Modern Data Analytics in the Cloud Era

Dissertation
zur Erlangung des akademischen Grades

Doktor–Ingenieur (Dr.–Ing.)

vorgelegt der

TECHNISCHEN UNIVERSITÄT ILMENAU
Fakultät für Informatik und Automatisierung

von
M.Sc. Steffen Kläbe

Gutachter:

1. Prof. Dr.–Ing. habil. Kai-Uwe Sattler
2. Prof. Dr. Tilmann Rabl
3. Prof. Dr. Carsten Binnig

Tag der Einreichung: 14. Dezember 2022
Tag der wissenschaftlichen Aussprache: 05. Mai 2023

DOI: 10.22032/dbt.57434
URN: urn:nbn:de:gbv:ilm1-2023000087

Abstract

Cloud computing has been the groundbreaking technology of the last decade. The ease-of-use of the managed environment in combination with nearly infinite amount of resources and a pay-per-use price model enables fast and cost-efficient project realization for a broad range of users. Cloud computing also changes the way software is designed, deployed and used. This thesis focuses on database systems deployed in the cloud environment. We identify three major interaction points of the database engine with the environment that show changed requirements compared to traditional on-premise data warehouse solutions.

First, software is deployed on elastic resources. Consequently, systems should support elasticity in order to match workload requirements and be cost-effective. We present an elastic scaling mechanism for distributed database engines, combined with a partition manager that provides load balancing while minimizing partition reassignments in the case of elastic scaling. Furthermore we introduce a buffer pre-heating strategy that allows to mitigate a cold start after scaling and leads to an immediate performance benefit using scaling.

Second, cloud based systems are accessible and available from nearly everywhere. Consequently, data is frequently ingested from numerous endpoints, which differs from bulk loads or ETL pipelines in a traditional data warehouse solution. Many users do not define database constraints in order to avoid transaction aborts due to conflicts or to speed up data ingestion. To mitigate this issue we introduce the concept of PatchIndexes, which allow the definition of approximate constraints. PatchIndexes maintain exceptions to constraints, make them usable in query optimization and execution and offer efficient update support. The concept can be applied to arbitrary constraints and we provide examples of approximate uniqueness and approximate sorting constraints. Moreover, we show how PatchIndexes can be exploited to define advanced constraints like an approximate multi-key partitioning, which offers robust query performance over workloads with different partition key requirements.

Third, data-centric workloads changed over the last decade. Besides traditional SQL workloads for business intelligence, data science workloads are of significant importance nowadays. For these cases the database system might only act as data delivery, while the computational effort takes place in data science or machine learning (ML) environments. As this workflow has several drawbacks, we follow the goal of pushing advanced analytics towards the database engine and introduce the Grizzly framework as a DataFrame-to-SQL transpiler. Based on this we identify user-defined functions (UDFs) and machine learning inference as important tasks that would benefit from a deeper engine integration and investigate approaches to push these operations towards the database engine.

Kurzfassung

Cloud Computing ist die dominante Technologie des letzten Jahrzehnts. Die Benutzerfreundlichkeit der verwalteten Umgebung in Kombination mit einer nahezu unbegrenzten Menge an Ressourcen und einem nutzungsabhängigen Preismodell ermöglicht eine schnelle und kosteneffiziente Projektrealisierung für ein breites Nutzerspektrum. Cloud Computing verändert auch die Art und Weise wie Software entwickelt, bereitgestellt und genutzt wird. Diese Arbeit konzentriert sich auf Datenbanksysteme, die in der Cloud-Umgebung eingesetzt werden. Wir identifizieren drei Hauptinteraktionspunkte der Datenbank-Engine mit der Umgebung, die veränderte Anforderungen im Vergleich zu traditionellen On-Premise-Data-Warehouse-Lösungen aufweisen.

Der erste Interaktionspunkt ist die Interaktion mit elastischen Ressourcen. Systeme in der Cloud sollten Elastizität unterstützen, um den Lastanforderungen zu entsprechen und dabei kosteneffizient zu sein. Wir stellen einen elastischen Skalierungsmechanismus für verteilte Datenbank-Engines vor, kombiniert mit einem Partitionsmanager, der einen Lastausgleich bietet und gleichzeitig die Neuzuweisung von Partitionen im Falle einer elastischen Skalierung minimiert. Darüber hinaus führen wir eine Strategie zum initialen Befüllen von Puffern ein, die es ermöglicht, skalierte Ressourcen unmittelbar nach der Skalierung auszunutzen.

Cloudbasierte Systeme sind von fast überall aus zugänglich und verfügbar. Daten werden häufig von zahlreichen Endpunkten aus eingespeist, was sich von ETL-Pipelines in einer herkömmlichen Data-Warehouse-Lösung unterscheidet. Viele Benutzer verzichten auf die Definition von strikten Schemaanforderungen, um Transaktionsabbrüche aufgrund von Konflikten zu vermeiden oder um den Ladeprozess von Daten zu beschleunigen. Wir führen das Konzept der PatchIndexe ein, die die Definition von unscharfen Constraints ermöglichen. PatchIndexe verwalten Ausnahmen zu diesen Constraints, machen sie für die Optimierung und Ausführung von Anfragen nutzbar und bieten effiziente Unterstützung bei Datenaktualisierungen. Das Konzept kann auf beliebige Constraints angewendet werden und wir geben Beispiele für unscharfe Eindeutigkeits- und Sortierconstraints. Darüber hinaus zeigen wir, wie PatchIndexe genutzt werden können, um fortgeschrittene Constraints wie eine unscharfe Multi-Key-Partitionierung zu definieren, die eine robuste Anfrageperformance bei Workloads mit unterschiedlichen Partitionsanforderungen bietet.

Der dritte Interaktionspunkt ist die Nutzerinteraktion. Datengetriebene Anwendungen haben sich in den letzten Jahren verändert. Neben den traditionellen SQL-Anfragen für Business Intelligence sind heute auch datenwissenschaftliche Anwendungen von großer Bedeutung. In diesen Fällen fungiert das Datenbanksystem oft nur als Datenlieferant, während der Rechenaufwand in dedizierten Data-Science- oder

Machine-Learning-Umgebungen stattfindet. Wir verfolgen das Ziel, fortgeschrittene Analysen in Richtung der Datenbank-Engine zu verlagern und stellen das Grizzly-Framework als DataFrame-zu-SQL-Transpiler vor. Auf dieser Grundlage identifizieren wir benutzerdefinierte Funktionen (UDFs) und maschinelles Lernen (ML) als wichtige Aufgaben, die von einer tieferen Integration in die Datenbank-Engine profitieren würden. Daher untersuchen und bewerten wir Ansätze für die datenbankinterne Ausführung von Python-UDFs und datenbankinterne ML-Inferenz.

Danksagung

An dieser Stelle möchte ich allen Menschen meinen Dank aussprechen, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Zuerst möchte ich meinem Betreuer Kai-Uwe Sattler danken. Kai gab mir bereits während meines Studiums die Möglichkeit an spannenden Themen des Fachgebiets mitzuarbeiten und Erfahrungen zu sammeln. Nach Abschluss meines Studiums ermöglichte er mir eine Promotion zu beginnen. In dieser Zeit durfte ich viele Menschen kennenlernen und in spannenden Projekten arbeiten. Ohne seine Ratschläge, Ideen und Motivation wäre diese Arbeit nicht möglich gewesen.

Darüber hinaus danke ich meinen Gutachtern Tilmann Rabl und Carsten Binnig für ihre Zeit zur Erstellung der Gutachten.

Besonderer Dank gilt der Firma Actian. Ein Teil dieser Arbeit entstand im Rahmen des Kooperationsprojektes “Readying VectorH for the Cloud” zwischen Actian und der TU Ilmenau. Nach meinem Wechsel zum Vector-Team von Actian bekam ich weiterhin die Freiheit selbstständig zu forschen und Ideen auszuprobieren und umzusetzen. Auch ohne diese Freiheit wäre diese Arbeit nicht möglich gewesen. Ich freue mich auf unsere weitere Zusammenarbeit.

Auch das tolle Arbeitsumfeld, das ich sowohl an der TU Ilmenau als auch bei Actian erleben durfte, hat einen wesentlichen Einfluss auf diese Arbeit. Hier durfte ich viele tolle Kollegen kennenlernen. Besonders möchte ich mich bei Stephan, Michael, Stefan und Philipp bedanken für die vielen konstruktiven Diskussionen, Mittagspausen, Kaffeerunden und Betriebsausflüge. Aber auch allen anderen Kollegen gilt mein Dank für das Schaffen einer motivierenden und freundschaftlichen Arbeitsatmosphäre.

Weiterhin möchte ich mich bei meiner Familie bedanken. Ihr habt mich immer unterstützt und mir ermöglicht, diesen Weg zu gehen. Dafür bin ich euch unendlich dankbar!

Zuletzt möchte ich der wichtigsten Person in meinem Leben danken, meiner Verlobten Luisa. Du warst immer für mich da, hast mir Rückhalt gegeben, mich motiviert, aufgemuntert und immer Verständnis gehabt. Ich freue mich auf unsere gemeinsame Zukunft.

Contents

Chapter 1
Introduction

1.1 Cloud Computing

Cloud computing gained extraordinary importance over the last decade with the advent of enabling technologies like distributed systems, virtualization or fast network technologies. Being at a market size value of around 480 Billion USD in 2022, the global cloud computing market is expected to rise up to a revenue of around 1.5 Trillion USD by 2030 [34]. Reasons for the success of the technology are the benefits cloud computing offers. Due to cloud service providers hosting large data centers, users can easily access a nearly inexhaustible amount of resources elastically. With this elasticity it is possible to flexibly adapt hardware requirements to the actual workload, i.e., acquiring additional resources to handle load peaks or even shutting down all resources during times where a service is not needed. In combination with a pay-per-use pricing model the cloud environment can also lead to reduced business costs. Another factor of success is the ease-of-use of cloud services. Cloud service providers offer a managed environment where users do not need to take care of underlying complexity of setting up hardware, software or typical tasks like networking. Instead, users can benefit from the guaranteed service availability cloud providers offer, and may even be protected against natural disasters when replicating over different geographical regions. This also holds for data, which can be automatically replicated and is therefore better protected against data loss due to hardware failures. With all this being transparently provided, cloud computing can also significantly lower the hurdle to start projects and therefore enable a larger group of users to benefit. Additionally, cloud computing makes it easy to access different hardware in order to optimize service performance or easily develop software for heterogeneous technologies. There are several compute instances available optimized for networking, CPU speed or main memory, or containing specialized hardware like GPUs or FPGAs. Overall, cloud computing consequently enables to easily get access to resources, flexibly adapt to changes in workloads or requirements with a convenient pricing model, compared to an on-premise setup where hardware needs to be bought, set up and maintained.

On the contrary, cloud computing also introduces new challenges. When using cloud computing, users need to trust in the cloud providers and the technology to keep data private and secure. Possibly confidential user data stored in the cloud means that it is accessible from everywhere protected by a cloud provider's encryption and access control. This fundamentally differs from an on-premise solution where data might be physically accessible only from a certain network or computer. Besides that, software design needs to consider the cloud computing characteristics. While an existing software can be deployed on hardware hosted in the cloud, this approach misses to really benefit from the advantages described above. The major point here is elasticity, which should be supported by the system in a native way to benefit

from the flexible hardware environment. As hardware is able to scale dynamically driven by the demand, software should also be able to do so, i.e., being able to add workers in order to increase parallelism to handle more load or decrease runtime. Additionally, software should separate storage and compute resources. With this, compute resources can be spun up, dynamically scaled and shut down while data is persistent in the storage, i.e., cloud file system.

1.2 Database Systems in the Cloud

In this thesis, we focus on analytical database management systems (DBMS) in the context of the cloud environment. According to the cloud computing reference architecture [26] by the National Institute of Standards and Technology (NIST) there are three major service models, which can be seen as abstractions how cloud services can be used. Infrastructure as a Service (IaaS) describes the model where generic resources like storage, networking or compute can be provisioned by the user, who is then able to deploy arbitrary operating systems and software on the infrastructure. Platform as a Service (PaaS) offers pre-installed applications that can be used for, e.g., software development tasks. These applications can include development kits, language extensions, web servers, etc. With PaaS it is possible to rapidly set up a working environment for different tasks. Software as a Service (SaaS) is the highest abstraction layer according to the NIST definition and describes a model where certain applications are deployed by the cloud provider and users can interact with the application over the application-specific interfaces using clients or web interfaces. A database system can be categorized as SaaS with users and clients interacting and querying the database using SQL and standard interfaces. In order to create a database instance in a SaaS model, a user would simply use a web frontend to select the initial compute resources, connect his storage bucket containing his data if available and the cloud environment would automatically provision hardware resources, install an operating system and deploy the database system on the hardware. Afterwards, the user can immediately start accessing the database and run queries without the need to manage the hardware environment, care about the storage or configure the DBMS. More recently, Function as a Service (FaaS) evolved as a fourth service model. Here a user does not even need to select the amount of resources, but simply runs a stateless function and pays by runtime or the number of function calls. Consequently, this model is even more convenient to use as the SaaS model. Database vendors like Amazon with Redshift Serverless [13] or Google with BigQuery [107] adapt this model towards a Database as a Service (DBaaS) model or "serverless" database. Users do not need to create DBMS instances but can simply send queries against data, retrieve results and pay per query or the amount of scanned data. Especially in analytics, where queries may read large amounts of data and perform expensive operations, this does not require users to scale the database instance to a reasonable size that is able to cope with the data volume and query complexity, and is therefore even more convenient to use. While database systems in a SaaS model may have stateful workers, i.e., explicit responsibilities for parts of the data and explicit catalog information maintained by the workers, the DBaaS model requires stateless workers. In the latter, workers do

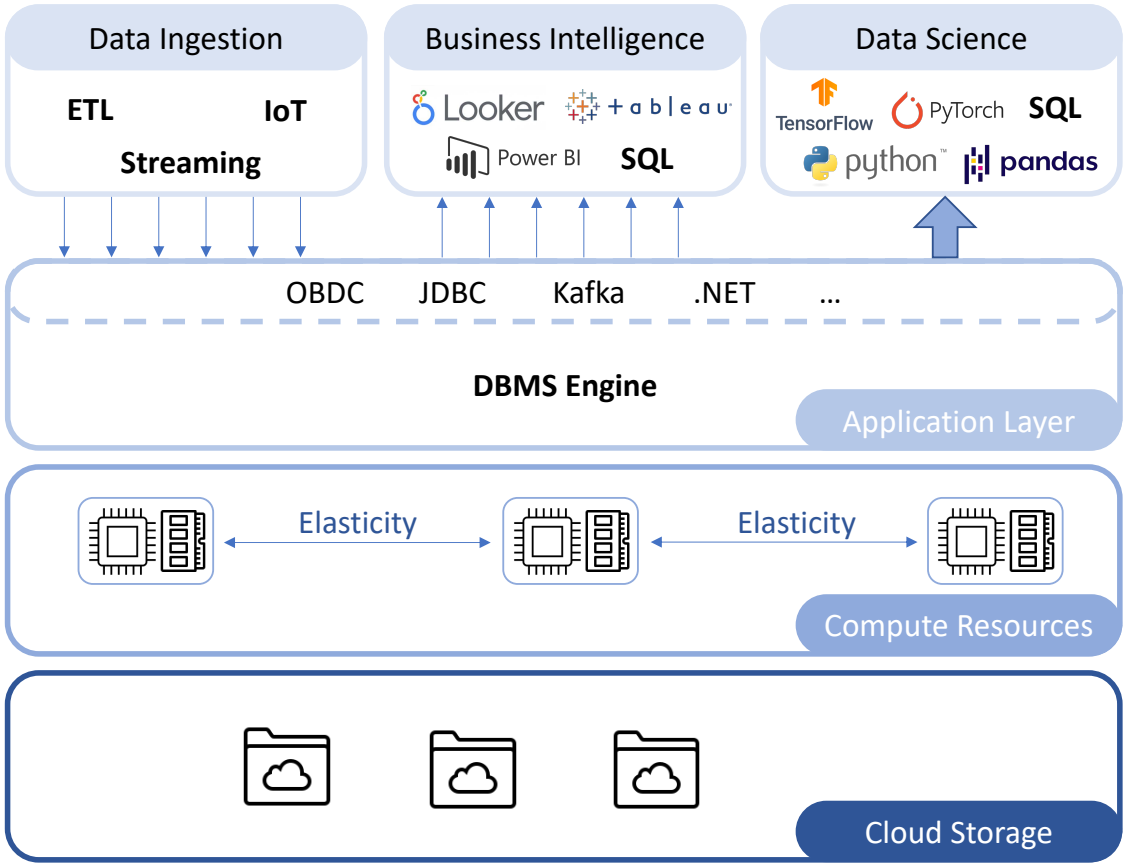


Figure 1.1: A database management system in the hierarchy of cloud computing layers

not store any catalog metadata but rather query a meta store during query execution or receive metadata attached to the task description.

Figure 1.1 shows how a DBMS interacts with different system layers in a cloud deployment in a SaaS or FaaS/DBaaS model. We can identify four major interaction points that lead to challenges in the design of a DBMS:

Compute Resources As desribed above, software and therefore also database systems must be able to exploit the elastic hardware environment. Based on the query load (either concurrent queries or very complex queries), the DBMS should be able to elastically scale. Data is stored in cloud storage that can be accessed by the elastic compute layer. As computing resources may also be shut down, this automatically enforces that the database system separates the compute logic from the storage logic.

Data Ingestion A major benefit of the cloud environment is the availability and accessibility of the system from nearly everywhere. Compared to traditional, on-premise data warehouse deployments where data is ingested using periodical bulk loads and ETL pipelines, this leads to more fine-grained, more frequent data ingestions. Cloud databases may serve as the data collection point for a huge amount

of devices producing data, like Internet-of-Things (IoT) networks with numerous sensors that produce and send data continuously. Consequently, the DBMS must be able to handle fine-grained data ingestions from numerous endpoints.

Business Intelligence Analytical database systems are used for business critical decision making processes. These processes typically need insights into a large data volume and results are aggregated to answer questions precisely. Queries for business intelligence are typically complex SQL queries written by data analysts or generated by business intelligence tools to produce dashboards. They are sent to the DBMS over standard interfaces and due to the need for precise answers, aggregated results are typically small and can be sent back to the client easily. These use cases do not differ from traditional on-premise data warehouse applications. Computing mainly takes place inside the DBMS engine, which has been optimized over decades to efficiently run these kinds of workloads. With more fine-grained and more frequent data ingestion, data freshness becomes a requirement in the business intelligence use case. Consequently, ingestion performance is an additional requirement for data ingestion.

Data Science Besides traditional analytical workloads expressed in SQL, more advanced workloads evolved in parallel to the evolution of cloud computing. Data science and Machine Learning (ML) workloads aim at finding hidden insights in the data by training models and using them for predictions on unknown data. These operations are not natively supported by traditional database engines. Consequently, modern analytics pipelines containing these kind of operations need to bulk download data from the database into a separate environment.

1.3 Contribution

The discussions in Section 1.2 showed that cloud computing not only changed the way database systems are deployed on hardware, but also changed the way how they are used and what they are used for. Based on these observations, this thesis provides the following core contributions:

1. **Elasticity.** We show how a database system can be scaled elastically during runtime and solve the challenge of adapting the state of workers using a partition manager that minimizes partition assignments, preserves partition locality and provides load balancing. Furthermore, we show how cold start problems after scaling can be mitigated using a buffer pre-heating strategy. The explanations for this contribution are based on [88].

2. **Data Ingestion.** Ingesting data from numerous endpoints can lead to low data quality, e.g., by violating constraints. In order to avoid failing ingestion tasks and skip expensive constraint checking in favor of data freshness, database schemas in cloud applications typically contain sparse schema information leading to non-optimal query performance. We focus on this problem of sparse schema information due to data ingestion from various endpoints

and present the PatchIndex approach that allows database systems to define approximate constraints, i.e., constraints that hold for almost all tuples. Consequently, constraints are allowed to be violated by the numerous ingestion tasks. PatchIndexes are generically defined and can be used to handle arbitrary constraints. They are easily updatable to perform data ingestions tasks efficiently and can be integrated into query optimization and query execution to increase query performance. We further show how PatchIndexes can be used to realize approximate sorting and approximate uniqueness constraints or to capture partition locality in an approximate multi-key data partitioning. The ideas behind these contributions are based on [83, 84, 85, 86, 82].

3. **Modern Workloads.** While traditional business intelligence workloads can be efficiently handled by database systems, more modern data science tasks are typically not supported and need to be performed in separate runtime environments. As this has several drawbacks, we investigate how to push modern workloads into the DBMS engine. First, we present the Grizzly framework that translates Python pipelines built on the popular Pandas DataFrame API into SQL and pushes these workloads towards the engine. Based on this, we identify user-defined functions (UDFs) and Machine Learning model inference as popular advanced analytics tasks and present approaches how to support Python UDFs and ML model inference natively by the DBMS engine. The elaborations for Grizzly are based on [65, 80, 64], while investigations on Python UDFs were published in [87] and in-database ML inference was examined in [81].

1.4 Outline

The remainder of this thesis is organized as follows. We start with a motivation of the contributions made in this thesis as well as a presentation of the Actian Vector Ecosystem in Chapter 2, where we provide an overview over the X100 query engine used to implement prototypes in this thesis and an overview over products built on the X100 engine. Chapter 3 focuses on the elasticity requirement. We describe an elastic scaling mechanism, a partition manager for the elastic environment and a buffer pre-heating strategy as an optimization to the scaling process. For the data ingestion requirement we present the PatchIndex approach in Chapter 4, which enables database systems to maintain approximate constraints. We describe the generic index design in Section 4.2, which can be applied to arbitrary constraints, and give examples of applying the idea to traditional database constraints in Section 4.3 or to the idea of an approximate multi-key data partitioning in Section 4.4. Chapter 5 shows a first approach to shift advanced workloads towards a DBMS. It introduces the Grizzly framework, which is a DataFrame-to-SQL transpiler that allows to push DataFrame operations towards the database system. We further utilize this framework to identify operations that are often included in typical modern data analysis pipelines. Based on these observations, Chapter 6 investigates approaches to natively integrate Python UDFs and ML model inference into the database engine in order to efficiently perform these advanced operations on data in the database. We thereby describe how Python UDFs can benefit from on-the-fly compilation, vectorization and parallelization and

compare approaches for in-database ML inference on different levels of abstraction, starting from a naive solution using Python UDFs up to a deep integration of a native model inference operator. In the final Chapter 7 we conclude our work and discuss open research questions in an outlook.

The chapters of this work have different views on the database engine within the cloud environment, similarly to the interaction points we identified before. We consequently also separated discussions about related work to these chapters and performed independent evaluations for our solution approaches to capture the use cases that are important for the respective features.

Chapter 2
Overview

2.1 Motivation

In the introduction in Chapter 1 we described that software can be easily deployed in the cloud environment, but needs to be rethought in different components to exploit the opportunities the cloud environment offers. In the following we therefore want to motivate the need for our contributions listed in Section 1.3. Figure 2.1 highlights the major interaction points of a database engine in the cloud environment and therefore is the foundation for the following discussion.

2.1.1 Elasticity

The first interaction point of the database engine shown in Figure 2.1 is the interaction with the elastic resource environment. Elasticity is a major benefit of cloud environments as discussed in Section 1.1. In contrast to scalability, which is a static system property describing the system's behaviour on a certain configuration [5], elasticity is a dynamic system property describing how a system is able to adapt to changing demands by automatically (de-)provisioning resources [71]. In on-premise deployments resources must be provisioned statically based on an observed load behaviour. As shown in Figure 2.2, scaling can be done in order to fit any expected load peaks (overprovisioning), leading to high costs and potentially unused resources. On the other hand scaling to match an average workload (underprovisioning) can lead to lower costs in exchange for potentially unmatched demands. While unmatched demands might be tolerable depending on customer service level agreements, choosing a good amount of provisioned resources is still a challenging task. With the flexible environment the cloud offers it is now the goal of elasticity to match the resource demand at any point in time while minimizing unused resources and consequently unnecessary costs.

In order to be a successful, cost-effective offer every system that is deployed in the cloud should exploit the elastic cloud environment. For a data-intensive system like an analytical database system supporting elasticity translates to solving the following general challenges:

1. **Support for runtime growing and shrinking:** Computing resources, e.g., cluster nodes, need to be added or removed efficiently with minimized system downtime.

2. **Providing mechanisms to rebalance load after elastic scaling:** Workload responsibilities must be balanced between computing resources after scaling. This might be realized in an explicit way in stateful systems by assigning responsibilities or implicitly by making a load balancer aware of the scaled topology in stateless systems.

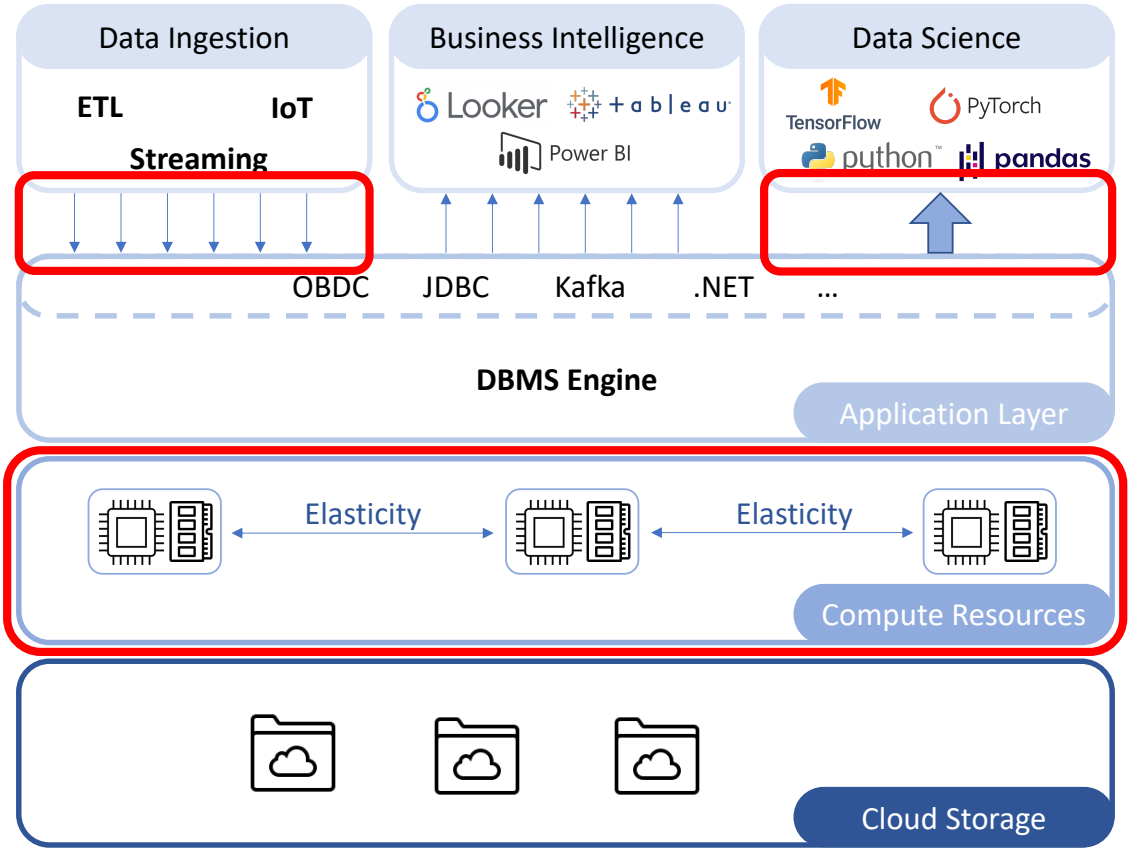


Figure 2.1: Database engine interaction points with the cloud environment

3. **Hiding effects of load balancing during elastic scaling:** Data-intensive systems make use of caching to achieve good query performance. When load is rebalanced, cached information might become invalid and performance might suffer from a cold start. This effect should be masked in order to make scaling transparent for the user.

While the first task is a technical issue and requires a solution tailored for the respective system design and technology, tasks 2 and 3 are generic issues and can therefore be solved in a conceptual way and applied to different systems.

2.1.2 Support for Approximate Constraints

The second interaction point highlighted in Figure 2.1 is data ingestion. In cloud environments datasets are likely to be unclean as data is ingested from numerous endpoints. Therefore it becomes challenging, if not even impossible, to define "perfect" constraints, i.e., constraint that match for all tuples, like primary keys, uniqueness constraints or sorting constraints. This results in database schemes with sparse schema information. As constraints are used in query optimization, e.g., selecting the actual join algorithm or estimating cardinalities of intermediate results, non-optimal query plans and non-optimal query performance are a consequence of sparse schema information. Additionally, even if constraints are defined, maintaining and enforcing

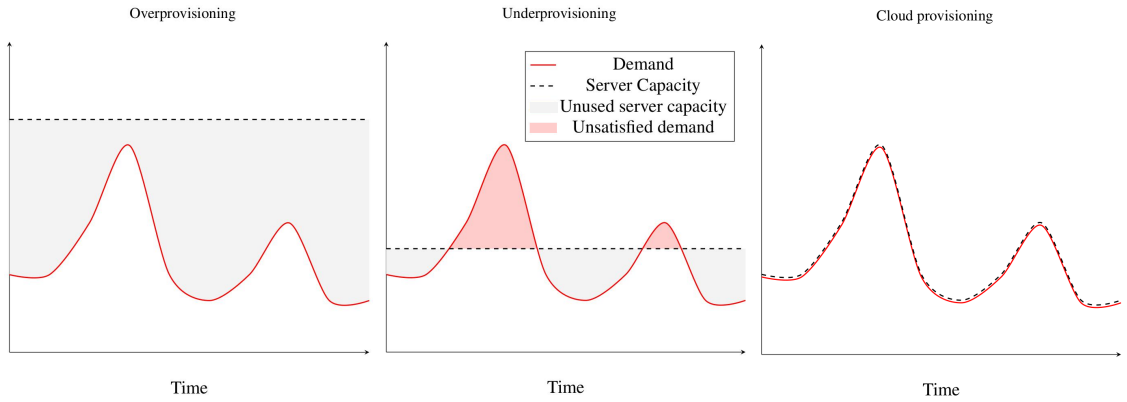


Figure 2.2: Resource provisioning for given load

them is expensive and therefore often avoided by users when running regular inserts or updates.

Besides potentially non-existing perfect constraints, these datasets may include approximate constraints, which are constraints that hold for nearly all tuples except a number of exceptions. These constraints occur for different reasons in big data environments, e.g.:

- **Data integration:** Data is integrated from various sources. This may lead to duplicates or an arbitrary order based on the order of integration.
- **Missing values:** Integrating different data schemas into one common schema may lead to NULL values.
- **Real world anomalies:** Real world datasets may not contain perfect constraints, e.g., equal names of persons, shared telephone numbers or shared addresses.

Approximate constraints contain valuable information that cannot be used for query execution, as the constraint definition is prohibited by a small fraction of the values. In order to prevent these information from being lost and to take advantage of them, allowing the definition and usage of approximate constraints is worth investigating.

Common real world use cases further underline the need for handling approximate constraints. In many business intelligence use cases, which are also listed in Figure 2.1, users combine analytical database systems with dashboard generation tools to simplify reporting, provide an overview over business data for management processes or utilize the built-in interactive query generators. While definitely increasing the usability of database management systems, these dashboard tools convert user interactions to SQL queries that are often very large and complicated in many cases. Furthermore these tools query characteristics of the data like distinct values to fill their APIs with e.g., drop-down selections, checkboxes, controllers or similar. An example query graph commonly generated by these tools is shown in Figure 2.3. Here, each subtree is a distinct query on an arbitrary column of the database. These queries would significantly benefit from any additional information on the processed data, like the described approximate constraints. Furthermore, many real world datasets

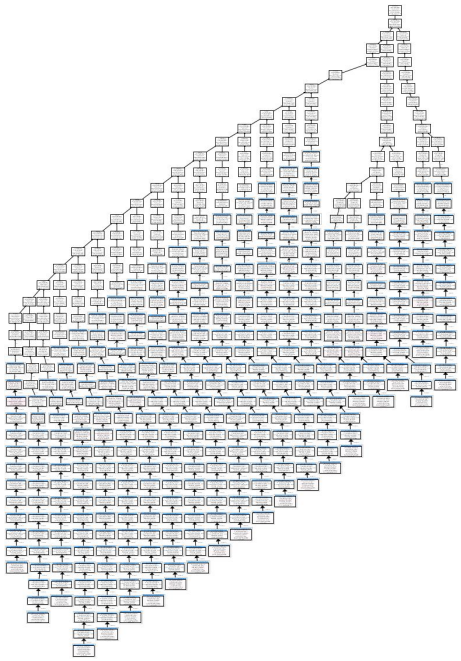


Figure 2.3: Overview over a query generated by a dashboard tool. Every branch is a distinct query. Graph generated using Actian Vector.

are timestamp-based, as they are generated by e.g., sensor networks. Not only being sorted on the timestamp, these datasets often show an approximate co-sorting of other columns and the timestamp column, e.g., auto-generated values, version numbers or increasing measurements. These nearly co-sorted columns also occur in sales datasets, e.g increasing order numbers leading to increasing order dates, or increasing order dates leading to increasing ship dates. For these cases, knowing about the approximate sorting of the columns would significantly accelerate sorting queries on these columns.

Overall data ingestion in the cloud environment shifts from traditional centralized ETL pipelines with bulk data loads towards more fine-grained and very frequent data ingestion from many endpoints. In combination with the lack of a database administrator caused by the ease-of-use of cloud database systems, database schemas in these environments often contain sparse schema information. While perfect data constraints might not exist in the data, approximate constraints can help increasing schema quality and accelerate query performance for different workloads.

2.1.3 DataFrame API

The cloud environment drives database systems towards being data platforms with a rich ecosystem. They are not only used through their SQL frontend for business intelligence anymore, but also with more advanced analytics tools with opportunities to define data processing pipelines in different ways. Seamlessly integrating with such tools in a convenient way is an important factor for the success of a cloud database system. Otherwise large amounts of data need to be downloaded from the database into separate environments to be processed. In comparison to business intelligence,

the database system here only acts as data delivery, wasting its capabilities for efficient query processing. Furthermore, data is not aggregated in these use cases and volumes can be very large compared to the small result sets of business intelligence queries. Compared to pulling data out of the database and executing the model in a separate environment, pushing these tasks into the database system has several advantages:

- **Reduced data transfer:** Instead of transferring large amounts of data to a client, data remains in the database system and only query results need to be transferred.
- **Exploiting server hardware:** Client hardware is typically weaker than server hardware. Hence, pushing query execution to the database server avoids expensive computations on a potentially weak client machine.
- **Scalability:** Database systems are designed to cope with large volumes of data and are optimized to handle larger-than-RAM data sets. Therefore, complex computations are placed best in this environment instead of handcrafting, e.g., buffering approaches in different external environments.
- **Query integration:** Once data is materialized to a separate processing environment, subsequent operations must also be performed there wasting efficient query processing capabilities of database systems.
- **Accessing sensitive data:** In some use cases data is not allowed to be moved out of the database system and can only be accessed under access control. Running model inference in Python might not even be possible in these cases. Pushing model inference into the DBMS enables subsequent operations, e.g., aggregations to only return aggregated and therefore non-critical inference results.

One of these frontends is Python, which became one of the most widely used programming language for Data Science and Machine Learning. According to the TIOBE index the languages popularity has steadily grown and was awarded language of the year in 2007, 2010, 2018, 2020 and 2021 [149]. The popularity obviously comes from its easy-to-learn syntax which allows rapid prototyping and fast time-to-insight in data analytics.

Python's success is also founded in the vast amount of libraries that help developers in their tasks. Nowadays, the most popular framework for loading, processing, and analyzing data is the Pandas library. Pandas had a huge success as it has connectors to read data in different file formats and represents it in a unified DataFrame abstraction. The DataFrame implementation keeps data in memory and comes with a variety of operators to filter, transform, join the DataFrames or executing different kinds of analytical operations on the data. However, the in-memory processing of Pandas comes with serious limitations and drawbacks:

- Data sizes are limited to the main memory capacity of the client machine, as there is no way of automatic disk-spilling and buffer management as found in almost every database systems.

- Even if the data to process resides in a database system on a powerful server, Pandas will load all data onto the data scientist's computer for processing. This is not only time consuming, but one can also assume that a companies sales table will quickly become larger than the memory of the data scientist's work station.

- Operations on a Pandas DataFrame often create copies of the DataFrame instead of performing the operation in place, occupying additional precious and limited memory.

In order to solve the memory problems, many users try to implement their own buffer manager and data partitioning strategies to only load parts of the original input file. However, we believe that scientists trying to find answers in the data should not be bothered with data management and optimization tasks, but this should rather be addressed by the storage and processing system. This motivates the need to seamlessly integrate a DataFrame API with database engines in order to push DataFrame operations towards the engine.

2.1.4 Python UDF and Machine Learning Support

Evolving a database engine towards a data platform can be accomplished by integrating tools and environments around the actual engine in a convenient way. Many workloads however also benefit from natively integrating them into the database engine. Similar to having a DataFrame API as a frontend solution to a database engine as described in Section 2.1.3, native support helps to push expensive computation towards the engine. We chose to investigate the native integration of Python user-defined functions (UDFs) and Machine Learning model inference as predominant operations in modern analytics pipelines.

UDFs play an important role in modern data analytics, as they offer a flexible, modular and reusable way to get business insights tailored for the user's needs. Many popular data warehouse systems integrate UDFs in different flavors to extend the functionality of the SQL frontend. Depending on the actual system, users can define UDFs using C/C++, Java, Python, Scala or Javascript code and use them in their SQL queries. Additionally, there are two types of UDFs, namely scalar UDFs, that produce a single result per input tuple, and table UDFs, that produce a new (potentially empty) table with an arbitrary number of results for a given input table. As described in Section 2.1.3 Python has been very successful due to its ease-of-use and wide range of available libraries. It is therefore used by a wide range of users and makes it less important to be a SQL expert to analyse data.

