* automatically deploys the model on the available resources in an optimal manner.

In order to enable scalable deep learning, *XAI* comes with different components. At the core of *XAI*, we have implemented a distributed optimizer that takes the model and data as input and finds an optimal

Table 7.1.: Popular Deep Neural Networks

Network	# Parameters	Depth	Year
AlexNet	62,378,344	8	2012
VGG16	138,357,544	16	2014
GoogleNet	6,797,700	22	2014
Resnet-50	25,636,712	50	2015
Resnet-152	60,344,232	152	2015
Inception V3	23,851,784	159	2015

configuration of the distributed training procedure that maximizes throughput of the training procedure for a given set of hyper-parameters (e.g., batch-size and learning rate). In addition to the optimizer, *XAI* comes with two more components: First, *XAI* implements a component which automates the deployment of the model based on the distribution parameters (e.g., number of parameter servers) determined by the optimizer. Currently, we have implemented adapters to support Apache MXNet and TensorFlow as recent deep learning platforms and Kubernetes and Slurm that are today being used typically in Cloud-based and HPC-based clusters. Second, *XAI* additionally implements an adaptive executor component which not only monitors the available resources (CPUs, GPUs, and network utilization) but also adaptively changes the deployment if over- or under-utilization of resources is detected.

In summary, we make the following contributions in this chapter: (1) We present *XAI*, a novel middleware to simplify scalable distributed training that we plan to open-source early 2019. (2) We discuss the design of a distributed optimizer which is a core component of *XAI* to automatically find an optimal deployment strategy for a given model and available resources. (3) We show in an extensive evaluation, that *XAI* can be used to train different DNN architectures at scale and support various possible deployments including different deep learning platforms, different cluster environments, as well as different hardware generations without hard-coding a new cost model for each new setup.

Outline

The remainder of this chapter is structured as follows. Section 7.2 first discusses the background of distributed deep learning. Section 7.3 then gives an overview of the architecture of *XAI* before we then explain the details of the different components in Sections 7.4 to 7.6. The evaluation, in Section 7.7, shows the result of using *XAI* to scale-out deep learning using a variety of different workloads. Finally, we conclude with an overview of the related work in Section 7.8 and a summary of the findings and possible avenues of future work in Section 7.9.

7.2 Distributed Deep Learning

In this section, we give a brief overview of deep neural networks (DNNs) and how typically distributed training works for DNNs.

7.2.1 Deep Neural Networks

DNNs represent a class of machine learning models that has been rapidly evolving over the last couple of years and have shown to apply to a wide area of domains like image classification, object detection, speech recognition, or machine translation. The first approaches of simple neural networks, so-called feed-forward networks, were already published in the 1950's [94]. However, the accuracy of those models was worse than classical machine learning methods [76].

With increasing computational power and more massive datasets, it was possible to train deeper neural networks [56] and outperform classical machine learning algorithms. One of the first breakthroughs was the accomplishment of [69] with a neural network containing eight layers. This Neural Network, so-called AlexNet, excelled at the ImageNet competition [39] and achieved the highest accuracy until then [69]. To exemplify the growth in complexity of DNNs in the last years, Table 7.1 shows a selection of popular neural networks for image classification with their number of layers and parameters.

7.2.2 Distributed Training

While the computational power and capacity of modern GPUs have been continuously growing [23], recent frameworks additionally support distributed training of DNNs to leverage the capacity of GPUs across multiple machines.

Distributing the workload across multiple nodes, however, involves splitting the training procedure across multiple machines, which typically comes in two forms for DNNs [32]: First, model-parallelism splits the model across machines (e.g., based on its layers), and every node trains a part of the model with the full dataset. Second, with data-parallelism, the datasets are split across workers, and every worker trains the full model but only using a part of the data. While new deep learning frameworks support both schemes, this thesis focuses on data-parallelism, which has seen wider adoption in practice than model-parallelism.

For data-parallelism, typically, a centralized parameter server infrastructure is used to synchronize the model across multiple machines [36]. The idea is that each worker node sends the model parameters to the centralized parameters server infrastructure, which merges the updates from different workers and sends back the updated parameters to the workers for the next mini-batch, which is being trained. To avoid that the centralized parameter server infrastructure being a bottleneck, the parameters can be sharded across multiple parameter servers.

A major challenge when using a centralized parameter server infrastructure for distributed training is to balance computation and communication to best leverage all resources (e.g., GPUs and the available network) and enable scalability when more workers with additional GPUs are being added. Deep learning frameworks, therefore, come with a variety of parameters that influence the ratio of computation and communication, such as model consistency (i.e., asynchronous or synchronous updates [65]), mini-batch size, but also the number of parameter servers being used to shard the update load. If these parameters are not chosen carefully, the overall scalability is limited, as we will show in our experimental evaluation in Section 7.7.

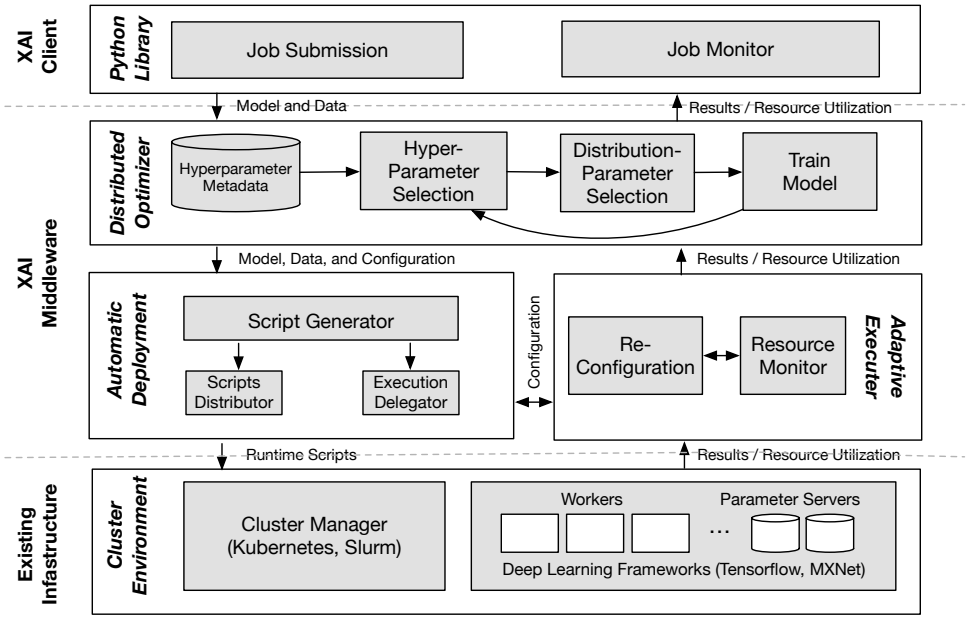


Figure 7.1.: *XAI* System Architecture

7.3 System Overview

XAI is built as middleware on top of existing machine learning frameworks such as Apache MXNet and TensorFlow. The purpose of *XAI* is to facilitate the process of running deep learning training by introducing a high-level interface, which hides the complexity of deploying a DNN in a distributed manner. Figure 7.1 shows the system architecture of *XAI*. In the following, we briefly discuss each component.

Client: *XAI* comes with a thin Python-based interface that initiates the training job where the user only has to specify the model (e.g., AlexNet) and the data set which is being used for the training procedure. That way, users can initiate a distributed training job without dealing with the low-level details such as selecting hyper-parameters (e.g., batch size) but also distribution-parameters (e.g., number of parameter servers). Additionally, the *XAI* client provides a simple interface for monitoring the training results as well as the resource utilization (e.g., of GPUs, network) online during training.

Distributed Optimizer: A major component of *XAI* is the distributed optimizer that can find an optimal configuration for a given DNN model and data set (training and test data). An optimal configuration consists of the hyperparameters that maximize the model accuracy as well as the distribution-parameters (e.g., number of parameter servers) that maximize the overall throughput in a distributed setup. The main challenge to maximize the overall throughput is to estimate the network bandwidth when running the training job in order to derive the minimum number of parameter servers required to not slow-down the training procedure. Section 7.4 explains the details of the distributed optimizer of *XAI*.

Automatic Deployment: Given a configuration from the optimizer, the DNN model is then automatically deployed in a cluster environment. A cluster environment is defined by the framework (e.g., Tensorflow and MXNet) as well as the cluster manager (e.g., Kubernetes and Slurm) that should be used for executing the training job. Based on the configuration and the given cluster environment, the *XAI*'s automatic model deployment component generates and distributes the required training scripts to all nodes (workers and parameter servers) and then delegates the training to the available cluster manager (Slurm or Kubernetes). Section 7.5 explains the details of the model deployment of *XAI*.

Adaptive Executor: The execution of a distributed DNN training is monitored by an adaptive executor in *XAI*. The purpose of the adaptive executor is to make the execution more robust towards shared environments or non-optimal decisions by the optimizer. To that end, the adaptive executor comes with a monitoring component that continuously analyzes the utilization of resources of all nodes (workers and parameter servers). Based on the monitored utilization, the adaptive executor can change a running training job by check-pointing the results of the last mini-batch and continuing the training job with a newly modified configuration (e.g., by increasing the number of parameter servers). Section 7.6 explains the details of the adaptive executor of *XAI*.

7.4 Distributed Optimizer

In this section, we explain the details of our distributed optimizer, which is the core component of *XAI*.