Offering Python UDFs increases a database system's usability for a wide range of users. However, the major bottleneck of UDFs is scalability, as shown in the analysis of [62], where UDFs take the major part of the query runtime. This especially holds for Python UDFs when integrated in a naive way due to several reasons. First, data needs to be converted from the internal database representation to a Python format and vice versa for the UDF results. Second, data needs to be transferred to the Python interpreter outside the database engine and results are sent back to the engine. Third, the interpreted code execution in the Python interpreter is typically

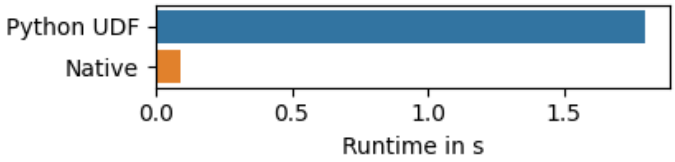


Figure 2.4: UDF vs. native performance for modulo operation on 8 million tuples

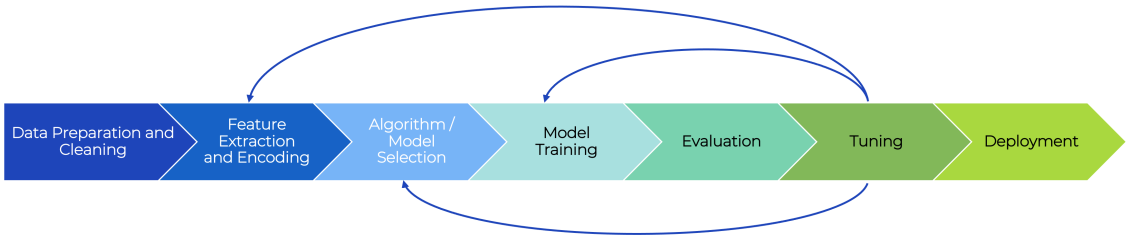


Figure 2.5: A typical Machine Learning Pipeline

slower than the execution of compiled and optimized code in the database engine. For operations that can be expressed both as a UDF and as a native database operator there is a large gap between the performance of a UDF and the respective native implementation of the operation, as shown in Figure 2.4. This motivates the need for more efficient execution opportunities for UDFs.

Besides user-defined functions, Machine Learning models play an important role in many modern data analytics pipelines. Machine Learning is a great tool able to discover unknown knowledge from datasets or capture correlations in high dimensional data that a human might not be able to grasp arithmetically. Figure 2.5 shows a typical ML pipeline, starting from data collection, feature encoding, model design and training towards evaluation and finally model deployment. Training a model that is suitable for a given problem is an iterative and time-consuming task. While the training process itself can be very long-running and hardware-intensive, the process of finding a suitable model might also consist of repeating preceeding steps to adapt the model architecture or the selected model features after evaluating the trained model. Consequently, this step is typically performed by ML experts.

On the other hand, deploying and using a model for inference can be done by arbitrary users. Trained models can be exported and shared with many users or even downloaded at market places or open-source repositories like the ONNX model zoo [118]. From a database engine perspective this makes ML model inference a good candidate for a more native integration into query processing withing the database engine.

2.2 X100 and the Actian Vector Ecosystem

Throughout this thesis we implemented prototypes of solutions to the problems presented in Section 2.1 into the X100 query engine [28] and used these prototypes to evaluate the solutions. However we would like to emphasize that the all approaches are generically applicable to arbitrary database systems. The X100 query engine has been one of the first columnar and vectorized relational execution engines. It originated from the Centrum Wiskunde & Informatica (CWI) Amsterdam and was spun off as the start-up Vectorwise in 2008. Since Vectorwise was acquired by the Ingres Corporation (later Actian) in 2011, X100 builds the base of an ecosystem of analytical database systems within Actian.

X100 was first integrated into the transactional database system Ingres [75] under the product name Vector. Vector is a single-server system that focuses on achieving multi-core parallelism. As the number of cores per CPU and per server is limited, multi-node systems of commodity hardware gained traction for big data analytics with the release of Hadoop [9] and Spark [11]. Consequently, Vector was extended to run on Hadoop clusters with the scale-out product Vector on Hadoop (VectorH in short) [36] starting in 2014. With the increasing importance of the cloud environment, VectorH was released as a cloud product called Avalanche starting from 2019, with recent releases focusing on supporting the cloud environment more natively.

In this chapter we present the core concepts of the X100 engine as well as the adaptions made to move it forward to scale out on clusters or to be deployed in the cloud. The aim of this chapter is to provide a common understanding of the engine's core design ideas. We consequently do not claim completeness of discussed concepts, but focus on areas that can help understanding the contributions of this thesis.

2.2.1 Storage Management

Columnar Data Layout X100 [28] and C-Store [146] pioneered the columnar storage layout for online analytical processing (OLAP) workloads. The traditional row-based layout has been the commonly used approach for online transactional processing (OLTP) workloads over decades due to its support for fine-grained update operations. With increasing data sizes and the need for data analytics the need for a read-optimized data layout rose. Storing tuples of a relation in a columnar fashion offers great opportunities for read-intensive OLAP workloads.

First, the columnar layout is read-optimized by design, as it enables to scan only necessary columns and exploit prefetching. OLAP queries frequently contain only a subset of columns of a table, which cannot be read separately in a row-wise storage layout. In contrary, scanning data from a columnar storage layout enables to utilize full bandwidth (both when accessing RAM or I/O) for relevant data, as pages only contain data of a single column. Second, the columnar layout offers great possibilities for data compression as shown in [167]. Columns can be compressed separately with compression algorithms tailored towards the data type, which leads to higher compression ratios compared to a row-wise layout and consequently further reduces expensive I/O effort.

As a downside, the columnar data layout requires an additional result materialization step to compose the initially decomposed columns of a query result, which is performed using a positional join. More severely, update support is a weakness of the columnar data layout. While inserting a single tuple leads to a single page access in the row-wise data layout, it leads to a page access for every table column in the columnar data layout. The PAX data layout [6] as a combination of a row-wise layout across pages but a columnar layout within pages tries to mitigate this issue.

Commonly, the update support issue is handled by holding a separate write-optimized update store. This update store is merged on-the-fly during query execution to achieve a consistent view of the input data. Updates are periodically propagated to the stable storage to reduce memory pressure as well as the effort for on-the-fly merging. While this update store might be a regular row-wise table store in common cases, X100 features positional delta trees (PDTs) [70] as a specialized update structure. Instead of maintaining track of tuple updates over their key values, PDTs organize differential updates by the tuple positions in a B-Tree-like data structure and thereby exploit the columnar data layout. PDTs are integrated into the table scan operators in X100 to efficiently and transparently merge updates on-the-fly.

Indexing Table indexing is an important database feature to identify relevant data for a given query. Pruning irrelevant data for queries containing predicates is crucial to reduce I/O effort and consequently improve query performance. X100 features two types of indexes: (co-)clustered indexes and Minmax indexes. A clustered index physically reorders data according to the specified key column. In X100 a clustered index is a SortKey, i.e., it enforces a physical sort order of tuples. If tables are connected over a foreign-key relationship and one table has a clustered index defined, X100 propagates the sort order to the connected table when creating a clustered index on the second table. Consequently, the tables are stored in a co-clustered fashion leading to efficient merge joins in query execution. As an example, a table clustered on a date column can determine the sort order of a foreign-key related table. Maintaining the sort order under updates is efficiently supported by using PDTs.

X100 does not offer tree-like index structures like B-Trees, which are typically used as clustered or non-clustered primary and secondary indexes. Instead, the engine uses Small Materialized Aggregates (SMA) [111], also called Minmax indexes or Zone Maps to achieve data pruning. Minmax indexes logically separate columns into buckets and maintain the minimum and maximum column value per bucket. By intersecting this range with either point or range predicates of a query it can be efficiently determined whether a bucket qualifies as input for a given query. In case of multiple predicates intersecting the chosen buckets can further reduce the amount of scanned data. The effectiveness of a Minmax index is dependent on the number of buckets as well as the data distribution. Regarding the latter, creating a clustered index on the table helps to achieve tighter ranges per bucket.

Minmax indexes are used in three different ways in X100, namely predicate pushdown, static range propagation and dynamic range propagation. We draw an example shown in Figure 2.6 of an accounts table with the respective Minmax index that logically separates the table into four buckets. Predicate pushdown is the

Accounts			
SID	**acctno**	**name**	**balance**
00	019	Isabella	269.38
01	038	Jackson	914.11
02	072	Lucas	346.61
03	153	Sophia	266.55
04	156	Mason	850.90
05	282	Ethan	521.60
06	302	Emily	647.38
07	314	Lily	119.40
08	332	Chloe	526.08
09	389	Emma	497.19
10	533	Aiden	22.03
11	592	Ava	140.67
12	808	Mia	383.69
13	896	Jacob	899.41

Accounts.MinMax							
bucket	**SID**	**acctno**		**name**		**balance**	
		min	**max**	**min**	**max**	**min**	**max**
0	00	019	153	Isabella	Sophia	266.55	914.11
1	04	156	314	Emily	Mason	119.40	850.90
2	08	332	592	Aiden	Emma	22.03	526.08
3	12	808	896	Mia	Jacob	383.69	899.41

Figure 2.6: Example of Minmax index for a given table

obvious way of evaluating point or range predicates of a query against the respective minimum and maximum values of the predicate columns to determine relevant data for a query. For an example query

```
Q := SELECT acctno FROM Accounts WHERE balance < 200
```

it is clear by comparing the predicate to the minimum and maximum index that only buckets 1 and 2 are relevant for answering the query, i.e., tuples with storage ID SID 4 to 11. We now nest query Q into a join with an arbitrary table like

```
J := SELECT * FROM Transactions T, (Q) as A WHERE T.acc = A.acctno
```

Static range propagation is a concept of pushing ranges produced by Minmax over equi-joins during the query optimization step. We already know from predicate pushdown that only buckets 1 and 2 of the accounts table qualify, which can be translated to *acctno* values in ranges [156, 314] and [332, 592] by again querying the Minmax index. These value ranges can be pushed to the transactions table on the join key to prune data that is ensured to not find a join partner. Dynamic range propagation goes one step further to push more detailed ranges over equi-HashJoins during query execution. If the accounts table is set to be the build side of the HashJoin, the scan of the transactions table is not started until the hash table build phase is finished. After building the hash table we have even more detailed information about the actually existing join keys. In the example only tuples with SIDs 7, 10 and 11 survive the filter and are inserted to the hash table, so only acc values of the transactions table in (314, 533, 592) need to be scanned, probably further reducing I/O effort.

Buffer Management Buffer management is an important task of database systems. It aims at keeping important pages in memory in order to reduce I/O effort necessary to execute queries. Database systems thereby do not rely on the operating system's paging algorithms but rather allocate the whole buffer memory on system start and maintain it on the application level. The main reason for this paradigm is that database systems have additional information about the page contents which allow to apply more sophisticated eviction policies than the operating system.

The main task of a buffer manager is to find an optimal page set for a workload, if the workload is known, or a page set that has a high probability to be useful for a predicted or expected future workload. The theoretical MIN replacement policy [19] achieves a provably [95] optimal page set for a given workload by evicting pages that have their next usage the farthest in the future. As this is not applicable for the common use case of an unknown future workload X100 uses the predictive buffer management (PBM) [147]. PBM predicts information about future page requests when long-running scans occur, while falling back to a traditional least recently used (LRU) strategy otherwise. It internally manages pages in priority queues of different importance.

Distributed Storage support Hadoop evolved to the de-facto standard for big data processing on commodity compute clusters in the mid 2010s. With the introduction of VectorH [36] X100 was extended with support for the Hadoop Distributed Filesystem (HDFS). HDFS combines storage devices of a shared-nothing compute cluster to a logical storage layer, where data is transparently accessible from every node either over local short-circuit reads or remote reads. This results in a scalable storage approach which is extended with fault tolerance by the HDFS replication policy, holding a total of three replicas per block by default distributed over different physical storage devices. X100 exploits this block replication and placement strategy to guarantee short-circuit reads while benefitting from the HDFS fault tolerance. Additionally, PDTs mitigate the issue of HDFS being an append-only storage system by design.

Cloud Filesystem Support The first version of Avalanche consisted of a VectorH deployment in the cloud environment, which consequently relies on storage tied to compute nodes. An important factor for the success of a cloud database system is the separation of storage and compute, i.e., storage and compute resources exist and can be scaled independently from each other. As a consequence, compute clusters can be stopped to save costs in the pay-per-use pricing model without loosing data persisted in the storage layer.

Cloud vendors offer persistent cloud storage like Amazon S3, Microsoft ABS or Google GCS. These storage systems are cheap and nearly infinite from a user perspective but differ from traditional file systems in their characteristics. They offer a PUT/GET API instead of POSIX file handling, weak eventual consistency but also high durability ($> 99.9999\%$ for S3) and disaster recovery across multiple global regions. From a performance perspective cloud filesystems have a high access latency and low single-threaded throughput, but can be used in a highly parallel fashion to achieve high throughput. X100 supports cloud filesystems for stable table storage. With aggressive prefetching and parallelism across nodes, partitions and columns it is thereby optimized to achieve very high scan throughput of tens of GB/s from cloud storage.

2.2.2 Query Processing

Execution model X100 follows the Volcano iterator model [56] for query execution, which is one of the predominant execution models besides compiled execution engines like Hyper [79]. Volcano describes an extensible schema to implement operators as iterators following a common interface. The interface consists of an *open()*, *next()* and *close()* operation that can generally encapsulate any operator logic. While *open()* initializes the operator state and may allocate required data structures, *close()* is called for query finalization in order to free allocated memory or export metadata like profiling information. The *next()* operation contains the actual operator logic. Queries are represented as trees with leafs typically being any scan or data input operators and inner nodes being algebra operators. Calling *next()* on an operator leads to *next()* calls on the operator's children and applying the operator-specific logic to the data chunks returned from the children, before returning the intermediate result to the parent operator. This leads to a pull-based execution model, i.e., iteratively calling *next()* on the root operator triggers query execution and produces the query result in a step-wise fashion. Operators in X100 are either unary (e.g., filters, projections, aggregations) or binary (e.g., joins), but Volcano also allows for more children (e.g., for multi-way joins).

Parallel Query Processing For parallel query execution X100 also follows the Volcano iterator model by achieving intra-operator parallelism using horizontal data partitioning. Consequently, multiple instances of each operator can be executed in parallel in order to exploit multi-core CPUs. Volcano additionally describes an exchange operator that allows to control parallelism. With this exchange operator X100 controls both the degree of parallelism as well as the data partitioning criteria, which makes exchange operators synchronization points between subqueries. For example, a query may scan a partitioned table in parallel, apply some filters and projections before reaching an exchange operator that repartitions the data on a given grouping key and at the same time reduces parallelism to finally execute an aggregation and return the result. Due to Volcano being an iterator model the query can be executed in a pipelined fashion, i.e., the aggregation might already start executing before the table scan is finished. Certain operators, we call them pipeline breakers, however require the full input dataset to produce any result, e.g., sorts, aggregation or hash table builds.

Vectorization The traditional Volcano iterator model is described to work in a tuple-at-a-time fashion, i.e., a *next()* call returns a single tuple to be passed to the next operator. Tuple-at-a-time processing leads to low instructions per cycle (IPC) due to call interpretation overhead and makes it impossible to use vector extensions for Single Instruction Multiple Data (SIMD) processing. On the other hand, column-at-a-time processing like MonetDB leads to high materialization overhead. As a compromise X100 uses vector-at-a-time processing [28], returning a vector of tuples per *next()* call, while still preserving the columnar layout. This way data independent operations like filters can be processed in parallel using vector extensions. In combination with the ability to exploit pipelining of modern CPUs this can lead to a typical IPC larger than

1. Data vectors are passed between operators as pointers which enables the results to be CPU cache resident when moving up in the operator tree and avoids fetching intermediate results from main memory. The vector size is chosen to match cache line granularity and typically set to 1024, which was experimentally determined as the sweet spot between column-at-a-time processing and tuple-at-a-time processing in [28].

Distributed Query Processing For VectorH and Avalanche the Volcano exchange operator is extended to a distributed exchange operator as described in [35]. Each node runs a X100 worker which receives an optimized query plan after the query optimization phase was finished on the leader node. Distributed exchange operators are the synchronization points for subqueries in these plans. Similar to regular exchange operators there are different types of distributed exchange operators like union or merge for reducing parallelism, broadcast for sharing intermediate results or hash splits for changing the partitioning on-the-fly.

For synchronization and data exchange among cluster nodes X100 uses the Message Passing Interface (MPI) [109]. MPI offers point-to-point communication as well as collective communication and is based on the concept of groups. Nodes within a group are identified using a rank starting at 0 and they are able to communicate with each other using a so called intra-communicator related to this group. In addition to that, so called inter-communicators allow communication among groups.

Chapter 3
Elastic Scaling

In this chapter we present solutions to challenges described in Section 2.1.1 and show how they are applied to the X100 query engine. The main contributions of this chapter are as follows:

- We show how X100 as a MPI-based system can be scaled elastically.
- X100 relies in data partitioning for distributed query execution. We present a partition assignment strategy suitable for the elastic environment. In particular, our partition assignment provides load balancing while minimizing partition reassignments.
- We descibe the buffer matching and filling approach which mitigates the problem of invalid data buffers and a cold start after elastic scaling by identifying important data and pre-heating buffers during scaling.

The remainder of this chapter is organized as follows. We discuss related work in Section 3.1 before presenting the design of the elastic cluster resize feature in Section 3.2. We investigate approaches for partition management and present our approach of a partition manager for the elastic environment in Section 3.3. Section 3.4 describes the cold start problem after scaling and the buffer matching and filling mechanism as a solution to the problem. Last, we evaluate the solution and show benefits of elastic scaling in Section 3.5.

3.1 Related Work

In the field of commercial systems, Snowflake [41, 155] was build from scratch for the cloud environment. It uses so called micro-partitions of several MB size to automatically partition and cluster data. Based on that, work distribution among nodes is realized using consistent hashing. In combination with work stealing, this approach automatically handles node failures as well as elastic scaling, while also minimizing the reassignment of micro-partitions as a property of the consistent hashing algorithm. For elasticity, Snowflake uses a multi-cluster architecture. It maintains a set of clusters of different pre-defined sizes to be able to instantly switch to a different cluster size. When buffered data is relocated, Snowflake does not immediately read the data from cloud storage, but does this in a lazy way. While the multi-cluster architecture provides good performance isolation, it leads to bad resource utilization and becomes cost-ineffective with the upcome of per-second billing. Therefore, Snowflake recently moved forward to resource sharing, which introduces new challenges like isolation and buffer sharing.

As a second example, Amazon Redshift [60, 13] divides compute nodes into the abstract concept of slices and assigns data to these slices. For scaling, the system offers

two possibilities. While the "classic resize" deploys a new cluster in the background and sets the system in a read-only mode for several hours, the "elastic resize" reduces the downtime by saving a snapshot to the cloud file system, adding/removing nodes and reassigning work by reshuffling the abstract slices between nodes. This way, the downtime for the scaling process is reduced to several minutes. Nevertheless, user queries are on hold during the scaling. The moved partitions are read asyncronously in the background into the buffers of the nodes. For more concurrency, Redshift is also able to attach more independent compute clusters to a single endpoint and balance concurrent queries between these clusters. Recently, Amazon also introduced a serverless option of Redshift. In this case, instances are automatically maintained, so users to not need to take care about instance sizes and configuration, but can start querying data immediately.

Google BigQuery [107] also follows the serverless approach. The compute layer Borg maintains virtual scheduling units. After query planning, these workers get a ready-to-execute query execution plan. This might be a partial plan and intermediate results are written to a persistent shuffle layer, where following query plans can read from again. Similarly, Microsoft Azure Synapse [15] also follows a serverless architecture and offers autoscaling.

The field of elastic DBMS design for the cloud is a fast-paced research area. While the main contributions of this chapter were introduced in 2019, the serverless architecture came up in the very recent years. In our approach presented in the following, we rely on the limitation that X100 has stateful workers managing catalog information and metadata about their assigned data partitions. Consequently, a serverless approach is not realizable without removing this limitation. We therefore focus on scaling an cluster of stateful workers on disaggregated storage, show how to minimize the partition reassignments and present an approach to pre-heat buffers after scaling.

3.2 Elastic Cluster Resize

In this section we describe the design of the elastic cluster resize feature for the X100 query engine. A solution to this problem is very system-specific and needs to be tailored to the system design and used technologies. X100 uses the Message Passing Interface (MPI) described in Section 2.2.2 for distributed communication, synchronization and processing. The engine was originally designed to be deployed on on-premise clusters based on a static cluster configuration. In order to scale the system, the cluster configuration has to be changed and the system has to perform a restart. This "inelastic" scaling approach has two major drawbacks. During the start process the system replays the write-ahead log, which might be a long-running operation depending on the log size. In addition to that, allocated memory is freed during the system shutdown. As a result, cached data in buffers is lost after a restart and data needs to be read again from storage. This cold start impacts the performance of the first queries. Therefore, we need a solution to scale a running MPI application without performing a full restart. We added an *add_nodes* and a *remove_nodes* system call encapsulating the scaling process.

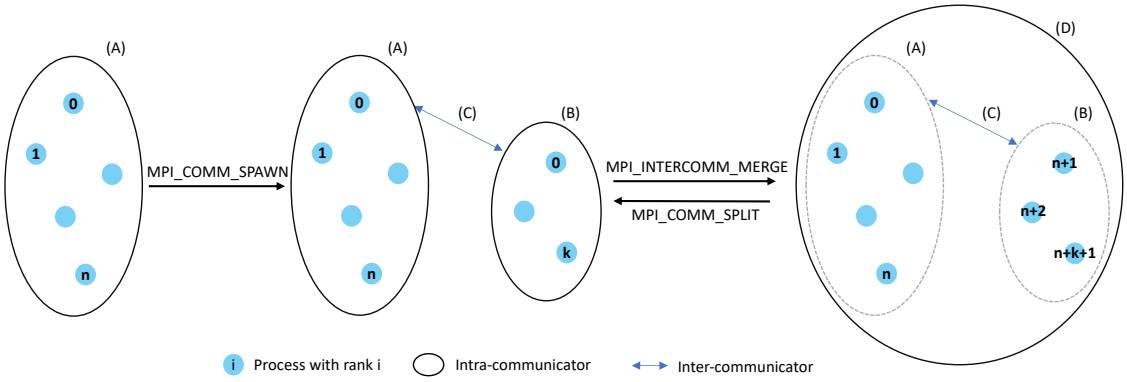


Figure 3.1: Schematic overview of cluster resize

The cluster resize feature exploits the opportunities of the MPI group and communicator management. The basic approach of adding nodes using MPI routines is illustrated in Figure 3.1. Starting from an existing group of current nodes with an intra-communicator (A), a new group of processes is spawned by starting the application on the newly provisioned cluster nodes using the *MPI_COMM_SPAWN* routine. All nodes of the new process group are able to communicate with each other using their own intra-communicator (B) and with the old group of nodes using the inter-communicator (C) returned by the MPI routine. In order to abstract from these different types of communication in a second step, both groups form a new intra-communicator (D) using the *MPI_INTERCOMM_MERGE* routine and replace group communicators (A), (B) and (C).

X100 workers are stateful, i.e., they maintain metadata about partitions they are responsible for. After new nodes are added in the *add_nodes* call, the leader node has to broadcast the current partition mappings as described in Section 3.3 in order to provide the new nodes with the correct initial partition assignments. Afterwards the current nodes have to assist the new nodes' startup, as there are various synchronization points within the startup procedure. To simulate a collective start of all servers, the current servers have to perform all of these synchronizations to make the added servers finish their initialization. The cluster resize feature has to be compatible with the following optimization made in X100. In order to reduce memory consumption, the creation of storage objects and metadata is skipped for partitions a node is not responsible for. After adding nodes, this responsibility assignment changes as some partitions of the current nodes get assigned to the newly added ones. As a result, current nodes hold structures for partitions they are not responsible for anymore. Each node checks for these kind of unused structures by iterating over all tables in their catalogs, dropping information and freeing memory whenever possible. This could also be done in a lazy way by checking for unused information within a partition responsibility check during query execution. But as these checks are very frequent operations and are called multiple times within each query, the decision was made to cleanup the unused structures directly as part of the *add_nodes* operation.

Removing nodes reverses the presented mechanism. In the first step the nodes are divided into two distinct groups S and R with corresponding intra-communicators

using a MPI_COMM_SPLIT routine. While nodes of group S perform a collective shutdown as a second step, the nodes of group R form the new cluster. Passing a list of hostnames to the function call, each node checks whether it is included in S and should terminate or not. It is currently not allowed to terminate the leader node, so *remove_nodes* returns an error in this case. The remaining nodes (group R) now update their partition mappings as described in Section 3.3. Removing nodes assigns more partitions to the remaining nodes, and, as the nodes were not responsible for these partitions before, the creation of storage structures and metadata was skipped. In order to reconstruct the missing information, an adapted replay of the write ahead log is performed. Within this replay, all log actions are skipped except those related to storage and metadata for tables the node is now responsible for and was not responsible for before. This information is provided by the partition manager.

With the design of the elastic cluster resize feature we achieved the possibility to scale the X100 cluster without restarting the existing nodes. We therefore ensured that the scaled system state is similar to the state before the scaling in terms of communication, metadata and catalog state. Furthermore, we automatically achieve support for in-memory updates that are resident in in-memory PDT structures mentioned in 2.2.1, as the log replay that is included in both *add_nodes* and *remove_nodes* also covers update log actions. As a result, the process reconstructs the update information needed to ensure data consistency.

Our implementation relies on the following assumption. For the *add_nodes* functionality binaries, configuration files and user data are accessible from the new nodes in the same way (especially using the same directory paths) as for the already running nodes. Triggering the operation over a front-end, e.g., a web-based management tool, would invoke starting the new nodes (dependent on the cloud service provider) and synchronizing necessary data before the *add_nodes* operation is started. A functionality for synchronizing the X100 installation directory already exists and is used for cluster deployment, so this assumption is no restriction.

Besides assumptions the presented implementation comes with some limitations. First, the resize functionality must be issued in a separate session with no concurrent sessions. The system is able to block/hold incoming session requests when a scaling request is made. As a consequence, the system can not be actively used during scaling. However, the elasitic cluster resize operation is expected to run much faster than a full restart with all consequences discussed above. Second, the hosts to be removed are restricted to be the ones with the highest ranks. As all nodes except the leader node are treated equally and the user does not call this function directly but through a frontend only providing the cluster size he wants to reach, the frontend can choose the nodes to remove as the ones that where added most recently. This ensures that only the highest ranks are selected. In case this is considered as too restrictive, one could extend the algorithm such that it first rearranges the nodes before the resize operation and assigns ranks in a way that fulfills the restriction. Third, it is currently not allowed to remove the leader node as worker nodes are not able to replace a missing leader node (which also holds for leader node failure).

3.3 Partition Management

In this section, we present a partition management approach that is suitable for the elastic cloud environment, which corresponds to the second goal stated in Section 2.1.1. Scaling can be invoked by the user to achieve one of the following goals. Either performance should be increased while keeping the data size fixed, or the system should be enabled to handle larger data sizes (while keeping performance constant). While the first goal refers to Amdahl's law, the second one is the use case for Gustafson's law (both in [63]). For X100 work distribution among cluster nodes is realized using partitioning and the workload assignment is static, i.e., each X100 worker has a fixed set of partition responsibilities. Consequently, the number of available partitions is important for the ability of the cluster to scale elastically. During scaling, we do not want to perform an expensive data repartitioning, but redistribute work by reassigning responsibilities.

The optimal number of partitions per table is approximately the total number of parallel threads the cluster offers. Assuming a homogeneous cluster, this is equal to the number of nodes multiplied with the number of physical cores per node. In order to support scaling while avoiding an expensive repartitioning we use the concept of overpartitioning, initially splitting tables into more partitions than necessary for the initial cluster configuration. The chosen number of partitions is crucial in terms of performance and the ability to scale the cluster size. Figure 3.2 shows the runtime of the TPC-H [27] query set on scale factor 1000 GB as a function of the number of partitions. The experiments were made on a cluster of 4 to 6 nodes with 24 cores each described in Section 3.5, providing a parallelism of 96 to 144 threads. First, the results show that increasing the cluster size while keeping the data size constant can lead to a performance benefit. Second, one can observe that increasing the number of partitions above the optimal partitioning of one partition per thread comes with an increasing performance penalty, which is caused by the introduction of an Union operator on top of table scans. Furthermore, increasing the number of partitions heavily impacts update performance, as update operations have to be performed on more fine-grained partitions. As a consequence, the number of partitions should not be chosen too large. Third, one can observe that X100 is able to handle underpartitioning to a certain degree by performing node-local splits during table scans, achieving the assignment of one partition per thread. However, this resplitting harms node-local join processing and is therefore not a desired behaviour.

3.3.1 Partition Assignment Approaches

We start by comparing basic approaches for partition assignment. The comparison is based on the following requirements, prioritized from most important to least important:

1. **Load balancing:** Assigning an equal number of partitions to each node is crucial to achieve an optimal query performance. As partitions are already built using the partitioning method, the assignment strategy can only affect the number of partitions per node, not the size of each partition.

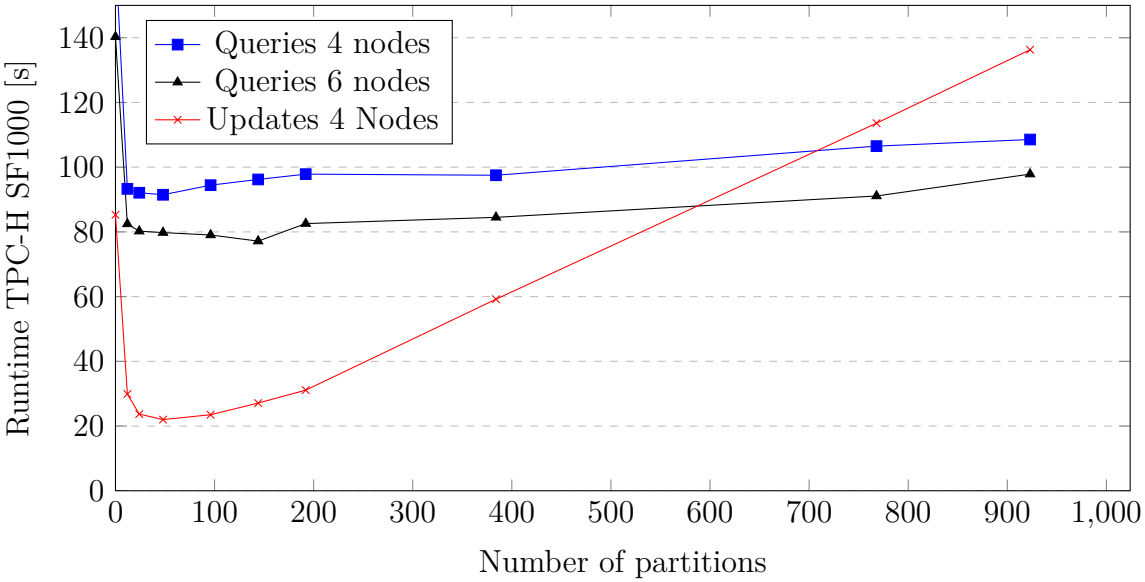


Figure 3.2: Dependency of Partitioning on TPC-H SF1000 runtime

2. **Lookup time:** The mapping *partition* → *node* is evaluated numerous times within each query and should therefore be an efficient operation.

3. **Update time:** As resizing the cluster changes partition assignments, the structures of the partition assignment should be updatable in an efficient way.

Besides these main objectives, the partition assignment strategies must fulfill the following side conditions for performance reasons:

- Keep co-locality of foreign-key related tables
- Minimize the number of reassigned partitions on cluster resize

Keeping the co-locality of related tables is a key for achieving optimal query performance by exploiting node-local joins and is therefore an important demand. Reassigning partitions has several effects and should therefore be minimized. First, it leads to storage access for reading the partition, as the data is not present in the node's buffer. Second, nodes have to update their catalog information when becoming responsible or loosing the responsibility for a new partition, as described in Section 3.2.

We can state a lower bound for the minimum number of partitions that have to be reassigned on cluster resize. Let d be the total number of partitions for an arbitrary table and we assume that partition assignment is balanced before a resize operation. When adding n nodes to an existing cluster of n_{old} nodes with $n_{new} = n_{old} + n$, it is clear that every node has to be responsible for $\frac{d}{n_{new}}$ partitions after resizing to achieve load balancing. We assume that n_{new} is a divider of d and if not, every node gets one additional partition until the remaining partitions are assigned. As every new node gets $\frac{d}{n_{new}}$ partitions, the minimum total number of reassigned partitions is $\frac{d}{n_{new}} \cdot n$. For removing n nodes from an existing cluster of n_{old} nodes it can be easily

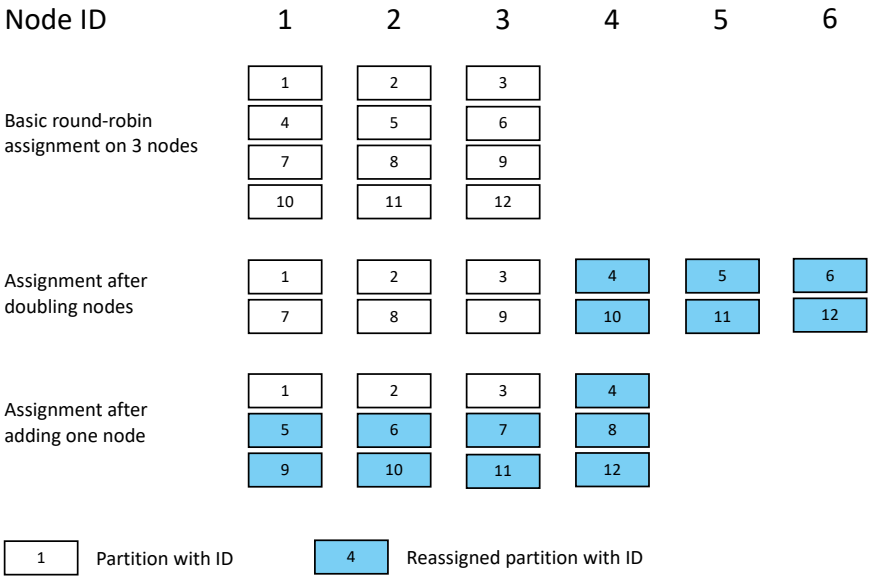


Figure 3.3: Round-robin partition assignment on cluster resize

seen that $\frac{d}{n_{old}} \cdot n$ partitions have to be reassigned, as every node was responsible for $\frac{d}{n_{old}}$ partitions before resizing. Overall we can state the lower bound of

$$reassigned\_partitions \geq \frac{d}{n_{max}} \cdot n$$

with $n_{max} = max\{n_{new}, n_{old}\}$ for adding/removing n nodes.

Round-robin assignment Round-robin assignment is the currently used assignment strategy in X100. It is clear that this strategy provides load balancing and as it can be evaluated using an arithmetic operation, has a very fast lookup time while not needing additional data structures. Nevertheless, with respect to the additional cluster resize functionality, round-robin is one of the worst choices as it reassigns nearly all partitions when not resizing the cluster with a factor that is a power of two, as shown in Figure 3.3. Doubling the number of nodes leads to a reassignment of half of the partitions, which is basically the minimum for achieving load balancing. But increasing the number of nodes by a factor being an arbitrary number and not being a power of two leads to a reassignment of nearly all partitions. Therefore, this strategy is not applicable for the elastic environment.

Consistent hashing The requirement of minimizing the number of reassignments on cluster resize directly leads to the method of consistent hashing, which places elements and buckets on a logical ring representing the set of hash values produced by the hash function. In order to improve load balancing, buckets can be replicated on the logical ring. It is shown that removing/adding one hash bucket leads to a reassignment of $\frac{k}{n}$ keys, with k being the total number of keys and n being the number of hash buckets [96]. With keys being partitions and hash buckets being nodes, this is the lower bound of reassigned partitions we stated before. If a node is

	Consistent hashing	**Partition manager**
Load balancing	Good, not guaranteed	Best possible
Partition reassignment	Minimized	Minimized
Lookup time	$O(\log(nodes \cdot replication))$	$O(1)$
Update	Re-sort array	Adapt every mapping $O(equi\_classes \cdot partitions)$
Memory consumption	$O(nodes \cdot replication)$	$O(equi\_classes \cdot partitions)$

Table 3.1: Comparison of partition assignment strategies

removed, only the partitions assigned to the removed node have to be reassigned and adding a node leads to a reassignment of all partitions between the added node and its last predecessor on the logical ring. Consistent hashing can be implemented by holding a sorted array or list of nodes, so a lookup operation to find a node responsible for a partition takes time $O(\log(n \cdot r))$ with r being the replication factor and using binary search on that list. Updating the number of nodes leads to resorting the list with a complexity of $O(n \cdot \log(n))$, for example using insertion sort for adding a few nodes or merge sort for adding a sorted list of nodes. Holding the list consumes memory in the size of $O(n \cdot r)$, which is independent from the number of partitions.

Explicitly storing and maintaining the assignment mapping This approach tries to provide a minimal lookup time and a best possible load balancing by explicitly maintaining and storing the mapping $[d] \to [n]$ from the set of partitions to the set of nodes for each possible partitioning using a partition manager structure. The mapping can be stored as an array with the size d, providing lookup time $O(1)$. Tables with equal number of partitions d_i build an equivalence class $i \in Eq$, so as a side effect, this guarantees co-location of foreign-key related tables when assuming them to have equal partition numbers (otherwise partition-local joins would not be possible anyway). Maintaining the mapping explicitly ensures that the best possible load balancing is achieved. As a drawback, this approach has a quite high memory consumption of $O(\sum_{i \in Eq} d_i)$, which is especially not independent from the number of partitions and increases with the number of distinct numbers of partitions. Nevertheless, the number of different partitionings and therefore the number of equivalent classes is typically small in user scenarios.

Comparison Table 3.1 compares the approaches of consistent hashing and partition manager. The partition manager approach outperforms consistent hashing in the most important categories load balancing and lookup time, while also minimizing partition reassignment. Assuming that the number of different partitionings is quite small and hence the number of equivalence classes is small, the time for updating the structure and the memory consumption is justifiable. As an example, a database consisting of 1000 tables sharing the same partitioning schema of 1000 partitions would lead to a memory consumption of around 5KB for 1000 4-byte-integer values and 1000 boolean values regarding the partition manager design shown in Section 3.3.2. Even for 1000 different partitioning schemas with a maximum of 2000 partitions

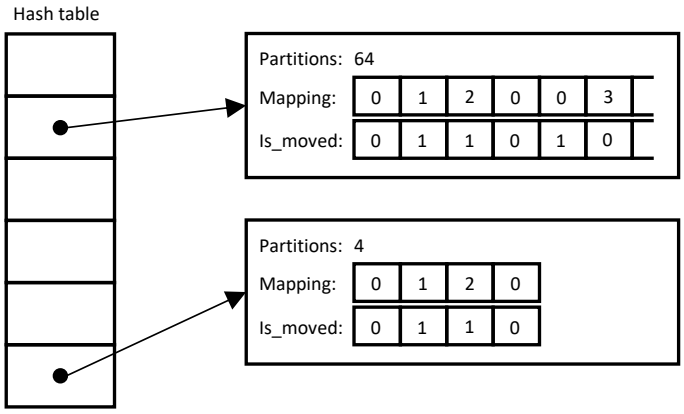


Figure 3.4: Partition manager overview

each we would get a few MB of memory consumption. Therefore, the decision has been made towards the partition manager approach.

3.3.2 Partition Manager Design

The partition manager structure, illustrated in Figure 3.4, realizes the approach of explicitly maintaining partition assignment mappings and maintains *partition mapping* objects for each equivalent class of table partitions, which consist of the number of *partitions*, a *mapping* array and an *is_moved* array, both of the size of the specific number of partitions. Each position i in *mapping* holds the node ID of the node responsible for partition i. The mappings are adapted during each cluster resize operation to maintain load balancing. In addition to that, the boolean *is_moved* value at position i indicates, whether the partition was moved during the last cluster resize operation, which is important to determine partitions to replay from the log when removing nodes or to delete the storage objects from when adding nodes. On top of the *partition mapping* objects, the partition manager maintains a hash table of *partition mapping* pointers to efficiently find the mapping for a given number of partitions.

For implementing a partition assignment, two assumptions are stated. First, it is assumed that co-local partitions have the same partition ID. This assumption is fulfilled by the hash partitioning method, as tuples with same keys (either primary or foreign keys) are mapped to the same hash value, which is used as partition ID. Second, it is assumed that tables with a foreign key relationship have the same number of partitions specified. Having an unequal number of partitions while using the currently implemented hash partitioning violates the requirement of co-locality. Especially, it is not ensured by the current hash function that having a table T with k times the number of partitions than it's join partner S results in a partitioning that maps one partition of S to exactly k partitions of table T.

Initialization: During server startup, the partition manager structure is initialized to a global variable by creating an empty hash table. When the partition manager gets queried for a partition mapping that is not present in its hash table, a

partition mapping object for the queried number of partitions is created and inserted into the hash table using the number of partitions d as key. The *mapping* is initialized using a round-robin strategy, so for partition ID i we get $mapping[i] = i \bmod d$. The choice of this initial strategy is arbitrary and could be replaced by any other strategy that provides a balanced assignment for n nodes. The *is_moved* array is initialized with *FALSE* at every position. The described process has time complexity $O(d)$ to initialize a single partition mapping.

Lookup: Knowing the structure of the partition manager, the lookup implementation is straight forward by a single hash table access and a single array access. Due to the design, the lookup operation has complexity $O(1)$.

Update: Whenever adding or removing nodes, all partition mappings have to be adapted. Therefore, we iterate over all entries in the partition manager's hash table and adapt every mapping using an algorithm divided into the following steps:

1. Compute the optimal load balancing for the new cluster state by computing *partitions_per_node* and a *remainder* if the number of nodes is not a divider of the number of partitions.

2. Compute a *diffs* array, with *diffs*$[i]$ indicating whether node i has to get additional partitions (positive entry) or get partitions removed (negative entry) to achieve the computed load balancing. The sum over all entries in the *diffs* array is 0, as the total number of partitions does not change.

3. Adapt the actual partition mapping by iterating over the *mapping* array. If we find a partition whose responsible node has a negative *diffs* entry, we move the partition to a node with a positive *diffs* entry and adjust the respective *diffs* entries.

With step three being the dominant step in terms of runtime, the algorithm runs in $O(d)$. As we adapt the mapping of every equivalence class, the update operation has a total runtime of $O(\sum_{i \in Eq} d_i)$.

Exchange: After new nodes are added to the system, they check during startup whether they are added to an existing cluster and if so, they prepare to receive the current partition assignments. The current nodes trigger this exchange functionality within the execution of the *add_nodes* query. The exchange operation is implemented using *MPI_BCAST* (broadcast), so it has to be performed collectively. After broadcasting the number of mappings, the number of partitions and the *mapping* array are broadcasted for each partition mapping. This way it is ensured that all nodes call the broadcast the same number of times.

As a result, the chosen approach minimizes the reassignment of partitions in the case of scaling while providing load balancing and efficient lookup and update functions.

3.4 Buffer Matching and Filling

We now motivate the problem of decreased performance after a cluster resize operation and present the design and the implementation of the buffer matching and filling

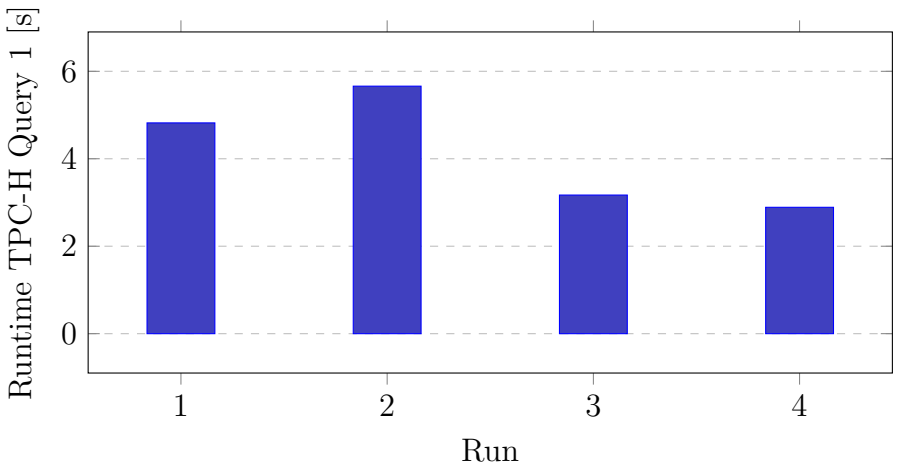


Figure 3.5: Runtime of TPC-H SF300 Query 1 in consecutive runs with run 1 on 8 nodes and runs 2,3 and 4 on 16 nodes after a cluster resize

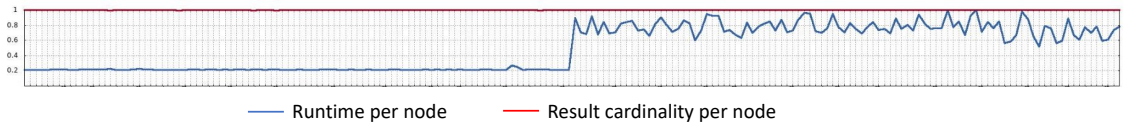


Figure 3.6: Qualitative TPC-H scan performance of query 1 for run 2

mechanism, solving the performance problem and therefore being a solution to the third goal stated in Section 2.1.1.

For a brief intermediate evaluation of the cluster resize functionality and the partition manager described in Sections 3.2 and 3.3, the TPC-H benchmark [27] was run on scale factor 300 using 8 of the 16 cluster nodes described in Section 3.5. Afterwards, the remaining 8 nodes were added and the benchmark was run again. Overall, it was observed that queries are slower after adding nodes than before. As an example, we pick query 1 of the benchmark, which is a selection and aggregation query on the lineitem table. The runtimes are shown in Figure 3.5. The query was run on 8 nodes with filled buffers (run 1) before adding 8 additional nodes and running the query several times again (runs 2, 3, and 4). Doubling the number of resources, we would expect a speedup up to factor 2, but the results clearly show an increase in runtime for the first run after the resize operation. However, the performance increases as expected with more consecutive query runs.

From a user perspective, this behavior is not satisfactory, as he pays for extra resources without getting any immediate benefit. The observed performance is a direct consequence of the buffer management. Figure 3.6 shows the qualitative performance (runtime and output cardinality) of the scan operation of TPC-H query 1 for all threads on all nodes. The right half of the plot, which covers the 8 nodes added by the cluster resize, shows a runtime that is up to 5 times slower with respect to the left half of the plot, which covers the 8 nodes that were already running before the cluster resize. This is caused by filled buffers of the old nodes, while the added nodes start with empty buffers. In order to smoothen this runtime, we present the buffer matching mechanism to fill the buffers of added nodes during the

cluster resize step. Doing so, we move the overhead of filling the buffer from user queries to the cluster resize operation, which is more justifiable to the user, as the elapsed time between the user toggling a cluster resize until finishing the resize also involves demanding new cluster nodes from the cloud service provider, which is also a potentially long-running operation and dependent on the actual provider. General approaches for buffer pre-filling must answer the following questions:

- Which data should be chosen to fill into the buffers?
- How is data brought to the buffers?

Especially the second question is important for the cloud setup. As cloud storage systems offer higher latencies than node-local storages, sending buffered data between nodes is also worth considering next to reading data from storage.

The chosen solution for this problem is our buffer matching mechanism. In order to discuss the mechanism from an abstract point of view, we identify a sender side and a receiver side, dividing the set of nodes into two distinct sets. Nodes of the sender side are characterized by losing the responsibility of partitions and having blocks in their buffer they are not responsible for anymore, while nodes of the receiver side become responsible for new partitions and do not have any buffered data for them. Note that because of our partition manager design in Section 3.3, a node is either a sender or a receiver. During data exchange, each node of the sender side can possibly have a connection to each node of the receiver side.

Block selection The first step of buffer matching is to identify for each node the set of blocks that needs to be sent to other nodes, as well as each block's specific destination. First we get a list of all blocks currently resident in the buffer memory sorted by importance. The importance of a block is determined by the actually used buffer replacement policy (PBM in the case of X100 as described in Section 2.2.1). In addition to that, we identify the destination of the blocks by querying the partition manager described in Section 3.3 to get the responsible node. Blocks that belong to a partition for which the node remains responsible are not sent and therefore dropped from the list. All other blocks are appended to a list of blocks per receiver, so as the result of the block selection step, each sender node holds a (potentially empty) list of blocks for each receiver node. One special case needs to be handled. Due to data distribution or due to buffering blocks of only a few partitions caused by selection predicates, it might occur that receiver nodes are intended to receive more blocks than they can actually fit into their buffers. The calculated cardinality difference is balanced between all senders to this receiver node and each sender is informed about the number of blocks to send before starting to send data. As the block lists are sorted by importance, the sender just drops the end of the list in this case.

Data exchange We now want to answer the question how buffer data is brought to nodes. As reading data from cloud storage might be slow compared to usual local disks or network transfer, the decision was made towards explicitly sending data to nodes over the network. The implemented data exchange mechanism follows three basic steps:

1. Exchange the number of blocks to transfer between each sender and receiver node.

2. Exchange block metadata.

3. Exchange buffer content.

The first step is important to establish synchronization between senders and receivers. Each node of the receiver side has to know about the number of blocks to receive from each node of the sender side. After the block selection step, each sender node holds a list of blocks per receiver. The length of these lists is shared with the respective receivers using MPI_GATHER routines, called within a loop over all added nodes. A receiver node with rank i becomes the receiver of the MPI_GATHER call in exactly one loop iteration. In this round, all other nodes send the length of their list i, indicating the number of blocks to send to node rank i. As a result, node i has the complete information about the number of blocks to receive after success of loop iteration i. In the second step, we transfer the blocks metadata (e.g., used bytes, columnID or the ID of the commit creating the block) to the receiver nodes. It is important to note that metadata and actual data of a block cannot be transferred all-in-one using a single MPI call by default, as metadata and actual data are not placed in consecutive memory areas due to the VectorH buffer management, which preallocates the whole buffer memory during startup. Constructing an additional structure holding both metadata and buffer data would lead to copying major parts of the buffer, which is not desirable. Sending the metadata is important for two reasons. First, the respective block can be searched in the receiver's catalog. X100 uses a replicated catalog, so a each block is already created on the receiver side. Second, the metadata contain information about the buffer content, like the actual data size or a flag to indicate whether data has been changed. This information is created during data load, so it needs to be sent as we do not load data from storage. Sending the metadata arrays is realized using non-blocking MPI point-to-point communication. This eliminates the need for explicit synchronization in this step. After receiving the metadata, each receiver performs catalog lookups to get pointers to the block structures and demands memory in the buffer memory for each block, before the blocks are inserted into the buffer replacement policy. In the third step, we transfer the actual buffer data. As the data for blocks is not placed in consecutive memory areas, they have to be sent one-by-one. Using non-blocking MPI communication like in the second step would therefore lead to k communication calls per sender and receiver with k being the number of blocks to transfer between sender and receiver, making it difficult to handle for the MPI environment when scaling the problem up. Therefore, we use synchronous, blocking communication for this step. To avoid deadlocks and reduce waiting time, we handle the communication in a multi-threaded way. For each point-to-point connection between a sender and a receiver node, having a non-zero number of blocks to transfer, both sender and receiver node open a separate thread running their side of the communication, resulting in a communication network. Each thread blocks until the respective counterpart of the communication is called. Upon receiving buffer data for one block, the data is copied to the buffer memory of the receiver node.

Fault tolerance The buffer matching mechanism is an addition to the cluster resize functionality. The success of the buffer matching step is not indispensable for the success of the whole cluster resize operation, but should not lead to an undefined state on the occurrence of errors. Therefore, the mechanism is designed to be fault tolerant. We distinguish between different times of error occurrence. If an error is detected before the block metadata in the catalog of a receiver is changed, we can simply perform a collective abort, as no durable changes were done yet. This is realized using a synchronization point between the second and the third step. If an error occurs during data exchange within third step, we have to ensure that the system handles the block's buffer content in the right way. After receiving a single block's data and copying it to the buffer memory, we verify the correctness of the data using an already existing magic number in the block's data. This magic number is a fixed constant which is used to discover transmission failures. This mechanism could be further improved using a checksum. If the verification succeeds, the block is flagged to be in memory. If the verification fails or an error occurs during communication, the current block is flagged as "LOAD", leading the system to not use the buffer content before an IO-thread loads the data from storage (and adjusts the metadata again). Furthermore, all pending blocks that have not been transferred yet are also flagged as "LOAD". All other communication threads are not affected and may succeed.

Integration The described mechanism is integrated into the *add_nodes* and the *remove_nodes* call. For adding nodes, the sender side is formed by the current nodes, as they lose responsibilities for partitions and may have buffered data for them, while all newly added nodes form the receiver side. During the system startup of the added nodes, buffer matching is integrated after the partition mapping exchange and the log replay, but before the server is able to handle user connections. This way the server already has the full catalog information. The current nodes perform buffer matching after following the server startup communication of the new nodes. For scaling the cluster size down, the removed nodes form the sender side of the buffer matching mechanism, while the remaining nodes form the receiver side. In order to enable removed nodes to determine the target of their buffered blocks, they update their partition mappings according to Section 3.3. Afterwards, they perform buffer matching while the remaining nodes perform it during execution of the *remove_nodes* call. In order to isolate the buffer matching communication from all other communication, a separate MPI communicator is build, being only valid during the buffer matching step. Finishing the buffer matching step, this communicator is destroyed. To provide the user the possibility to toggle the buffer matching mechanism on/off, an additional parameter is introduced into the X100 configuration API.

Optimizations After describing the basic ideas behind the buffer matching mechanism, we want to introduce two additional optimizations to the concept: the deletion of unused blocks at the sender side and the use of an alternative data exchange implementation. The strategies of the buffer policies are designed to keep the most important elements in the buffer by using priority queues. After performing buffer

matching, blocks a sender node is not responsible for anymore may remain in its buffer queues. Due to the behavior of the strategies, these blocks are displaced at some time in the future. Nevertheless, a block may remain a long time in the queues once it is categorized as very important, depending on the actual displacement strategy. As a consequence, this buffer page is useless for a long time, blocking possibly important blocks from finding their way into the buffer. Therefore, we explicitly drop sent blocks from the buffer replacement policy on the sender side.

The third step of the buffer matching mechanism uses blocking MPI calls to send the block's data one-by-one, as the buffered data of multiple blocks are not placed in consecutive memory areas (in this case, one call pointing to the start of the memory area would suffice). The MPI environment can be configured to use several communication protocols and uses the Transmission Control Protocol (TCP) in the X100 integration. Therefore, each call to send/receive a data block invokes communication setup, as well as the common TCP slow start phase, which is unnecessary overhead. As an optimization, we introduce a second, socket-based data exchange implementation. Similar to the described mechanism in the third step of the data exchange step, each sender-receiver pair with non-zero number of blocks to be transferred opens a thread on each side. Instead of starting MPI communication, the nodes establish a TCP stream socket connection. The receiver node creates a socket, sends the socket address information to the sender node using MPI and listens for an incoming connection. The sender node connects to the socket and sends data over the socket. As this single connection keeps alive until all data is sent, the overhead of communication establishment and slow start phase is reduced compared to the MPI implementation. In order to provide the same level of fault tolerance, each side of the socket checks the socket status using *select* before sending/receiving data. Furthermore, data blocks can be send in chunks, and only after a full block is received, the receiver verifies the block. Similarly to the fault tolerant behavior, blocks are flagged on connection or communication errors.

3.5 Evaluation

During the evaluation, we want to prove the superiority of the implemented cluster resize feature over the inelastic way of scaling. Furthermore, we want to show that the usage of the buffer matching and filling mechanism improves query performance after a cluster resize operation and is, therefore, a useful extension. The initial evaluation was done on a private cluster of 16 commodity nodes, each consisting of an AMD Opteron Processor 3380 @2600MHz with 4 modules of 2 cores each, 32 GB DDR3 RAM, 3.5 TB disk space, distributed among 4 HDDs and running CentOS-7 64 Bit. The nodes are connected over a 1GBit/s ethernet connection and Hadoop 2.7.1 is installed on the cluster. In addition, the TPC-H benchmark [27] on scale factor 1000 GB provided test data and test queries. This benchmark covers a well-understood synthetical workload in order to evaluate and compare data warehouse solutions with a dataset inspired by real world applications. The large tables of the benchmark are partitioned into 192 partitions, which is an overpartitioning for the cluster of 16 nodes with 8 cores each.

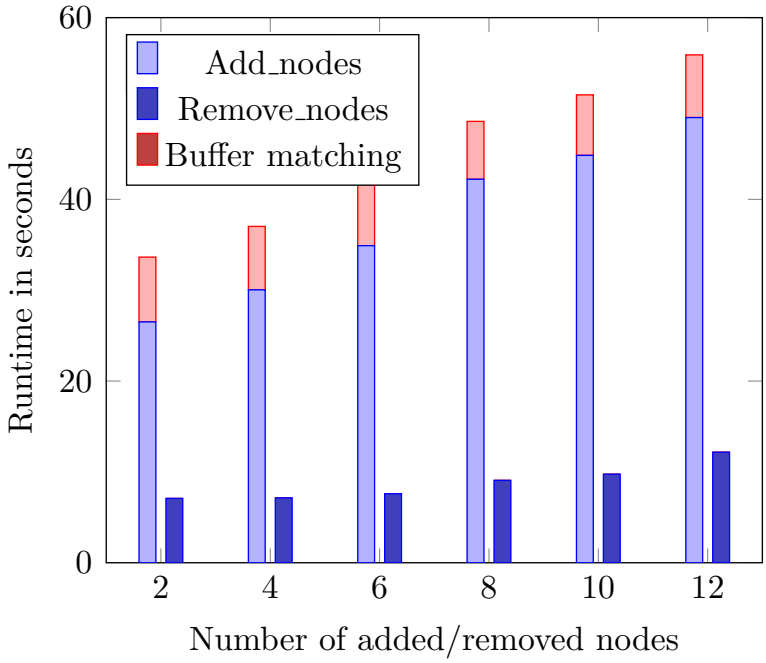
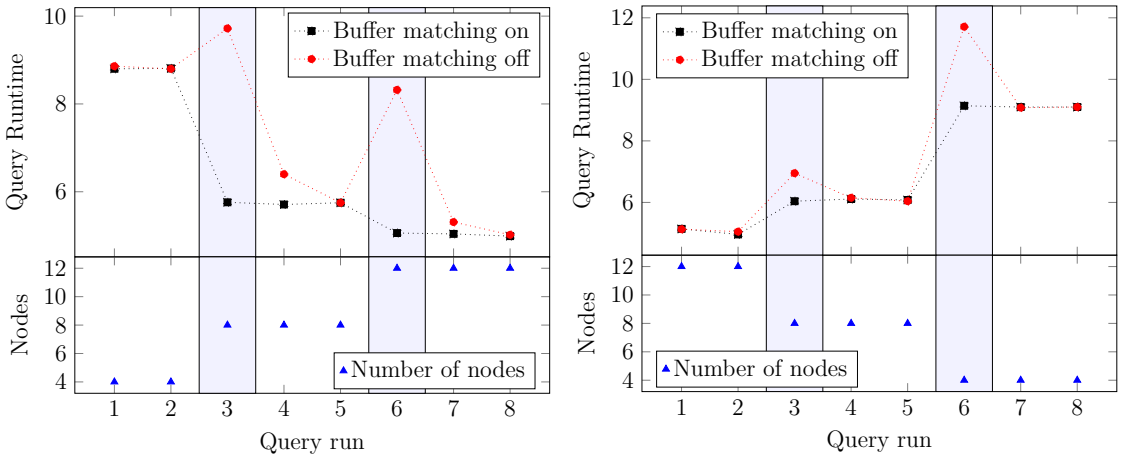


Figure 3.7: Scaling performance for adding and removing nodes

Scaling performance: The first experiment evaluates the performance of the implemented cluster resize feature. For the investigation of the upscaling process we start with a cluster of 4 nodes with filled buffers and vary the number of added nodes, while we start with 16 nodes and vary the number of removed nodes for the downscaling process. For these experiments, we keep the size of the bufferpool at 10 GB and the block size at 1 MB. Figure 3.7 illustrates the results of the experiment. One can observe that the runtime for adding nodes without using buffer matching increases in a linear way with the number of added nodes. The main reason for this behavior is the collective startup of the nodes. Starting more nodes at the same time increases the impact of the various synchronization points within the startup process. The buffer matching mechanism adds a nearly constant overhead to the measured *add_nodes* runtime. In a more detailed consideration one can observe that the buffer matching mechanism shows its minimum runtime when adding 8 nodes. Adding more nodes also increases the buffer matching data exchange parallelism (the number of receiver nodes per sender node), so the minimum runtime is expected to be at the number of physical cores, which is 8 in the used hardware setup. The downscaling runtime shows a slight increase when removing more nodes, which is caused by the adapted log replay the remaining nodes have to perform. Within this step, removing more nodes leads to more log entries that have to be replayed in the remaining nodes. The buffer matching mechanism was not applied for the downscaling process, as buffers where totally filled before scaling, so there was no buffer space left in the remaining nodes to receive blocks from removed nodes. Overall we can state that downscaling takes significantly less time than upscaling, because the synchronization effort for downscaling is lower. Once the nodes are split into two groups within the *remove_nodes* process, the group of removed nodes can simply perform a shutdown.



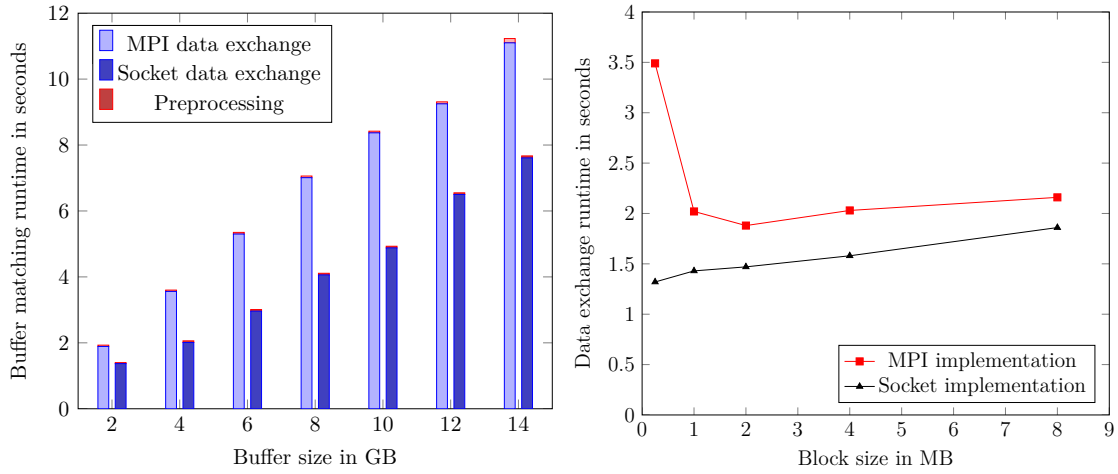
(a) Add_nodes

(b) Remove_nodes

Figure 3.8: Scaled query performance (buffer matching impact highlighted)

Scaled query performance: This experiment investigates the impact of the cluster resize operations on query performance. For this we repeatedly run a query while changing the cluster size between runs. As the main impact is expected to be within the scan operators, we use a query scanning two columns of the lineitem table. In order to fit the data into the buffer of the smallest cluster configuration, we limit the number of tuples to 500 million using a selection predicate, while setting the buffer memory to 20 GB per node. This way we avoid I/O access that would create an unintended bias in the measurements. Moreover we minimize the network traffic by applying an aggregation on each column, which is executed locally on each partition and results in a single tuple that needs to be send to the leader node.

Figure 3.8 show the measured query runtimes. Regarding the case of upscaling in Figure 3.8a, we can observe that adding nodes accelerates the query runtime as expected. However, the behavior varies between different buffer matching configurations. With activated buffer matching, query runtime drops and stays on the same level for the respective query runs after cluster resize, because the buffers already contain the needed data. On the contrary, not using buffer matching leads to significantly slower queries, especially for the first run after a cluster resize. The reason for this behavior is that added nodes have to read data from storage. In the following runs the query performance improves as buffers of added nodes fill. For the case of downscaling, we have to distinguish between two use cases. On the one hand, the reason a user triggers a downscale operation can be that the system is in an overprovisioning state, so the system underutilizes the provided hardware. In this case, removing server capacity should not have an impact on the systems performance. For X100 we neglect this case as we assumed an overpartitioning at every point in time. If this condition is violated, a repartition operation is triggered by the system. On the other hand, the user could invoke the downscaling to save costs while accepting slower system performance, e.g., when the expected load becomes less during specific times of a day. This case is expressed by Figure 3.8b. When scaling down the cluster, we can observe a performance degradation, again varying between buffer matching configurations. Similar to the upscaling case, the runtime



(a) Depending on buffer size (1 MB blocks) (b) Depending on block size

Figure 3.9: Buffer matching performance with constant buffer size of 10GB when upscaling from 8 to 16 nodes

of the first query run after cluster resize is significantly slower when not using buffer matching, as remaining nodes become responsible for data of removed nodes and have to read it from storage. With activated buffer matching, data is sent to the remaining nodes, leading to an immediately fast runtime after resize. Overall, this experiment proves that using the buffer matching mechanism during cluster resize perceptibly increases query performance after resizing.

Buffer matching performance: In this experiment, we want to evaluate the buffer matching performance for both implemented data exchange mechanisms, using MPI and using data streams over sockets. The two main parameters that have an impact on the buffer matching performance are buffer size and block size. While the buffer size affects the amount of data that is shipped during the buffer matching data exchange, the block size affects the granularity of shipped blocks and as a result also the communication overhead. Trying to touch as much data as possible, we use query 9 of the TPC-H benchmark for this experiment, as it touches five of the 8 tables in the benchmark and scans about seven billion tuples for the used scale factor of 1000 GB.

The plot in Figure 3.9 shows the runtime of the buffer matching mechanism as a function of the buffer size per node or the block size for both data exchange implementations. For these measurements, we used the upscaling step from 8 to 16 nodes. Figure 3.9a show that the data exchange takes the major part (about 98%) of the whole buffer matching step. Furthermore, the plot illustrates the expected runtime increase when increasing the buffer size and shows that they correlate in a linear way. Comparing both data exchange implementations, we can observe that the socket implementation is faster in terms of runtime and also shows a smaller grow in runtime when increasing the buffer size. As discussed in Section 3.4, this behavior is presumably caused by the fact that the MPI implementation has to re-initiate the communication for each block, as the blocks may be randomly placed within

the buffer memory space. The socket implementation on the other hand initiates the communication once before sending a data stream and therefore reduces the communication overhead.

In a further experiment we evaluate the impact of varying block sizes on the buffer matching mechanism. For this we keep the buffer size constant at 10 GB and we only consider the pure data exchange runtime, as we have seen in Figure 3.9a that preprocessing only takes a minor part of the overall runtime. Increasing the block size implies an increase of the necessary memory to allocate blocks. Therefore, we had to switch to scale factor 300 GB for this experiment as the overall available memory did not suffice for the largest tested block size of 8 MB and scale factor 1000 GB. Figure 3.9b shows the results of this experiment. For both implementations we can observe a slight increase in runtime when increasing the block size. This is caused by increased data volume that has to be exchanged, as larger block sizes come along with larger unused space or padding. Besides that, the socket implementation is not heavily impacted by varying block sizes, as data is simply written to stream sockets not considering any block boundaries. On the contrary, the MPI implementation profits from larger block sizes, as the communication overhead shrinks with the decreasing number of blocks to be sent. As a result, the difference in runtime between both implementations also shrinks with larger block sizes. Nevertheless, the choice of the block size also impacts other parts of the system, so this choice is usually fixed around a value of 1MB and can not be changed after database creation. For these block sizes, the socket implementation is surely the better choice compared to the MPI implementation.

Cluster resize usability: In the last experiment, we compare the implemented cluster resize functionality with the "inelastic" scaling, which involves the following steps:

1. Shutdown of the system
2. Adjustment of a list that holds the X100 node names
3. Restart of the system
4. Run a query

These steps are encapsulated in a script to reliably measure the runtime. We define the start state of the experiment as a running X100 system with filled buffers. Furthermore, we define the end state as the moment we get a query result from a scaled X100 instance. The runtime between start and end state is measured for the cluster resize feature with and without activated buffer matching, as well as for the inelastic scaling process. As query workload we choose query 1 and query 9 of the TPC-H benchmark. The buffer size is set to 10 GB per node and we investigate the cases of scaling from 8 to 16 nodes and vice versa.

Table 3.2 shows the measured runtime results of the experiment. For all cases, the implemented cluster resize feature outperforms the inelastic scaling up to a factor 4. In addition to that, activated buffer matching shows a benefit in runtime for the cases of scaling the system up, caused by the buffer pre-filling. The amount of benefit

	add_nodes		remove_nodes	
	Q1	Q9	Q1	Q9
BM on	57.74 s	77.55 s	32.02 s	51.06 s
BM off	58.70 s	87.95 s	32.56 s	52.04 s
Inelastic	167.68 s	223.30 s	123.20 s	147.94 s

Table 3.2: Runtime for scaling the system using the inelastic scaling process or the cluster resize feature with and without buffer matching (BM)

is dependent on the actual query for this experiment, so query 9 shows a better speedup than query 1 using buffer matching. For the case of removing nodes, buffer matching has a negligible impact, as buffers of the remaining nodes are already filled. Therefore, a buffer merging strategy could be a possible optimization in the future. Moreover, we can state that adding nodes to the system is slower than removing nodes, both for inelastic scaling and the cluster resize feature. The reason for this is that started servers perform a collective startup with several synchronization points and do a full log replay. Increasing the number of servers, this startup time also increases, leading also the inelastic scale-up to be slower than the scale-down. In the contrary, removed nodes can shutdown independently after the remaining servers form a new communicator (see Section 3.2), not influencing the further query processing.

The evaluation is highly dependent on the cluster configuration (e.g., network speed) as well as the size of the write-ahead log that has to be replayed. Therefore, this experiment is intended to show a qualitative difference between the scaling methods. The scaled query performance experiment was done again before the elastic cluster resize feature got released in the product. The TPC-H benchmark on scale factor 1000 was run on a cluster of eight virtual machines in the Google Kubernetes Engine (GKE), consisting of 16 CPU cores and 256 GB RAM each. We take query 1 as an example. Starting from a hot cluster of two nodes a node was repetitively added with the query being run two times in between to capture performance directly after scaling. Figure 3.10 shows the results of the experiment. Similar to the results presented before, we can observe that buffer matching has a huge impact on query performance after scaling. As nodes are substantially faster in this experiment, I/O overhead of reading data from storage has an even higher relative impact when buffer matching is disabled. With enabled buffer matching, query performance improves smoothly with added nodes which was the goal of the feature. We can also observe that scaling itself is significantly faster than in the initial experiments on the private cluster of commodity hardware. As a result the buffer matching runtime is now a significant part of the overall scaling runtime, compared to Figure 3.7. Scaling runtime decreases with increasing cluster size due to the increasing parallelism in buffer matching. Consequently, the decision towards buffer matching is not as clear as before. The mechanism still hides the cold start problem after scaling in the scaling runtime, which might be better justifiable towards a user instead of slow performance after scaling. However, in this experiment on more modern hardware it introduces more runtime to the scaling operation than it reduces runtime for subsequent queries. The reason for this is the improved I/O bandwidth of this hardware, so reading from

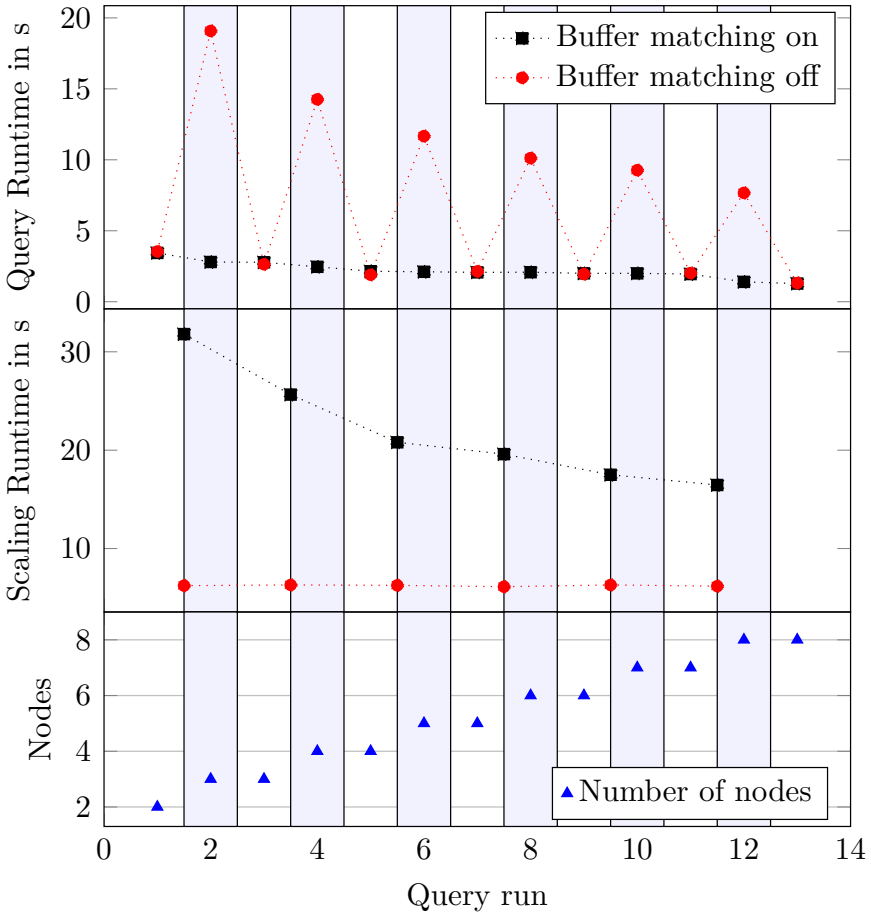


Figure 3.10: Updated scaling experiment in cloud environment. TPC-H SF1000 query 1. Buffer matching effect after scaling highlighted.

storage is not significantly slower than performing buffer matching anymore. In the release version of the feature, the third step of the buffer matching mechanism (data exchange) was adapted towards an asynchronous block prefetching mechanism from storage.

Overall, the evaluation proves that the implemented cluster resize feature significantly outperforms the "inelastic" scaling method using a restart while also enabling further optimizations like an incremental log replay or buffer pre-heating in order to achieve optimal query performance after scaling.