7.4.1 Overview of the Optimizer

As shown in the architecture of *XAI* in Figure 7.1, the optimizer is composed of three steps that are executed iteratively to explore the search space: (1) hyper-parameter selection, (2) distribution-parameter selection, and then (3) model training. The overall aim of the optimizer is to find a model with high accuracy using minimal runtime.

The idea behind the iterative search procedure is that the first step determines a set of hyper-parameters (e.g., batch size, learning rate, etc.) that should be used for training the next DNN in the next iteration. In *XAI*, we currently implement a state-of-the-art approach based on to select hyper-parameters [43], which is also available in Auto-sklearn. This approach uses a random-forest-based Bayesian optimization method such as Sequential Model-based Algorithm Configuration (SMAC) to find the best instantiation of hyper-parameters. The approach additionally employs meta-learning to start Bayesian optimization from proper configurations evaluated on previous similar datasets and stores the results in our hyper-parameter Metadatabase.

Once a set of hyper-parameters for the next iteration is selected, the second step determines a set of distribution-parameters to minimize the runtime (i.e., maximize the throughput) of the distributed training procedure. This step is not considered in the existing AutoML approaches, which typically only focus on hyper-parameter selection. The main contribution of our optimization procedure is to combine the existing AutoML approaches for hyper-parameters selection with a selection of distribution-parameters, which minimize the runtime of distributed training. The details about the selection of distribution-parameters are discussed next in Section 7.4.2.

Afterward, once a set of hyper-parameters and distribution-parameters are determined, the optimizer trains the given DNN for a pre-defined number of epochs (using the automatic model deployment and the adaptive execution component in *XAI*) and based on the accuracy results it decides whether or not to start the next iteration of optimization using the same procedure as discussed before.

7.4.2 Distribution-Parameter Selection

In the following, we describe our procedure for selecting a set of distribution-parameters to minimize the runtime for a given set of hyper-parameters. In this chapter, we focus on distributed DNN training using data-parallelism and a centralized parameter server infrastructure with multiple servers where each hosts a shard of parameters.

The two main distribution-parameters of interest in our first version of the optimizer are the number of parameter servers being used as well as the update strategy to synchronize the parameters between workers and parameter servers. However, in future versions, we also plan to consider other distributed schemes, including model-parallelism as well as other approaches for data-parallelism to distributed parameters (i.e., using replication approaches based on MPI to broadcast the parameters, etc.).

When using data-parallelism and a centralized parameter server infrastructure, the aggregated network bandwidth available between workers and parameter servers is an essential factor when it comes to scalability. The main idea behind the optimizer is that for a given number of workers, each having one or multiple GPUs, we use a cost model to estimate the minimal number of parameter servers required to sustain the update load.

In order to do so, the optimizer estimates the expected average network bandwidth requirements between all workers and the centralized parameter server infrastructure for a given DNN architecture that should be trained. Based on those estimated bandwidth-requirements, the number of parameter servers is determined by simply dividing the required bandwidth by the bandwidth each parameter server can provide, as we show later in Section 7.4.2.

Cost-Model Calibration

Different from optimizers known from databases, the optimizer in *XAI* does not use hard-coded cost-models to estimate these values. Instead, in *XAI*, we rely on a short calibration phase that determines basic cost model parameters experimentally. Using a calibration phase is typically not a problem for distributed DNN training since the training phase for one set of hyper-parameters already takes hours or even days. Compared to the time required for training, the time required for the calibration phase is negligible.

The goal of the cost-model calibration is to define the basic parameters such as the outgoing network load each worker can produce as well as the incoming network load a parameter server can consume. Furthermore, the second parameter of interest is the ratio of computing to communication time required for a given DNN model architecture. This ratio is an important parameter in our cost-based parameter selection since it allows us to determine the number of parameter servers required to minimize the overall training runtime, as we will see in the next subsection.

There are different factors that influence the ratio of computing and communication time. First, hyper-parameters such as the batch size or learning rate determine the overall update load and thus the transfer time. Second, the GPU and networking hardware being used in the cluster setup play another vital role. Thus, calibration needs to be re-executed when different hyper-parameters or a different hardware setup is being used. However, if the same DNN has already been trained before using the same hyper-parameters and hardware setup, we can reuse the cost-models of previous calibration runs. That way, our optimizer

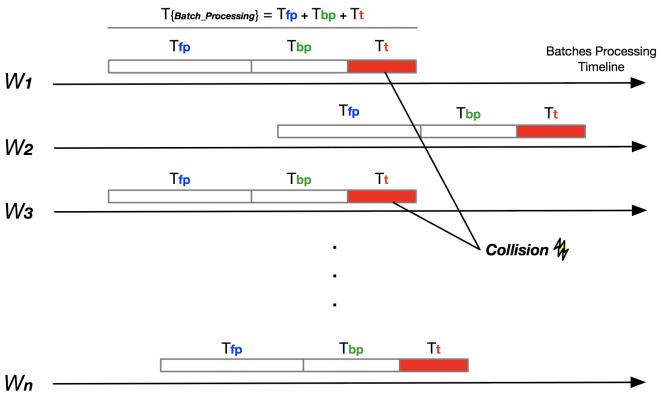


Figure 7.2.: Collision Model of our Optimizer

can determine an optimal distributed setup for unseen DNN architectures as well as new hardware generations without the need for adapting a hard-coded cost model.

In order to find out the ratio of compute and communication time required for a given DNN model architecture, the cost-model calibration trains first the DNN with the given hyper-parameters on one worker. It uses all available GPUs without using a parameter server at all (i.e., all training is executed locally) for only a few mini-batches (i.e., we use 10 mini-batches at the moment to mitigate the effect of outliers). The calibration phase then monitors the runtime of the local training and divides it by the number of batches. The time required to train one batch is then used as an estimate to represent the total training time, including the forward propagation time T_{fp} and the backward propagation time T_{bp}. Since the training is performed locally, there is no time used for transferring weights to a central parameter server infrastructure.

Afterward, the same procedure is performed using a distributed setup with one worker and an increasing number of parameter servers. We use our monitoring capabilities of *XAI* to see when the outgoing network bandwidth of the worker is saturated. The purpose of this step is to find the total batch processing time T, including the ideal transfer time T_t if the network is not a bottleneck. The difference between the batch processing time with the local training is used as an estimate for the transfer time to send the weight updates from one worker over the network to one parameter server, i.e., $T_t = T - (T_{fp} + T_{bp})$. Furthermore, based on this step of the calibration phase, we can also identify the outgoing network bandwidth load that one worker BW_w can produce.

Finally, as a last step of the calibration phase, we run the training in a distributed setup with one parameter server and an increasing number of workers. Using our monitoring capabilities, we can thus determine the maximum network bandwidth BW_{ps} that a parameter server can sustain.

Cost-Based Parameter Selection

The goal of the cost-based parameter selection is to find the minimum number of parameter servers required to cover the network load generated by n workers under different consistency models (asynchronous and synchronous updates). In our current version, we use asynchronous updates as default, while *XAI* can also be configured to use synchronous training. However, asynchronous updates have shown to provide an overall better runtime but might result in a slower convergence. Modeling the dependency between throughput and convergence for asynchronous and synchronous updates is left for future work.

Estimating the number of parameter servers required for synchronous updates is trivial. If we assume that all workers send and receive data from parameter servers at the same time, then we can compute the required number of parameter servers as $n \cdot BW_w / BW_{ps}$.

When using asynchronous parameter updates, it is more complicated, since each worker sends its updates independently. In the ideal case, if the communication of workers is not overlapping, we would only require BW_w / BW_{ps} parameter servers independent of the number of workers n being used. However, with an increasing number of workers, the likelihood that two workers send/read parameters from a centralized parameter server infrastructure at the same time increases. In the following, we show how we can see this likelihood.

If we have n workers and 1 parameter server, the range of workers that are transferring data at the same time can, in general, vary between 1 and n workers. Figure 7.2 shows the basic idea of our collision model that we use to estimate the collision likelihood that m workers (where $1 < m \leq n$) transmit their parameters at the same time.

As primary input to estimate the likelihood that m out of n workers collide, we use the following estimates that we computed as a part of the calibration phase: T which represents the total time to train a mini-batch in one worker including the transfer time as well the transfer time T_t. Based on these parameters, we can compute the probability P_t that a worker transfers data as:

$$P_t = \frac{T_t}{T} \tag{7.1}$$

If we look to the workers as being independent, then the probability that any possible combination of two workers ($\binom{n}{2}$) in a cluster with n workers are sending data to a parameter server at the same time is defined by the following equation:

$$P_t(n) = \binom{n}{2} (P_t)^2 \tag{7.2}$$

This formula can be generalized to the probability $P_t(n,m)$ that any possible combination of m workers is sending at the same time.

$$P_t(n,m) = \binom{n}{m} (P_t)^m \tag{7.3}$$

To calculate the probability that only one worker sends data at any point of time during training, we use equation 7.4:

$$P_t(n, m = 1) = 1 - \sum_{m=2}^{n} \binom{n}{m} (P_t)^m \tag{7.4}$$

The purpose of calculating the overall likelihood of collisions is to estimate the expected bandwidth E_{BW} that the workers could need to transmit parameter updates. The following equation defines how to compute the expected bandwidth for a number of workers based on the discussions before:

$$E_{BW}(n) = \sum_{m=1}^{n} m \cdot P_t(n,m) \cdot BW_w \tag{7.5}$$

After calculating the expected bandwidth $E_{BW}(n)$ for n workers, we can now estimate the number of the parameters servers $PS(n)$ required for n workers as follows:

$$PS(n) = \left\lceil \frac{E_{BW}(n)}{BW_{ps}} \right\rceil \tag{7.6}$$

In some cases, when the transfer time T_t is dominating the batch processing time T, Equation 7.6 results in an overestimate of the parameter servers. This problem is explained in [73], which discusses the probability of intersecting intervals. Based on their results, our equations above only hold if $T_t < \frac{T}{n-1}$.