3.6 Conclusion

Adapting a system to exploit the elasticity of cloud hardware is a major factor for the success of the system as a cloud product. As the first goal, we implemented an elastic cluster resize feature for the X100 query engine, enabling adding and removing nodes during system uptime and therefore avoiding the drawbacks of a full system restart, e.g., full log replay and empty buffers. For the implementation of the feature we utilized group and communicator management offered by the Message

Passing Interface (MPI), which is used for node-to-node communication within X100. Second, we designed a partition manager that is suitable for the cloud environment. By using overpartitioning and explicitly managing partition-to-node mappings for equivalence classes of partitionings, the implemented solution minimizes partition reassignments, keeps partition co-locations, balances load on partition level and provides efficient lookup and update functions. The partition manager replaces the round-robin partition assignment in X100, which showed to be not suitable for the elastic cloud environment. While evaluating the cluster resize feature in combination with the implemented partition manager, queries did not show the expected speedup immediately after the resize, which was caused by empty buffers. As an optimization, we introduced the buffer matching and filling mechanism into the cluster resize feature. After changing partition mappings, nodes scan their buffers and send buffered data to other nodes in order to pre-fill their buffers, leading to immediate speedup after cluster resize. For the buffer matching data exchange, we implemented two different approaches using MPI communication and using data streams over sockets and evaluated them against each other.

The experiments showed that the elastic cluster resize feature significantly outperforms the inelastic scaling process using a system restart. Activating the buffer matching mechanism further increases the performance after the cluster resize by hiding the effort for buffer pre-heating in the scaling runtime. This enables users to immediately profit from additional resources. A repeated experiment on recent hardware however showed that the buffer matching mechanism adds more runtime to the scaling process than it reduces runtime in query execution. The relative impact on scaling is significantly higher compared to the first experiments, caused by the scaling process itself taking much less time in modern cloud environments.

Chapter 4
Database Support for Approximate Constraints

In order to exploit approximate constraints in analytical databases we propose to handle tuples violating constraints separately to accelerate query execution. In this chapter, we therefore introduce the generic PatchIndex structure supporting arbitrary approximate constraints. PatchIndexes allow to materialize information about tuples violating given constraints and query them separately during query execution. Constraints can be maintained under regular inserts or updates, which is a common case for data applications running in the cloud. Even perfect constraints are allowed to become approximate over time. The main contributions of this chapter are as follows:

- We describe the design of the generic PatchIndex structure and consider design decisions regarding memory footprint, update support and efficient query integration using the PatchIndex scan. The PatchIndex approach can be applied to arbitrary constraints by implementing a defined interface.

- We apply the PatchIndex approach to sorting and uniqueness as traditional database constraints, explain how these constraints can be discovered and describe possibilities to benefit from the approximate constraints in query optimization and query execution.

- We show that not only traditional database constraints can benefit from PatchIndexes by applying the approach to the problem of multi-key data partitioning. We therefore relax the problem to a patched multi-key partitioning problem and provide a possible solution for the problem based on PatchIndexes.

The remainder of this chapter is organized as follows: We discuss related work in Section 4.1. Section 4.2 describes the basic design of the index structure and discusses design choices regarding the underlying data layout. It presents the PatchIndex scan and the interface to apply PatchIndexes to arbitrary constraints. In Section 4.3 we introduce the definition of “Nearly unique columns” and “Nearly sorted columns” as examples for approximate versions of traditional database constraints. We provide approaches to discover the constraints and incorporate them in query optimization to benefit from the approximate constraint information. Section 4.4 defines the problem of a patched multi-key partitioning and describes how the problem can be mapped to traditional graph partitioning algorithms and a solution can be achieved by using PatchIndexes.

4.1 Related Work

Materialization The materialization of information is a widely investigated topic in the database research field, particularly in the context of materialized views. While significantly impacting query performance [23], the major drawback of materialized views is their update support. For the subset of select-project-join (SPJ) views there are efficient ways to support table updates and keeping views consistent [21, 22, 61, 150]. They typically try to avoid base table access by determining update sets and performing differential update operations. Besides that, materialized information should be chosen based on the expected benefit, so intermediate results of frequent queries are a good candidate for materialization. A benefit-based approach for choosing intermediate results is given in [113], while [66] also includes the probability of a materialized result to be usable in future queries to the cost model. Nevertheless, these approaches do not take updatability into account.

Patch Processing PatchIndexes are a generic approach to materialize arbitrary constraints. The approach is based on the concept of patch processing, which is commonly used in compression. The PFOR, PFOR-DELTA and PDICT compression schemes [167] show robustness against outliers by handling exceptions to certain distributions separately. Furthermore, white-box compression [52] learns distributions and properties of data in order to choose appropriate compressing schemes. Here, tuples that do not follow a certain behaviour can be compressed using another compression scheme than remaining tuples, significantly improving compression rates as a result. These approaches modify the way data is physically stored to handle exceptions, which significantly differs from our approach. By not changing the physical order of data, our concept of PatchIndexes allows multiple approximate constraint definitions per table, so for example the definition of multiple approximate sort orders.

Traditional Database Constraints After presenting the PatchIndex design in the following, we show how PatchIndexes can be applied to traditional database constraints like uniqueness and sorting. Besides materialized views, there are several specialized materialization approaches like SortKeys described in Section 2.2.1, materializing a certain tuple order, or JoinIndexes [151] and Bitwise Dimensional Co-Clustering (BDCC) [17] for join materialization. While JoinIndexes materialize foreign key joins by maintaining an index to the join partner as an additional table column, BDCC physically co-locates join partners of different tables in distributed databases. Our approach materializes aggregate, join and sort results directly at the table level, which makes it usable for a wide range of queries while still providing efficient update support. Particularly updating uniqueness constraints, which are the results of aggregations, differs from the general approaches for SPJ views.

Research on approximate constraints evolved from the field of constraint discovery. For the uniqueness constraint, approaches for unique column combinations (UCCs) [2] and respective discovery algorithms [68, 119] are introduced to handle perfect constraints. In order to also handle approximate uniqueness constraints, the concepts of “possible” and “certain” keys are presented in [90] to enforce constraints by

replacing violating tuples. Besides that, embedded uniqueness constraints (eUC) [158] separate uniqueness from completeness to enforce the uniqueness constraint only on a subset of tuples. Recent publications [123, 103] also cover discovery approaches for approximate denial constraints, a more general class of data-specific constraints. As these approaches mainly focus on the discovery of approximate constraints, we integrate approximate constraints into query execution in an updatable way with our work. We therefore combine the concept of approximate constraints with the concept of materialization.

Physical database design for OLAP As a second example for applying PatchIndexes we relax the partition locality constraint in data partitioning and introduce the patched multi-key partitioning constraint. Data placement has been an investigated problem for decades, as co-locating data is crucial for query performance [164]. Besides approaches for transactional workloads [39, 122], physical database design for analytical workloads is typical workload driven, either transparent for the database system [51] or integrated into a DBMS's optimizer [114]. DBDesigner [152] recommends sort keys and column encodings besides partition keys and also takes storage footprint and fault tolerance into account. [72] presents a learned partition key advisor that learns and adapts while observing the workload and [164] presents a workload-agnostic approach besides a workload-driven one, making decisions based on the database schema and the full data. Amazon published a distribution key recommender [120] based on mapping the join history of tables to a linear program, showing that solving the problem is NP-complete. The described approaches advise optimal partition keys or partition strategies for a given workload, but are limited to single-key partitionings due to limitations of the underlying DBMS. Multi-dimensional partitioning can be achieved by holding multiple partition schemes in parallel using replication [101] or using hierarchical partitioning [145]. Orthogonally to partitioning, multi-dimensional clustering [20] or Bitwise Dimensional Co-Clustering (BDCC) [17] can be used to place similar tuples close to each other to reduce I/O effort of queries or efficiently support joins and aggregations by exploiting the closeness of similar tuples. In our work we envision to mitigate the limitation of DBMS's to a single partition key per table. We show that hierarchical partitioning is not suitable and achieve multi-dimensional partitioning without data replication. This opens the possibility for partition advisors to consider multiple partition keys , so queries on different partition keys can be performed without data repartitioning.

Graph partitioning We show that we can map the problem of finding a solution for the patched multi-key partitioning constraint to existing graph partitioning algorithms. Finding a balanced graph partitioning is known to be NP-hard [7] and typically based on graph cuts, i.e., edge-cut or vertex-cut algorithms. Conceptually, edge-cut algorithms assign vertices to partitions and remove edges from the graph that connects vertices of different partitions. Vertex-cut approaches on the other hand assign edges to partitions and remove vertices with adjacent edges of different partitions. The problem is defined and surveyed in [29, 4], discussing the challenges of distributed graph algorithms and the state-of-the-art approaches like Metis [78], JA-BE-JA [132, 131] or DFEP [59]. With Distributed Neighbour Exchange (Dis-

tributedNE) [67], a distributed graph partitioning algorithm was presented recently that achieves a proven upper bound in partition quality and outperforms the state-of-the-art approaches, scaling to trillion edge graphs and offering approaches for both edge-cut and vertex-cut. Recently, [45] showed how to incrementalize existing graph partitioning algorithms, e.g., DistributedNE, adding update support to existing graph partitionings. In our example, we do not focus on capabilities of graph partitioning approaches, but show how to map the problem of multi-dimensional relational data partitioning to graph partitioning to make use of the existing approaches.

Robust Query Performance Multi-key data partitioning can be used to achieve robust query performance, which is a challenge in modern database optimizer design. A topic outline and a discussion of challenges are given in [57, 160]. In general, robust query performance describes the behaviour of a system to deliver good query performance for many different workloads, but trying to avoid bad performance as much as possible. Consequently, query optimizers should not try to strive for an optimal query plan, but find a reasonable good plan that tolerates wrong decisions, like cardinality or selectivity estimation errors. Many directions exist to achieve robust performance, like runtime collection of profiling information to allow adaptive query performance (e.g., in [1, 77]), using constraint information, handling estimation errors [16, 162] or providing only a small set of physical operators to reduce the chance for bad decisions [160]. In our example we aim at achieving robust performance on the physical data layout level by using our patched multi-key partitioning constraint to reduce repartitioning as much as possible. For all chosen partition keys we only repartition a small amount of tuples in order to achieve reasonable good performance, but never bad performance caused by a full table repartitioning. Our query optimization techniques are based on existing operators, not increasing the search space of physical operator plans as a result.

4.2 PatchIndex Design

4.2.1 Definitions and Overview

We design the PatchIndex as an index structure that can be defined on a certain subset of table columns for an arbitrary constraint. For each indexed column it maintains the exceptions for the given constraint. Exceptions are identified over a unique identifier, which is the row identifier RID in our case. Please note that a row identifier might change under updates. In compliance to related work on patched processing, we call the set of exceptions "set of patches" and define it as follows:

Definition 4.2.1. Naming conventions

R	Relation
$t \in R$	Tuple of relation R
$dom(c)$	Set of possible values of a column c
id	Column of row identifier RID
$id(t) \in \mathbb{N}$	Row identifier of t
$c(t) \in dom(c)$	Value of column c of tuple t

Additionally, we define a projection function
$PROJ(R, c) : relation \times column \rightarrow relation$
as a projection of relation R on column c, which similarly to the SQL operator performs no duplicate elimination and therefore differs from the relational algebra operator π.

Definition 4.2.2. Set of patches
For a column c we define a set of patches $P_c \subseteq \{id(t) \mid t \in R\}$. Based on this, we define $R_P = \{t \in R \mid id(t) \in P_c\}$ as the set of tuples of R whose row identifiers are in P_c and $R_{\setminus P} = \{t \in R \mid id(t) \notin P_c\}$ as the set of tuples of R whose row identifiers are not in P_c.

The PatchIndex data structure is intended to maintain the set of patches P_c for the column c it is defined on. It identifies exceptions over their row identifier, which enables an efficient join of the PatchIndex information with the actual data by position in the PatchIndex scan, but leads to some overhead to maintain positions under updates. Identifying tuples over a unique identifier like a tuple identifier or a primary key would lead to an additional value-based join to merge the PatchIndex information with the data. For the realization of PatchIndexes we implemented two basic approaches, which are the identifier-based approach and the bitmap-based approach. For the identifier-based approach, we store the 64 Bit tuple identifiers of all tuples in P_c in an array, which is similar to a sparse approach. As a result, the memory consumption of this approach is proportional to the cardinality of P_c. On the contrary, the bitmap-based approach is similar to a dense way of storing data. With n being the number of tuples in c (or the respective relation R), the PatchIndex holds a bitmap of size n for the column c, which is in particular independent of the cardinality of P_c. The element at position i within this bitmap indicates whether tuple i belongs to P_c or not.

Similar to the decision between a sparse and a dense way of storing, deciding between the identifier-based and the bitmap-based approach is based on the cardinality of P_c. As we require 1 Bit per element for the bitmap-based approach and 64 Bit per element for the identifier-based approach, we can expect that the identifier-based approach has a lower memory consumption for all cases where $|P_c|/|R| \leq 1/64 = 1.56\%$.

4.2.2 Update-conscious Data Layout

In our initial experiments we found that approximate constraints are beneficial for far larger sets of patches than 1.56% of the tuples, leading to the choice for a dense,

bitmap-based approach for storing the sets of patches. An important fact for an index structure to be usable in real world use cases is update support. Especially with the upcome of cloud based data analytics, data freshness and real-time reporting gained a lot of importance. Therefore optimizing a data structure for update support while maintaining read performance is an important factor for its usability.

In database systems, there are three types of table updates, namely insert, modify and delete operations. With the bitmap-based approach inserts and modifies can be handled efficiently. For inserts, reallocating/resizing the bitmap and setting single bits is sufficient, while for modifies single bits need to be changed. Delete operations however are the main challenge for the bitmap structure, as it needs to be ensured that read access to specific bits remains efficient after update operations. As described in Section 4.2.1, we build the PatchIndex on row identifiers to ensure efficient read performance, and deleting a tuple descreases row identifiers of all subsequent tuples. Although it can be realized by shifting the bitmap towards the deleted position, this potentially shifts large amounts of memory and therefore results in poor update performance. On the opposite simply marking bits as inactive leads to bad read performance, as a tuple with row identifier i is not represented by the bit at position i, but by the bit at position i minus the number of deleted tuples in the range $[0, i]$ of row identifiers.

To mitigate the issue of poor update or read performance, we present our approach of a sharded bitmap that efficiently supports RID-preserving delete operations while keeping additional memory consumption low. We base our design approach on existing update-conscious bitmap approaches and extend them with parallel and vectorized delete and bulk delete support.

Sharded bitmap design

Update Conscious Bitmap Indices [32] faced the problem of degrading read performance under updates [14]. HICAMP [156] and UpBit [14] proposed different solutions to this problem for multi-dimensional bitmap indexes. While HICAMP divides bitmaps into slices and maintains access to them over a directed, acyclic graph, UpBit uses delta-structures for update maintenance and "fence pointers" for fast access to arbitrary bit positions. Both approaches share the concept of slicing, either explicitly or implicitly over pointers, to keep updates local and allow parallel operations on the bitmap.

Similar to UpBit, in our sharded bitmap design we implicitly divide bitmaps into virtual shards in order to keep delete operations local and to enable parallelism in update operations. The upper part of Figure 4.1 shows the design concept of the data structure. The bitmap is realized using an array of addressable elements. While these elements are 64 bit types in our implementation, we reduced the size to 8 bits in Figure 4.1 for clarification. A virtual shard then consists of multiple addressable elements and a single additional integer value indicating the index of the first bit in a shard, similar to UpBit's fence pointers. Besides keeping delete operations local, this approach also facilitates fine-grained locking and logging for efficient concurrency control as described in Section 4.2.5.

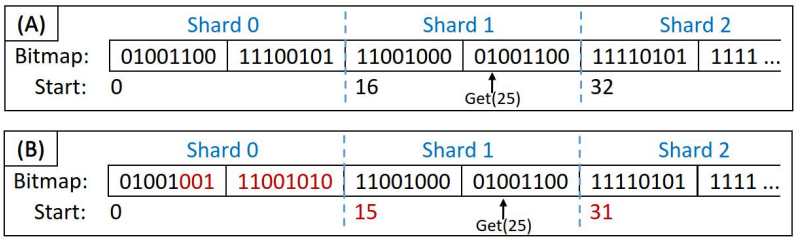


Figure 4.1: Sharded bitmap design before (A) and after (B) deleting the bit at position 5.

Sharded bitmap operations

Bit access In an ordinary bitmap, single bit access methods *set()*, *get()* and *unset()* are performed in two steps. First, the position of the bit is calculated by getting the addressable element using a division of the position by the size of an addressable element (realized using a bit shift) and getting the position within an element using a bitwise AND operation with a bitmask. In a second step, the bit is changed or returned using another bitwise operation.

In order to access a single bit in the sharded bitmap structure, we first compute the shard containing the bit. This can be efficiently realized using a bit shift and additional comparisons with the start values of the upcoming shards. The additional checks are necessary as a bit may be contained in a subsequent shard due to previous delete operations. Then the bit can be accessed using the position relative to the start value of the shard in a similar way than the ordinary bitmap. This way, accessing a single bit in the sharded bitmap is only slightly slower than in ordinary bitmaps.

Delete Deleting a single bit in the sharded bitmap is divided into three steps:

(a) Determine the position of the bit like described in the bit access operation.

(b) Shift all subsequent bits within the shard by one position towards the deleted bit.

(c) Decrement the start values of all subsequent shards.

As an example, the result of deleting the bit at position 5 is shown in the lower part of Figure 4.1 with the changed elements highlighted in red color. Note that after deleting a bit, the values of subsequent elements change, which is the desired semantic of the delete operation. In the example, the bit at position 25 after the delete operation is the bit at position 26 before the delete. Compared to ordinary bitmaps, steps (b) and (c) are tradeoffs, as we limit the impact of a delete operation to a single shard in step (b), but need additional effort afterwards in step (c) to adjust metadata. Therefore, the choice of the shard size is crucial for the data structure.

Step (b) is the main challenge of this approach as it involves cross-element bit shifts, which can be realized by a sequence of bit masking and shifting operations. In order to further accelerate this step, we designed a vectorized cross-element shifting algorithm that uses Advanced Vector Extensions Version 2 (AVX2) intrinsics and is based on shifting, masking and permutation intrinsics to enable data exchange

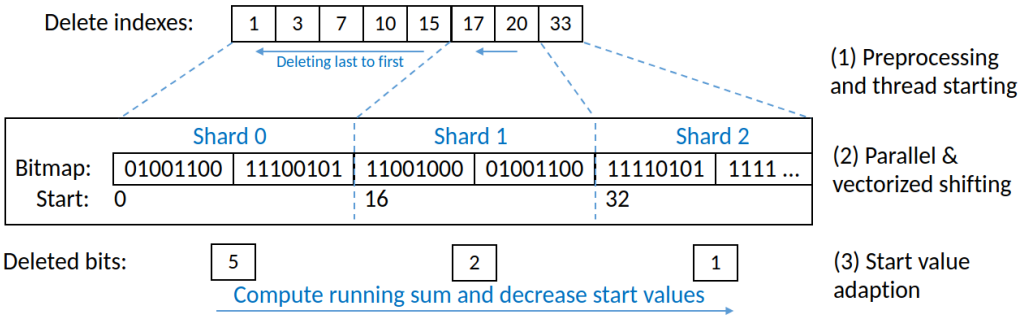


Figure 4.2: Parallel & vectorized bulk delete operation

```
// Load data to avx vector
__m256i x = _mm256_loadu_si256((__m256i*)data);
// Get last bit of each element
__m256i y = _mm256_and_si256 (x, bit_mask);
// First bit will be last bit of prev. element
y = _mm256_sllv_epi64(x, shift_mask63);
// Save element at pos 0 for next iteration
bits = _mm256_blend_epi32(bits, y, 0x03);
// Rotation Mask: (3, 3, 2, 1) is 1111001
__m256i rotated=_mm256_permute4x64_epi64(y, 0xF9);
// Copy element from last iteration to pos 3
rotated = _mm256_blend_epi32(rotated, bits, 0xC0);
// Saved element to pos 3 for next iteration
bits = _mm256_permute4x64_epi64(bits, 0x24);
// Shift data
x = _mm256_srlv_epi64(x, shift_mask1);
// Insert bit from next element
x = _mm256_or_si256 (x, rotated);
// Store back to data
_mm256_storeu_si256((__m256i*)data, x);
```

Listing 4.1: Cross-element bit shift using AVX2

between AVX lanes. The algorithm that is used inside the loop over the data of a shard is shown in Listing 4.1 (assuming pre-defined constant bit masks for shifting and masking).

Bulk delete In order to reduce the effort of changing the start value array and to exploit the opportinities of the sharding approach for parallelism, we introduce a bulk delete operation to the sharded bitmap structure. The basic concept of the bulk delete operation is shown in Figure 4.2. After a preprocessing step to determine the shards that belong to the elements, step (b) of shifting within the single shards can be performed in parallel using threads, as bitshifts remain local to the shards due to the design of the data structure. A thread is hereby created for each shard that contains indexes to be deleted, so the total number of threads depends on the number of shards and the location of deleted positions. The actual bitshift is again realized

using the vectorized algorithm shown in Listing 4.1. At the end of the operation, all start values are adapted in a single array traversal by holding a running sum over deleted bits of all preceeding groups. This way, the vectorized and parallelized bulk delete operation minimizes the additional effort of step (c) while further accelerating step (b).

The bulk delete operation is order sensitive, as deleting a bit within a shard shifts subsequent bits, changing their position. Consequently, positions of bits that are intended to be deleted are not correct if bits at smaller positions were deleted before. To overcome this, delete operations are performed in descending order, starting from the largest position. In our implementation, the PatchIndex structure buffers the rowIDs of tuples that should be deleted and performs a bulk delete operation at the end of the query. It thereby ensures the order of the rowIDs to enable an efficient bulk delete.

Condense With each single delete operation, a bit at the end of a shard gets lost, as the subsequent bit from the following shard is not shifted to this position. This is the main drawback of the sharded bitmap design and follows from limiting the impact of a delete operation to a single shard. In order to overcome this drawback, we introduce a condense operation to the sharded bitmap design, which shifts the elements of the bitmap between shards and resets the utilization of the structure as a result. The condense operation is realized using a single traversal over the bitmap. For each shard, data of subsequent shards is shifted to the bits that were lost due to delete operations and the start values are adapted accordingly. Condensing can be triggered manually or automatically by monitoring the utilization of the bitmap and triggering once a certain threshold is reached.

Sharded Bitmap Microbenchmarks

Choosing the shard size is crucial for the performance and the memory overhead of the sharded bitmap data structure. A small shard size leads to a large memory overhead due to the additional start values and deletion performance overhead for their adaption. On the contrary, a large shard size leads to a negligible memory overhead, but results in shifting large amounts of memory, which was the problem we intended to avoid with the design of the sharded bitmap. We ran microbenchmarks in order to determine a reasonable shard size on a machine consisting of two Intel(R) Xeon(R) CPU E5-2680 v3 with 2.50 GHz, offering 12 physical cores each, 256 GB DDR4 RAM and 12 TB SSD. The runtime of bulk deleting 1M elements (randomly chosen but fixed) from a sharded bitmap of size 100M is shown in Figure 4.3 for the parallel and parallel & vectorized bulk delete implementation. First, we can locate a clear minimum runtime at a shard size of 2^{14} bits. Below this minimum, the overhead of preprocessing and thread starting is not worth the benefit of multithreading and above this minimum the shifting effort starts to dominate the runtime again. Second, we can observe that vectorization impacts performance only in a minor way at these relatively small shard sizes, but gets more impactful the larger the shard size is chosen. Regarding memory consumption, a 64 Bit start value is stored for each

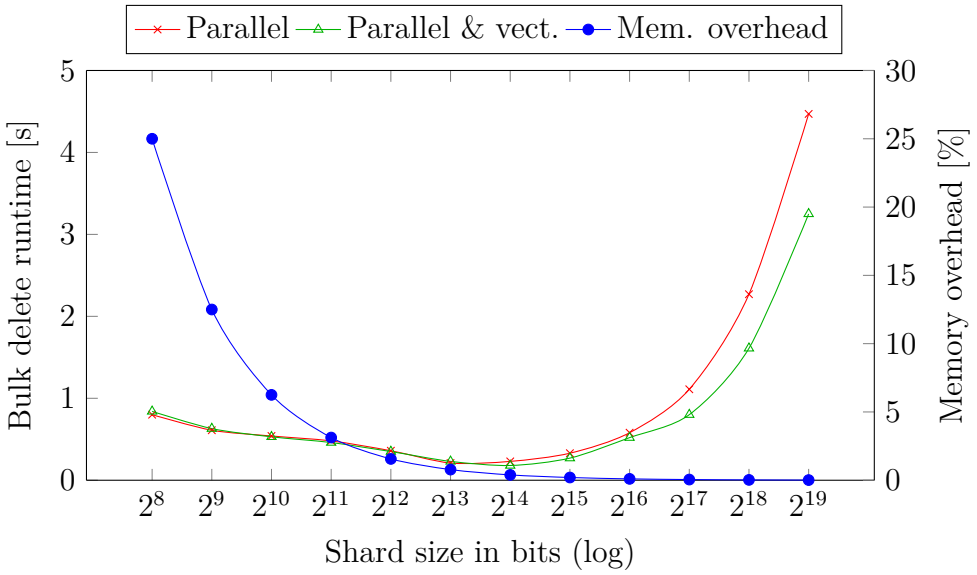


Figure 4.3: Sharded bitmap bulk delete runtime for 1M elements and memory overhead of sharding depending on shard size

	Bitmap	**Sharded bitmap**
Sequential Set	8.1 ns	16 ns
Sequential Get	5.1 ns	10.1 ns
Seq. Delete	$3.1 \cdot 10^6$ ns	3400 ns
Seq. Bulk Delete	–	180 ns

Table 4.1: Bitmap operator runtimes per element for bitmap with 100M elements and shard size 2^{14} bits

shard, resulting in a memory overhead of $\frac{64}{shard\_size} \cdot 100\%$. Choosing the shard size of 2^{14} bits therefore leads to a memory overhead of 0.39% for sharding.

Table 4.1 shows the latency per element of the bitmap operators that are relevant for the PatchIndex. Comparing an ordinary bitmap with the sharded bitmap approach, the virtual sharding leads to a small overhead for bit access operations, which is caused by determining the group an element belongs to. In terms of delete operations, the sharded bitmap performs three orders of magnitude faster than the ordinary bitmap, which improves to another order of magnitude when using the bulk delete operation to delete 1M elements. For the ordinary bitmap structure the delete runtime is size dependent and linearly increases/decreases when changing the bitmap size. Consequently, the sharded bitmap data structure significantly outperforms the ordinary bitmap in terms of deletion support, which particularly holds for large bitmap sizes, and justifies the small bit access overhead and small memory overhead for use cases where update support is important.

4.2.3 PatchIndex Scan

In order to benefit from PatchIndexes in query execution, we first have to efficiently apply the patch information to the dataflow of a query execution tree (QET).

Therefore, we introduce the PatchedScan. Scan operators typically support the definition of scan ranges to prune data and reduce I/O effort. These scan ranges are determined by evaluating selection predicates and using small materialized aggregates [111] that are guaranteed to be filtered out during query optimization. As tuples that harm constraints and are therefore classified as patches might be scattered across the data, translating PatchIndex information into scan ranges would result in numerous fine-grained ranges and non-negligible overhead.

Therefore we decided to realize the PatchedScan by combining an ordinary table scan with a specialized PatchSelect operator placed directly on top of the scan. In order to apply the information of the PatchIndex, we introduce selection modes *use_patches* and *exclude_patches* for these PatchSelect operators. As the PatchIndex works on row identifiers, we ensure that row identifiers are equal to the tuple positions in a datastream by placing the selection operators directly on top of scan operators, as intermediate operators could harm this assumption by filtering tuples. As a result, the output dataflow of the scan operator is logically splitted by the PatchSelect operators resulting in a dataflow containing the tuples of the set of patches P_c and a dataflow containing all tuples except the patches. An advantage of this approach is that the physical way tuples of a table are stored is not needed to be changed when creating a PatchIndex on that table. As a consequence, it becomes possible to create multiple PatchIndexes on a single table, which significantly differs from the typical limitation of one sort key per table.

Once during the query build phase, selection operators with modes *use_patches* and *exclude_patches* query the PatchIndex in order to receive a pointer to the patch array or to the bitmap holding the patch information. The pointer as well as other metadata like processed tuples are stored in a state variable of the operator. During query execution, both modes pass the incoming dataflow to the next operator while applying the patch information on-the-fly using a bitmap iterator. Using an iterator on the bitmap mitigates the overhead of a generic lookup operation per tuple and is possible as tuples are scanned in the order of their row identifier. For *exclude_patches*, applying patch information means skipping a matching tuple and mode *use_patches* only passes elements that match elements of the patch array.

Partition support PatchIndexes also support several common techniques of analytical database systems. First, they support partitioning by creating a PatchIndex for each partition separately. This way, partitioning is transparent to the actual index implementation. During query execution, subqueries are executed on partitions as far as possible and we do not harm this ability by placing the specific selection operators directly on top of the scan operators.

Scan range support Second, scan ranges used for I/O pruning (e.g., by small materialized aggregates [111]) are supported by the PatchedScan. While building the QET, *exclude_patches* and *use_patches* selection operators fetch the scan ranges from the scan operators below. During query execution, they merge the scan ranges on-the-fly with the patches by rebasing the bitmap iterator in order to skip patches outside the ranges. Applying scan ranges to scan operators decreases the number of scanned tuples. As we computed the set of patches P_c on the full set of values

for the indexed column c, we probably loose the minimality of the set of patches. As an example values might be unique also for tuples after applying the *use_patches* selection because duplicates were pruned by scan ranges. However, this does not harm the correctness but might introduce some unnecessary work in the worst case for not applying constraint-specific optimizations on these tuples. Mitigating this issue would lead to recomputing the sets of patches for each query with its respective scan ranges and is therefore not an option.

4.2.4 Durability

The PatchIndex is currently designed as an in-memory data structure. The index creation is logged to a write-ahead log (WAL), so the index can be reconstructed when performing the log replay in case of a system restart or failure. The determined patches are not written to the WAL in order to keep it slim, so the index is reconstructed from the data using the same mechanisms as for index creation. There are several alternatives to the in-memory approach that should be evaluated in the future. First, the index data could be materialized to disk, which has the advantages of durability, easy recovery and reducing the main memory consumption, as not all PatchIndexes have to be held in-memory at the same time. On the contrary, reading the relevant PatchIndex data from disk during query execution might decrease query performance, harming the desired benefit from the PatchIndex usage. Second, the PatchIndex information could be materialized as a bitmap column to the table. This way, reading the information could be done using ordinary table scan operators.

4.2.5 Concurrency

In terms of concurrency control, PatchIndexes seamlessly integrate into a system's snapshot isolation mechanism [31]. Although snapshot isolation is a very coarse-grained way to achieve serializability of transactions, it is often sufficient for read-optimized DBMS. Nevertheless, PatchIndexes offer opportunities for a more fine-grained concurrency control due to the underlying sharded bitmap data structure. As shards are independent from each other, fine-grained locking can be used to avoid concurrent access without locking the whole data structure. Adapting the start values of a sharded bitmap produces no conflicts, as the only operation that adapts the start values is the delete operation which uses decrement operations. Concurrent decrements are no conflicts, as different orders of executing a series of decrements produce the same result.

4.2.6 Extensibility

The PatchIndex data structure itself is independent from the actual approximate constraint it should maintain, i.e., it just holds the sets of patches for the indexed columns in a generic way. The structure defines an interface that needs to be implemented in order to use apply the approach to a given constraint. In an abstract view, the interface consists of the following functions:

- **Initial filling:** This function describes how to determine an initial set of patches. It is invoked either on the first time data is loaded into an empty table with a defined PatchIndex or on the creation of a PatchIndex on an existing table and creates the initial sharded bitmaps for the indexed columns.
- **Insert support:** Describes how inserted tuples impact the existing set of patches. While there might be additional exceptions within the inserted tuples, the added tuples might also lead to new exceptions from the existing tuples. Sharded bitmaps are resized in this step.
- **Update support:** Similar to insert support, this describes how updating existing values of indexed columns impact the sets of patches. New exceptions might be added or existing exceptions might be dropped. The sharded bitmaps remain at the same size.
- **Delete support:** Similar to insert and update support, but deleting tuples is likely to not add any new exceptions for most constraints. Deleting might lead to remaining tuples not being exceptions anymore, e.g., in the case of deleting a duplicate for a uniqueness constraint. However updating the set of patches could be skipped here, which might lose the minimality of the set of patches in exchange for delete performance. Sharded bitmaps are shifted as described in Section 4.2.2.
- **Query optimization rule:** An optimization rule needs to be added to actually benefit from the PatchIndex information in query execution. Typically, a dataflow is splitted using the PatchedScan described in Section 4.2.3 and optimized separately, e.g., by dropping expensive operations for tuples that already match a given constraint.

We discuss different approximate constraints in the following sections, which translates to finding approaches to implement the described interface.

4.3 Traditional Database Constraints

We described that database constraints contain valuable information for query optimization, but can be expensive to maintain and enforce in update cases. Update operations might be rejected when constraints are not met, and this might happen more often in cloud warehouse applications where data is ingested from a large number of data sources. Consequently, users may skip constraint definition in order to favour data freshness over data integrity.

With the concept of PatchIndexes we now have the tool to solve this dilemma. We can maintain constraint information without the need to enforce the constraints, i.e. it allows constraints to become approximate over time. In this section we choose the uniqueness constraint and the sorting constraint as important traditional database constraints. We define the concepts of “Nearly Unique Columns” (NUC) and “Nearly Sorted Columns” (NSC), describe the discovery of these constraints and how query processing can benefit from them. In the evaluation we compare PatchIndex against

different materialization approaches for these constraints in terms of query and update performance.

4.3.1 Definitions

Based on definitions 4.2.1 and 4.2.1 we introduce the definitions of "Nearly Unique Columns" (NUC) and "Nearly Sorted Columns" (NSC). Values of these columns fulfill the uniqueness constraint or the sorting constraint, respectively, except a set of tuples. We collect the row identifiers of tuples that violate the constraint in a set of patches P_c for a column c, so that the respective constraint is fulfilled by all remaining column values of c. An example is shown in Figure 4.4. The shown integer values of c are unique if we exclude, e.g., tuples with row identifiers in $\{1, 2, 3, 7\}$, while they are sorted if we exclude tuples with row identifiers in $\{4, 7\}$.

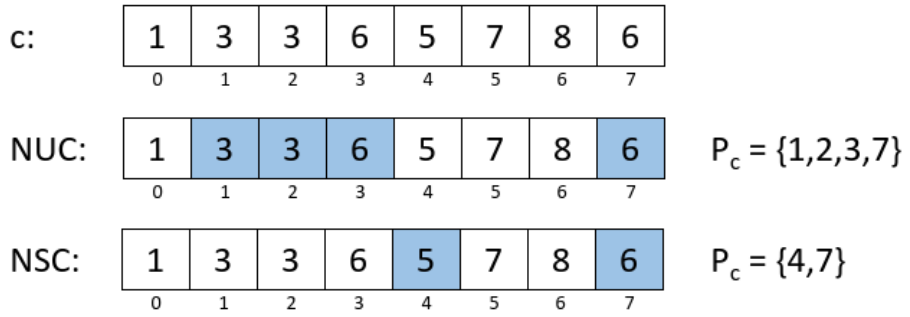


Figure 4.4: Example for NUC and NSC for a given dataset

Definition 4.3.1. Threshold variables
We define threshold variables *nuc_threshold* and *nsc_threshold*, both in $[0, 1] \subset \mathbb{R}$.

Definition 4.3.2. Nearly unique column (NUC)
A column c is a nearly unique column (NUC), when there is a set of patches P_c such that all of the following conditions are fulfilled:

(NUC1) $PROJ(R_{\setminus P}, c)$ is unique

(NUC2) $PROJ(R_{\setminus P}, c) \cap PROJ(R_P, c) = \emptyset$

(NUC3) $|P_c|/|R| \leq$ *nuc_threshold*

As an intuition, we want values of c (described using the projection operator $PROJ$ on c) to be unique after we excluded all tuples with row identifiers in P_c. The second condition (NUC2) is important to ensure the correctness of the query result, as we later want to utilize the PatchIndex in query execution by querying R_P and $R_{\setminus P}$ separately from each other. In column c of Figure 4.4, values 3 and 6 are duplicates, so they have to be excluded to make the column unique. As we have to exclude all occurrences of the duplicate values according to (NUC2), P_c consists of four identifiers. Therefore, the column c would be classified as a NUC if *nuc_threshold* ≥ 0.5. The choice of a minimal set is obviously unambiguous.

Definition 4.3.3. Nearly sorted column (NSC)
Given a column c, let $\triangleleft$ be an arbitrary order relation on $dom(c)$. Column c is a nearly sorted column (NSC), when there is a set of patches P_c such that both of the following conditions are fulfilled:

(NSC1) $\forall t_i, t_j \in R_{\setminus P} : id(t_i) < id(t_j) \Rightarrow c(t_i) \triangleleft c(t_j)$

(NSC1) $|P_c|/|R| \leq nsc\_threshold$

As an intuition, we want values of c to be sorted according to the given order relation $\triangleleft$ based on the order of their row identifiers after we excluded tuples with row identifiers in P_c. In order to fulfill the sorting constraint for column c of Figure 4.4, we can exclude the two identifiers shown by P_c to get a sorted sequence, so column c could be classified as a NSC if $nsc\_threshold \geq 0.25$. Nevertheless, this choice is ambiguous, as we can find another set of patches $P_c = \{3, 7\}$ with the same cardinality of two. In the discovery mechanism of NSC presented in Section 4.3.2, we are interested in a smallest set P_c.

4.3.2 Constraint Discovery

Based on the definitions in Section 4.3.1, the problem of deciding whether a given column c is a NUC or NSC translates to finding a set of patches P_c such that the respective constraints are fulfilled. We give general approaches for both problems that can be also integrated into arbitrary automatic database administration tools to automatically discover the constraints.

In order to discover a NUC, we need to find all column values that are not unique, which can be realized using a hash table. Instead of realizing this, we can reuse existing operators of database systems, as the described approach is equal to a distinct aggregation. Therefore, we can simply realize the NUC discovery on SQL level. It is not sufficient to simply compute a distinct query or query all values that occur more than once, as we need all occurences of a non-unique column value in order to meet condition (NUC2). Therefore, we join the result of an aggregation query with the actual table on the examined column c to get the row identifiers of all tuples with non-unique values for c, which is equal to the set of patches P_c we want to compute. We need to pay special attention to NULL values, as they are not joined with each other but should be assigned to the set of patches. Therefore, the join operation is realized using an outer join with a subsequent selection, resulting in the following query:

```
select tab.tid from tab
left outer join
    (select c from tab group by c
    having count(*) > 1) as temp
on tab.c = temp.c
where temp.c is not null or tab.c is null
```

By checking for non-null values in *temp* we ensure that we only select values with matching join partners. Based on the result of the query, the classification of a column c as a NUC can be made based on the third condition $|P_c|/|R| \leq nuc\_threshold$.

In order to discover a NSC, we utilize the longest sorted subsequence algorithm [50], for which we need the full data of a given column c as a prerequisite. The algorithm then maintains arrays to keep the length and the predecessor of the last element in the longest sorted subsequence of data $[1, \ldots, k]$ at position k. For each of the n elements in the array, the algorithm performs a binary search on the already computed results, resulting in an overall worst case runtime of $O(n \cdot \log(n))$. In order to compute P_c, the resulting list of indexes that are included in the longest sorted subsequence is inverted (with respect to the examined relation R). This way we ensure that $R \setminus P_c$ is sorted on column c and as we computed the longest sorted subsequence, we also ensure that the cardinality of P_c is minimal. NULL values are also assigned to P_c in order to ensure correctness of sorting queries. The classification of a column c as a NSC can then be based on the second condition $|P_c|/|R| \leq nsc\_threshold$.

4.3.3 Query Processing

Analytical Queries

The main idea to integrate PatchIndexes into query execution is the PatchedScan described in Section 4.2.3, merging the PatchIndex information on-the-fly with the dataflow and splitting it into a flow of tuples satisfying the constraint and a flow of exceptions, and work on both dataflows separately. The PatchedScan is realized using an additional selection operator on top of a scan operator, merging the PatchIndex information on-the-fly to the dataflow according to selection modes *exclude_patches* and *use_patches*. The key for query optimization using PatchIndexes is cloning query subtrees for data and patches and optimizing both subtrees separately. Typically, we can exploit the PatchIndex information in the dataflow that excluded the patches and achieve a speedup in this subtree, as a constraint is fulfilled here. Finally, both subtrees are combined again to ensure transparency for the remaining query execution tree.

With NUCs and NSCs, PatchIndexes can be integrated into the optimization of distinct, join and sort queries. For distinct queries, we can exploit the information that tuples are unique in a NUC after excluding the patches, so the most expensive aggregation operator to compute the distinct values can be dropped from this subtree as shown on the left side of Figure 4.5. Here "X" is an arbitrary subtree that does not contain any joins or aggregations. In case of a query that contains a grouping, this approach can also be applied by decoupling the aggregation for grouping from the distinct aggregation.

Knowing about the sorting of NSC in join queries, we can replace the generic HashJoin operator with the faster MergeJoin operator in the subtree that excluded patches, like shown in the right part of Figure 4.5. This optimization requires the subtree "X" to be sorted on the join key, which is a frequent case in joins between fact tables and dimension tables in data warehouse applications. While subtree "Y" must preserve the tuple order and is therefore not allowed to contain aggregations, both "X" and "Y" may contain join operators that are order preserving, so e.g., being the probe side of a HashJoin. Under these requirements the PatchIndex information can be propagated through a query tree. In order to further optimize this approach,

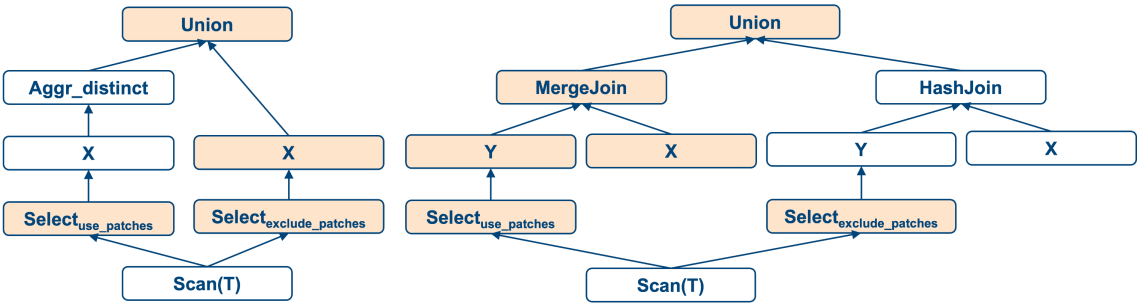


Figure 4.5: Query plans for distinct (left) and join (right) queries before and after PatchIndex optimization with the newly inserted operators highlighted.

the result of subtree "X" is buffered instead of computed twice. Additionally, join sides can be swapped to further accelerate the HashJoin. As the number of patches is known during optimization time, we can choose the build side of the HashJoin as the one with the lowest cardinality to improve the time and space for hash table building. Building the hash table on the patches is often the best decision as the number of patches is typically small. The join sides of the MergeJoin are chosen similarly to ensure the combinability of both subtrees. As the MergeJoin operator is not sensitive to join sides, this has no impact on performance.

Besides the join operator, the information about a NSC can also be used to accelerate sort queries, as we already know that a major part of the data is already sorted. Here we follow an approach similar to distinct queries, so the query plan is equal to the left plan of Figure 4.5 exchanging the aggregation with the sort operator. If there is a PatchIndex defined on the sort column using the same ordering as the sort operator, the sort operator becomes obsolete in the subtree that excludes patches, as these tuples are already known to be sorted. The sort operation only needs to be performed on the patches, which is intended to be the minor part of the data. In order to preserve the sort order, the results of both subtrees are combined using a Merge operator instead of a Union operator.

Cost Model

The use cases for query optimization presented in the preceeding subsections can be easily integrated into the cost models of arbitrary query optimizers, as cardinalities and operator output estimates are known during optimization time and we use ordinary query operators for the optimization, except the newly introduced selection modes for the PatchedScan. The costs for query trees (build and execution costs) produced by the PatchIndex optimizations can therefore be estimated by query optimizers and if the estimated costs are smaller than the costs of the subtree before the optimization, the PatchIndex should be used for the query. The selection operators with modes *exclude_patches* and *use_patches* merge the PatchIndex information on-the-fly to the output dataflow of the scan operator. This is particularly independent from the data types of the input data, as the decision of passing or dropping a tuple is based on a tuple's row identifier. As a consequence, the operator's overhead is

	Nearly unique column	**Nearly sorted column**
Insert	Scan inserted tuples, join them with the table, merge the results with the existing patches.	Determine a new sorted subsequence extending the already existing one.
Delete	Drop tracking information about deleted tuples.	Drop tracking information about deleted tuples.
Modify	Scan modified tuples, join them with the table, merge the results with the existing patches.	Merge all modified tuples with the existing patches.

Table 4.2: Design principles for update operations

fixed for every tuple. In our experiments, the selection operators for both selection modes took a minor part of query runtimes (typically below 1%).

Update Queries

Handling table update operations like inserts, deletes or modifies is the most significant problem of any materialization and therefore also important for the approximate constraint materialization using PatchIndexes. As table updates were not present at the time a general materialization was computed, it reaches an inconsistent state whenever an update occurs. The common way to handle this inconsistency is refreshing, often leading to an expensive recomputation. For some use cases, minor inconsistencies might be acceptable and refresh cycles can be chosen quite loose, while other applications rely on consistent and up-to-date data and therefore demand for very tight or just-in-time refresh cycles.

On the contrary, PatchIndexes are designed to efficiently support table update operations. We avoid getting inconsistent states by handling updates immediately after they occur. In order to perform these updates efficiently, the design of the update handling mechanisms is driven by the goal of avoiding an index recomputation and avoiding a full table scan while preserving the invariant of holding all exceptions to a given constraint. The basic ideas for handling inserts, deletes and modifies for the specific constraints are summarized in Table 4.2.

Besides the concepts of delta structures, small materialized aggregates and range propagation described in Section 2.2.1 we utilized the concept of intermediate result caching for the design of the update handling mechanisms.C aching can be used within queries to avoid expensive recomputations. In our further considerations, we use the Reuse operator to encapsulate this behaviour. Intermediate results are materialized in main memory by the ReuseCache operator and read again from the ReuseLoad operator.

Insert Handling insert operations in the PatchIndex translates to answering the question which tuples have to be added to the existing patches. For the uniqueness constraint, this is a difficult question, as this constraint relies on a global view of the table. Inserting a single tuple might produce a collision with another tuple that was already in the table and had a unique value in the indexed column before. As we need

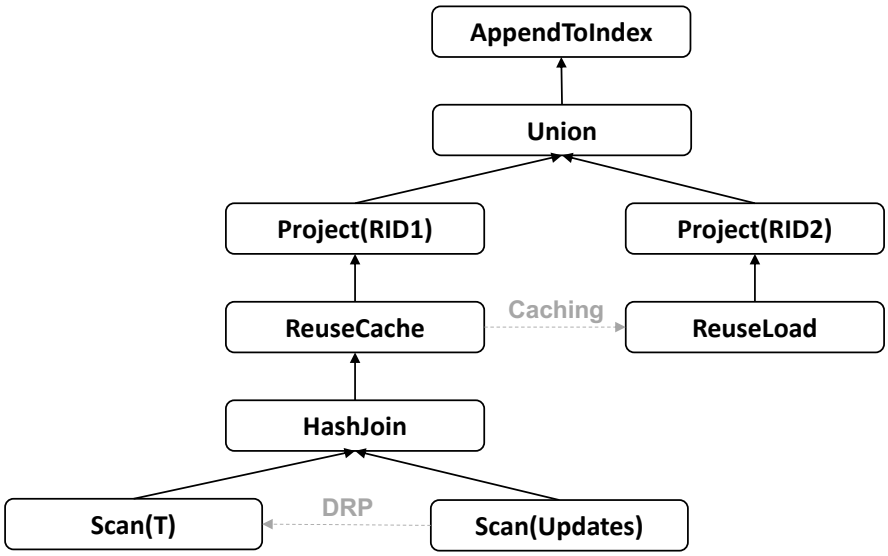


Figure 4.6: Insert handling query on table T with dynamic range propagation (DRP) and intermediate result caching

to keep track of all occurences of non-unique values to ensure correctness, we perform a join query after the insert operation, joining the inserted tuples with the actual table (including inserted values, as duplicates might also occur in the inserts) like shown in Figure 4.6. Afterwards we project row identifiers of both join sides using intermediate result caching. The row identifiers are then merged into the existing patches. This way, we preserve the capability of holding the minimum set of patches to make the table unique when excluding them, while also avoiding a recomputation of the full index. In order to avoid the full table scan we utilize dynamic range propagation. After the build phase of the join operator is finished, scanning the full table is reduced to only the blocks that contain potential join partners. This significantly reduces I/O overhead and is therefore a major improvement of the insert handling mechanism.

For the sorting constraint, we focus on a local view on the inserted tuples instead of a global view on the whole table by extending the already existing sorted subsequence with inserted values instead of recomputing a globally longest sorted subsequence. The PatchIndex keeps track of the last value of the subsequence (i.e., the largest value of an ascending sort order or the smallest value of a descending sort order). During execution of the insert query, the PatchIndex temporaryly stores inserted values and starts computing a longest sorted subsequence with all values being larger/smaller than the last value of the existing sorted sequence. We hereby utilize the same longest sorted subsequence algorithm [50] as used in the discovery of a NSC in Section 4.3.2. The row identifiers of tuples that are not included in this extending sorted subsequence are added to the set of patches. With this mechanism, we probably lose the optimality of the index, namely to keep track of the longest sorted subsequence. The reason for that is the fact that combining two sorted sequences that have a maximum length for two parts of the data might not form a longest sorted sequence for the combination of the two parts. An example for this

can be easily constructed, e.g., the table holding values (1, 2, 10) and inserted values (3, 4). Nevertheless this does not lead to wrong query results when applying the PatchIndex optimization for the sorting constraint and should be negligible for the typical use cases. However, monitoring the exception rate and triggering a global recomputation once a certain threshold is reached is a possible solution for that. From an implementation point of view, scanning the inserted values is realized by scanning the PDTs of the current query. Furthermore, merging the determined results with the existing patches translates to reallocating the bitmap and setting the respective bits for the new patches in the bitmap-based approach.

Modify For the uniqueness constraint, handling modify operations is done similarly to insert handling described above. For the sorting constraint, all modified tuples have to be added to the set of patches, as modifying values might destroy the sorting of the computed subsequence. The only difference in the actual realization is that reallocating the bitmap becomes obsolete as the table cardinality does not change during modify operations.

Delete Deletions are handled by the PatchIndex by dropping the information about deleted tuples from the patch information without taking a global picture of the table into account. For both uniqueness and sorting constraint, dropping values from the table does not violate the constraint. For the uniqueness constraint, we might lose the optimality this way, as a value that was not unique before the delete operation might become unique afterwards if the other occurrences of the value was deleted. Nevertheless, it would remain in the patches to keep the delete handling mechanism simple and would obviously not lead to wrong query results when applying the PatchIndex to queries. The same holds for the sorting constraint, probably losing optimality when deleting values from the existing longest subsequence. Again monitoring and recomputing the index is a possible solution if this becomes a problem. From an implementation point of view, deleting values raises the problem that rowIDs of subsequent tuples decrease for each deleted tuple. This is considered by the bulk delete operation of the sharded bitmap data structure as described in Section 4.2.

4.3.4 Evaluation

In this section, we evaluate our solution in different experiments to prove the performance impact of PatchIndexes and its updatability. We therefore show experiments on different integration levels. First, we present a set of PatchIndex microbenchmarks for a fine-grained evaluation of aspects like performance impact, creation time or update operations. Afterwards we show the impact of PatchIndexes on query performance using a query subset of the well-known TPC-H benchmark. For the evaluation, we integrated PatchIndexes into the Actian Vector 6.2 built on the X100 [28] engine. The system runs on a machine consisting of two Intel(R) Xeon(R) CPU E5-2680 v3 with 2.50 GHz, offering 12 physical cores each, 256 GB DDR4 RAM and 12 TB SSD. For all measured results, we used queries on hot data, which means that data resides in the in-memory buffers of the system. This way, we reduce the

I/O impact and focus on the pure query execution time. Additionally, we did not trigger any sharded bitmap condense operations but started every single experiment with a freshly build index structure for comparability reasons.

In the evaluation, we compare the generic PatchIndex approach against different specialized materialization approaches, namely materialized views, SortKeys and JoinIndexes described in Section 4.1. For distinct queries, we use materialized views as a comparison, which is a widely used technique in database systems to pre-compute partial queries like the distinct query in our example. While leading to a significant performance benefit if matched by a user query, the major drawback of materialized views is their ability to handle updates. Typically, they need to be re-computed when updates occur to keep them consistent with the actual database. Alternatively, these expensive recomputations can be delayed and run regularly, if minor inconsistencies are acceptable for the user application. As materialized views are not offered by Actian Vector, we simulate this approach by storing the materialized information in a separate table and manually rewrtiting queries. For sort queries, we compare PatchIndexes against SortKeys, which is not implemented as a B^+-Tree in Vector, but physically sorts data on the given SortKey column. This way, sort queries can be translated to simple scan queries. As a drawback, physically reordering data is a very costly operation and maintaining this order in case of updates requires additional effort. Last, we evaluate join queries by comparing the PatchIndex approach against JoinIndexes, which materialize foreign key joins as an additional table column. If a SortKey is defined on the table holding the primary key of the join, the foreign key related table is ordered similarly, so that a MergeJoin becomes possible to join both tables.

Microbenchmarks

For our microbenchmarks, we designed a data generator[1] that varies the exception rates to given constraints. The data consists of 1B tuples with two columns, a unique *key* column and a *value* column that shows the desired data distribution. With the tuple width of 128 Bytes this results in a dataset size of 128 GB. In order to exploit parallel data processing, we partition the datasets on the *key* column into 24 partitions. As the *key* column is unique, this results in partitions of nearly equal size. For the uniqueness constraint, exceptions of the *value* column are equally distributed into 100K values, while the remaining values are unique and differ from the values of the exceptions. For the sorting constraint, exceptions are randomly chosen and all remaining values form a sorted sequence in ascending order. For both constraints, exceptions are randomly placed in the datasets. As the datasets are generated once, the randomness does not impact the comparability of the evaluation results. For the evaluation of update operations, we chose the dataset with exception rate $e = 0.5$. This choice has no impact on update query performance, as bits of updated patches have to be accessed independently from being patches before the update.

The materialization is realized in different ways for the constraints in the microbenchmarks. For the uniqueness constraint, we materialized a table containing all unique values of the *value* column using a distinct query. As a result, a distinct query

[1] Available under `https://github.com/Sklaebe/Approximate-Constraint-Data-Generator`

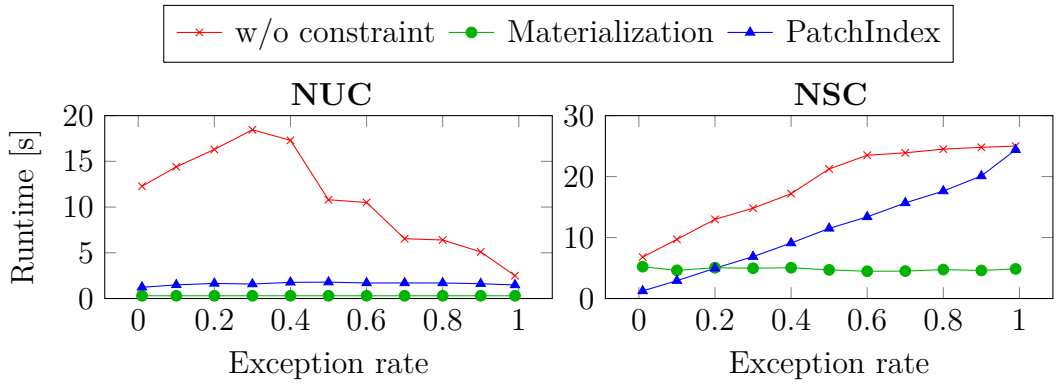


Figure 4.7: Runtimes of a distinct query on a NUC or a sort query on a NSC with varying exception rate

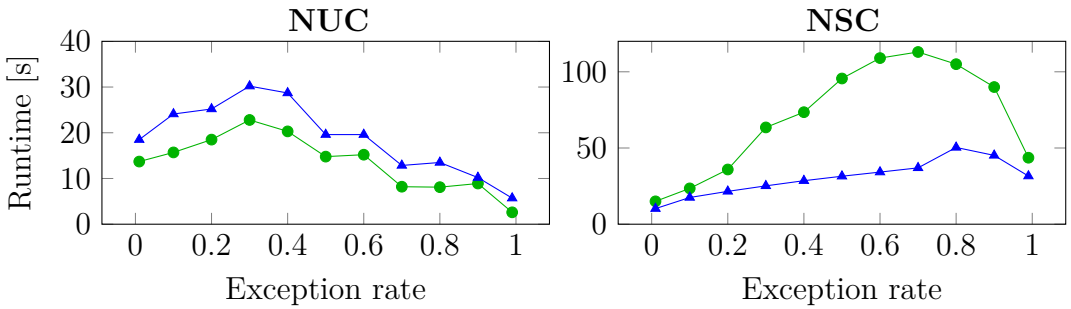


Figure 4.8: Runtime for materialization/index creation for varying exception rate

on the *value* column is replaced by a scan query on the materialized view without the need for an expensive aggregation. For the sorting constraint, we materialized the order information using a SortKey on the *value* column. This way, the data of the real table is physically re-ordered according to the *value* column. Queries that include a sort operator therefore just scan the table. As the table is partitioned, an additional merge step of the tuples from each partition is necessary to preserve the global order of the query results.

Performance impact First we want to prove the query performance improvement for different exception rates when using PatchIndexes. We run a distinct query and a sort query respectively while varying the exception rates for the uniqueness and sorting constraint in the dataset. Figure 4.7 shows the results of this experiment. For the uniqueness constraint, reference runtimes without any constraint definition increase with increasing exception rates, before decreasing starting from an exception rate of 0.3. With increasing exception rates, the number of distinct tuples and therefore the number of aggregation groups decrease. The runtime behaviour is then caused by the inference of a reduced hash table size and the increased communication costs, as the system uses a shared hash table build plan for the aggregation. The use of a materialized view shows a nearly constant runtime as the query only scans the materialized result. Using a PatchIndex shows a significant performance benefit compared to the reference runtimes with performance comparable to the materialized

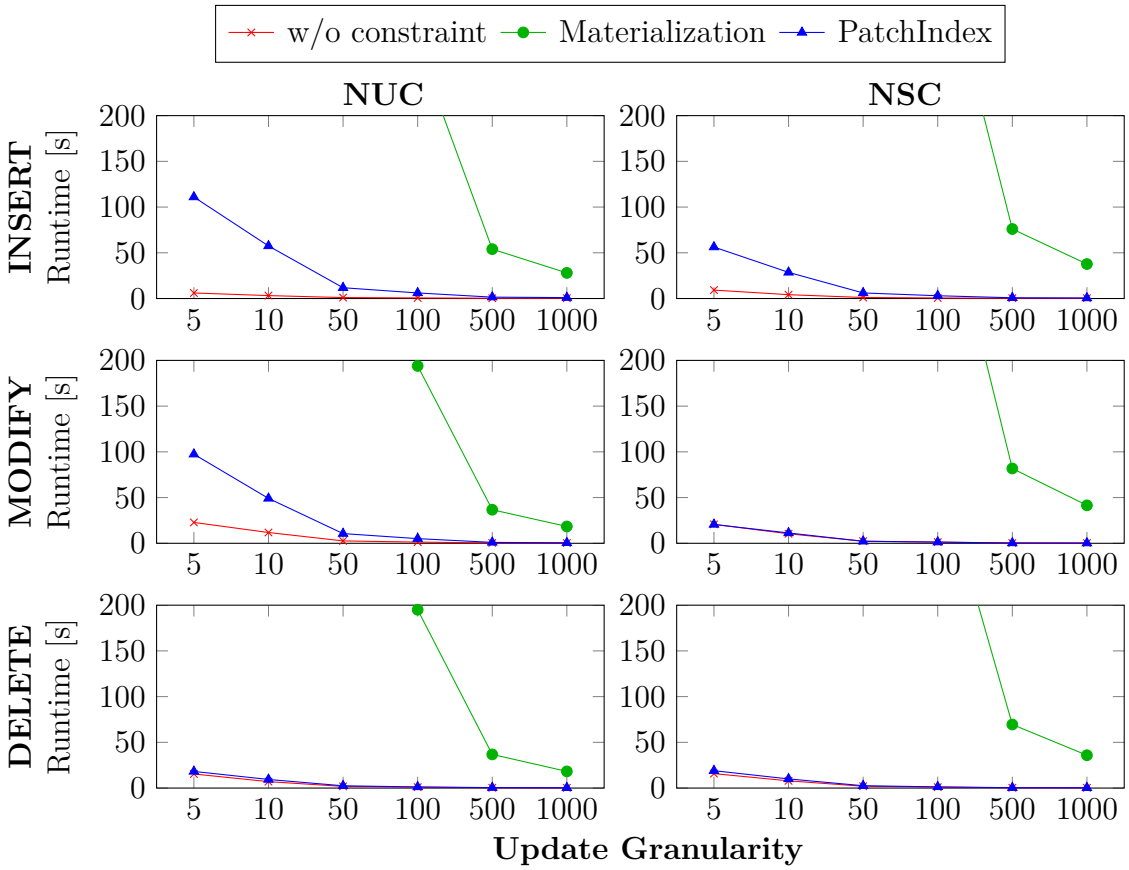


Figure 4.9: Update performance for inserting/updating/deleting 1000 tuples and varying update granularities

view. Performance slightly increases with increasing exception rates caused by more tuples being processed in the aggregation. As a result, the actual performance gain compared to the reference runtime shrinks with increasing exception rates, but using a PatchIndex does not impact runtimes in a negative way for the evaluated cases.

For the sorting constraint, reference runtimes increase with increasing exception rates, which is caused by the pivoting strategy of the internal QuickSort implementation, behaving better the more sorted the input sequence already is. Using a SortKey as materialization shows a constant runtime, although slightly slower than the scan query for the uniqueness constraint, as merging the sorted partitions is necessary. Additionally, the query still performs a sort operator to ensure the sorting, resulting in a slightly worse performance than using a PatchIndex for small exception rates. Using the PatchIndex shows a significant performance gain compared to the reference runtimes. Runtimes increase as expected with increasing exception rates as more tuples have to be processed by the sort operator, so the relative performance gain shrinks with increasing exception rate. Again, using a PatchIndex does not impact runtimes in a negative way for the evaluated cases.

Memory consumption The memory consumption of a PatchIndex is independent from the materialized constraint and shown in Table 4.3. The bitmap-based approach

	PI_bitmap	**PI_identifier**	**Mat. view (NUC)**
General	$t/8 \cdot 1.0039\ B$	$e \cdot t \cdot 8\ B$	$(10^5 + (1-e) \cdot t) \cdot 8\ B$
$e = 0.01$	125.48 *MB*	80 *MB*	7.9 *GB*
$e = 0.2$	125.48 *MB*	1.6 *GB*	6.4 *GB*

Table 4.3: Memory consumption for example dataset of 128 GB / $t = 10^9$ tuples

has a constant total memory consumption (1 bit per tuple + sharded bitmap overhead). For completeness, we also show the memory consumption of an identifier-based sparse approach of materializing exceptions here, which grows linearly with the number of exceptions. The bitmap-based approach has a lower memory consumption for cases with exception rate $e > 0.0158$. The performance experiments prove that PatchIndexes impact query performance for far larger sets of patches, so a sparse storage approach is not suitable here as initially stated in Section 4.2.1. In contrast, the materialized view for the uniqueness constraint materializes every unique value (100K unique values + exceptions), leading to a significantly higher memory consumption than the PatchIndex for most cases. A SortKey does not lead to any storage overhead and is therefore not included in Table 4.3. It physically reorders the existing tuples and is not realized over an additional index structure like a B^+-Tree.

Creation time Another important fact for the usability of data structures is their creation effort. Both PatchIndex and materialization are intended to be used multiple times after being created. Otherwise the creation effort would not be worth compared to the achieved query speedups. Figure 4.8 shows the comparison of the runtimes for creating the PatchIndex and the materialization for both constraints. For the uniqueness constraint, the runtimes follow the reference runtimes of Figure 4.7, as the distinct query is pre-computed and materialized. The effort to create a PatchIndex is slightly higher than the materialization, as the information about exceptions have to be filled into the index structure. For the sorting constraint, creating the SortKey takes a huge amount of time, as this physically reorders the table data. In comparison, creating a PatchIndex is more efficient. The increasing exception rate leads to an increasing number of comparisons in the longest sorted subsequence algorithm, while decreasing the length of the sorted sequence and therefore decreasing the effort to reconstruct it. The inference of both parts leads to the observed runtimes. Next to the worse creation performance of a SortKey, it's definition is also limited to one per table, as it physically reorders the data. In comparison, PatchIndexes can be defined multiple times per table on different columns, as it does not change the way data is physically stored.

Insert Based on the dataset with an exception rate $e = 0.5$, we inserted 1000 tuples per run into the database and varied the granularity of the insert operations between 5 tuples per operation (200 queries in total) and 1000 tuples per operation (1 query in total) to capture the impact of trickle and bulk inserts. The first row of Figure 4.9 shows the total runtimes to insert the 1000 tuples. First, we can observe that recomputing or maintaining the materialization for each insert operation produces

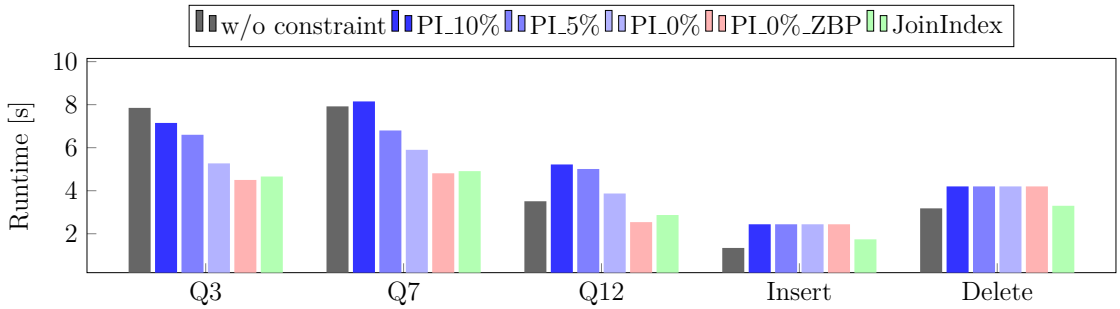


Figure 4.10: TPC-H query performance

a tremendous overhead to the reference runtime without any constraint definition, making it especially not usable for trickle updates. In contrast, PatchIndexes support insert handling in a more efficient way. For the uniqueness constraint, every insert operation invokes the insert handling query shown in Figure 4.6, resulting in an overhead for fine-grained inserts. Similarly, the sorting constraint invokes the execution of the longest sorted subsequence algorithm on the inserted tuples for every insert query. Nevertheless, the added overhead is smaller compared to the uniqueness constraint. For insert granularities of 50 tuples per operation or higher, the overhead is negligible for both constraints and defining a PatchIndex on a column does not impact insert runtimes in a remarkable way. Comparing the plots for PatchIndex and materialization, this result also leads to another consequence. Keeping the runtime nearly equal by fixing a value on the y-axis, update cycles can be chosen 50 times more frequently for the uniqueness constraint and 100 times more frequently for the sorting constraint when using a PatchIndex instead a materialization. This increases the ability of the system to keep the materialized information consistent with the actual dataset.

Modify Similar to the insert experiment, we here updated 1000 tuples of the dataset with exception rate $e = 0.5$ for different granularities. For the uniqueness constraint, we can observe a similar behaviour than for the insert support, as both perform the same query to handle the updates and keep the PatchIndex accurate. For the sorting constraint, updates must be included to the exceptions in the PatchIndex, as they may destroy the sorted subsequence . This can be done efficiently without the need for an additional query, leading to nearly no overhead compared to the reference runtime.

Delete The last row of Figure 4.9 shows the total runtime for deleting 1000 tuples from the dataset with exception rate $e = 0.5$ with varying granularities. As the PatchIndex just drops information about deleted tuples and the shareded bitmap offers an efficient delete operation, handling deletes is a very efficient operation and adds nearly no overhead to the operation runtime.

TPC-H

The TPC-H benchmark [27] is a well-known and well-understood benchmark used for performance evaluation of analytical DBMS for many years. Besides analytical queries, the benchmark also contains update sets for table inserts and table deletes that can be used to evaluate update support. In our experiments, we used the benchmark at scale factor SF 1000.

We decided to focus on the largest join in the benchmark, which is the join between the lineitem and orders table. Although the benchmark only contains clean data with perfect constraints, we manually manipulated the data order of the lineitem table in order to introduce exceptions to the sorting constraint, resulting in three datasets with 0%, 5% and 10% exceptions. While we stored the orders table in a sorted way, we evaluated the impact of a PatchIndex and a JoinIndex defined on the lineitem table for query performance and chose a subset of three queries that includes this join. Besides different exception rates, we evaluated two options for the dataset without exceptions. The first option uses the PatchIndex optimizations described in Section 4.3.3 and is therefore the general case that is comparable against the reference runtime in a fair way. Nevertheless, these queries contain unnecessary overhead as the constraints in the benchmark do not contain any exceptions. In the second option we therefore enable zero-branch-pruning (ZBP), which is a technique of removing subtrees of a query plan that are ensured to not produce any results. During query optimization query plans are annotated with cardinality estimations. If these estimates are ensured to be zero, e.g., by evaluating integrity constraints or indexes, the query rewriter can drop the respective subtrees from the query plan. This way, the subtree that would process the patches is pruned from the query plan, resulting in better performance as it drops all overhead introduced by cloning the query subtrees. Although not being the general use case, this option can therefore be used to compare results against the JoinIndex runtimes.

Figure 4.10 shows the results of the experiment for queries as well as for update sets. In general, the exception rate has no impact on reference runtimes without constraint, JoinIndex runtimes or update runtimes with PatchIndexes. We can observe that the PatchIndex benefit on join queries depends on the size of the join and the exception rate. For Q3, which contains the largest join, query runtime decreases with decreasing exception rate and impacts query performance in a positive way even for an exception rate of 10%. The same holds for Q7, which however shows worse performance compared to the reference runtime for 10% exceptions. As the PatchIndex impact grows with lower exception rates, PatchIndex runtimes nearly reach the JoinIndex runtimes for an exception rate of 0%. Additionally activating zero-branch-pruning reduces the overhead introduced by the Patchindex optimization, leading the queries to run 43% and 40% faster compared to the reference runtime. With zero-branch-puning, runtimes are also slightly faster than using the JoinIndex, which is a full materialization of the join. This is caused by a small additional scan effort in the JoinIndex query, as here the index is materialized in an additional table column. Query Q12 shows a different behaviour, as this already short-running query is impacted in a negative way when using a PatchIndex. In opposite to queries Q3 and Q7, the join in this query is very small due to prior selections. Therefore, the added overhead of cloning subtrees is larger than the gained benefit of exchanging the

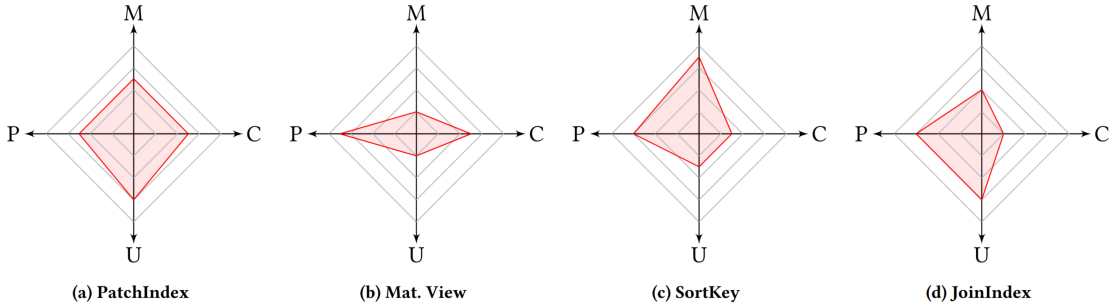


Figure 4.11: Qualitative comparison of PatchIndex against evaluated approaches in terms of Creation effort (C), Memory/Storage overhead (M), Performance impact (P) and Updatability (U). (Higher score means "better")

HashJoin with a MergeJoin in the PatchIndex optimization. Nevertheless, enabling zero-branch-pruning leads to a performance benefit of 28%, which is slightly faster than the JoinIndex query again. In terms of update support, zero-branch-pruning does not have any impact on performance. For the insert set (inserting 0.5M tuples) and the delete set (deleting 6M tuples) we can observe a slight performance overhead for both materialization approaches. Here the JoinIndex performs slightly better than the PatchIndex, as updates are handled in-memory by the Positional Delta-Tree [70] structure. The creation effort for both approaches is excluded from Figure 4.10 for scaling purpose. Here, creating a PatchIndex takes 100 seconds, which is significantly faster than creating a JoinIndex, which takes around 600 seconds.

In contrary to distinct and sort queries, this experiments shows that using PatchIndexes in join queries is a trade-off, as the overhead of cloned subtrees might negate the benefit of using a MergeJoin for most tuples. As described in Section 4.3.3, these plans would not be chosen by the optimizer. Although the benchmark (theoretically) only contains perfect constraints, it is an example for another advantage of the PatchIndex approach. Even if a dataset is clean at a point in time, it may become unclean in the future by update operations. While these updates would be aborted with the definition of usual constraints, PatchIndexes would allow the updates and the respective transition from a perfect constraint to an approximate constraint.

Resume

We compared the generic PatchIndex approach against materialized views, SortKeys and JoinIndexes as specialized materialization approaches for different queries. Figure 4.11 shows a qualitative comparison of the discussed approaches, proving that the PatchIndex structure is an impactful compromise between not defining any constraints and the materialization of constraints. The effort to create a PatchIndex is in the order of materialized views, while SortKeys and JoinIndexes require significantly more time to create. As PatchIndexes only add a single bit per tuple, they show a moderate memory overhead, which is only surpassed by the SortKey, which reorders data and avoids storing additional metadata. For distinct, and sort queries, the PatchIndex reaches a performance impact comparable to the specialized

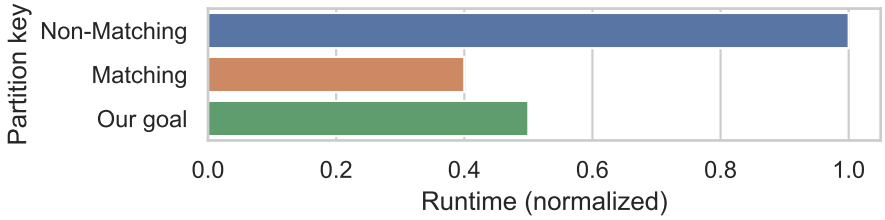


Figure 4.12: Groupby performance depending on matching partition key

materialization approaches, which even holds for quite high exception rates. While update support is a drawback of materialized views and SortKeys, the PatchIndex approach offers lightweight support for update operations.

4.4 Patched Multi-key Partitioning for Robust Query Performance

The key for efficient query processing in a distributed environment is data partitioning, i.e. distributing chunks of data over cluster nodes in order to perform subqueries on them in parallel. A partitioning of a relational table is defined as a disjunct split of tuples in the table and is realized using different partitioning strategies. While random partitioning or round-robin partitioning assign tuples to partitions randomly or based on their position in the table, value-based partitioning like hash partitioning or range partitioning distribute data based on the values of a chosen partition key column. The design of these partitioning strategies also aims at fulfilling the requirements of load balancing and exploiting partitioning information in query processing. While random and round-robin partitioning result in nearly perfectly balanced partition sizes, their information cannot be exploited in query optimization. On the other hand, range partitioning can be used for partition pruning in queries containing filters and hash partitioning can be exploited for partition-local execution of joins and aggregations when the partition key column matches the join or grouping key. However, load balancing hardly depends on the value distribution, leading to possibly imbalanced partitions for skewed data.