The intuition behind this is that when T_t is relatively large compared to T, there are at least *m_min* workers that send data in parallel. In other words, the case where only one worker transfers weights to parameter servers is not existing. To solve this issue, we compute the minimum number of intersected workers m_{min} that collide for sure. The idea is that all other $m \geq m_{min}$ are excluded from computing the overall likelihood of collisions. We extended our estimates for this case in the appendix. In our experiments in Section 7.7, we show that our estimate based on Equation 7.6 results in optimal selection of parameter servers for an asynchronous model updates.

7.5 Automatic Model Deployment

The aim of the automatic model deployment is that a *XAI* can deploy the training of a given DNN in different cluster environments. Currently, *XAI* supports the automatic deployment of a given DNN model using Tensorflow or MXNet as DNN frameworks and Slurm or Kubernetes as a cluster scheduler.

In the following, we briefly outline the challenges and ideas that we addressed when deploying a training job on Slurm or Kubernetes, respectively.

7.5.1 Deployment using Slurm

The main challenge when executing MXNet or Tensorflow in a Slurm-based environment is that resources (i.e., workers and parameter servers) must be mapped to physical resources (i.e., nodes) after a training job is deployed.

This departs from most DNN frameworks, which require a static assignment of resources before starting a training job. To start a distributed training job in Tensorflow or MXNet, it is necessary to provide the host addressees of the different nodes and their roles (workers and parameter servers) in a cluster specification.

When using Slurm, this so-called cluster specification can only be determined at runtime after the job is deployed in a cluster. We, therefore, generate startup scripts that dynamically create a cluster specification for each node being selected by Slurm.

7.5.2 Deployment using Kubernetes

The automatic deployment component also supports Kubernetes, by automatically generating the proper YAML-scripts for all the so-called pods that will be created on each node in a cluster. The challenges are similar to the Slurm environment.

To dynamically deploy workers and parameter servers, our deployment component automatically creates internal Kubernetes services for each pod, so that they can communicate through the pod addresses. A training job receives at startup a pool of pod addresses as an argument that identifies which pods are participating in the training and who is taking over the role of being a parameter server or a worker.

7.6 Adaptive Executor

The main component of the adaptive executor is the resource monitoring component. This component records essential metrics about the hardware performance of individual nodes (i.e., workers and parameter servers) to track possible bottlenecks. The identification of bottlenecks on a high level helps to quickly investigate the problem more accurately with specific tools or directly adapt the DNN training deployment to utilize all resources uniformly.

The currently selected metrics include CPU utilization, main memory consumption, in- and outgoing network traffic, as well as GPU load and GPU memory consumption. Even during developing *XAI*, the monitoring tool has often helped us to directly identify if we are running into a network or GPU bottleneck or to detect a skew on the parameter servers.

The monitoring component is implemented as a python program that runs on one CPU core in parallel to the DNN training process, records the given metrics, and writes them to a log file. The logs from all nodes are continuously collected, transformed and analyzed, so that the training can be adapted according to the monitoring results and, for instance, scale-in or scale-out the parameter server infrastructure if more resources are needed. For the manual investigation, we additionally provide a service for visualizing the analyzed data using the before-mentioned metrics.

7.7 Experimental Evaluation

In our experimental evaluation, we have trained neural networks in a distributed way on different clusters and deep learning frameworks using various hyper- and distribution-parameters. To give an overview and back up the need for a cost-based optimizer in *XAI*, the following sections will first illustrate how those different parameters significantly influence the performance of the DNN training. Furthermore, we also show the efficiency of our optimizer to select an optimal set of parameters as well as interesting findings that we are able to derive from using our monitoring component.

Setup and Workloads

In all our experiments, we have used the two cluster setups as shown in Table 7.2. We have chosen two different setups: one setup on an HPC-cluster with a fast network connection and one setup using

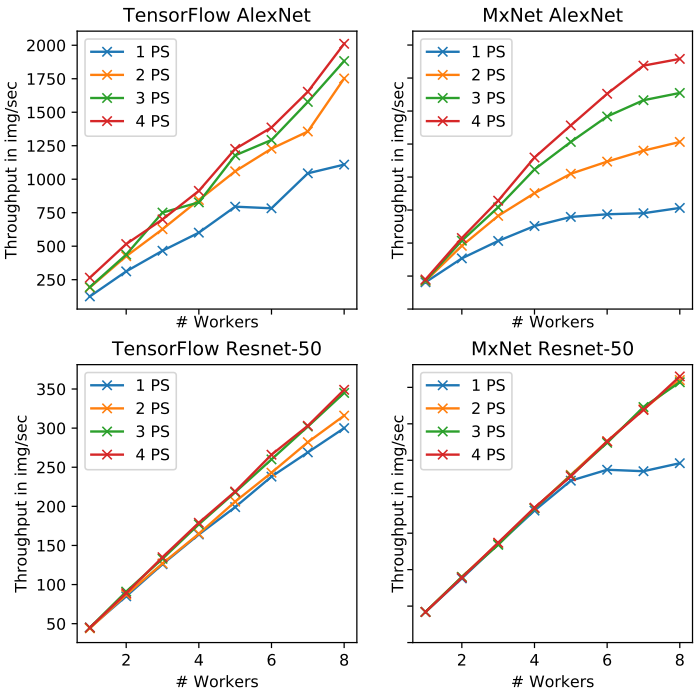


Figure 7.3.: Throughput Analysis for AlexNet and ResNet-50 using TensorFlow on the HPC Cluster with Asynchronous Training

an AWS-cluster with a slower network connection. Furthermore, both setups differ also in the GPU generation being used.

The DNN models we have used for the evaluation are listed in Table 7.1. These DNNs have been used over the last years for image classification representing different model architectures. We believe that our findings and cost models can also be generalized to other domains using different model architectures (e.g., sequence-to-sequence models for machine translation). Showing this, however, is part of our future work.

Furthermore, in our evaluation, we only show the effects of selecting different parameters on the overall throughput and not on model accuracy. The reason is that the main contribution of our optimizer is the cost-based model to find an optimal distributed setup, which aims to minimize runtime. Consequently, in all our experiments, we also used only synthetic data sets; i.e., images are represented as in-memory arrays with random values, to avoid running into bottlenecks (e.g., I/O limitations of shared file systems) not relevant for our evaluations.

7.7.1 Exp 1: Throughput Analysis

We have empirically evaluated several hundreds of distributed setups to show the sensitivity of the training throughput from different parameters. The results justify the need for a calibration phase to make *XAI* not only independent of the DNN architecture but also independent of the framework being used as well as the underlying hardware. In the following sections, we summarize the most important findings.

Table 7.2.: Machine Configuration in Different Clusters

Cluster	HPC	AWS (p2.xlarge)
CPU	2x Xeon E5-2670	4x vCPU
RAM	32 GByte	61 GByte
Bandwith	20 Gbit/s	1.3 Gbit/s
GPU	2x Tesla K20X	1x Tesla K80
OS	CentOS 7	Ubuntu 16.02
TensorFlow	1.10	1.10
MXNet	1.2.1	n.a.
CUDA	9.0	9.0
CUDNN	7.1.3	7.1.3

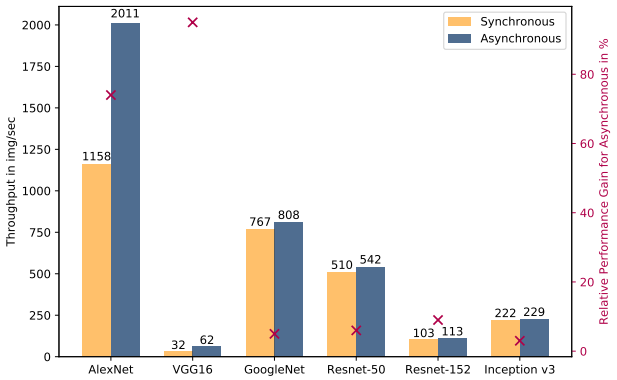


Figure 7.4.: Throughput Analysis in TensorFlow for different DNNs with 8 Worker and 4 Parameter Servers on the HPC Cluster

Different Frameworks

By training DNNs on different frameworks, we saw a difference in terms of throughput when using different Deep Learning Frameworks like TensorFlow or MXNet. In Figure 7.3, the upper plots show how the training throughput in TensorFlow for training AlexNet using asynchronous model consistency can achieve higher throughput with a lower amount of parameter servers (PS) than with MXNet. What is not shown in the plots, is that in synchronous mode, TensorFlow performs up to 30% worse than MXNet with a higher number of parameter servers, whereas TensorFlow still outperforms MXNet with a small number of parameter servers.

In the lower two plots of Figure 7.3, we additionally see that when using another DNN (ResNet-50) there is no big difference between the two frameworks. However, the previous observation also applies here. MXNet needs more parameter servers for scaling-out. This is indicated by the *1 PS* line in the lower right subplot, which stops increasing at a throughput of around 250 images per second. The general ability to achieve a better scale-out with fewer parameter servers in ResNet-50 in both frameworks is due to higher computational requirements for training this DNN. As a result, the ratio of communication to computation shifts to a less network-bound situation.