Value based partitioning strategies face a severe issue: The quality of partitioning heavily depends on the choice of the partition key to find a balanced partitioning, but also to allow partition-local execution of queries in as many cases as possible. When a user does not know which partition key to choose and the partition key consequently does not match the join or grouping column in a query, data needs to be repartitioned during query execution in order to allow parallel query execution. This repartitioning involves expensive network transfer between workers and can lead to bad query performance when choosing a wrong partitioning key, as shown for a simple aggregation query in Figure 4.12. Defining multiple partition keys (i.e. having a multi-key partitioning) is not possible here without replicating the data to store it in different ways.

Use cases for multi-key partitioning occur frequently in real world. Modern data warehouse applications contain a large number of fact and dimension tables

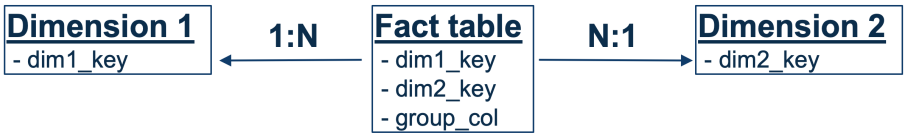


Figure 4.13: Minimized use case example: On which column to partition the fact table on?

combined in a complex schema. A minimal example for a fact table referencing two dimension tables is given in Figure 4.13. In addition to the foreign keys, there might be also columns in the fact table that are frequently used for aggregations, e.g. market segments or business units. So which column should be used as the partitioning key? Partitioning on one of the foreign keys results in a fast join with the respective dimension, but slower joins on other dimensions and slow aggregations on any other grouping column. Ideally, we would therefore like to partition the table on all dimension keys and expected grouping keys at the same time.

Robust query performance is an increasingly important design goal for data warehouses. It aims at achieving reasonably good (but maybe non-optimal) performance for many different workloads. Trying to find optimal query plans and chasing the last percentages of performance can lead to hitting worst-case query plans in unknown workloads due to complicated query optimizations. Robust query performance waives the aim to find optimal query plans and focuses on avoiding bad performance. For the example in Figure 4.12 robust performance could be slightly worse than optimal performance, but consistent for all different grouping keys. A workload-agnostic approach aims at reaching reasonable performance for an unknown workload. We do not want to optimize for a known workload that favors a certain set of e.g. grouping keys.

As partitioning has a major impact on query performance, we introduce a partitioning strategy that allows the definition of multiple partition keys without replicating the data in this section. We define the patched multi-key partitioning constraint, which captures the idea that a table is partitioned on multiple partition keys when excluding a set of exceptions for each of the columns. We discuss why current partitioning schemes are not sufficient for multi-key partitioning and show that we can find a patched partitioning of a table by mapping the table to a graph and applying existing graph partitioning algorithms. Furthermore, we present how a patched multi-key partitioning can be integrated into query optimization. In our experiments we show the opportunities of our approach to achieve robust query performance for multiple partition keys.

4.4.1 Multi-key Partitioning

In the following we propose definitions for a multi-key partitioning function and a balanced partitioning and discuss drawbacks of naive ways of achieving multi-key partitioning using hash partitioning. We focus on multi-key partitionings for two keys. However, all statements can be extended to more partition keys in an obvious way.

Definitions

Definition 4.4.1. Multi-key partitioning function (2 keys)
Let relation R be a set of tuples t, $cols(R)$ be the set of columns of R, columns A,$B \in cols(R)$ be the partition keys and n be the number of partitions. Further we denote $t(X)$ as the value of column X of tuple t. We define a multi-key partitioning function for two partition keys as a function $p : t \in R, n \in \mathbb{N} \mapsto i \in \{1, \cdots, n\}$ with properties:

(MK1) $\forall t_1, t_2 \in R : t_1(A) = t_2(A) \Rightarrow p(t_1, n) = p(t_2, n)$

(MK2) $\forall t_1, t_2 \in R : t_1(B) = t_2(B) \Rightarrow p(t_1, n) = p(t_2, n)$

We call **(MK1)** *Partition locality for A* and **(MK2)** *Partition locality for B*, which intuitively means that all tuples sharing the same attribute value in one of the partition keys are assigned to the same partition. The partition locality is required for partition-local execution of joins and aggregations. Only if tuples holding the same column value for a column X are assigned to the same partition, we can ensure that all matching join partners in joins or all group members in aggregations can be found in the same partition when joining or grouping on that column X. Consequently, expensive data repartitioning is not needed in this case. Combining **(MK1)** and **(MK2)** means that we aim at enabling this partition-local execution for all subkeys of a combined partition key, i.e., we want partition-local execution on both columns A and B separately if the table was partitioned on (A, B), independent on the column values of the respective other key. Please note that as a result of providing partition locality for each single-column subkey of the partition key, the multi-key partition function automatically provides partition locality for all multi-key (sub)keys of the partition key, including the whole partition key itself. This can be easily seen by the conjunction of **(MK1)** and **(MK2)** and also holds for all subkeys of multi-key partition keys on more than two columns. Besides the requirement to enable partition-local query execution, we also require a partition function to produce a balanced partitioning to avoid load imbalance.

Definition 4.4.2. Balanced partitioning
We call a partitioning of table into partitions P_i with $i \in \{1, \cdots, n\}$ balanced with an imbalance factor $\alpha = \frac{\max_{i \in \{1,\cdots,n\}}\{|P_i|\}}{\min_{k \in \{1,\cdots,n\}}\{|P_k|\}}$.
Obviously, we want to have an imbalance factor α near 1 indicating a nearly balanced partitioning.

Multi-key Hash Partitioning

Hash partitioning is the most commonly used partitioning strategy, as it is easy to compute and provides partition locality in case of a single partition key. There are multiple strategies for applying hash-partitioning on multiple keys without data replication. As a first option, the partition keys can be combined to a surrogate key that is used as the input for hashing. Consequently, partitioning on columns (A, B) a tuple t with $t(A) = a$ and $t(B) = b$ is assigned to partition P_i with $i = p(a \circ b, n)$, with

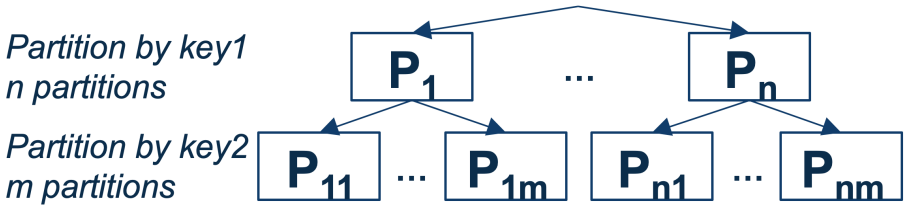


Figure 4.14: Hierarchical partitioning for two keys

p being a hash-based partitioning function mapping into the value range $\{1, \cdots, n\}$ and $\circ$ being a function to combine key values to a surrogate key, e.g., concatenating, interleaving or arithmetic functions. This strategy has two drawbacks: In case of A and B not sharing the same data type, we need a mapping or conversion of the column values to a unified type the partitioning function can be applied on. More severely, we can only ensure that the hash-based partitioning function p assigns tuples to the same partition if the arguments of p are equal for both tuples. To the best of our knowledge, there is no function $\circ$ that produces the same result combining a constant a and different values for b and combining a constant b with different values for a at the same time. An exception here is the constant function producing an arbitrary constant c, which would assign all tuples to a single partition and is therefore no option. As a result, this strategy does not match the desired partition locality requirement.

As a second strategy for applying hash partitioning on multiple keys, the hash function can be applied to the keys separately and the hashed values are combined afterwards. When partitioning on columns (A, B) a tuple t with $t(A) = a$ and $t(B) = b$ is assigned to partition P_i with $i = (p(a, m) \circ p(b, m)) \bmod n$, again with p being a hash-based partitioning function mapping into the value range $\{1, \cdots, m\}$ and $\circ$ being a function to combine the hash values. Similarly to the first strategy, this strategy does not match the partition locality requirement as there is no such function $\circ$ that ensures partition locality in both arguments at the same time.

A third option for achieving multi-key partitioning without data replication is hierarchical partitioning, as shown for two partition keys in Figure 4.14. Data is partitioned on a single key first, and each resulting partition is further partitioned using the remaining keys. For two partition keys, a partition P_{ik} contains all tuples that are assigned to partition i according to the first key and to partition k according to the second key. When requiring a partitioning on the first key in a query, a partition P_i consists of the union of all subpartitions $P_{ik}, k \in \{1, \cdots, m\}$, while a partitioning on the second key is constructed by the union of $P_{ki}, k \in \{1, \cdots, n\}$ for a partition P_i. This hierarchical approach has several drawbacks:

1. Hierarchical partitioning is prone to skew. If there is a correlation between partitioning keys, it might occur that only a subset of values occur in subpartitions after partitioning on the first key. This might lead to skewed partition sizes and can be shown using an example on the CommonGovernment table of the PublicBI benchmark [154] that we use in our evaluation in Section 4.4.6. Using hierarchical partitioning with $n = m = 4$ on *Vend_vendorname* first and on *Co_name* second leads to a partitioning where the largest partition is 3x larger

than smallest partition. Using *Bureau_name* as the first partition key before *Co_name* however leads to a factor 15 between the smallest and the largest partition size. The reason for this is the column correlation. *Bureau_name* is highly correlated to *Co_name* (see Figure 4.20), so 99% of the distinct *Co_name* values can be found in only one of the four partitions after the first partition step. *Vend_vendorname* however does not have a strong correlation to *Co_name*, so around 80% of the unique *Co_name* values can be found in all partitions after the first partitioning step, which is more similar to a random distribution and consequently leads to more balanced partitions after the second partitioning step.

2. Partition responsibilities change during query execution. An executor is responsible for partitions $P_{i\_}$ when requiring a partitioning on the first key, while being responsible for $P_{\_i}$ when requiring a partitioning on the second key. Nearly all partition responsibilities are moved between queries, making e.g., buffered data useless.

3. The number of partitions increases rapidly with the number of columns and cores. The total number of partitions is $\prod_{i=1}^{num\_cols} n_i$, with n_i being the number of partitions for key column i. In order to avoid unused cores during query execution, we require $n_i \geq num\_cores; 1 \leq i \leq num\_keys$. For an example machine of 128 cores and three partitioning keys we would need at least 2^{21} partitions. This leads to an non-negligible overhead in query execution due to additional metadata and partition assignments.

4. The order of partition columns impacts load balance.

5. Due to changing node responsibilities for partitions, a shared-disk approach is a requirement for hierarchical partitioning.

Changing data responsibilities impacting buffering is a major drawback of hierarchical partitioning. (3) and (4) could be mitigated by applying a physical allocation strategy of the partitions to physical partitions, e.g., in [145]. (5) is a minor drawback here, as data access for all nodes on all data is typically ensured by either a cloud file system or a shared file system in an on-premise cluster setup. The access however might be non-uniform, having local and remote reads like in HDFS.

4.4.2 Patched Multi-key Partitioning

In order to mitigate the drawbacks discussed in Section 4.4.1, we relax the multi-key partitioning definition and introduce patched multi-key partitioning, i.e., allowing exceptions to the partitioning constraint. We show how to find patched multi-key partitionings using graph partitioning algorithms or iterative algorithms.[2]

[2]Available under `https://github.com/dbis-ilm/Patched_Multi-key_Partitioning`.

Algorithm 1: Iterative graph partition assignment

Input : Relation $R = \{t_i | 0 \leq i \leq n\}$,
Partition keys $K \subseteq cols(R)$

Output : Set of partitions $P = \{P_i | P_i \subseteq R, i \in \mathbb{N},$
$\bigcup P_i = R, \forall i, k \in \mathbb{N} : P_i \cap P_k = \emptyset\}$

```
1  P ← {}
2  while R \ ⋃P ≠ ∅ do
3      ΔR ← R \ ⋃P
4      X ← {}
5      X' ← {t} for some t ∈ ΔR
6      while X ≠ X' do
7          X ← X'
8          for k ∈ K do
9              // No duplicates due to set semantics, so no need to adapt ΔR
10             X' ← X' ∪ {t ∈ ΔR | ∃t' ∈ X' : t(k) = t'(k)}
11     P ← P ∪ {X} ;
12 return P
```

Motivation

Based on Definition 4.4.1 there is a straight-forward and inevitable approach to find a multi-key partitioning on a given dataset. This approach is sketched in Algorithm 1 and serves as a starting point for further discussions. The main idea is to iteratively build data partitions by starting at a single tuple and adding tuples in order to match **(MK1)** and **(MK2)**, which means that in one step we add all tuples with matching values in one of the partition keys, and repeat until we do not find any tuples to add. If there are still unassigned tuples we repeat this behaviour starting with an unassigned tuple. The resulting partitions can then be used directly as table partitions or assigned to physical partitions using a physical allocation strategy as e.g., in [145]. An example sequence of this algorithm is shown in Figure 4.15. Starting with tuple t_1 in the first partition P_1, we add t_2 in the first step as t_2 also contains the column value $t_2(A) = 1$. The loop will now terminate after checking that no more tuples share values for columns A or B with tuples in partition P_1. As not all tuples are assigned yet, we start a second partition P_2 initially containing t_3. t_5 is added next due to a matching value in column A and t_4 is added due to matching value in column B. The final partitioning consequently consists of two partitions.

We implemented the algorithm using a recursion of SQL semi-joins and unions for initial experiments. These experiments revealed the major drawbacks of the naive approach, as it produces a single large partition for most datasets and partition keys. Even if it discovers multiple partitions in the data, the number of physical partitions and their size is determined by this "natural" number of partitions in the given dataset. We can allocate partitions to physical partitions to reduce the number of partitions, but we can not fill more partitions than "naturally" discovered. However, if we would agree that the value $A = 2$ is an exception and tuples holding

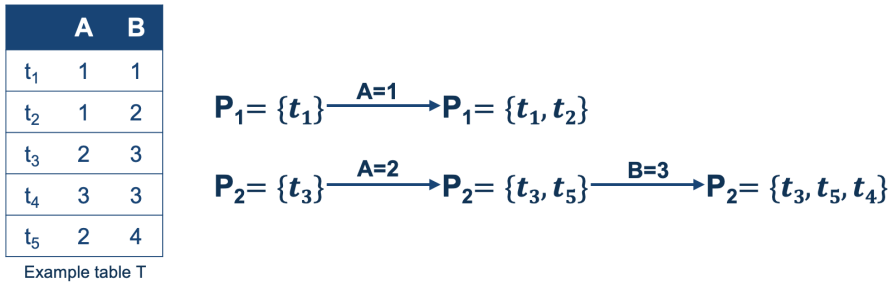


Figure 4.15: Sequence of the naive multi-key partitioning

the value $A = 2$ are not necessarily assigned to the same partition, tuple t_5 would not be assigned to P_2 and would consequently form a separate partition P_3. A query containing a grouping on column B could be executed partition-locally without the need for data repartitioning. Grouping on A also does not require to repartition all tuples, but only the tuples holding the exception value $A = 2$. This example motivates the definition of a patched multi-key partitioning.

Definition

We want to allow marking tuples as exceptions to the multi-key partitioning as defined in Definition 4.4.1 in order to find more balanced partitions and to be independent from the "natural" partitions of a given dataset. Keeping in mind that PatchIndexes allow the definition of approximate constraints by maintaining sets of patches, we therefore relax the definition of a multi-key partitioning as follows:

Definition 4.4.3. Patched multi-key partitioning (2 keys)
Let relation R be a set of tuples t, $cols(R)$ be the set of columns of R, columns $A, B \in cols(R)$ be the partition keys and n be the number of partitions. Further we denote $t(X)$ as the value of column X of tuple t and assume the existence of a set of patches P_c for every partition key column $c \in \{A, B\}$. We define a patched multi-key partition function for two partition keys as a function
$p : t \in R, n \mapsto i \in \{1, \cdots, n\}$ with the following properties:

(PMK1) $\forall t_1, t_2 \in R_{\setminus P_A} : t_1(A) = t_2(A) \Rightarrow p(t_1, n) = p(t_2, n)$

(PMK2) $\forall t_1, t_2 \in R_{\setminus P_B} : t_1(B) = t_2(B) \Rightarrow p(t_1, n) = p(t_2, n)$

Intuitively, **(PMK1)** and **(PMK2)** relax the *Partition locality* to tuples that are not included in the set of patches for the respective column. The definition also does not make any claims about the size of the set of patches, i.e., a trivial solution would be to assign all tuples to the sets of patches. The main goal is now to construct both a partition function and small sets of patches for the partition key columns, such that the defined constraints are met.

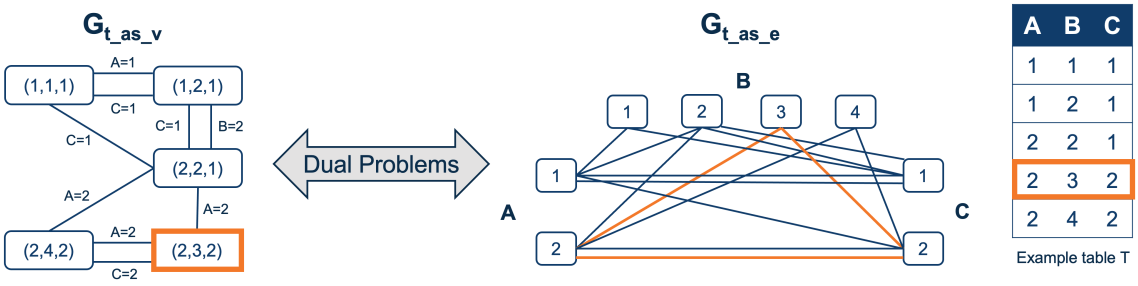


Figure 4.16: Graph constructions for example table with 3 partition keys

4.4.3 Graph Partitioning

The naive algorithm presented in Section 4.4.2 showed that the problem of finding a (patched) multi-key partitioning intuitively can be transferred to graph algorithms. The constraints that once a tuple is assigned to a partition, every other tuple sharing the same value in at least one of the partition key columns must be assigned to the same partition, are similar to following paths in a graph to discover its connected components. The "natural" partitions that Algorithm 1 discovers are thereby the connected components of a graph constructed from the tuples of the given dataset. We formally define a mapping of relations to graphs and exploit existing graph partitioning algorithms as a first approach for meeting the goal of finding sets of patches and a partitioning function such that Definition 4.4.3 is met. The main concept of using graph partitioning algorithms can be depicted as follows:

1. Map the input table to a graph based on the partition keys.
2. Apply a graph partition algorithm on the graph. This results in an assignment of vertices/edges to a graph partition.
3. Map the graph partitions back to table partitions.
4. Define set of patches based on overlapping vertices or edges.

By defining the mapping between graphs and tables, this approach can be generalized to arbitrary graph partitioning algorithms. In the following, we describe generally applicable mappings. Only the actual storage layout of the graph is dependent on the expected input layout of a given graph partitioning algorithm.

Table to Graph Mapping

Mapping a table to a graph is based on the chosen set of partitioning keys and can be done in two general ways: Modeling tuples of the table as vertices or as edges, resulting in dual representations depicted for an example table in Figure 4.16 and defined as follows:

Definition 4.4.4. Graph Construction, Tuples as Vertices
Assume a Relation R and a set of partition keys $P \subseteq cols(R)$. We define a multigraph $G_{t\_as\_v}(R) = (V, E)$ that represents the relation R with $V = \{t | t \in R(P)\}$ and $E = \{(u, v)_k | u, v \in R, u \neq v, k \in P : u(k) = v(k)\}$.

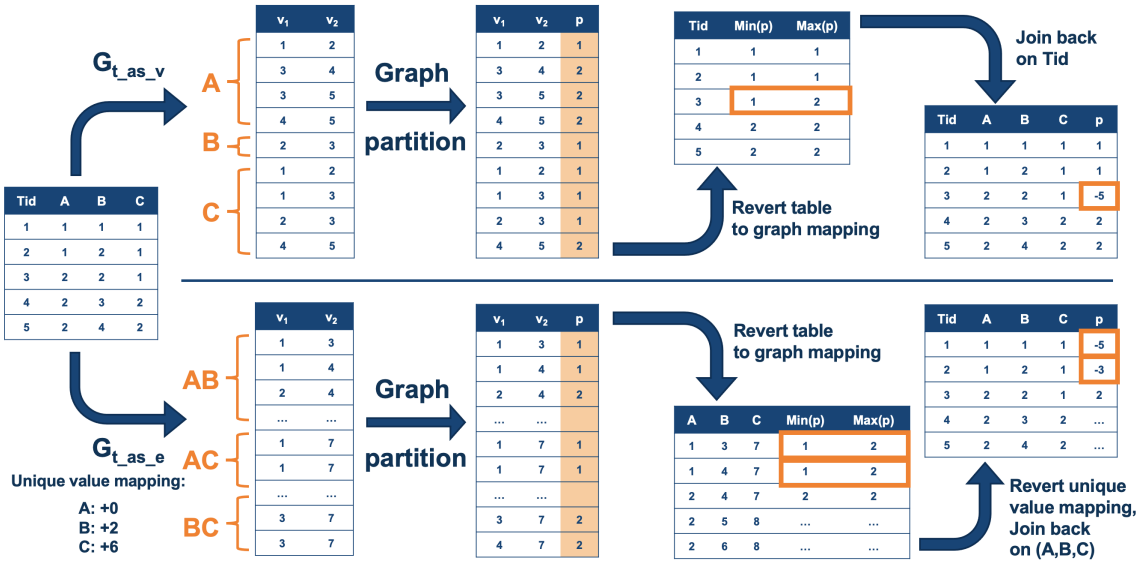


Figure 4.17: Graph partitioning example workflow

Definition 4.4.5. Graph Construction, Tuples as Edges
Assume a Relation R and a set of partition keys $P \subseteq cols(R)$. We define a multigraph $G_{t\_as\_e}(R) = (V', E')$ that represents the relation R with $V' = \{v | \exists k \in P : v \in R(k)\}$ and $E' = \{(u, v)_t | t \in R, k_1, k_2 \in P, k_1 \neq k_2 : t(k_1) = u \wedge t(k_2) = v\}$.

Intuitively, the graph $G_{t\_as\_v}(R)$ contains a vertex for each distinct combination of partition key values of R and an edge between two vertices for each shared partition key value. On the other hand, $G_{t\_as\_e}(R)$ is a multipartite graph and contains a vertex for each distinct partition key value and an edge between two vertices for each tuple in R which contains the respective partition key values. As columns might share column values leading to conflicts in vertex identifiers, we use a unique mapping between partition key values and vertex identifiers in the implementation. We defined both graph constructions as multigraphs, which could be replaced by constructing a weighted graph with weights indicating the number of tuples holding the pair of values in the case of $G_{t\_as\_e}$ or the number of equal partition key values in the case of $G_{t\_as\_v}$. Both approaches work similarly and the choice between them is only impacted by the expected input of the graph partitioning algorithm. Conceptually, an edge with a large weight indicates the same graph connectivity as multiple edges between vertices. In Figure 4.16, we used the multigraph construction, so there are two edges between the vertices $(2, 4, 2)$ and $(2, 3, 4)$ in $G_{t\_as\_v}$ or two edges between $A = 2$ and $C = 2$ in $G_{t\_as\_e}$ respectively. The resulting temporary graphs can be larger than the original table. With n being the number of tuples in the base table, d_i being the number of distinct values in column i and k being the number of partition key columns, graph $G_{t\_as\_v} = (V, E)$ has size $|V| = n$ and $0 \leq |E| \leq k \cdot \sum_{i=1}^{n-1} i$ (depending on the common partition key values of e tuples; minimum: all unique, maximum: all tuples equal) and graph $G_{t\_as\_e} = (V', E')$ has dimensions $|V'| = \sum_{i=1}^{k} d_i$ and $|E'| = n \cdot \sum_{i=1}^{k-1} i$, particularly larger than n as tuples can be represented by multiple edges for more than two partition key columns.

In our implementation we realized both constructions over SQL queries to create the graph representation from a given database table. The required graph representation is determined by the expected input of the used graph partitioning algorithm. In our case, we produce a flat file containing a list of edges. Figure 4.17 shows an example end-to-end workflow for the graph partitioning approach. For $G_{t\_as\_v}$ we use tuple identifiers as vertex identifiers and calculate edges by performing a self join of the table for every partition key separately and combining the results using a union. For $G_{t\_as\_e}$ we use the column values of the partition keys as vertex identifiers and apply a unique column mapping (in the example: adding the running sum of max values of other partition keys). We project all two-element subsets of the partition key, apply the unique column value mapping and union the results of every subset.

Running the Graph Partitioning

In the next step, we apply a graph partitioning algorithm on the constructed graph. As discussed in Section 4.1, there is the choice between vertex-cut and edge-cut algorithms, and both alternatives can be applied to both possible graph constructions. While vertex-cut algorithms assign edges to partitions and cut vertices from the graph, edge-cut algorithms assign vertices to partitions and cut edges from the graph. Cutting the equivalent of tuples in the graph, i.e., edge-cut in $G_{t\_as\_e}$ and vertex-cut in $G_{t\_as\_v}$, cuts whole tuples and consequently leads to a single set of patches for the whole table. Using the respective other variant leads to a single set of patches for each of the partition key columns. This offers more flexibility, so we expect smaller sets of patches when later querying columns of the partition key separately.

The remaining two options, i.e., performing an edge-cut on $G_{t\_as\_v}$ or a vertex-cut on $G_{t\_as\_v}$ also show some challenges. In $G_{t\_as\_v}$, edges represent a common column value between two tuples. Removing an edge from the graph however only has a local impact and does not impact other edges representing the same column value. As we described in Section 4.4.2 and will describe formally when describing query execution using PatchIndexes in Section 4.4.5, we need to repartition all tuples sharing a column value that was marked as an exception. Consequently, we would need to remove all edges with a common column value when removing one of them, e.g., removing all edges representing $A = 2$ in Figure 4.16. Conceptually, this issue is caused by the fact that a single column value is represented by multiple graph objects, i.e., multiple edges in this case. On the other hand, $G_{t\_as\_e}$ shares a similar challenge. Here, tuples are represented by multiple edges. Performing a vertex-cut on $G_{t\_as\_e}$ assigns edges to partitions and could therefore possibly assign a tuple to different partitions, which leads to conflicts when mapping the graph back to the table.

The property of finding a balanced partitioning and minimizing the number of cutted graph objects (and minimizing the sets of patches later) relies on the properties of the graph partitioning algorithm. As we focus on embedding any given graph partitioning algorithm into relational table partitioning in this approach, discussions about the quality of the partitioning produced by a choosen algorithm is out of scope of this work. As an example, we chose to use the vertex-cut algorithm Distributed Neighbor Extension [67] on both the $G_{t\_as\_v}$ and the $G_{t\_as\_e}$ construction, comparing the effects described above. DistributedNE starts with a random edge

per partition, iteratively grows partitions in parallel using neighbor expansion and performs synchronisation between the expansion rounds. The algorithm finds a local optimum, but due to the random starting point selection it does not guarantee a global optimum.

Graph to Table Mapping

Mapping the partitioned graph back to the relational table is the last step and includes solving the two main goals stated in Section 4.4.2, namely to find a partition function and sets of patches for partition key columns in order to meet the constraints of a patched multi-key partitioning. After applying the graph partitioning algorithm that assigned graph elements to partitions we reverse the graph construction by adding the assigned partition to the respective tuple as an additional table column. This way, we materialize the partitioning function as a mapping directly in the table. Afterwards we repartition the table on the graph partition column using regular hash partitioning. DistributedNE performs a vertex-cut, so it assigns edges to partitions. In order to reverse the table to graph mapping we create a table of schema $(vertex1, vertex2, partition\_id)$ from the partitioning result file like shown in Figure 4.17 in order to perform the mapping again over SQL. For $G_{t\_as\_v}$ we produce a mapping of partition identifiers to tuple identifiers by projecting $(vertex1, partition\_id)$ and $(vertex2, partition\_id)$ and union them, before grouping on the vertex column and computing the minimum and maximum partition_id. If minimum and maximum match, the tuple was not cut from the graph by the vertex cut and we can apply the partition_id. Otherwise we apply a negative partition_id and the tuple gets assigned to the sets of patches later. For $G_{t\_as\_e}$ we split the table to reconstruct the two-element partition key subset based on filter predicates (using the unique value mapping). Afterwards we join these parts to reconstruct a mapping table of schema $(partition\_key\_columns, partition\_id)$. Again, we group on the partition key columns and calculate the minimum and maximum partition_id to discover and mark conflicts like in the $G_{t\_as\_v}$ case. For both cases we need to ensure that the partition_ids calculated by the graph partitioning are also assigned to different physical table partitions, which is not necessarily ensured, i.e., it is not ensured that partition_ids between 1 and n are indeed assigned to n different physical partitions. We therefore update the partition_ids with values that are ensured to be assigned to different partitions. These can be easily calculated if the hash function used for hash partitioning is exposed over the SQL frontend of the database system. The reverted mapping tables are finally joined back with the fact table on the tuple identifier for $G_{t\_as\_v}$ or on the partition keys for $G_{t\_as\_e}$ in order to update the materialized partition_id column.

In the last step, we need to identify the sets of patches. Graph elements that have adjacent graph elements with different partitions are exceptions, which are removed from the graph in the graph partitioning. Similarly, we query the table for each partition key to identify partition key values holding more than one distinct partition_id or a negative partition_id indicating a conflict from the graph to table mapping as described in Section 4.4.5. These tuples are declared as exceptions and assigned to the set of patches for the respective partition key column. In the upper case of the example in Figure 4.17, all tuples with $A = 2$ would be included in the

set of patches P_A for column A, tuples with $B = 2$ in P_B and tuples with $C = 1$ in P_C respectively, because tuples holding these values have either more than one distinct partition_id or a negative partition_id.

Discussion

The approach of mapping a relational table to a graph, using existing graph partitioning algorithms to partition the constructed graph and infer the table partitioning from the graph partitioning is an elegant way of combining both worlds to apply the well-known and well-proven characteristics of graph algorithms to relational tables. However, it also has some drawbacks. First, the dataset must be completely loaded before graph partitioning can be invoked. This is fundamentally different from a typical data ingestion pipeline, where tuples are iteratively loaded (in parallel), and decisions on partition assignment are made locally to a tuple instead of considering the whole table. Consequently, we need to repartition the table after it was already loaded to apply the graph partitioning. Second, data layouts in analytical database systems fundamentally differ from data layouts in graph databases, so expensive data reorganisation or data transport would be needed. In our case, we export the constructed graph from the database in order to use it as an input for DistributedNE. Although iterative extensions [45] exist that would enable iterative partition assignments during loading, they would either require reconstructing the graph when data should be appended to an existing table or the constructed graph needs to be maintained as a separate copy of the data during the table lifetime. For both reasons, the approach described above might be hardly applicable in practice, but motivates the design of a iterative patched multi-key partitioning approach.

4.4.4 Iterative Patched Partitioning

Using graph partitioning algorithms require the full dataset to be present and are difficult to integrate due to the different data layouts of relational databases and graph systems. However, algorithms like DistributedNE ensure minimal sets of cuts (which translates to minimal sets of patches) under given balancing constraints. In order to make the approach more usable in practice, we design an iterative patched partitioning strategy as an alternative approach in this section which is able to decide on partition assignments upon coming across a single tuple at the price of loosing the optimality of the resulting partitioning. The main idea of the approach is to iteratively build and "color" a graph. Similar to hash partitioning, we thereby make a partition assignment decision just in the moment the tuple is loaded into the table. Our example algorithm shown in Algorithm 2 is based on the $G_{t\_as\_e}$ construction and intuitively holds an assignment of values to partitions for each column of the partition key. For each tuple, we need to distinguish between three cases. If there is no partition assignment for any of the partition key values yet, we apply a partition function f on it. If there are some key values that already have an assignment (because they already occured before) and all of these partition_ids match, the tuple is assigned to this partition, matching the intuition that tuples sharing a partition key value are assigned to the same partition. If however there are key values already assigned to a partition and these assignments do not match, we have a conflict. This

Algorithm 2: Iterative graph partition assignment

```
   Input  : *part_key_values, num_keys
   Output : partition_id
 1 lookups ← part_assignment_lookup(part_key_values);
 2 if allNULLOrExceptions(lookups) then
 3 |   part ← f(part_key_values) ;                          // Case 1
 4 else if allNULLOrExceptionOrEqual(lookups) then
 5 |   part ← firstAssignment(lookups) ;                    // Case 2
 6 else
 7 |   global rotating_idx ;                          // Case 3:  Conflict
 8 |   while lookups[rotating_idx] == NULL OR
   |     lookups[rotating_idx] == EXCEPTION do
 9 |   |   rotating_idx ← (rotating_idx + 1) % num_keys ;
10 |   part ← lookups[rotating_idx];
11 |   rotating_idx ← (rotating_idx + 1) % num_keys ;
12 global partition_mappings ;
13 for i = 0; i < num_keys; i = i + 1 do
14 |   if lookups[i] == NULL then
15 |   |   partition_mapping[i].insert(part_key_values[i], part)
16 |   else if lookups[i] ≠ part then
17 |   |   partition_mapping[i].update( part_key_values[i], EXCEPTION)
18 return part ;                              // Partition_id for given tuple
```

tuple would conceptually connect two connectivity components in the iteratively built graph. We can't revert previous decisions to join these connectivity components and decide for one partition key to be the decisive key, so the tuple will be an exception for every other column when later discovering the sets of patches. The choice for the decisive key follows a rotating schema in order to distribute exceptions over all partition key columns. In all cases, we insert the assigned partition_id into the assignment for every column that did not have an assignment before or update to the exception marker when hitting case 3, i.e., there is a partition_id for a value which was not chosen. The exception markers are handled similarly to NULLs in case 1 and case 2, because they are not decisive.

The described algorithm obviously has some drawbacks. Due to the "local" decision in the conflict case, it is not able to find a globally optimal partitioning with a minimum set of patches. Second, it does not have any guarantees about partition balancing. The first case thereby has the most impact on balancing. We track the current sizes of each partition (not shown in Algorithm 2) and the function f assigns a tuple to the smallest partition in the in order to fill partitions as equally as possible. Third, the algorithm is sensitive to the order of tuple insertion. As a simple example depicted in Figure 4.18, inserting tuples that represent a path through a graph would assign all tuples to a single partition, so we rely on the assumption that we hit the first case regularly during tuple insertion to assign tuples to different partitions. Inserting the tuple with partition keys $(2, 2)$ in the upper case of Figure 4.18 and hitting case

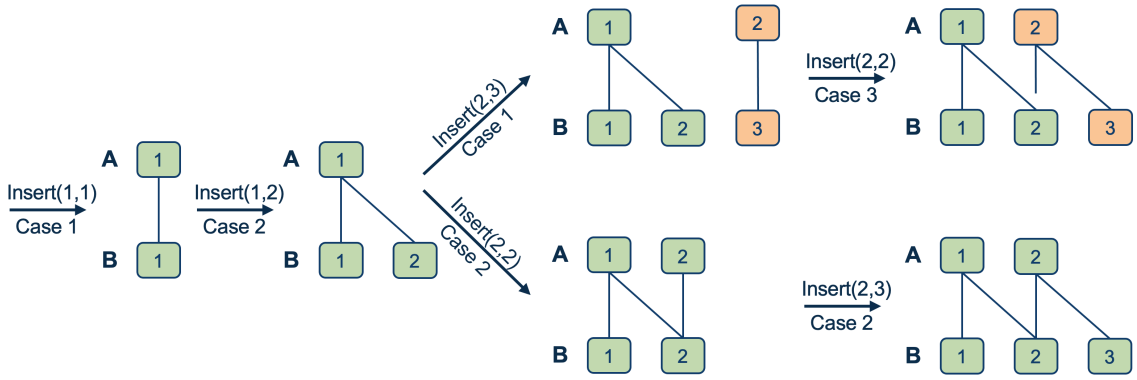


Figure 4.18: Order-sensitivity of iterative patched partitioning

3, the tuple (i.e., the edge) is assigned to the second partition according to key A (indicated by not connecting the edge), so all tuples with $B = 2$ will later belong to the set of patches P_B as they belong to different partitions. Due to the rotating index for choosing the decisive column, column B would be chosen as the decisive column when hitting case 3 for the next time in order to balance patches over all partition keys. However, having a decision making process "local" to a tuple enables on-the-fly partition assignments during data loading, avoiding expensive conversions and repartitionings of the graph partitioning approach presented in Section 4.4.3. The algorithm is designed as a single-threaded approach. In order to be integrated into parallel loading, access to the metadata structure must be secured using locks during Algorithm 2 between reading the partition assignments and updating them. As this is the major part of the algorithm, an optimistic approach would be favourable here, i.e., performing the partition assignment lookup again before updating to ensure that no updates were performed in the meantime. In order to make it openly available, our implementation contains a standalone version of the algorithm not integrated into data loading operators inside a database kernel. The standalone version processes input files and produces an output file with an additional column for the computed partition_id.

4.4.5 Query Processing

As described in Section 4.2.6, PatchIndexes are generic data structures that can be extended to arbitrary constraints by implementing an interface for PatchIndex creation, maintenance under updates and adding an optimizer rule to exploit PatchIndex information in query optimization. In the following we describe the respective adaptions for integrating the patched multi-key partitioning constraint.