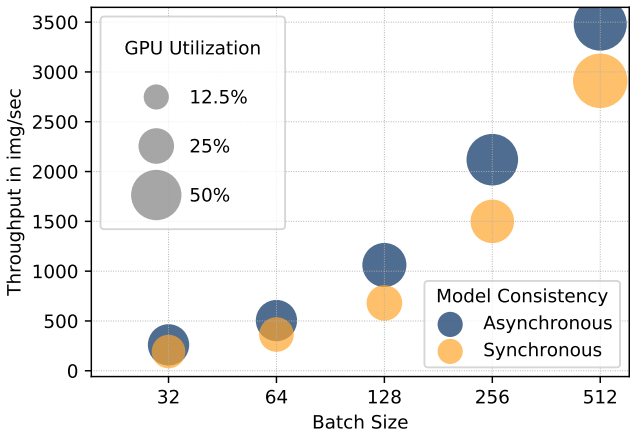


Figure 7.5.: Effect of Batch Size on the Throughput for training AlexNet with 8 Workers and 1 parameter server on the HPC Cluster

Different DNNs

Figure 7.4 illustrates the throughput difference when training the different DNNs mentioned in Table 7.1. It is noticeable that the difference in throughput is related to the difference in computational complexity of the DNNs, which is not the same as the number of parameters a DNN has but also depends on the types of layers used. As a result, for some DNNs with low computational complexity, the throughput is much higher if a fast network interconnection is being used because the ratio of computation over communication shifts and makes the overall training network-bound.

Different Consistency Models

In this experiment, we analyze the effect of different consistency models on the overall throughput, as shown in Figure 7.4. For some DNNs like AlexNet and VGG16, we can see a high increase for asynchronous over synchronous training of up to 80%, shown by the red markers in Figure 7.4. This is caused by the high amount of parameters of those networks, while the depth (i.e., number of layers) is comparably low compared to other DNNs. As a result, we can see that the performance gains of asynchronous training over synchronous training depend heavily on DNN architecture.

Different Batch Sizes

Figure 7.5 shows that the mini-batch size influences the throughput during distributed training. The x-axis shows the batch size for a single worker on a logarithmic scale, and the y-axis shows the throughput. In this experiment, we only show the result of AlexNet. For this DNN, the throughput scales almost linearly with an increase in mini-batch size. What we can also see is that an increase in mini-batch size leads to a

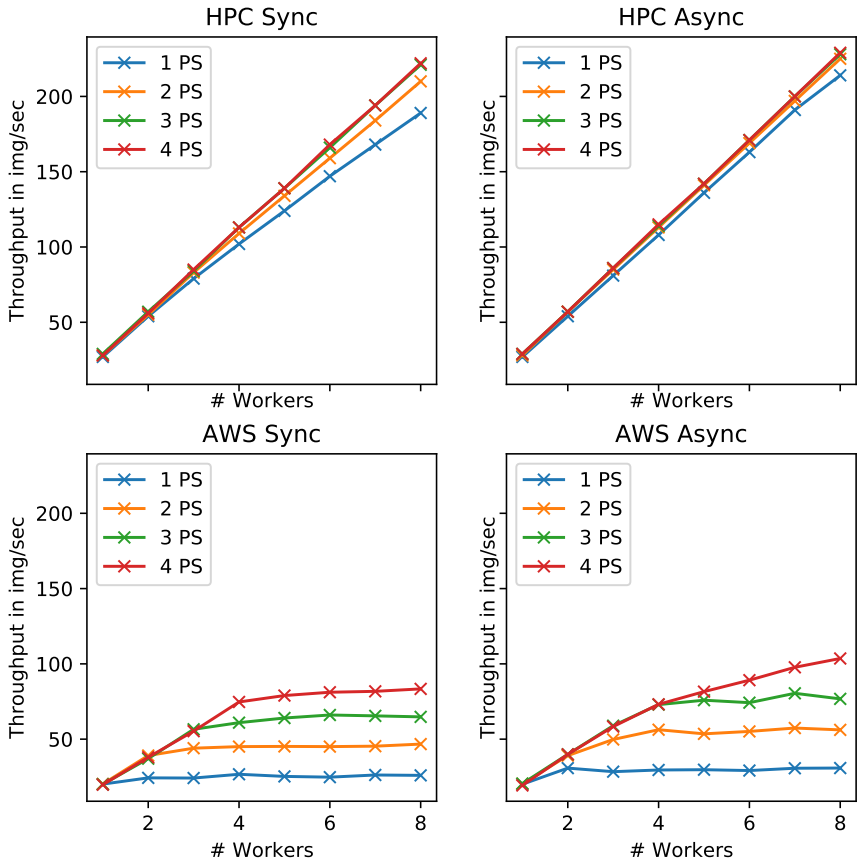


Figure 7.6.: Training of Inception v3 on different Clusters

higher GPU utilization of the workers, since a single worker processes more images per batch such that the ratio between communication over computation decreases. It is further noticeable that asynchronous training performs on average 38% better than synchronous training.

Different Cluster Environments

In this experiment, we show the effects of using different cluster setups (HPC vs. AWS). The lower plots in Figure 7.6 indicate that the DNN training on AWS with a comparably slow network connection (see Table 7.2) shows effects of network congestion. The increase of parameter servers for AWS thus helps to mitigate the congestion to a certain extent and better scale-out. The DNN training on the HPC cluster (upper plots in Figure 7.6), on the other, shows much better scalability with fewer parameter servers. Only in the case of synchronous training, we also see that the network becomes a bottleneck when using only one parameter server for 8 workers. This originates from network peak requirements in synchronous training since all workers send their updates and receive new parameters (almost) simultaneously.

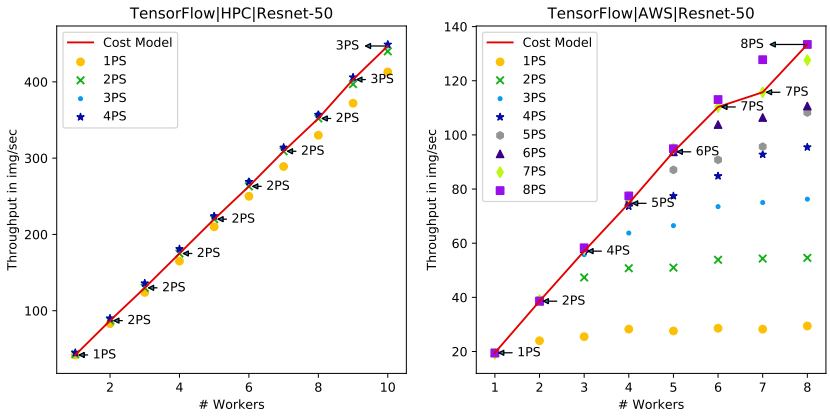


Figure 7.7.: Accuracy of the Optimizer for Different Clusters

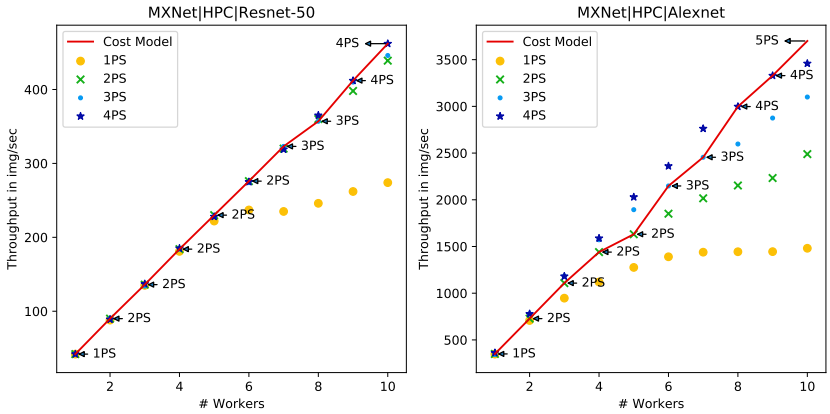


Figure 7.8.: Accuracy of the Optimizer in Different DNNs

7.7.2 Exp 2: Accuracy of Optimizer

The goal of this experiment is to show the accuracy of our cost model. To show that, we executed the distributed training procedure with a varying number of workers where we first manually varied the number of parameter servers and then compared it to the training in *XAI* where our optimizer determined the number of parameter servers for a given number of workers. The idea is that the optimizer neither under-estimates nor over-estimates the number of parameter servers required to sustain the load of the workers.

In the following, we show the results when applying our cost model not only for different DNN models but also when using deep learning frameworks as well as different cluster setups. However, due to space limitations, we only show the results of the cost model for asynchronous training, which is also more challenging to model, as explained in Section 7.4.

Figure 7.7 shows the result of training ResNet-50 on the HPC and the AWS clusters using TensorFlow. The goal is to show the accuracy of the cost model for different clusters using different hardware setups. The red line shows the result of our cost model, where each point is annotated with the parameter servers that our model predicted. As we see from the plot, our cost model predicts the minimal number of parameter servers that allow us to scale-out almost linearly (i.e., it neither under- nor over-estimates the

number of parameter servers required). For example, for the plot on the right-hand side, we can see that the cost model suggests using 5 parameter servers for 4 workers. Using more parameter servers would not increase the throughput, but using less than 5 servers would significantly decrease the overall throughput. What is also interesting is that in the HPC cluster (left-hand side), where we have high network bandwidth, the optimizer, in general, recommends a fewer number of parameters servers compared to training the same DNN on the AWS cluster (right-hand side) where we have only a slow network.

Figure 7.8 shows the result of training ResNet-50 and AlexNet on the HPC cluster using Apache MXNet. The goal is to show the accuracy of the cost model for DNNs with different computational complexity. In ResNet-50, our optimizer recommends using 2 parameter servers if up to 6 workers are being used. If we look at the throughput for 2 workers, we can easily see that 2 parameter servers are an optimal choice because using 3 or 4 parameter servers would not increase the throughput. Moreover, in AlexNet, where the number of model parameters is higher as for ResNet-50, the likelihood of collisions (i.e., two workers send/receive their parameters at the same time) is also higher. Thus, our optimizer selects to use more parameters servers for the same number of workers.

7.7.3 Exp 3: Resource Monitoring

The resource monitoring component provides information about several metrics for each node of the cluster, as explained in Section 7.6. This component is beneficial in identifying unexpected behaviors during training and helped us to point out potential bottlenecks.

For instance, Figure 7.9 shows the monitored network data received for the parameter servers during the training of AlexNet when using 4 parameter servers and 5 workers. In the upper plot, we see that the network utilization for parameter servers (PS) 2 and 3, represented by green and orange lines, are much higher, while the network utilization for parameter servers 1 and 4 is much lower. This information led us to investigate how the parameters of AlexNet were distributed among the four parameter servers. The reason turned out to be skew in the way how weights were distributed to parameter servers.

Further investigations led us to the following findings. AlexNet has three fully-connected layers: Two with 4096 neurons each and one with only 1000 neurons [69]. Since each fully-connected layer is one big operation in the computation graph, the load balancer of Tensorflow was assigning the parameters in a layer-wise manner to parameter servers. This layer-wise assignment was then causing the skew on the network and consequently reducing the overall training performance.

To solve the issue, we introduced a new partitioner for AlexNet to split the layers into equal-sized parts according to the number of parameters servers, thus, sharding the network load equally across servers. The results after using our partitioner for AlexNet can be seen in the lower plot of Figure 7.9. As a main result, we see that the network load is better distributed across all parameter servers, and the overall training time is thus reduced.

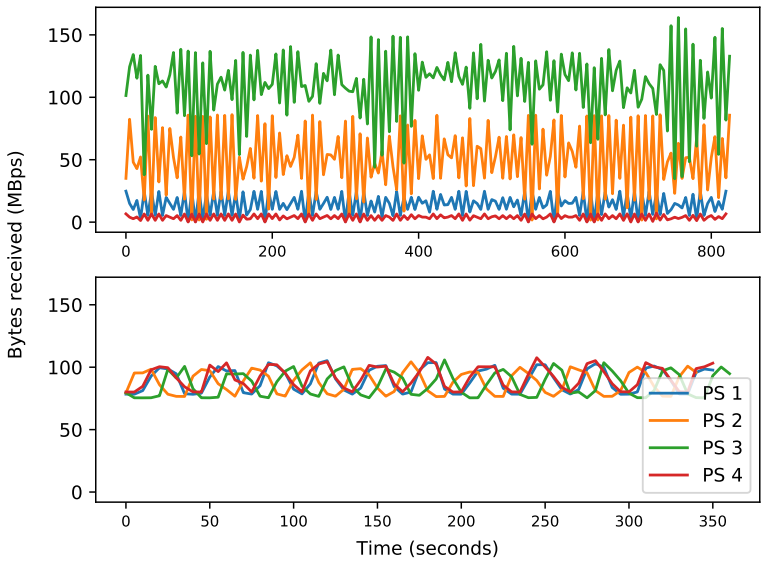


Figure 7.9.: Network Data received by Parameter Servers with and without Skew.

7.8 Related Work

In this section, we discuss related work. We first focus on modern systems and libraries that have a similar goal as *XAI* to enable scalable AI. Afterward, we discuss the broader area of automated machine learning (AutoML), which is also relevant for this chapter.

Scalable AI

Many recent deep learning frameworks such as TensorFlow [16], MXNet [21], or CNTK [75]) support the distribution of the training process across multiple machines in a cluster of nodes. However, even if existing well-established models (such as AlexNet, GoogleNet, or ResNet) are being used, it is still a challenging task to efficiently scale-out distributed deep learning.

A system that takes similar approaches as *XAI* to simplify the distributed training of DNNs is Horovod [101]. However, there are significant differences between *XAI* and Horovod. First, Horovod is currently only supporting TensorFlow as a framework, while *XAI* is built as middleware and can support different deep learning frameworks. Second, *XAI* comes with an optimizer that automatically defines the optimal number of parameter servers, which has to be manually tuned in Horovod.

Another direction to scale out deep learning more efficiently is to provide libraries that allow deep learning frameworks to implement a more efficient communication stack. One example is the Intel Machine Learning Scaling Library (MLSL) [104]. MLSL uses an implementation of the MPI allreduce primitive to make communication more efficient. Furthermore, MLSL also comes with some adaptive execution strategies to better overlap computation and communication. All these optimizations are orthogonal to the goals of *XAI* and could be integrated into any of the supported frameworks of *XAI*.

Automated Machine Learning

There have been several attempts to automate machine learning to make it more accessible. However, these approaches typically concentrate on the hyper-parameter selection and not on the complete automated deployment of distributed machine learning, as we do in *XAI*. One notable example is Auto-Weka [109].

Auto-WEKA aims to automate the use of Weka [72] by applying recent derivative-free optimization algorithms, in particular, SMAC [60], to the hyperparameters tuning problem; a small sub-problem of *XAI*. Furthermore, there are also Cloud services like Google AutoML [49]. These services, however, are often only usable in a limited number of scenarios as they significantly restrict the type of models that can be trained. Moreover, cloud services often enforce other limits, such as a maximum training data-size.

7.9 Conclusions and Outlook

In this chapter, we presented *XAI*, a middleware on top of existing deep learning frameworks (Apache MXNet and Tensorflow), to easily scale-out distributed training of DNNs. At the core of *XAI*, we have implemented a distributed optimizer that takes the model and the available cluster resources as input and finds an optimal distributed setup of the training procedure. In the first version of *XAI*, we only support distributed training using data-parallelism with a centralized parameter server. In the future, we will extend *XAI* to also support not only other frameworks but also other forms of data-parallelism within those frameworks (e.g., by replicating the parameters). Another exciting route would be to include automatic model-parallelism in *XAI* as well.

Part IV.
Summary

8 Conclusion and Outlook

In this thesis we considered the problems of performing scalable data analytics and machine learning in the cloud. In Chapter 3, we proposed *XDB*, a parallel shared-nothing database architecture for scalable data analytics in the cloud. *XDB* combines the scalability of the shared-nothing architecture with the elasticity of the cloud. Moreover, *XDB* is implemented using a middleware approach over an existing single-node database system, using its efficient features in query processing. Recently, several cloud vendors such as Amazon and Microsoft implemented similar ideas in *XDB* in their cloud analytics services Amazon Redshift [3], and to a lesser extent, Azure Synapse [5].

XDB integrates two novel concepts: elastic partitioning and cost-based fault-tolerance. We have dedicated Chapter 4 and Chapter 5 to highlight them. In the discussion on elastic partitioning, we presented `PREF`, a novel partitioning scheme that produces co-partition tables with a given set of join predicates. Furthermore, based on `PREF`, we also proposed two automatic partitioning design algorithms that maximize data-locality while minimizing data-redundancy. The first algorithm is schema-driven that relies on the schema to derive the join predicates, and the second algorithm is workload-driven, which, in addition to the schema, uses a set of queries as inputs. Our experiments show that while the schema-driven algorithms work reasonably well for small schemata, the workload-driven design algorithm is more efficient for complex schemata with a larger number of tables. Furthermore, we presented our novel cost-based fault-tolerance scheme for parallel data processing. Our cost model can select the optimal set of intermediate results to be materialized such that the query runtime is minimized in the presence of mid-query failures. The experiments show that our fault-tolerance scheme is efficient with a variety of different queries and clusters (i.e. different MTBF). As a relevant contribution, we presented *Spotgres* as a variant of *XDB*. *Spotgres* is a novel parallel database that can be deployed reliably and economically on Amazon's Spot Instances. In *Spotgres*, we benefited from our above-mentioned, cost-based fault-tolerance scheme to recover from mid-query failures. Additionally, we implemented in *Spotgres*, a constrained-based bid advisor that finds an optimal set of bids based on a given set of constraints.

On the topic of machine learning, we presented in Chapter 7, *XAI* as a middleware on top of existing deep learning frameworks (Apache MXNet and Tensorflow) that enables an easy scale-out of distributed training of DNNs in the cloud. Recently, cloud providers like Google AutoML [49] and Amazon SageMaker [4] are delivering similar services like *XAI*. These services, however, are often only usable in a limited number of scenarios as they significantly restrict the type of models that can be trained, moreover, they lack a distribution optimizer that optimizes the number of parameter servers and workers. The aim of *XAI* is that data scientists can use a simple interface to specify the model that needs to be trained as well as the resources available (e.g., number of machines, number of GPUs per machine). Based on this input, *XAI* automatically deploys the model on the available resources optimally. In *XAI*, we implemented optimizers to optimize the training parameters such as distribution- and hyper-parameters to maximize the overall of the training process and accuracy, depending on the DNN and the cluster setups.