PatchIndex Creation

After running one of the graph partitioning algorithms described in Section 4.4.2 we create PatchIndexes on the partition key columns in order to maintain the exceptions to the partitioning. We follow the approach of having sets of patches per column, so we create separate index instances on them. The sets of patches maintain the tuple identifiers of all exceptions to the partitioning, i.e., all tuples that have a partition

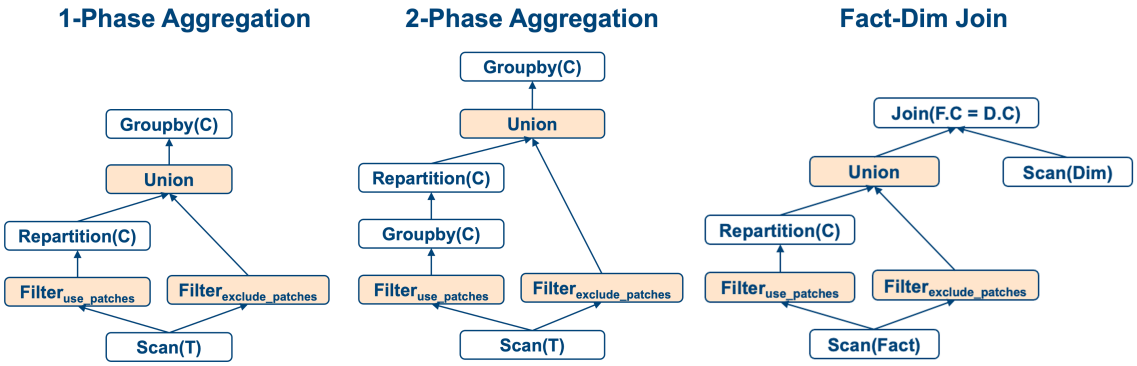


Figure 4.19: Patched query plans after applying the PatchIndex optimization rule (added operators highlighted)

key value that is present in more than one partition or have a negative partition_id indicating a conflict when mapping the graph back to the table. We discover the partition key values violating the partition constraint by (1) running an aggregation query counting the distinct partition_ids of column values and selecting the ones having more than one distinct partition_id and (2) running a filter query to find all values with negative partition_ids. The union of (1) and (2) is joined back with the table on the respective key column to find all tuple identifiers to be inserted into the PatchIndex. In the upper case of Figure 4.18 all tuples with $B = 2$ are exceptions, so tuple identifiers of tuples $(1, 2)$ (assigned to the first partition) and $(2, 2)$ (assigned to the second partition) are inserted into the PatchIndex holding the set of patches P_B. Partitioning is transparent for the PatchIndex, so an index instance is created per partition.

Analytical Queries

Materializing approximate constraints with PatchIndexes generally follows the idea to split query execution using the PatchIndex scan between tuples that match a certain constraint and exceptions to the constraints. The patched multi-key partitioning constraint defined in Definition 4.4.3 allows us to avoid expensive repartition/shuffle operations as we know that tuples not included in P_X for a given column X match the Partition locality criteria. Consequently, these tuples do not need to be reshuffled.

Partition locality plays an important role for aggregations and joins. For executing a partitioned aggregation, two basic approaches exist if the grouping column does not match the partition key of the table. First, data can be reshuffled on the grouping key and the subsequent aggregation can be executed partition-locally. This approach introduces large network overhead for repartitioning. Second, one could perform a pre-aggregation on the existing partitions, then reshuffle data and perform a post-aggregation afterwards, which is necessary as not all elements of a group are initially placed in the same partition. Here one needs to carefully consider the aggregation function, that is required to be decomposable, e.g., an average can't be executed in a stepwise fashion without breaking it into a sum and a count. The 2-phase aggregation is favorable when data contains few distinct values according to statistics, so the additional aggregation effort is amortized by the less effort to

shuffle data. Both variants can be enhanced by the patched partitioning constraint. As shown in Figure 4.19, we can skip the repartitioning (for the 1-phase aggregation) or the pre-aggregation and repartitioning (for the 2-phase aggregation) for all tuples that are not included in the set of patches when aggregating on a patched partition key. This is possible because either all tuples or no tuples sharing the same partition key value are handled as exceptions. Consequently we reduce the number of tuples that need to be either reshuffled or pre-aggregated and reshuffled. Splitting the datastream however also introduces overhead. As the 2-phase aggregation is typically chosen when the grouping column only has a small amount of unique values, we expect that the benefit of the PatchIndex optimization might be too small to amortize the overhead of added operators. As the number of exceptions are known during query optimization, the costs of both plans with and without the optimization can be estimated and the optimizer can decide between them. Except adding linear costs for the filter operators with modes *use_patches* and *exclude_patches*, no changes are needed in the cost model of an arbitrary optimizer.

Join operators can also be performed in a partitioned way if tables are partitioned on the join key, otherwise an expensive repartitioning is needed again. In typical data warehouse applications, many joins are performed between fact tables and dimension tables. While the latter ones are typically partitioned on their primary keys, choosing the partition key of fact tables consisting of many foreign keys is typically difficult. When we included the foreign key in the patched multi-key partitioning and have a PatchIndex on the join column, we can again avoid the repartitioning for all tuples not included in the set of patches. This case however requires the second table to be co-partitioned, which is automatically ensured by using the same partitioning function in e.g., hash-partitioning, but is not guaranteed in our graph partitioning approach. We therefore derive the partitioning of the primary key side of the join from the fact table when the foreign key was included in the multi-key partitioning computation. Consequently, dimension key values that do not belong to the respective set of patches in the fact table derive the same partition_id, while exceptions are assigned using common hash partition to be co-partitioned with the fact table exceptions after repartitioning. Deriving the partitioning is a problem for shared dimensions, as only one fact table can be decisive for the dimension table partitioning. However, it is conceptually better to repartition a smaller dimension table than a large fact table.

In contrast to hash partitioning where data is not required to be reshuffled only if the partition key matches the grouping/join key, the described optimizations work with every key that was part of the patched multi-key partitioning. So instead of a fast query when hitting the right key and slow queries for all other keys in case of hash partitioning, we aim at reaching robust performance for aggregate/join queries on all keys of the multi-key partition key. The performance will depend on the amount of tuples that are categorized as exceptions and assigned to the sets of patches. Additionally, partition responsibilities of nodes do not change when querying different partition keys, so patched multi-key partitioning also works for shared-nothing architectures. Node-local buffers are not invalid for different queries, which is the major drawback of hierarchical partitioning approach described in Section 4.4.1.

The queries shown in Figure 4.19 are also allowed to have additional operators below the groupby/join operator, e.g., additional filters or projections. These additional operators would be replicated to both datastreams including/excluding patches, so they are applied to both datastreams separately. As a current limitation, these operators are not allowed to be aggregations or joins, so the PatchIndex optimization rule is currently only applied on the lowest aggregation/join in the query tree.

Update Queries

Updatability is an important factor to make a concept applicable in practice. Therefore, we need to maintain the patched partitioning constraint also under update operations, i.e., inserts, modifies and deletes. The PatchIndex data structure itself is designed in a generic way. The underlying sharded bitmap structure supports update operations and we simply need to implement an interface to define how the set of patches is maintained under updates. Inserts are handled similar to the initial loading, so tuples get a partition_id assigned by the partition assignment algorithms described in Section 4.4.2. However, it might happen that an inserted tuple connects two connectivity components of the graph representation, meaning that partition key column values become exceptions. This can similarly happen for modifies on partition key values. We discover these new exceptions using a join of the inserted/modified tuples with the table itself, performing the same distinct aggregation than during initial partitioning to find values with more than one unique partition_id assigned. We keep the join small by only scanning the inserted/modified tuples on the one side and performing data pruning on the full table based on the observed join keys and small materialized aggregates [111] on the table, to avoid a full table scan. Delete operations do not connect graph components but might split components, which does not harm the partitioning constraint. In general update operations might degenerate the graph and can lead to loosing the optimality of the graph partitioning. By monitoring exception rates this might lead to a recomputation of the partitioning after some time.

4.4.6 Evaluation

With our experiments we show that patched multi-key partitionings can be found in real-world datasets while comparing the presented partitioning approaches and highlight properties of data and algorithms that impact partition quality. Additionally we show opportunities for query performance. We integrated the support for patched multi-key partitionings using PatchIndexes into the Actian VectorH 6.2 system. The system runs on a four node cluster, each node consisting of a Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, 256 GB RAM and a 10 GBit/s network interface. We use the PublicBI benchmark dataset [154] representing real user Tableau workbooks and picked the CommonGovernment workbook as an example. The workbook consists of 110 GB raw data and 43 queries including 13 different group-by columns. For performance evaluation we focus on these analytical queries, as the update mechanism is similar to and extensively evaluated in Section 4.3.4.

Table 4.4: Runtime(in s), imbalance factor and exception rates(in %) of Common-Government table for different partitionings

				1M Tuples						5M Tuples					
Partition keys				Graph			Iterative			Graph			Iterative		
Column1	Column2	Column3	Parts	Time	Imbal.	Exc.	Time	Imbal.	Exc.	Time	Imbal.	Exc.	Time	Imbal.	Exc.
Ag_name	Bureau_name	-	4	46	4.9	(2,0)	39	2.3	(3,0)	100	2.2	(1,0)	175	2.5	(3,0)
Ag_name	Co_name	-	4	43	3.2	(10,1)	39	2.1	(10,1)	102	3.1	(43,1)	174	2.9	(23,1)
Bureau_name	Co_name	-	4	43	3.4	(1,0)	39	2.9	(3,0)	94	3.8	(21,1)	174	3.5	(3,0)
Vend_dunsnumber	Vend_vendorname	-	4	241	2.9	(0,1)	39	1.1	(16,23)	1050	3.1	(1,1)	179	1.01	(17,24)
Ag_name	Bureau_name	Co_name	4	49	2.6	(63,27,13)	40	2.3	(8,1,1)	121	4.2	(76,51,40)	178	2.5	(7,1,1)
Ag_name	Bureau_name	Co_state	4	47	2.4	(98,93,99)	40	1.2	(90,49,96)	110	2.7	(99,97,99)	176	1.4	(90,43,96)
Ag_name	Bureau_name	-	96	102	10K	(37,0)	60	15K	(2,0)	215	2.6K	(14,0)	198	22K	(3,0)
Ag_name	Co_name	-	96	109	13K	(82,1)	66	477	(12,1)	214	388K	(85,1)	199	680	(11,1)
Bureau_name	Co_name	-	96	118	13K	(33,1)	62	450	(3,0)	211	601K	(65,1)	198	557	(3,1)
Vend_dunsnumber	Vend_vendorname	-	96	207	21	(1,1)	63	6.1	(21,34)	560	28	(1,3)	206	8.5	(23,35)
Ag_name	Bureau_name	Co_name	96	117	111	(98,95,82)	60	128K	(8,1,1)	214	1.2	(100,100,100)	229	11K	(20,2,1)
Ag_name	Bureau_name	Co_state	96	114	905	(99,95,99)	63	5.5	(92,81,96)	215	1.3	(100,100,100)	223	5.6	(92,82,97)

We measure execution times for partitioning using the following workflows. For graph partitioning we require the presence of the full dataset. Therefore we start with a pre-loaded table, perform the table to graph mapping, the graph partitioning and the graph to table mapping, before repartitioning the table on the determined partition_id. For the iterative partitioning approach we do not require the full table. We read a fixed flat file line by line and decide the partition assignment per tuple, so the order-sensitivity of the iterative approach does not have an impact on different runs in the experiments. We write the data with the added partition_id back to a flat file and load the resulting table in a single bulk load.

Experiment 1: Partition quality

In the first experiment we show the existence of patched multi-key partitionings. We ran the partitioning on all pairs and triples based on the 13 grouping columns. Table 4.4 shows a subset of the candidates, chosen in order to present properties of our approach, with their respective statistics, i.e., partitioning runtime, imbalance factor and exception rates $e_{Col} = (|P_{Col}|/|R|) \cdot 100\%$ for both partition algorithms. All results show the median of three runs, which is particularly of interest for the graph partitioning algorithm starting at a random initial partitioning. We only use the graph partitioning with the $G_{t\_as\_e}$ mapping, as we observed that the selfjoin in $G_{t\_as\_v}$ leads to a very huge table when joining on columns with only a few unique values and out-of-memory situations. Table 4.4 shows different cases: The "ABC" columns *Ag_name*, *Bureau_name* and *Co_name* only have few hundred distinct values. Consequently, the resulting graph consists of few vertices with dense connectivity. In contrast, the "vendor" columns *Vend_dunsnumber* and *Vend_vendorname* have over 100K distinct values, leading to a graph of many vertices with sparse connectivity. In all cases, the number of edges is equal to the number of tuples by the definition of $G_{t\_as\_e}$.

We want to find a good partitioning indicated by both an imbalance factor near 1 and low exception rates for all columns. From the results we can outline properties that impact partition quality:

- **Column correlation:** If columns correlate, they tend to have value combinations that are more likely than others. This leads to graphs with parts that are more densely connected than others and results in less exceptions when cutting edges from the graph. If columns are not correlated, the resulting

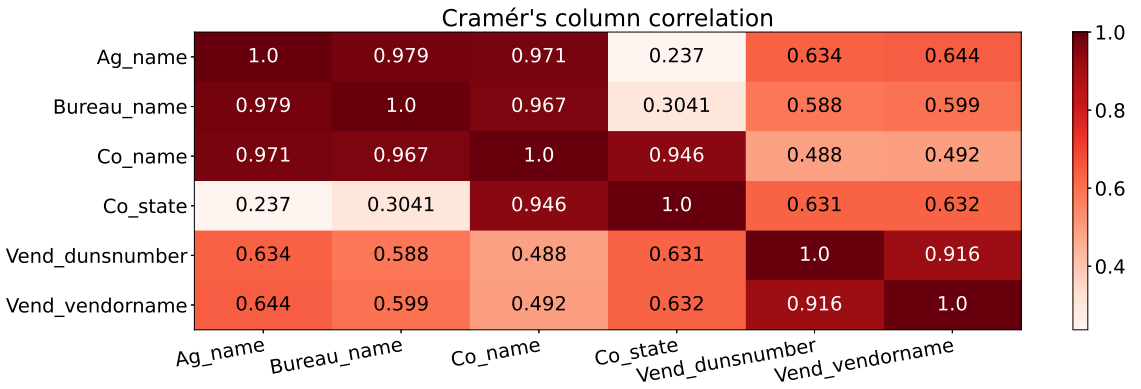


Figure 4.20: Column correlations in CommonGovernment

graph has a more random distribution of edges, leading to a high number of exceptions. Figure 4.20 shows correlations of the nominal "ABC" and "vendor" columns based on the symmetric Cramér's V[37]. We can observe that correlating columns can lead to good partitionings in Table 4.4. *Co_state* does not correlate with *Ag_name* or *Bureau_name* columns and consequently leads to a bad partitioning when included.

- **Number of distinct values:** As the number of edges is fixed by the number of tuples, the number distinct partition key values determine the number of vertices and consequently the degree of connectivity of the graph. We can can find a more balanced partitioning with less exceptions if there are many distinct partition key values in the data, which is the case for the "vendor" columns compared to the "ABC" columns.

- **Number of partitions:** Similar to the number of distinct values, the number of desired partitions has an impact on partition quality. While we can find a reasonable good partitioning for the "ABC" columns with 4 partitions, running the algorithms for 96 partitions leads to very bad partitionings indicated by either a high imbalance factor or high exception rates. Imbalance factor and exception rates are a trade-off here, as shown for, e.g., $(Ag\_name, Bureau\_name)$ at 96 partitions and 5M tuples. The graph partitioning leads to higher exception rates but lower imbalance factor compared to the iterative approach. However, all partitionings with such imbalance factors are not usable. As the "vendor" columns contain more unique values and consequently more vertices in the graph, they are suitable to find a partitioning with more partitions. As a side note, the graph partitioning algorithm runs slower on the "vendor" columns for 4 partitions than for 96 partitions, as communication overhead for 4 partitions dominate the benefit of parallelism here.

- **Apriori property:** From the results we can also suggest that the choice of partition key columns follow an apriori property. A partition key of k columns can only lead to a good partitioning if all $k-1$ subkeys provide a good partitioning. This is shown by the "ABC" columns in the iterative approach. While we get a good partitioning on $(Ag\_name, Bureau\_name, Co\_name)$,

exchanging *Co_name* with *Co_state* results in a bad partitioning, as none of the subkeys containing *Co_state* is a good partitioning. Graph partitioning on three columns leads to high exception rates for the "ABC" columns due to the conflict resolution when mapping $G_{t\_as\_e}$ back to the table. Due to the apriori property the example given in Table 4.4 is limited to three partition key columns, as there are no more columns that correlate with the "ABC" columns in the dataset. Please note that this is a property of the dataset, not of our approach.

From a runtime perspective, the graph partitioning runtime is mainly dependent on the complexity of the graph, i.e., the number of vertices, as it propagates partition assignments over paths in the graph. Consequently, runtime on the "ABC" columns is significantly lower than on the "vendor" columns. The runtime thereby splits into several parts: The mapping from table to graph takes around 5-12s depending on the table size, running DistributedNE takes between 2-950s depending on the graph size, mapping the graph back to the table takes 4-9s and repartitioning the table takes 40-70s, so for small graph sizes repartitioning is the dominating part. On the other hand, iterative partitioning runtime is mainly impacted by the number of tuples as it iteratively processes the tuples. It is therefore constant for the same table size independent of the chosen partition keys. Runtime splits into iterating over the flat file and loading the resulting flat file into the database, which took around 20s for 1M tuples or 100s for 5M tuples. The iterative approach is favorable over the graph partitioning in the currently presented version, both in terms of runtime and partitioning quality.

This experiment also reveals a weakness of the current approach. Starting from a good partitionining (*Ag_name*, *Bureau_name*), adding *Co_state* destroys the whole partitioning instead of just excluding the values from the uncorrelated *Co_state* column. This is caused by the fact that the underlying partitioning algorithm is not aware that the graph is multipartite, i.e., it has subsets of vertices (one subset per column) and all edges are between subsets, but never within a subset. For our application, it would be better to simply cut a whole subset instead of destroying the whole partitioning. In the example, it would be better to have 100% exceptions in *Co_state* while keeping the good partitioning in (*Ag_name*, *Bureau_name*). With this, queries that require a partitioning on the column cutted by the algorithm would run similar than before using repartitioning, but queries on the remaining columns would be accelerated.

Experiment 2: Query performance

Out of all queries of the CommonGovernment workbook we chose two groups. Queries 9, 29 and 41 contain aggregations on *Vend_dunsnumber* and *Vend_vendorname*, while queries 1, 11 and 12 contain aggregations on *Ag_name*, *Bureau_name* and *Co_name*. We loaded the full dataset with hash partitionings on each of these columns as baselines and compared them against patched multi-key partitionings using the iterative partitioner on *(Vend_dunsnumber, Vend_vendorname)* named as *patched_vendor* and *(ag_name, bureau_name, co_name)* named as *patched_ag*, which were the column combinations with useful partitionings from the first experiment.

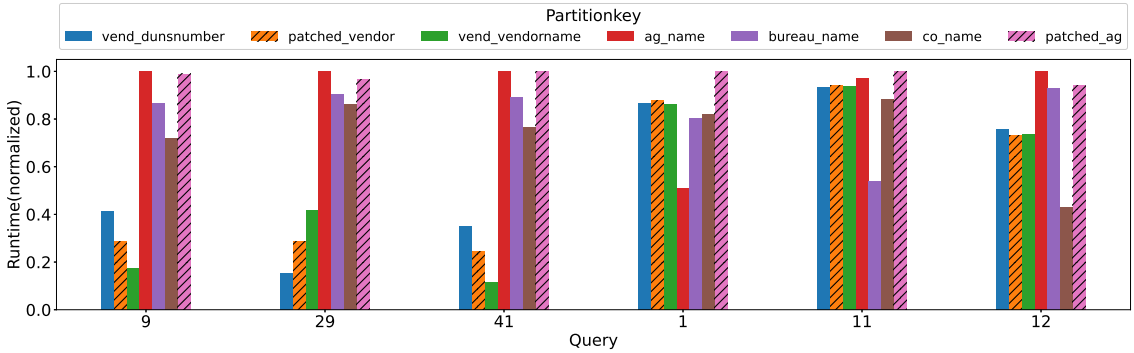


Figure 4.21: Query runtimes on different table partitionings

All tables have 96 partitions. The results are shown in Figure 4.21. We can first observe that the baselines show different runtimes, although they are expected to perform similarly. Partitionings on the vendor columns are faster than partitionings on *Ag_name*, *Bureau_name* or *Co_name*, caused by the small amount of unique values of the latter columns leading to imbalanced partitionings.

Overall the partitioning that matches the grouping column of the respective query shows the fastest runtime as expected. For the first set of queries (9, 29, 41), the *patched_vendor* partitioning shows performance slightly worse than the best partitioning, but better than the respective other vendor partition key. This is exactly the behaviour we want to achieve. We achieve a consistent performance with a patched multi-key partitioning for this set of queries.

For the second set of queries (1, 11, 12) we can observe a different behaviour. Here, the patched multi-key partitioning is not able to achieve reasonable good performance cuased by the small number of unique values of the partitioning columns. The aggregations are performed as a two-phase aggregation and the overhead of adding a PatchIndex scan is too large to amortize it's benefits. Additionally, partitions are even more imbalanced than in the respective baselines and the exception rates are very high as shown in the first experiment. Consequently, using a patched multi-key partitioning on partition keys with small number of distinct values (leading to bad imbalance factors and high exception rates) does not show the robust performance we are aiming at and should not be used.

Besides that, robustness also means reasonable performance in cases where partition keys do not match grouping keys. We can observe that the *patched_vendor* partitioning behaves similar than the other vendor partitionings in the second set of queries (1, 11, 12), while the *patched_ag* partitioning performs similarly than the partitionings on the "ABC" columns on the first set of queries (9, 29, 41). In conclusion we can state that patched multi-key partitioning shows robust performance for columns with a quite large amount of unique values. It avoids bad query performance caused by full table repartitionings when matching the grouping keys while performing similar to the baselines when not matching the grouping key. Please note that the case of replication is also covered in the experiment, i.e., one could achieve best performance in all queries when holding replicas for all partition keys leading to 5x storage and update cost.

Table 4.5: TPC-DS partition key column combinations and exception rates

Table	Partition Key Columns	Exception Rates
Item	(i_item_sk, i_item_desc, i_formulation, i_product_name)	(0%, 0.2%, 0%, 0.01%)
Item	(i_item_sk, i_item_desc, i_formulation, i_product_name, i_manufact_id, i_manufact)	(4.7%, 5.1%, 5.1%, 0%, 83%, 67%)
Store_returns	(sr_customer_sk, sr_cdemo_sk, sr_ticket_number)	(3.8%, 0.07%, 3.7%)
Catalog_returns	(cr_refunded_customer_sk, cr_refunded_cdemo_sk, cr_returning_cdemo_sk, cr_order_number)	(3.7%, 1.7%, 2.1%, 0.6%)
Catalog_returns	(cr_refunded_addr_sk, cr_returning_cdemo_sk, cr_order_number)	(5.2%, 2.4%, 1.1%)
Web_returns	(wr_item_sk, wr_refunded_cdemo_sk, wr_returning_cdemo_sk)	(5.1%, 0.8%, 0.7%)
Web_returns	(wr_refunded_customer_sk, wr_refunded_cdemo_sk, wr_returning_customer_sk, wr_returning_cdemo_sk)	(0.4%, 0.4%, 0.5%, 0.4%)
Web_returns	(wr_refunded_cdemo_sk, wr_refunded_hdemo_sk, wr_returning_cdemo_sk, wr_returning_hdemo_sk)	(1.1%, 6.2%, 1.1%, 6.2%)
Web_returns	(wr_refunded_cdemo_sk, wr_refunded_addr_sk, wr_returning_cdemo_sk, wr_returning_addr_sk)	(0.7%, 1.0%, 0.7%, 1.0%)
Web_sales	(ws_bill_customer_sk, ws_bill_cdemo_sk, ws_bill_addr_sk, ws_ship_customer_sk, ws_ship_cdemo_sk, , ws_order_number)	(13.7%, 0.7%, 22.8%, 13.2%, 0.8%, 0%)
Web_sales	(ws_bill_customer_sk, ws_bill_cdemo_sk, ws_ship_customer_sk, ws_ship_cdemo_sk, ws_order_number)	(7.3%, 0.4%, 7.3%, 0.4%, 0%)
Catalog_sales	(cs_bill_customer_sk, cs_bill_cdemo_sk, cs_ship_customer_sk, cs_ship_cdemo_sk, cs_order_number)	(8.1%, 1.7%, 8.1%, 1.6%, 0.1%)
Catalog_sales	(cs_bill_cdemo_sk, cs_bill_addr_sk, cs_ship_cdemo_sk, cs_ship_addr_sk, cs_order_number)	(1.8%, 17.3%, 17.3%, 17.3%, 0.1%)
Store_sales	(ss_customer_sk, ss_cdemo_sk, ss_ticket_number)	(10.2%, 3.8%, 0.5%)
Store_sales	(ss_cdemo_sk, ss_addr_sk, ss_ticket_number)	(3.4%, 18.8%, 1.2%)

Experiment 3: TPC-DS

In order to show the robustness of our approach over different datasets, we investigated the well-known TPC-DS[126] benchmark dataset. Although being synthetic and therefore rarely contatining "natural" clusters we found different useful partition key combinations in the dataset on scale factor 1. We used the column correlation requirement and the apriori property to find the combinations: In the fact tables we calculated the column correlations of all interesting columns (i.e., foreign keys or nominal values) and searched for combinations of columns that are all pairwise correlated with a correlation of over 0.9 and fulfill the apriori property. The resulting column combinations are candidates and were partitioned using the iterative partitioning algorithm into 4 partitions. Table 4.5 shows several partitionings with their respective exception rates, all showing a partition balance factor below 1.1.

4.5 Conclusion

Datasets in the cloud typically contain sparse schema information, caused by data ingestion from numerous endpoints leading to conflicts or users not defining constraints. The latter follows the goal to speed up ingestion or to avoid aborts due to conflicts on constraints. In order to overcome this issue we introduce the concept of PatchIndex, a generic index structure that allows the definition of approximate constraints. By simply implementing an interface the PatchIndex can be applied to arbitrary constraints. PatchIndexes maintain exceptions to constraints and make them usable in query optimization and execution to avoid expensive operations on tuples matching a constraint. The index is based on a sharded bitmap data structure, which allows efficient updates on the materialized information. With this updatability the concept also allows columns that are expected to fulfill a certain constraint to become approximate over time.

We showed how PatchIndexes can be applied to sorting and uniqueness constraints as examples of traditional database constraints. In this cases we can avoid distinct aggregations for the uniqueness constraint and avoid sorts or replace HashJoins with MergeJoins in the case of the sorting constraint. The evaluation showed that distinct aggregations and sorts can significantly benefit from using PatchIndexes even for high exception rates, while exchanging HashJoins with MergeJoins is only beneficial for small exception rates. Compared to traditional materializations like materialized

views, SortKeys or JoinIndexes, PatchIndexes deliver similar query performance while also offering more efficient updates.

In a second example we showed how PatchIndexes can be used to achieve a multi-key data partitioning. Here we want to obtain partition locality for multiple partition keys, i.e., all equal values of all keys can be found in the same partition. With PatchIndexes it is possible to maintain exceptions to the partition locality. We show how the problem of finding such a partitioning can be mapped to a graph partitioning problem and provide an example workflow. The evaluation showed that patched multi-key partitioning can lead to robust query performance by avoiding full table repartitionings in queries with different partitioning requirements.

Chapter 5
Grizzly
- A DataFrame-to-SQL Transpiler

In this chapter we present the Grizzly framework, which provides a DataFrame API similar to Python Pandas, but instead of shipping the data to the program, the program is shipped to where the data resides in order to exploit the capabilities of the database engine for efficient query processing. We argue that for many scenarios data is already stored in a (relational) database and used for different applications. Therefore, analysts familiar with Python or the DataFrame API should neither be bothered with learning SQL to access this data nor with implementing buffer management strategies to be able to process this data with Pandas in Python.

By transpiling DataFrame operations to SQL we push the execution of modern analytics towards the database engine, which was one of our major goals described in Chapter 1. The Grizzly framework is a frontend solution to achieve this goal, i.e., it acts as a connector to the database system and uses the well defined SQL API without requiring changes in the database engine. Due to this portability, it can be easily deployed in a managed cloud environment with any SQL-compliant database system. However, we also extend the Grizzly framework with API extensions for more advanced analytics. While these extenstions are similarly transpiled to SQL, they are also used to discuss opportunities for native support of modern analytics that we show in Chapter 6. The main contributions of this chapter are as follows:

- We present a framework that provides a DataFrame API similar to Pandas and transpiles the operations into SQL queries, moving program complexity to the optimized environment of a DBMS.
- We extend the API with the feature of processing external files directly in the database by automatically generating code to use DBMS specific external data source providers. This especially enables the user to join files with the existing data in the database directly in the DBMS.
- As a different extension, user-defined functions (UDFs) are also shipped to the DBMS by exploiting the support of the Python language for stored procedures of different database systems.
- Adding on the Python UDF support, Grizzly can automatically generate UDF code to apply pre-trained ML models (the ModelJoin) to the data inside the database. This way, use cases like classification or text analysis on the data inside the database can be pushed towards the engine and executed in a scalable way.

The remainder of the chapter is organized as follows: We discuss related work and compare existing systems with a Pandas-like DataFrame API in Section 5.1.

	Modin	**Koalas**	**IBIS**	**AFrame**	**Grizzly**
Approach	Python optimization	Transpiling	Transpiling	Transpiling	Transpiling
Backends	Ray, Dask	Spark	Arbitrary DBMS with SQL-API	AsterixDB	Arbitrary DBMS with SQL-API
Query Evaluation	Eager	Lazy	Lazy	Lazy	Lazy
UDF Support	On local Pandas DataFrames	Over Pandas UDFs	In-DBMS execution for Impala and BigQuery	In-DBMS execution for AsterixDB	In-DBMS execution for Postgres and Vector
Ext. File Support	Read to DataFrame	Read to DataFrame	Read to DataFrame	Read to DataFrame	In-DBMS support using ext. tables/ foreign data wrappers
ML Model Support	Handcrafted UDFs	Handcrafted UDFs	Handcrafted UDFs	API for Scikit models	API for ONNX, Tensorflow, PyTorch

Table 5.1: Comparison of available systems with Pandas-like API.

In Section 5.2 we present the architecture of our Grizzly framework as well as the transpilation of DataFrame operations to SQL code. Afterwards the important features of Grizzly are explained in detail, namely the external data source support in Section 5.3.1, the UDF support in Section 5.3.2 and the ModelJoin feature in Section 5.3.3. We evaluate the performance impact and scalability of Grizzly in Section 5.4 before concluding and motivating opportunities for native support of modern analytics in Section 5.5.

5.1 Related Work

There have been several systems proposed to translate user programs into SQL. The RIOT project [165] proposed the RIOT-DB to execute R programs I/O efficiently using a relational database. RIOT can be loaded into an R program as a package and provides new data types to be used, such as vectors, matrices, and arrays. Internally objects of these types are represented as views in the underlying relational database system. This way, operations on such objects are operations on views which the database system eventually optimizes and executes. Another project to perform Python operations as in-database analytics is AIDA [40], but focuses mainly on linear algebra with NumPy as an extension besides relational algebra. The AIDA client API connects to the AIDA server process running in the embedded Python interpreter inside the DBMS (MonetDB) to send the program and retrieve the results. AIDA uses its TabularData abstraction for data representation which also serves to encapsulate the Remote Method Invocation of the client-server communication.

Several projects have been proposed to overcome the scalability and performance issues in the Pandas framework. These projects can be categorized by their basic approaches of optimizing the Python execution or transpiling the Pandas programs into other languages. Modin [125] is the state-of-the-art system for the Python optimization approach. By offering the same API as Pandas, it can be used as a drop-in-replacement. In order to accelerate the Python execution, it transparently partitions the DataFrames and compiles queries to be executed on Ray [112] or Dask [42], two execution engines for parallel and distributed execution of Python programs. Additionally, Modin supports memory-spillover, so (intermediate) DataFrames may exceed main memory limits and are spilled to persistent memory. This solves the memory limitation problem of Pandas. However, Modin also uses eager execution like Pandas and still requires the client machine to consist of powerful hardware, since data from within a database system is fetched onto the client, too.

In the field of systems that use the transpiling approach, Koalas [89] brings the Pandas API to the distributed spark environment using PySpark. It uses lazy evaluation and relies on Pandas UDFs for transpiling. Koalas is in maintenance mode only nowadays as Databricks moved over to use PySpark as a DataFrame API. The creators of the Pandas framework also tackle the problem of the eager client side execution in [74]. IBIS collects operations and converts them into a (sequence of) SQL queries. Additionally, IBIS can connect to several (remote) sources and is able to run UDFs for Impala or Google BigQuery as a backend. Though, tables from two different sources, such as different databases, cannot be joined within an IBIS program. With a slightly modified API, AFrame [141] transpiles Pandas code into SQL++ queries to be executed in AsterixDB. Independent from the DataFrame API, [48] shows how to compile arbitrary Python code to SQL. In contrast, in Grizzly we produce standard SQL which can be executed by any SQL engine and use templates provided in a configuration file to account for vendor-specific dialects.

An approach to integrate Machine Learning into columnar database systems was proposed in [130]. The approach uses handcrafted Python UDFs to train the model inside the database, store the model as a DBMS-specific internal serialized object and apply the model by deserializing it again. In comparison, Grizzly supports pre-trained, portable model formats, automatically generates code to apply the models and also introduces a caching approach to cache the model, which reduces the loading (or deserialization) overhead and is therefore of major importance for deep and complex neural networks.

The main features of the presented systems are compared to Grizzly in Table 5.1. Grizzly also uses the approach of transpiling Python code to SQL, making it independent from the actual backend system and therefore being more generic than Koalas or AFrame. Similar to the proposed systems, Grizzly provides an API similar to the Pandas DataFrame API with the goal to abstract from the underlying execution engine. However, Grizzly extends this API with two main features that clearly separates it from the other systems. First, it provides in-DBMS support for external files. This enables server-side joins of different data sources, e.g., database tables and flat files. As a consequence, performance increases significantly for these use cases compared to the client-side join of the sources that would be necessary in the other systems. Additionally, the result of the server-side join remains in the database,

enabling subsequent operations to be also executed in the DBMS instead of the client machine. Second, Grizzly offers an easy-to-use API for applying Machine Learning models directly in the DBMS. Compared to the other systems where this feature could be simulated by handcrafting UDF code to apply the model on the client side, Grizzly exploits the UDF feature of DBMS and automatically generates code to cache the loaded model in memory and apply the pre-trained models directly on the server side. Furthermore, applying the model requires several (Python) functions, which can only be realized non-optimally using handcrafted UDFs. Again, as results remain in the DBMS, subsequent operations can be executed efficiently by the DBMS before returning the result to the client. Besides the performance aspect, this feature makes it significantly easier to apply Machine Learning models to the data compared to handcrafted UDFs.

5.2 Architecture

There are two major paradigms of data processing: data shipping and query shipping [91]. While in the data shipping paradigm, as found in Pandas, the possibly large amount of data is transferred from the storage node to the processing node, in query shipping the query/program is transferred to where the data resides. The latter is found in DBMSs, but also in Big Data frameworks such as Apache Spark and Hadoop. In this section we discuss the architecture of our Grizzly framework which is designed to maintain the ease-of-use of the data shipping paradigm in combination with the scalability of the query shipping approach. In its core, Grizzly consists of a DataFrame implementation and a Python-to-SQL transpiler. It is intended to solve the scalability issues of Pandas by transforming a sequence of operations on DataFrames into a SQL query that is executed by a DBMS. However, we would like to emphasize that the code generation and execution is realized using a plug-in design so that code generator for other languages than SQL or execution engines other than relational DBMSs (e.g., Spark or NoSQL systems) can be implemented and used. In the following, we show how SQL code generation is realized using a mapping between DataFrames and relational algebra.

Figure 5.1 shows the general (internal) workflow of Grizzly. As in Pandas, the core data structure is a DataFrame that encapsulates operations to compute result data. However, in Grizzly a DataFrame is only a hull and it does not contain the actual data. Rather, the operations on a DataFrame only create specific instances of DataFrames, such as `ProjectionDataFrame` or `FilterDataFrame`, to track the operations. A DataFrame instance stores all necessary information required for its operation as well as the reference to the DataFrame instance(s) from which it was created. This lineage graph basically represents the operator tree as found in relational algebra. The leaves of this operator tree are the DataFrames that represent a table (or view) or some external file. Inner nodes represent transformation operations, such as projections, filters, groupings, or joins, and hence, their results are DataFrames again. The actual computation of the query result is triggered via actions, whose results are directly needed in the client program, e.g., aggregation functions which are not called in the context of `group by` clause. Special actions such as `print` or `show` are available to manually trigger the computation.

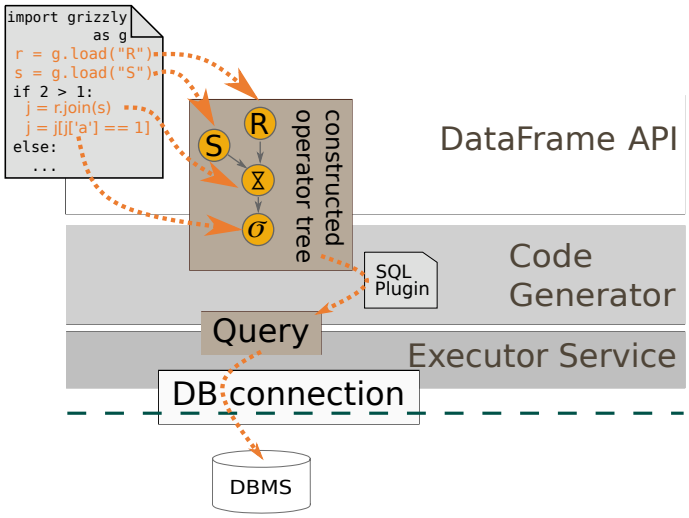


Figure 5.1: Overview of Grizzly's architecture.