One of the future work aspects on *XDB* is to extend it to support more complex workloads such as machine learning tasks as well as different data formats. Since *XDB* has been implemented based on core ideas in data partitioning and fault tolerance, we also seek to improve the above mentioned novel ideas. For data partitioning, one of the potential areas here is to adopt our automatic partitioning design to other operators (e.g., aggregations). Moreover, we want to support different kinds of workloads, such as OLTP. For fault-tolerance, we are looking to integrate other fault-tolerance strategies (e.g., check-pointing of

the operator state to also support mid-operator failures) into our cost-based fault-tolerance scheme. With regards to *XAI*, we will extend it to support not only other machine learning frameworks, but also other forms of data-parallelism within those frameworks (e.g., by replicating the parameters). One of the aspects of improvements of *XAI*, as well, is to support other distributions schemes such as model-parallelism and hybrid-parallelism.

A Elastic Partitioning Scheme

A.1 Estimating Redundancy

As discussed in Section 4.2, predicate-based reference partitioning might result in some redundancy in PREF partitioned tables. Since seed tables in our partitioning design algorithms will never contain duplicate tuples (and thus redundancy), in order to estimate the size of a database after partitioning, the expected redundancy in all tables which are PREF partitioned must be estimated. In the following, we explain our probabilistic method to estimate the size of a given PREF partitioned table.

Redundancy is cumulative, meaning that if a referenced table in the PREF scheme contains duplicates, the referencing table will inherit those duplicates as well. This can be best explained by the example in Figure 4.2. Table ORDERS has two copies of the tuple with orderkey=1. Table CUSTOMER, which is PREF partitioned by ORDERS inherits this duplicate (i.e., customer withcustkey=1 is therefore stored redundantly in partition 1 and 3). In other words, PREF partitioning can be viewed as walking a tree, where nodes are the tables (the seed table being the root) and each referenced table is the parent of its referencing table(s). The redundancy in the parent table results in the redundancy in the child table. Therefore, in order to estimate the redundancy in a PREF table, we should take into account the redundancy of all the tables along the path to the seed table.

To this end, we assign a *redundancy factor* (denoted by r) to each edge in the MAST. To find the redundancy factor of an edge, we use histogram of the join key in the referenced table (whereas we can use sampling to reduce the runtime effort to build histograms). For example, assume that we want to estimate the size of the table ORDERS in Figure 4.2 after partitioning. For each value in column orderkey of table LINEITEM, we calculate the expected number of duplicates of the corresponding tuple in table ORDERS we expect after partitioning (i.e. the number of partitions which contain a copy of that tuple). The idea behind this method is that tuples with lower frequency in the histogram are expected to end up in fewer number of partitions (i.e. fewer duplicates) as compared to tuples with higher frequency. Therefore, by calculating the *expected value* of copies of each distinct value in the join key of the referenced table, and then add them all together, we can find an estimate of the size of the referencing table after partitioning.

We now formally explain our method. Let the random variable X denote the expected number of copies of a tuple after partitioning, n represent the number of partitions and f denotes the frequency of that tuple in the histogram. Note that X can take any number between 1 and $m = min(n, f)$: $X = 1$ results when all references of that tuple happen to be in the same partition, and therefore, no replica is needed. On the other hand, $X = min(N, f)$ is the maximum value, since the number of copies of a tuple is either n (i.e. full replication) or f, when $f < n$ and each reference ends up in a separate partition. The expected number of copies of a tuple with frequency f is therefore as follows:

$$E_{f,n}[X] = 1 \cdot P_{f,n}(X = 1) + 2 \cdot P_{f,n}(X = 2) + \cdots + m \cdot P_{f,n}(X = m)$$

where $P_{f,N}(X = x)$ is the probability that the tuple has x copies (meaning that it will be replicated to x different partitions, out of the total N partitions). It can be calculated as follows:

$$P_{f,n}(X = x) = \frac{\binom{n}{x} \cdot x! \cdot S(f,x)}{n^f}$$

In the formula above, $S(f,x)$ is the *Stirling number of the second kind*, and is the number of ways to partition a set of x distinguishable objects (tuples) into n non-empty indistinguishable boxes (partitions). The numerator, therefore, is the number of ways to choose x partitions out of total n partitions, i.e. $\binom{n}{x}$, and then to partition f tuples into these x distinguishable partitions, which is why the Stirling number is multiplied by $x!$. The denominator is the total number of ways to put f distinct tuples into n distinct partitions.

The above mentioned formula requires a number of expensive recursions (because of the Stirling number). Since $E_{f,n}[X]$ depends only on f and n, the entire computation can be done in a preprocessing phase. Therefore, instead of actually calculating the expected number of copies for each tuple in run-time, only a fast look-up in a pre-loaded table is enough. Thus, the time complexity of finding $E_{f,n}[X]$ in run-time is $O(1)$.

We are now ready to calculate the *redundancy factor* of an edge in `MAST`. We define it as the after-partitioning size of a table divided by its original size. Let V_e denote the set of distinct values in the join key of the referenced table T_i over its outgoing edge e to table T_j(for example,the distinct values of column `orderkey` of table `LINEITEM` in Figure 4.2 is $\{1,2,3\}$).

The *redundancy factor* is defined as follows:

$$r(e) = \frac{\sum_{v \in V_e} E[v]}{|T_j|}$$

Note that $r(e)$ can be ranged between two extremes: 1 (no redundancy) and n (full redundancy). As mentioned before, the after-partitioning size of a referencing table is not determined only by the redundancy factor of the immediate edge coming from its referenced table, but also by the redundancy factor of all the edges along the path from the seed table.

Finally, we can now estimate the after-partitioning size of table T_i:

$$|T_i^P| = |T_i| \cdot \prod_{e \in path(T_{RF}, T_i)} r(e)$$

where $path(T_{RF}, T_i)$ consists of all the edges from T_{RF} to T_i in the `MAST`.

For example, assume that $r(\texttt{LINEITEM} \rightarrow \texttt{ORDERS})$ turns out to be 2, meaning that $|\texttt{ORDERS}^P|$ would be twice as big as its original size $|\texttt{ORDERS}|$. Now, if $r(\texttt{ORDERS} \rightarrow \texttt{CUSTOMER})$ is 3, the after-partitioning size of table `CUSTOMER` would be estimated to be $2 \cdot 3 = 6$ times its original size.

B XAI Distribution Cost-model

B.1 Cost Model Extension

In this section, we will explain in detail the extension of the cost model from section 7.4.2 to cover some critical cases when the network is slow (i.e., the transfer time dominates the training time).

According to [73], Equation 7.3 holds only iff $T_t < \frac{T}{n-1}$. The intuition behind this is that when T_t is relatively large compared to T, then having intersected workers which transfer data together to the parameter server without having an intersection somewhere else in the period T is not guaranteed, and thus, equation 7.3 is not valid to calculate the transfer probability in such cases. We saw that the condition $T_t < \frac{T}{n-1}$ does not hold normally on those clusters which have low network bandwidth, and consequently, high transfer time T_t compared to the computation time.

To solve this problem, we calculated the minimal number of workers m_{min} that can exclusively intersect and send updates together to the parameter server on the period T_t within a batch processing time T. Derived from the constraint itself, $n < \frac{T}{Tt} + 1$, and then, calculate the minimal m_{min} as follows:

$$m_{min} = n - (\left\lfloor \frac{T}{Tt} \right\rfloor - 1) \tag{B.1}$$

Figure B.1 shows an example of an exclusive intersection. The time to transfer updates to the parameter servers T_t is almost half of the total time of process a batch. According to equation B.1, $m_{min} = 3$. That means, from the total 4 workers, we can get any combination of 3 intersected workers on T_t, which exclusively intersect. Equation 7.3 can be applied then to calculate transfer probabilities of any number of intersected workers $m \geq m_{min}$. On the other hand, equation 7.3 will not be valid to be used with $m = 2$ sending workers. The reason is that there is no case where only 2 workers are sending data exclusively to the parameter server within time T without also having 2; other workers are intersecting somewhere within the same period T.

To solve the issue of calculating the transfer probabilities of those cases where the number of intersected workers $m < m_{min}$, we first calculate the probabilities for the cases where $m \geq m_{min}$ using the equation

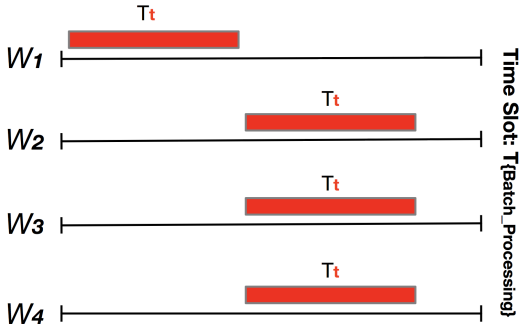


Figure B.1.: Exclusive Intersection of Worker m=3

7.3, and then as an estimate, we deal with the cases where $m < m_{min}$ as one single case with one transfer probability value which can be calculated as follows:

$$P_t(n, m < m_{min}) = 1 - \sum_{m=m_{min}}^{n} P_t(n, m) \tag{B.2}$$

To calculate the expected bandwidth of the cases where $m < m_{min}$, we need to calculate first the weighted average bandwidth for those cases as follows:

$$BW(n, m < m_{min}) = \frac{\sum_{m=1}^{m_{min}-1} m \cdot BW_w}{m_{min} - 1} \; m_{min} \geq 2 \tag{B.3}$$

Our final equation to calculate the expected bandwidth of n workers with taking in consideration the time constraint from [73] is as the following:

$$E_{BW}(n) = \begin{cases} P_t(n, m = 1) BW_w + \sum_{m=2}^{n} m P_t(n, m) BW_w & \\ & T_t < \frac{T}{n-1} \\ P_t(n, m < m_{min}) \frac{\sum_{m=1}^{m_{min}-1} m \cdot BW_w}{m_{min}-1} + & \\ \sum_{m=m_{min}}^{n} m P_t(n, m) BW_w & T_t \geq \frac{T}{n-1} \end{cases} \tag{B.4}$$

B.2 Distributed Optimizer Algorithm

In the following, we will explain the algorithm which we implemented to find the optimal distribution strategy for a distributed training job. The algorithm is based on our cost model which has been explained in section 7.4.2 and section B.1. Algorithm 1 describes the distribution parameters selection; at the very beginning, an enumeration step is performed to produce different possible distribution configurations. For example, if the cluster has 10 nodes, then the enumeration step will produce 9 different configurations starting from $1ps$ and $1w$ to $1ps$ and $9w$ because at least 1 parameter server can be used in a distributed training. These are default settings that will be evaluated in the next steps. In line 2, the transfer probability of sending data to a parameter server is calculated as described in equation 7.1. The value presents the transfer probability of a single, and an exclusive worker has transfer time T_t over a time period of length T.

The evaluation of the configurations list starts by iterating on the configurations one by one. We first check if the transfer probability in equation 7.3 can be used to calculate the transfer probability of different collisions. This step is done by checking the constraint $conf.W < \frac{T}{Tt} + 1$, if the condition does not hold, we calculate m_{min} as described in equation B.1 in section B.1, otherwise we assign 2 to m_{min} since 2 present the minimal number of intersection between workers. For each configuration, we calculate the transfer probability for all possible collisions between workers in that configuration. Once the probabilities of the configuration are calculated, we calculate the expected bandwidth according to the equation B.4.

The last step of the algorithm is to find the optimal configuration from the list of the evaluated configurations. To pick the configuration with maximum throughput, we sort the list of the distribution configurations ascendingly. We then iterate from the last configuration in the sorted list until we meet a configuration where the sum of its workers and recommended parameter servers can be obtained from the cluster, such that $conf.w + conf.PS \leq cluster.n$.

Algorithm 1: Distribution Parameters Selection

input : number of nodes in Cluster n, T_t, T, BW_w, BW_{PS}
output : Optimal distribution setup (W, PS) from a cluster

1 initialize: $distributionConfs \leftarrow enumerateConf(cluster.n)$;
2 initialize: $tProb \leftarrow T_t/T$;
3 **foreach** *conf in distributionConfs* **do**
4 **if** $checkTransferProbConstraint(conf.W, T_t, T)$ **then**
5 **for** $m \leftarrow 2$ *to* $conf.W$ **do**
6 $tProbMWorker \leftarrow calTProbMWorkers(m, conf.W, tProb)$;
7 $conf.insertTProb(m, tProbMWorker)$;
8 **end**
9 $tProbNoOrMinIntersection \leftarrow calTProbNoOrMinIntersections(conf, 2)$;
10 set $tProbNoOrMinIntersection$ in $Conf$;
11 $E_{BW} \leftarrow calExpectedBWForConf(conf, BW_{worker})$;
12 $PS \leftarrow calRecommendedPS(E_{BW}, BW_{PS})$;
13 $conf.setRecommendedPS(PS)$;
14 **else**
15 $m_{min} \leftarrow calExclusiveIntersection(conf.W, T_t, T)$;
16 **for** $m \leftarrow m_{min}$ *to* $conf.W$ **do**
17 Lines 6 - 7;
18 **end**
19 $tProbNoOrMinIntersection \leftarrow calTProbNoOrMinIntersections(conf, m_{min})$; Lines 10 - 13;
20 **end**
21 **end**

Bibliography

[1] TPC-DS. http://www.tpc.org/tpcds/, .

[2] TPC-H. http://www.tpc.org/tpch/, .

[3] Amazon Redshift Cloud Data Warehouse. https://aws.amazon.com/redshift/, .

[4] Amazon SageMaker. https://aws.amazon.com/sagemaker/, .

[5] Azure Synapse Analytics. https://azure.microsoft.com/en-us/services/synapse-analytics/.

[6] An Elastic Shared-Nothing Architecture . https://skylandtech.net/2015/04/28/an-elastic-shared-nothing-architecture/.

[7] Pivotal Greenplum Database. http://www.gopivotal.com/big-data/pivotal-greenplum-database.

[8] Apache Hadoop. http://hadoop.apache.org/.

[9] Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[10] Data Partitioning. https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning.

[11] Oracle Exadata. https://www.oracle.com/database/technologies/exadata.html.

[12] SAP HANA Database. www.sap.com/HANA.

[13] Teradata Database. http://www.teradata.com/.

[14] HP Vertica Database. http://www.vertica.com/.

[15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[16] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 265–283, 2016.

[17] Azza Abouzied, Kamil Bajda-Pawlikowski, Jiewen Huang, Daniel J. Abadi, and Avi Silberschatz. HadoopDB in action: building real world applications. In *SIGMOD Conference*, pages 1111–1114, 2010.

[18] AI and ML and DL. Clearing the confusion: Ai vs machine learning vs deep learning differences. https://towardsdatascience.com/clearing-the-confusion-ai-vs-machine-learning-vs-deep-learning-differences-fce69b21d5eb, 2018. Accessed: 2020-04-20.

[19] Fuat Akal, Klemens Böhm, and Hans-Jörg Schek. OLAP Query Evaluation in a Database Cluster: A Performance Study on Intra-Query Parallelism. In *ADBIS*, pages 218–231, 2002.

[20] Alex Smola. A. What is the Parameter Server? https://www.quora.com/What-is-the-Parameter-Server.

[21] Apache MXNet. Apache MXNet. https://mxnet.apache.org/, 2018. Accessed: 2018-09-28.

[22] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. Efficient processing of data warehousing queries in a split execution environment. In *SIGMOD Conference*, pages 1165–1176, 2011.

[23] Tal Ben-Nun and Torsten Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR*, abs/1802.0, feb 2018. URL http://arxiv.org/abs/1802.09941https://arxiv.org/abs/1802.09941.

[24] C. Binnig, A. Salama, E. Zamanian, M. El-Hindi, S. Feil, and T. Ziegler. Spotgres - parallel data analytics on spot instances. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 14–21, April 2015. doi: 10.1109/ICDEW.2015.7129538.

[25] Carsten Binnig, Robin Rehrmann, Franz Faerber, and Rudolf Riewe. FunSQL: it is time to make SQL functional. In *EDBT/ICDT Workshops*, pages 41–46, 2012.

[26] Carsten Binnig, Norman May, and Tobias Mindnich. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In *BTW*, pages 363–382, 2013.

[27] Carsten Binnig, Abdallah Salama, Alexander C. Müller, Erfan Zamanian, Harald Kornmayer, and Sven Lising. XDB: a novel database architecture for data analytics as a service. In *SoCC*, page 39, 2013.

[28] Carsten Binnig, Abdallah Salama, Alexander C. Müller, Erfan Zamanian, Harald Kornmayer, and Sven Lising. XDB: a novel database architecture for data analytics as a service. In *IEEE Big Data*, 2014.

[29] Carsten Binnig, Abdallah Salama, and Erfan Zamanian. DoomDB - Kill the Query. In *SIGMOD Conference*, 2014.

[30] Stéphane Bressan. *Distributed Query Optimization*, pages 908–912. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_708. URL https://doi.org/10.1007/978-0-387-39940-9_708.

[31] Eric A. Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[32] Víctor Campos, Francesc Sastre, Maurici Yagües, Míriam Bellver, Xavier Giró-I-Nieto, and Jordi Torres. Distributed training strategies for a computer vision deep learning algorithm on a distributed GPU cluster. *Procedia Computer Science*, 108:315–324, 2017. ISSN 18770509. doi: 10.1016/j.procs.2017.05.074.

[33] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[34] Jeffrey Dean and Greg S. Corrado. Large scale distributed deep networks. In *NIPS*, 2012.

[35] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[36] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, and Quoc V Le. Large scale distributed deep networks. *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012. ISSN 10495258. doi: 10.1109/ICDAR.2011.95.

[37] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1223–1231, Red Hook, NY, USA, 2012. Curran Associates Inc.

[38] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[39] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, jun 2009. ISBN 978-1-4244-3992-8. doi: 10.1109/CVPR.2009.5206848. URL `http://ieeexplore.ieee.org/document/5206848/`.

[40] Li Deng and Dong Yu. Deep learning: Methods and applications. *FTSP*, 7(3–4):197–387, 2014.

[41] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[42] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. Supporting table partitioning by reference in Oracle. In *SIGMOD Conference*, pages 1111–1122, 2008.

[43] K. Eggensperger, M. Feurer, F. Hutter, J. Bergstra, J. Snoek, H. Hoos, and K. Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, 2013.

[44] Flexera. Cloud management. `https://www.flexera.com/blog/cloud/2018/02/cloud-computing-trends-2018-state-of-the-cloud-survey/`, 2019. Accessed: 2019-03-08.

[45] Eric Friedman, Peter M. Pawlowski, and John Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[46] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*, pages 209–219, 1986.

[47] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience. *PVLDB*, 2(2):1414–1425, 2009.

[48] Ahmad Ghazal, Dawit Yimam Seid, Alain Crolotte, and Mohammed Al-Kateb. Adaptive optimizations of recursive queries in teradata. In *SIGMOD Conference*, pages 851–860, 2012.

[49] Google AutoML. Google AutoML. https://cloud.google.com/automl/, 2018. Accessed: 2018-09-28.

[50] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994. ISSN 1041-4347. doi: 10.1109/69.273032. URL `https://doi.org/10.1109/69.273032`.

[51] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994. ISBN 0201558025.

[52] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902.

[53] Joseph M. Hellerstein and Michael Stonebraker. *Readings in Database Systems: Fourth Edition*. The MIT Press, 2005. ISBN 0262693143.

[54] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a Database System*. Now Publishers Inc., Hanover, MA, USA, 2007. ISBN 1601980787.

[55] Herodotos Herodotou, Nedyalko Borisov, and Shivnath Babu. Query optimization techniques for partitioned tables. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 49–60, 2011.

[56] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science (New York, N.Y.)*, 313(5786):504–7, jul 2006. ISSN 1095-9203. doi: 10.1126/science.1127647. URL http://www.ncbi.nlm.nih.gov/pubmed/16873662.

[57] HP- The Machine. HP-The Machine. https://www.hpl.hp.com/research/systems-research/themachine/.

[58] Hui-I Hsiao and David J. DeWitt. A Performance Study of Three High Available Data Replication Strategies. *Distributed and Parallel Databases*, 1(1):53–80, 1993.

[59] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. Flexps: Flexible parallelism control in parameter server architecture. *Proc. VLDB Endow.*, 11(5): 566–579, January 2018. ISSN 2150-8097. doi: 10.1145/3187009.3177734. URL https://doi.org/10.1145/3187009.3177734.

[60] Frank Hutter et al. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, 2011.

[61] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker, and Stanley B. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE*, pages 779–790, 2005.

[62] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[63] Zachary G. Ives, Amol Deshpande, and Vijayshankar Raman. Adaptive query processing: Why, How, When, and What Next? In *VLDB*, pages 1426–1427, 2007.

[64] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks, 2018.

[65] Peter H. Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. How to scale distributed deep learning? *CoRR*, nov 2016. URL http://arxiv.org/abs/1611.04581.

[66] Sunirmal Khatua and Nandini Mukherjee. Application-Centric Resource Provisioning for Amazon EC2 Spot Instances. In *Euro-Par*, pages 267–278, 2013.

[67] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks, 2014.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[69] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012. ISSN 10495258. doi: http://dx.doi.org/10.1016/j.protcy.2014.09.007.

[70] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *Int. J. Rob. Res.*, 34(4–5):705–724, April 2015. ISSN 0278-3649. doi: 10.1177/0278364914549607. URL https://doi.org/10.1177/0278364914549607.

[71] Alexandre A. B. Lima, Marta Mattoso, and Patrick Valduriez. Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster. *JIDM*, 1(1):75–88, 2010.

[72] Machine Learning Group at the University of Waikato. Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/, 2018. Accessed: 2018-09-28.

[73] Math Pages. Probability of intersecting intervals. http://www.mathpages.com/home/kmath580/kmath580.htm, 2018. Accessed: 2018-09-24.

[74] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques and tools. *CoRR*, abs/1903.11314, 2019. URL http://arxiv.org/abs/1903.11314.

[75] Microsoft CNTK. The Microsoft Cognitive Toolkit. https://www.microsoft.com/en-us/cognitive-toolkit/, 2018. Accessed: 2018-09-28.

[76] Marvin Minsky and Seymour Papert. *Perceptrons; an introduction to computational geometry*. MIT Press, 1969. ISBN 9780262130431.

[77] Bernardo Miranda, Alexandre A. B. Lima, Patrick Valduriez, and Marta Mattoso. Apuama: Combining intra-query and inter-query parallelism in a database cluster. In *Proceedings of the 2006 International Conference on Current Trends in Database Technology*, EDBT'06, page 649–661, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540467882. doi: 10.1007/11896548_49. URL https://doi.org/10.1007/11896548_49.

[78] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning, 2017.

[79] Guido Moerkotte. *Building Query Compilers*. University of Mannheim, http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf, 2009.

[80] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. Sparknet: Training deep networks in spark, 2015.

[81] Rimma V. Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *SIGMOD Conference*, pages 1137–1148, 2011.

[82] Thomas Neumann and Cesar Galindo-Legaria. Taking the Edge off Cardinality Estimation Errors using Incremental Execution. In *BTW*, pages 73–92, 2013.

[83] Nils Nilsson. *The quest for artificial intelligence: A history of ideas and achievements*. 01 2010. ISBN 9780521122931. doi: 10.1017/CBO9780511819346.

[84] Oracle. Oracle Rdb. https://www.oracle.com/database/technologies/related/rdb.html.

[85] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, 2011.

[86] M. Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. *ACM Comput. Surv.*, 28 (1):125–128, March 1996. ISSN 0360-0300. doi: 10.1145/234313.234368. URL https://doi.org/10.1145/234313.234368.

[87] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011. ISBN 978-1-4419-8833-1.

[88] O'Reilly Podcast. How to train and deploy deep learning at scale. https://www.oreilly.com/ideas/how-to-train-and-deploy-deep-learning-at-scale/, 2018. Accessed: 2018-09-28.

[89] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD Conference*, pages 61–72, 2012.

[90] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *EDBT*, pages 430–441, 2013.

[91] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. Automating physical database design in a parallel database. In *SIGMOD Conference*, pages 558–569, 2002.

[92] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's Adolescence. *PVLDB*, 6(10): 853–864, 2013.

[93] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. In *ICDE*, 2014.

[94] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 1939-1471. doi: 10.1037/h0042519. URL http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519.

[95] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[96] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.

[97] Abdallah Salama, Carsten Binnig, Tim Kraska, and Erfan Zamanian. Cost-based fault-tolerance for parallel data processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 285–297, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2749437. URL http://doi.acm.org/10.1145/2723372.2749437.

[98] Abdallah Salama., Alexander Linke., Igor Pessoa Rocha., and Carsten Binnig. Xai: A middleware for scalable ai. In *Proceedings of the 8th International Conference on Data Science, Technology and Applications - Volume 1: DATA,*, pages 109–120. INSTICC, SciTePress, 2019. ISBN 978-989-758-377-3. doi: 10.5220/0008120301090120.

[99] Kai-Uwe Sattler. *Distributed Query Processing*, pages 912–917. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_704. URL https://doi.org/10.1007/978-0-387-39940-9_704.

[100] D. Sculley et al. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2503–2511, 2015.

[101] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR*, feb 2018. URL http://arxiv.org/abs/1802.05799.

[102] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1–2):703–710, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920931. URL https://doi.org/10.14778/1920841.1920931.

[103] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, page 926–934, Red Hook, NY, USA, 2013. Curran Associates Inc.

[104] Srinivas Sridharan, Karthikeyan Vaidyanathan, Dhiraj Kalamkar, Dipankar Das, Mikhail E Smorkalov, Mikhail Shiryaev, Dheevatsa Mudigere, Naveen Mellempudi, Sasikanth Avancha, Bharat Kaul, and Pradeep Dubey. On Scale-out Deep Learning Training for Cloud and HPC. *CoRR*, pages 16–18, 2018.

[105] Thomas Stöhr, Holger Märtens, and Erhard Rahm. Multi-Dimensional Database Allocation for Parallel Data Warehouses. In *VLDB*, pages 273–284, 2000.

[106] ShaoJie Tang, Jing Yuan, and Xiang-Yang Li. Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance, booktitle = IEEE CLOUD. pages 91–98, 2012.

[107] ShaoJie Tang, Jing Yuan, Cheng Wang, and Xiang-Yang Li. A Framework for Amazon EC2 Bidding Strategy under SLA Constraints. *IEEE Trans. Parallel Distrib. Syst.*, 25(1):2–11, 2014.

[108] Nesime Tatbul, Yanif Ahmad, Ugur Çetintemel, Jeong-Hyon Hwang, Ying Xing, and Stanley B. Zdonik. Load Management and High Availability in the Borealis Distributed Stream Processing Engine. In *GSN*, pages 66–85, 2006.

[109] Chris Thornton et al. Auto-weka: combined selection and hyperparameter optimization of classification algorithms. In *SIGKDD*, pages 847–855, 2013.

[110] P.A. Tobias and D. Trindade. *Applied Reliability, Third Edition*. Taylor & Francis, 2011. ISBN 9781584884668.

[111] Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD Conference*, pages 241–252, 2011.

[112] Florian M. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *BIRTE (Informal Proceedings)*, 2008.

[113] Florian M. Waas. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database. In *BIRTE (Informal Proceedings)*, 2008.

[114] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

[115] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD Conference*, pages 13–24, 2013.

[116] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel Madden. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE*, pages 657–668, 2010.

[117] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *IEEE CLOUD*, pages 236–243, 2010.

[118] Sangho Yi, Artur Andrzejak, and Derrick Kondo. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE T. Services Computing*, 5(4):512–524, 2012.

[119] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13:55–75, 08 2018. doi: 10.1109/MCI.2018.2840738.

[120] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[121] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 17–30, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2723718. URL http://doi.acm.org/10.1145/2723372.2723718.

[122] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: parallel databases meet MapReduce. *VLDB J.*, 21(5), 2012.