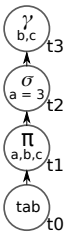
	Python Pandas	SQL
Projection	`df['A']`	`SELECT a FROM ...`
	`df[['A','B']]`	`SELECT a,b FROM ...`
Selection	`df[df['A'] == x]`	`SELECT * FROM ...WHERE a = x`
Join	`pandas.merge(df1, df2,`	`SELECT * FROM df1`
	`left_on='x', right_on='y',`	`inner|outer|right|left join df`
	`how='inner|outer|right|left')`	`ON df1.x = df2.y`
Grouping	`df.groupby(['A','B'])`	`SELECT * FROM ...GROUP BY a,b`
Sorting	`df.sort_values(by=['A','B'])`	`SELECT * FROM ...ORDER BY a,b`
Union	`df1.append(df2)`	`SELECT * FROM df1`
		`UNION ALL SELECT * FROM df2`
Aggregation	`df['A'].min()`	`SELECT min(a) FROM ...`
	`max()|mean()|count()|sum()`	`max(a)|avg(a)|count(a)|sum(a)`
	`df['A'].value_counts()`	`SELECT a, count(a) FROM ...`
		`GROUP BY a`
Add column	`df['new'] = df['a'] + df['b']`	`SELECT a + b AS new FROM ...`

Table 5.2: Basic Pandas DataFrame operations and their corresponding SQL statements.

Building the lineage graph of DataFrame modifications, i.e., the operator tree, follows the design goal of lazy evaluation behavior as it is also found in the RDDs in Apache Spark [163]. When an action is encountered in a program, the operator tree is traversed, starting from the DataFrame on which the action was called. While traversing the tree, for every encountered operation its corresponding SQL expression is constructed as a string and filled in a SQL template. For this, we apply a mapping of Pandas operations to SQL statements. This mapping is shown in Table 5.2. Based on the operator tree, the SQL query can be constructed in two ways:

1. generate nested sub-queries for every operation on a DataFrame, or
2. incrementally extend a single query for every operation found in the Python program.

```python
# load table (t0)
df = grizzly.read_table("tab")
# projection to a,b,c (t1)
df = df[['a','b','c']]
# selection (t2)
df = df[df.a == 3]
# group by b,c (t3)
df = df.groupby(['b','c'])
```




```sql
SELECT t3.b, t3.c FROM (
  SELECT * FROM (
    SELECT t1.a, t1.b, t1.c FROM (
      SELECT * FROM tab t0
    ) t1
  ) t2 WHERE t2.a = 3
) t3 GROUP BY t3.b, t3.c
```

(a) Source Python code. (b) Operator tree. (c) Produced SQL query.

Figure 5.2: Steps for transpiling Python code to a SQL query: The operations on DataFrames (a) are collected in an intermediate operator tree (b) which is traversed to produce a nested SQL query (c).

In Grizzly we implement variant (1), because variant (2) has the drawback to decide whether the SQL expression of an operation can be merged into the current query or a sub-query has to be created. Though, the SQL parser and optimizer in the DBMSs have been implemented and optimized to recognize such cases. As an example, Figure 5.2 shows how a Python script is transformed into a SQL query. Although the nested query imposes some overhead to the optimizer for unnesting in the DBMS and bears the risk that it fails to produce an optimal query, we believe they are very powerful and mostly well tested, so that it is not worth it to re-implement such behavior in Grizzly. The generated query is sent to a DBMS using a user-defined connection object, as it is typically used in Python and specified by PEP 249[124]. Grizzly produces standard SQL with vendor-specific statements to create functions or access external data as we will discuss below. The vendor-specific statements are taken from templates defined in a configuration file. By providing templates for the respective functions one can easily add support for arbitrary DBMSs. We currently support Actian Vector and PostgreSQL.

Besides the plain SQL queries, Grizzly needs to produce additional statements in order to set up user-defined functions and connectors to external data sources. If the Python code uses, e.g., UDFs, this function must be created in the database system before it can be used in the query. Thus, Grizzly produces a list of so-called pre-queries. A pre-query to create a UDF is the `CREATE FUNCTION` statement including the corresponding function name, input and output parameters as well as the function body of course. For an external data source, the pre-query creates the necessary DBMS-specific connection to the data source, as described in the next section.

One design goal of the Grizzly framework is to serve as a drop-in replacement for Pandas. In order to achieve this goal, Grizzly offers a transparent fallback mode to Pandas. By hooking into the `__getattr__` method of Grizzly's DataFrame class we can automatically detect if a given operation is not supported yet. In this case code generation is triggered, the generated query is sent to the DBMS and the result is transformed into a Pandas DataFrame. Consequently, the operation can be transparently called on the resulting DataFrame. However, once execution was triggered and the result was materialized there is currently no way of pushing

execution again to the DBMS afterwards. In the worst case (i.e., the first operation after scan is already not supported) a Grizzly pipeline therefore behaves exactly similar to a Pandas pipeline. Besides that, the fallback mode enables stepwise extensibility of the framework, i.e., support for operations can be added over time. Due to the behaviour of `__getattr__` of only checking the existence but not the list of supported parameters of a function, an added operation requires to be fully API-compatible with the respective Pandas version before being released.

5.3 API-Extensions for Modern Analytics

Every operation that is not supported in Grizzly triggers the materialization of the intermediate result and leads to further processing in the Python environment of the client machine. With the goal of pushing as much computational effort as possible to the DBMS, we investigated typical use cases in modern analytical pipelines and added pushdown support for them in Grizzly. In this section, we discuss how we integrated support for external file joins, user-defined functions and ML model inference.

5.3.1 Support for External Data Sources

In typical data analytics tasks data may be read from various formats. On the one hand, (relational) database systems are used to store and archive large data sets like company inventory data or sensor data in IoT applications. On the other hand, data may be created by hand, exported from operational systems or shared as text files like CSV or JSON. For these files it is not always necessary, intended or beneficial to import them into a database system first, as they might be only for temporary usage or need to be analyzed before loading them into the database. As a consequence, there is a gap between tables stored in a database system and plain files in the filesystem, and both sources need to be combined. In Pandas, one would need to read the data from the database as well as the text files and combine them in main memory. Since it is our goal to shift the complete processing into the DBMS, the files need to be transferred and imported into the DBMS transparently. In our framework, we achieve this by using the ability of many modern DBMS to define a table over an external file as defined in the SQL/MED standard from 2003 [108].

As an example, PostgreSQL offers *foreign data wrappers* (FDW) to access external sources such as files, but also other database systems. A PostgreSQL distribution includes FDWs for, e.g., CSV files, files in HDFS as well as other relational and non-relational DBMSs. Own FDWs for other sources can easily be installed as extensions. Internally, the planner uses the access costs to decide for the best access path, if such information is provided by the FDW.

Besides PostgreSQL, all Actian products built on X100 offer the external table feature, which is realized using the Spark-Vector-Connector[143]. Here X100 handles external tables as meta data in the catalog with a reference to a file path in the local filesystem or HDFS. Whenever a query accesses an external table, Vector exploits the capabilities of Apache Spark to read data efficiently in parallel, leading to fast scans of external tables.

In Grizzly, we offer easy-to-use operations to access external data sources. These operations return a DataFrame object and are the leaves of the operator lineage graph described in Section 5.2, similar to ordinary database tables. Here a user has to specify the data types of the data in the text files as well as the file path. During SQL code generation, a pre-query is automatically generated that creates the external table/foreign data wrapper for each of these leaves. As the syntax of these queries might be vendor-specific, we maintain templates to create an external data source in the configuration file. The pre-queries are then appended to the pre-query list described in Section 5.2, so they are ensured to be executed before the actual analytical query is run. In the actual query, these tables are then referenced using their temporary names.

An important point to highlight here is that the database server must be able to access the referenced file. We argue that with network file systems mounts, NAS devices or cloud file systems this is often the case. Even actively copying the file to the server is not a problem since such data files are rather small, compared to the amount of data stored in the database.

5.3.2 Support for Python UDFs

Another important part of data analytics is data manipulation using user-defined functions. In Pandas, users can create custom functions and apply them to DataFrames using the `map` function. Such functions typically perform more or less complex computations to transform values or combine values of different columns. These UDFs are a challenge when transpiling Pandas code to SQL, as their definitions must be read and transferred to the DBMS. This requires that the Python program containing the Pandas operations can somehow access the function's source code definition. In Python, this can be done via reflection tools like the inspect module. Most DBMS support stored procedures and some of them, e.g., PostgreSQL and X100, also allow to define them using Python (language PL/Python). This way, functions defined in Python are processed by Grizzly, transferred to the DBMS and dynamically created as a (temporary) function. Note that most systems only offer scalar UDFs at the moment, which produce a single output tuple from a single input tuple. Consequently, Grizzly only supports this class of functions and does not offer any support for table UDFs, which produce an output tuple for an arbitrary number of inputs.

The actual realization of the UDF support is hereby different for different DBMS and shows some limitations that needs to be considered. First, Python UDFs are only available as a beta version in PostgreSQL and X100. The main reason for this is that there are severe security concerns about using the feature, as especially sandboxing a Python process is difficult. As a consequence, users must have superuser access rights for the database or demand access to the feature from the administrator in order to use the Python UDF feature. While this might be a problem in production systems, we argue that this should not be an issue in the scientific use cases where Python Pandas is usually used for data analytics. Second, the actual architecture of running Python code in the database differs in the systems. While some systems start Python processes per-query, other systems keep processes alive over the system uptime. The

per-query approach has the advantage that it offers isolation in the Python code between queries, which is important for ACID-compliance. As a drawback, the isolation makes it impossible to cache user-specific data structures in order to use it in several queries, which needs to be considered when designing the ModelJoin feature in Section 5.3.3. On the contrary, keeping the Python processes alive allows to cache such a user context and use it in several queries. However, this approach violates isolation, so UDF code has to be written carefully to avoid side effects that might impact other queries.

Although the DBMS supports Python as a language for user defined code, SQL is a strictly typed language whereas Python is not. In order to get type information from the user's Python function, we make use of type hints, introduced in Python 3.5. A Python function using type hints looks like this:

```
def repeat(n: int, s: str) -> str:
  r = n*s # repeat s n times
  return r
```

Such UDFs can be used, e.g., to transform, or in this example case combine, columns using the `map` method of a DataFrame:

```
# apply repeat on every tuple using columns name, num as input
df['repeated'] = df[['num','name']].map(repeat)
```

Using the type hints and a mapping between Python and SQL types, Grizzly's code generator can produce a pre-query to create the function on the server. For PostgreSQL, the generated code is the following:

```
CREATE OR REPLACE FUNCTION repeat(n int, s varchar(1024))
RETURNS varchar(1024)
LANGUAGE plpython3u
AS 'r = n*s # repeat s n times
return r'
```

Currently, we statically map variable-sized Python types to reasonable big SQL types, which is a real limitation and should be improved in the future. The command to create the function in the system is vendor-specific and therefore taken from the config file for the selected DBMS. We then extract the name, input parameters, source code and return type using Python's `inspect` module and use the values to fill the template. The function body is also copied into the template. Similar to external data sources in Section 5.3.1, the generated code is appended to the pre-query list and executed before the actual query. The `map` operation is translated into a SQL projection creating a computed column in the actual query:

```
SELECT t0.*, repeat(t0.num, t0.name) as repeated
FROM ... t0
```

As explained above, the previous operation from which `df` was derived will appear in the `FROM` clause of this query.

5.3.3 Machine Learning ModelJoin

In Grizzly, we expand the API of Python Pandas with the functionality to apply different types of Machine Learning models to the data. In the following, we name this operation of applying a model to the data a "ModelJoin". Instead of realizing this over a `map`-function in Pandas, which leads to a client-side execution of the ModelJoin and therefore faces the same scalability issues as Pandas, we exploit the recent upcome of user-defined functions in popular database management systems and realize the ModelJoin functionality using Python UDFs. As a consequence, we achieve a server-side execution of the ModelJoin directly in the database system, allowing automatic parallel and distributed computation.

Note that we talk about the usage of pre-trained models in this section, as database systems are not optimized for model training. However, applying the model directly in the database has the advantage that users can make use of the database functionality to efficiently perform further operations on the model outputs, e.g., grouping or filters. Additionally, users may use publicly available, pre-trained models for various use cases. For our discussions, we assume that necessary Python modules are installed (which should not be an issue in a managed cloud environment) and the model files are accessible from the server running the database system. In the following, we describe the main ideas behind the ModelJoin concept as well as details for the supported model types, their characteristics and their respective runtime environments, namely PyTorch, Tensorflow, and ONNX.

ModelJoin Concept

Performing a ModelJoin on an DataFrame triggers the generation of a pre-query as described in Section 5.2, which performs the creation of the respective database UDF. As the syntax for this operation is vendor-specific, the template is also taken from the configuration file. The generated code hereby has four major tasks:

1. Load the provided model.
2. Convert incoming tuples to the model input format.
3. Run the model.
4. Convert the model output back to an expected output format.

While steps 2-4 have to be performed for every incoming tuple, the key for an efficient ModelJoin realization is caching the loaded model in order to perform the expensive loading only if necessary. (Re-)Loading the model is necessary if it is not cached yet or if the model changed. These cases can be detected by maintaining the name and the timestamp of the model file. However, such a caching mechanism must be designed carefully under consideration of the different, vendor-specific Python UDF realizations discussed in Section 5.3.2.

We realize the caching mechanism by attaching the loaded model, the model file name and the model time stamp to a globally available object, e.g., an imported module in the UDF. The model is loaded only if the global object has no model attribute for the provided model file yet or the model has changed, which is detected

```python
def input_to_model(a: str):
        ...

def model_to_output(a) -> str:
        ...

df = grizzly.read_table('tab') # load table
# apply model to every value in column 'col'
# using provided input and output conversion functions
# store model output in computed column 'classification'
df['classification'] = df['col'].apply_model("/path/to/model", input_to_model,
↪  model_to_output)
# group by e.g., predicted classes
df = df.groupby(['classification']).count()
df.show()
```

Listing 1: Python code of ModelJoin example

by comparing the cached timestamp with the filesystem timestamp. In order to avoid that accessing the filesystem to get the file modification timestamp is performed for each call of the UDF (and therefore for every tuple), we introduce a magic number into the UDF. The magic number is randomly generated for each query by Grizzly and cached in the same way as the model metadata. In the UDF code, the cached magic number is compared to the magic number passed and only if they differ, the modification timestamps are compared and the cached magic number is overwritten by the passed one. As a result, the timestamps are only compared once during a query, reducing the number of file system accesses to one instead of once-per-tuple. With this mechanism, we automatically support both Python UDF realizations discussed in Section 5.3.2, although the magic number and timestamp comparisons are not necessary in the per-query approach, as it is impossible here that the model is cached for the first tuple. We exploit the isolation violation of the second approach that keeps the Python process alive and carefully design the ModelJoin code to only produce the caching of the model and respective metadata as intended side effects.

Model types

Grizzly offers support for PyTorch, Tensorflow and ONNX models. All three model formats have in common, that the user additionally needs to specify model-specific conversion functions for their usage in order to specify how the expected model input is produced and the model output should be interpreted. These functions are typically provided together with the model by creators. With A, B, C, D being lists of data types, the conversion functions have signatures $in\_conv : A \rightarrow B$ and $out\_conv : C \rightarrow D$, if the model converts inputs of type B into outputs of type C. With A and D being set as type hints, the overall UDF signature can be infered as $A \rightarrow D$ as described in Section 5.3.2. With this, applying a model to data stored in a database can be done easily and might look like the example in Listing 1.

As the conversion functions are typically provided along with the model, users only need to write a few lines of code. It is thereby mandatory to specify the

```
CREATE OR REPLACE FUNCTION apply_model_123(col varchar(1024))
 RETURNS varchar(1024)
 LANGUAGE plpython3u AS 'def input_to_model(a: str):
         ...

 def model_to_output(a) -> str:
         ...

  #apply model here
' parallel safe;

SELECT t2.classification, count(*) FROM (
  SELECT *, apply_model_123(t1.col) as classification FROM (
    SELECT * FROM tab t0
  ) t1
) t2 GROUP BY t2.classification
```

Listing 2: Generated SQL code of ModelJoin example

input parameter types of the `input_to_model` function as well as the output type of the `model_to_output` function. In this example, the resulting UDF would have signature `str -> str`. Running this example, Grizzly automatically generates the UDF code and triggers its creation in the DBMS before executing the actual query. The produced query along with the pre-query to setup the UDFs in PostgreSQL is shown in Listing 2. The actual code for model application is generated from templates and varies for the different model types described in the following.

PyTorch The PyTorch library is based on the Torch library, originally written in Lua. PyTorch was presented by Facebook in 2016 and has gained popularity as it enabled programs to utilize GPUs and integrate other famous Python libraries. In its core, PyTorch consists of various libraries for Machine Learning that have different functionality.

A trained model can be saved for later reuse. For saving, two options exist. The first option is to serialize the complete model to disk. This has the disadvantage that the model class definition must be available for the runtime when the model is loaded. In Grizzly, this would mean that users who want to use a pretrained model in their program also need the source code of the model class. The second option for storing a trained model is to store only the learned parameters. Although this option is more efficient during deserialization, the user code must explicitly create an instance of the model class. Thus, the source code of the model class must be available for the end user. Additionally, in order to instantiate the model class, users need to provide initial parameters values to the model's constructor which may be unknown and hard to set for inexperienced users. Besides the challenges for using PyTorch with foreign models, Grizzly allows to load PyTorch models where only the learned parameters have been stored (option 2 from above).

Tensorflow Tensorflow is a another famous framework for building, training and running Machine Learning models. Tensorflow models are directed, acyclic graphs (DAGs) that are statically defined. With placeholder variables attached to nodes of the model graph, inputs and outputs can be mapped to the graph nodes. A `tensorflow.Session` object is used as the main entrance point and allows to run the model after configuring.

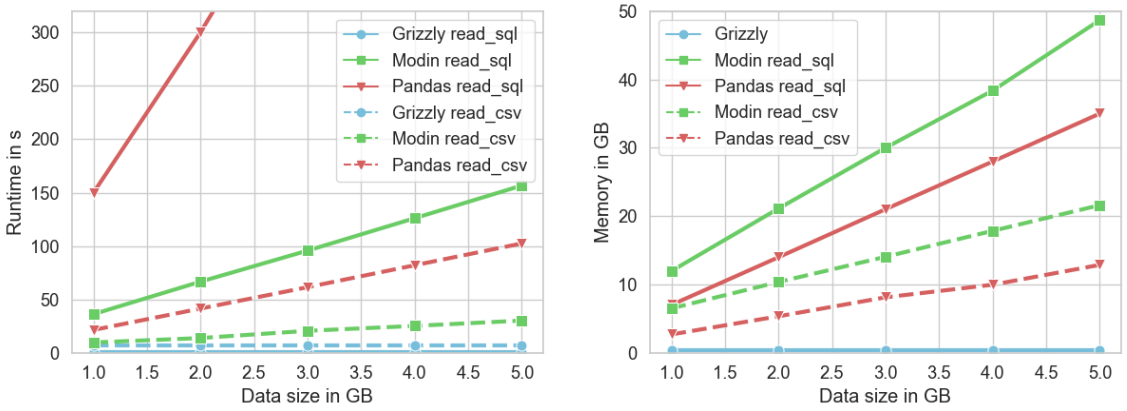
During the training phase, arbitrary states of the model graph can be exported as a `checkpoint`, which is a serialized format of the graph and its properties. Grizzly supports these `checkpoints` as an model type and generates code to restore the model graph as well as the `tensorflow.Session` object. However, users have to know the names of placeholders defined in the model in order to map inputs and outputs to the respective model nodes. Additionally, users can specify a vocabulary file to translate inputs to an expected model input format if necessary. As these restrictions require in-depth knowledge about the model, Grizzly offers additional possibilities to automatically generate the conversion functions. Nevertheless, this harms the ease-of-use slightly. Starting with version 2, Tensorflow offers different possibilities to exchange trained models with the introduction of the Tensorflow Hub library or support for the Keras framework. In the future, we aim at integrating support for these formats in Grizzly.

ONNX ONNX is a portable and self-contained format for model exchange, that is able to be executed with different runtime backends like Tensorflow, PyTorch or the onnxruntime. The self-containment and the portability of models makes the ONNX format easy to use, which meets the design goals of Grizzly. In the generated model code, we rely on the onnxruntime as execution backend in order to be independent from Tensorflow or PyTorch. A broad collection of pre-trained models along with their conversion functions is available in the Model Zoo[118].

5.4 Evaluation

In this Section, we compare our proposed Grizzly framework against Pandas and Modin, the current state-of-the-art framework for distributed processing of Pandas scripts, as the other related systems presented in Section 5.1 do not offer all evaluated features. We present different experiments for data access as well as applying a machine learning model in a ModelJoin. Our experiments were run on a server consisting of a Intel(R) Xeon(R) CPU E5-2630 with 24 threads at 2.30 GHz and 128 GB RAM. This server runs Actian Vector 6.0 in a docker container, Python 3.6 and Pandas 1.1.1. Additionally we used Modin version 0.8 and experimentally configured it to the best of our knowledge, resulting in using Ray as the backend, 12 cores and out-of-core execution. For fairness, we ran the Pandas/Modin experiments on the same machine. As this reduces the transfer costs when reading tables from the database server, this assumption is always in favor of Pandas and Modin.

During our experiments, we discovered a bug in the parallel `read_sql` design of Modin, which produces wrong results for partitioned databases. The developers confirmed the issue. However, we used the parallel `read_sql` in order to not penalize



(a) Execution time

(b) Main memory consumption

Figure 5.3: CPU and RAM consumption for Pandas, Modin and our proposed Grizzly framework.

Modin in the experiments, not considering the wrong results. This assumption is therefore also in favor of the Modin results.

Data Access Scalability In this first experiment, we want to prove our initial consideration of Pandas' bad scalability with a minimal example use case. We used Actian Vector as the underlying database system and ran a query that scans data with varying size from a table or a `csv` file and performs a `min` operation on a column to reduce the result size. This way, we can compare the basic `read_sql` and `read_csv` operations of Pandas and Modin against Grizzly.

Figure 5.3 shows the execution time as well as the memory consumption of the evaluated query. For `sql table` access, Pandas shows an enormous runtime, linearly growing to 800 s for a data set size of 5 GB. Modin is significantly faster than Pandas and scales better. However, Grizzly is able to answer this query in a constant, sub-second runtime, as only the result has to be transfered to the client instead of the full dataset. Additionally, this query can be answered by querying small materialized aggregates [111], which are used by default as an additional index structure in X100. In comparison to Pandas/Modin, this shows that with Grizzly queries can also benefit from index structures and other techniques to accelerate query processing used in database systems. For `csv` access, we can observe a similar behavior of Modin being faster and scaling better than Pandas due to its parallel `read_csv` implementation. Grizzly again shows a nearly constant runtime slightly higher than the `sql table` access but faster than Modin and Pandas. The main reason for this is that Grizzly uses the external table feature of Actian Vector, which is based on Apache Spark. As a consequence, the runtime is composed of the fixed Spark delays like startup or cleanup [157] and a variable runtime for reading the file in parallel. As data sized are small here, the fixed Spark delays dominate the runtime.

Regarding memory consumption, Pandas again scales very poorly and memory consumption increases very fast, with the `read_sql_table` consuming more memory than the `read_csv` operation for the same data size. As a result, the memory

consumption might exceed the available RAM of a client machine even for a small dataset size. In comparison, Modin consumes even more memory than Pandas for `read_sql` and `read_csv` respectively, potentially caused by the multiple worker threads. The memory consumption of Grizzly is mainly impacted by the result size, which is very small due to the choice of the query. However, this is only half of the truth, as Grizzly shifts the actual work to the DBMS which also consumes memory. Nevertheless, modern DBMS are designed for high scalability and are able to handle out-of-memory cases with buffer eviction strategies in the bufferpool or disk-spilling operators for database operators with a high memory consumption. Therefore, this is the ideal environment to run complex queries, as it is not limited to the available memory of the machine. Note that Modin also supports out-of-memory situations by disk-spilling. Another advantage of Grizzly is that the DBMS can run on a remote machine while the actual Grizzly script is executed from a client machine, which is then allowed to have an arbitrary hardware configuration while still being able to run complex analytics.

Combining Data Sources An important task of data analysis is combining data from different data sources. We investigated a typical use case, namely joining flat files with existing database tables. We base our example on the popular TPC-H benchmark dataset [27] on scale factor SF100, which is a typical sales database and is able to generate inventory data as well as update sets. We draw the following use case: The daily orders (generated TPC-H update set) are extracted from the productive system and provided as a flat file. Before loading them into the database system, a user might want to analyze the data directly by combining it with the inventory data inside the database. As an example query, we join the daily orders as a flat file with the customer table (1.5M tuples) from the database and determine the number of orders per customer market segment using an aggregation. The evaluated Python scripts are similar except the data access methods. While Pandas and Modin use (parallel) `read_sql` and `read_csv` for table and flat file access, Grizzly uses a `read_table` and a `read_external_table` call. This way, an external table is generated in Actian Vector, encapsulating the flat file access. Afterwards, the join as well as the aggregation are processed in the DBMS, and only the result is returned to the client.

For the experiment, we varied the number of tuples in the flat files. The runtime results in Figure 5.4 show that Grizzly achieves a significantly better runtime than Pandas and Modin. Additionally, it shows that Pandas and Modin suffer from a bad `read_sql` performance, as the runtime is already quite slow for small number of external tuples. Regarding scalability we can observe that runtime in Pandas grows faster with increasing number of external tuples than in Grizzly, caused by the fast processing of external tables in Actian Vector. Overall we can conclude that Grizzly significantly outperforms Python Pandas and Modin by two orders of magnitude in this experiment and offers the possibility to process significantly larger datasets.

ModelJoin Performance There are various applications and use cases where machine learning models can be applied. As an example use case, we applied a sentiment analysis to a string column. We therefore used the IMDB dataset [104],

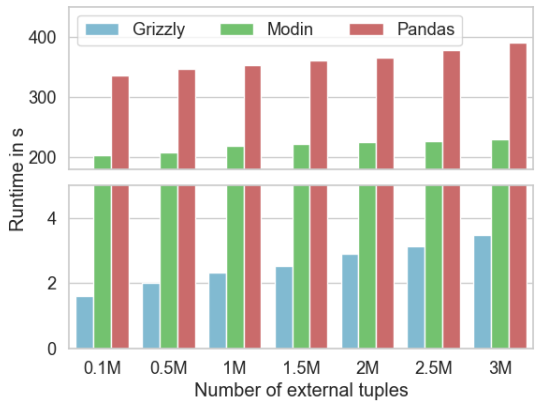


Figure 5.4: Query runtime with database tables and external sources

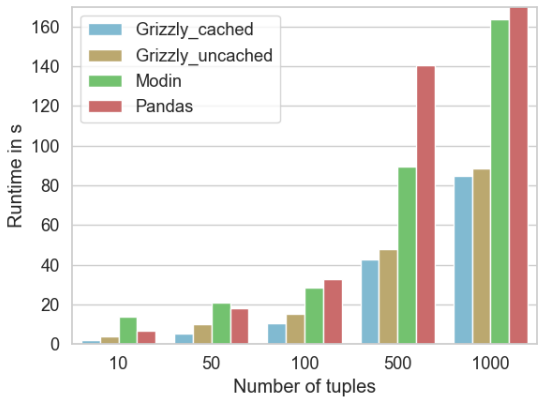


Figure 5.5: ModelJoin runtime.

which contains movie reviews, and applied the state-of-the-art RoBERTa model [102] with ONNX. The investigated query applies the model to the review column and groups on the output sentiment afterwards, counting positive and negative review sentiments. In Grizzly, we therefore use ModelJoin feature described in Section 5.3.3, while we handcrafted the function to apply the model for Modin and Pandas and invoked the function over the `map` function on the DataFrames after reading from the database.

Figure 5.5 shows the resulting runtimes for different number of review tuples. First, we can observe that Grizzly significantly outperforms Modin and Pandas in terms of runtime and scalability. For increasing data size Modin is also significantly faster and scales better than Pandas, while showing some overhead over Pandas for the very small data sets, as parallelism here introduces more overhead than benefit. While Modin achieves a speedup by splitting the DataFrames into partitions and applying the machine learning model in parallel, Grizzly achieves the performance improvement by applying the model directly in the database system. In Actian Vector, we used a parallel UDF feature (discussed in Section 6.2.4 and configured it for 12 parallel UDF workers, showing the best results for our setup in this experiment. Additionally, with the Python implementation of Actian Vector, which keeps the Python interpreters alive between queries, it is possible to reuse a cached model from a former query, leading to an additional performance gain of around 5 seconds.

Resume In our evaluation, we proved the superiority of our proposed Grizzly framework over Pandas and Modin, the existing state-of-the-art distributed DataFrame framework. In different experiments for data access, the combination of different data sources and the application of pre-trained machine learning models to a dataset we showed that Grizzly has significantly better query performance as well as scalability, as most operations are executed directly in the database system. This is reinforced by the fact that query performance benefits from index structures of the DBMS, while these indexes do not have any impact when only reading data in a Pandas script and performing operations on the client. Furthermore, Grizzly enables complex data analysis tasks even on client machines without powerful hardware by pushing operations towards database servers. Consequently, queries are not limited to client

memory, but are executed inside database systems that are designed to handle out-of-memory situations.

5.5 Conclusion

With the goal of pushing advanced analytics towards the database system, we presented Grizzly, a scalable and high-performant data analytics framework that offers a similar DataFrame API like the popular Pandas framework and can be used as an frontend alternative to SQL. As Pandas faces some severe scalability problems in terms of memory consumption and performance, Grizzly transpiles the easy-to-write Pandas code into SQL in order to push complexity to arbitrary database systems. This way, query execution is done in a scalable and highly optimized environment that is able to handle large datasets and complex queries. Additionally, by pushing queries towards remote database systems, complex analytics can be performed on client machines without extensive hardware requirements.

We extended the Pandas DataFrame API with several features in order to perform typical data analytics challenges in an efficient and easy-to-use way. First, Grizzly provides support for external data sources by exploiting the respective feature of several database systems and automatically generating code to create necessary tables or wrappers. This way, different sources like database tables or flat files can be combined and processed directly inside the DBMS instead of loading them into the client, improving performance and scalability. Second, Grizzly makes use of the recent upcome of Python user-defined functions in database systems in order to transparently push the execution of such functions on DataFrames into the DBMS. Third, applying pre-trained Machine Learning models to data is supported by Grizzly by automatically generating UDF code for Tensorflow, PyTorch or ONNX models. Pushing these operations to the database system not only increases performance and scalability, but also enables efficient processing of further operations on the model outputs, as they still remain in the database system. This allows a seamless integration of Machine Learning functionalities towards the vision of ML systems [133]. In our evaluation, we compared our proposed Grizzly framework to Pandas and Modin, the current state-of-the-art framework for distributed execution of Pandas-like DataFrames. In our experiments on data access scalability, combining different data sources, and applying Machine Learning models we proved that Grizzly significantly outperforms both systems while offering higher scalability and an easy-to-use API.

Grizzly can be seen as a frontend solution to push advanced analytics towards the database engine. While transpiling DataFrame operations to SQL is very portable, our mapping of user-defined functions or the ModelJoin to Python UDFs also limits performance. In the use cases from Section 5.4 the Python UDF operator took the major part of the runtime, often far beyond 90% of the total query runtime. As user-defined functions are an important feature for flexible analytics pipelines and ML inference is also an important part of many modern use cases, these two features are candidates for a more native integration into the database engine. We therefore focus on accelerating these workloads using native database engine support in the next chapter.

Chapter 6
Native Support for Modern Analytics beyond SQL

User-defined functions and ML model inference are important tasks in modern data analytics. As discussed in Section 5.5 these features would benefit from a deeper engine integration. In this chapter we discuss approaches for a native integration of Python UDFs and ML model inference. The remainder of this chapter is organzied as follows: We start with discussing related work in Section 6.1. Section 6.2 describes different integration levels of Python UDFs. We discuss compilation in Section 6.2.2, vectorization in Section 6.2.3 and parallelisation in Section 6.2.4 as orthogonal directions to optimize native UDF execution, before evaluating the approaches in Section 6.2.5. The second part in Section 6.3 covers an investigation on approaches for in-database ML model inference. We discuss neural network layer types and their applicability to typical relational data in Section 6.3.2. Section 6.3.3 presents a pure SQL-based solution to realize ML model inference, while we present a native operator in Section 6.3.4. We compare our approaches against existing approaches using pure Python, Python UDFs or an operator based on Tensorflow's C-API in the evaluation in Section 6.3.5. Finally, the chapter is concluded in Section 6.4

6.1 Related Work

Python and UDF compilation: Python transpilers like Cython [18] and Nuitka[115] work with arbitrary Python language constructs and especially also with external modules, but show slower performance compared to native code as they still rely on Python's C-API. Python compilers like Numba [93] translate Python code to native C code, which leads to efficient code execution but limits the usage to the supported language constructs. Numba supports a set of basic Python language constructs as well as Numpy code and offers an "object mode" as a fallback for working with, e.g., Strings, which also invokes the Python environment. However, importing external libraries is not allowed in Numba. Tuplex [144] focuses on compiling data analytics pipelines written in Python to a native binary. All these frameworks focus on compiling Python code to a binary and running it as a standalone process. In our work, we investigate how these frameworks can be integrated into an interpreted query engine during runtime, enabling the ad-hoc creation and usage of compiled Python UDFs. Besides these frameworks, GraalVM[55] supports compiling code into native images. However, it mainly focuses on Java code and only supports Python as a "guest language" embedded into Java code for compilation. The feature is in an experimental state and only supports a few Python modules. [48] shows how to compile Python code to SQL statements. Several approaches also compile UDFs that are not written in Python. [44] focuses on compiling interpreted PL/SQL UDFs into recursive SQL functions to avoid PL/SQL - SQL context switches, while [138] aims

at just-in-time compiling SQL lambda functions. The vectorized, column-oriented database system MonetDB also supports Python UDFs [129]. It uses the interpretation of Python code as the standard case, but optimizes for Numpy UDFs as Numpy and column-stores share a similar storage layout. As a result, the execution of Numpy UDFs reaches nearly native code speed. [129] also highlights the issue that compiling Python UDFs would require to compile and link the database engine against the compiled UDF, which is not possible during database runtime. In our approach we focus on arbitrary Python constructs and exploit dynamic loading to make compiled UDFs available to the query engine during runtime.

Python global interpreter lock: In Python there is the global interpreter lock (GIL), which was chosen in the design of Python in order to simplify low-level details like synchronisation or memory management of concurrent threads. While this leads to high single threaded performance, it limits the parallel or multithreaded execution of Python code [105]. Several solutions on the GIL issue in dynamic languages have been proposed like hardware transactional memory [116], software transactional memory [105] or the use of virtual machines [106]. In our approach, we achieve parallelism similary to virtual machines by spawning separate UDF engine processes in a parallel query engine. Working on independent data partitions, we do not need any type of synchronisation, which simplifies the GIL issue and makes separate Python processes a reasonable solution.

AI4DB vs. DB4AI: Data Science and Machine Learning have always been of interest in database research and different solutions have been proposed to support the required operations inside DBMSs. Especially technologies that are generally referred to as AI are more and more used in the database context, leading to two different fields: AI4DB and DB4AI. AI4DB understands projects and approaches that leverage AI technologies to improve database components like cardinality estimators [161], learned indexes [92] or natural language support [148]. Contrary to AI4DB, DB4AI comprises projects that *use* database systems to improve the application of AI technologies on existing data sets. Hence, this work falls into the DB4AI category, although we only consider one class of AI: machine learning with neural networks. DB4AI and AI4DB approaches have been surveyed in [97, 98].

ML on SQL: The MADlib [69] library implements different analytical operations in SQL, including data mining, machine learning, and deep learning algorithms. The library is limited to PostgreSQL-based systems only, heavily using PostgreSQL syntax. Besides being not portable, MADlib also does not support recurrent models that we identified as useful in the database context in Section 6.3.2. The PMML standard [58] defines an XML schema that defines model pipelines to be exchanged between applications and systems. With Bismarck [47], a unified architecture for analysis operators (e.g., Linear Regression and Support Vector Machines) in DBMSs was proposed with the goal to simplify the integration of new operators and to study performance optimization possibilities. While MASQ [30] translates selected *scikit_learn* classifiers and regressors to plain SQL, [136] shows how to build SQL statements for decision trees. In [117] Olteanu shows how to map the ML training

problem to a relational database problem and in [139], Schüle et al. describe how ML training pipelines can be expressed over SQL and how in-DBMS training can be accelerated using GPU implementations.

ML on UDFs or external engines: Many database vendors released support for model inference like BigQuery ML [53], Redshift ML [8], Vertica-ML [46] or SQLServer Machine Learning Services [110]. They mainly focus on providing an SQL frontend and performing computations either using UDFs or integrated ML frameworks like Tensorflow [54]. Raven [76, 121] follows this idea by integrating the ONNX runtime into SQLServer. Inference tasks consist of a SQL query and a Python program (the model pipeline). The optimizer shares information, like used attributes and result sizes, between the SQL operators and the Python code for cross optimizations like early pruning. Raven also automatically translates simple models like decision trees into SQL. Another example for integrating ML models into database systems by using Python UDFs is presented in [130]. Additionally, [166] presents different approaches to integrate ML inference with a DBMS using UDFs, direct file access or Spark, which differs from our approach of integrating it directly into the database engine. In our work, we compare both ML on SQL approaches as well as native integrations. We provide the ML-To-SQL framework that generates SQL statements for neural network inference and a native ModelJoin operator that is not based on any external runtime, and compare them against existing approaches.

Language support for ML: With MLearn [140] another declarative language to define machine learning pipelines was proposed that can be transpiled to Python or SQL (over UDFs). However, only HyPer and PostgreSQL are supported. When dealing with large data sets, an alternative to centralized solutions is Apache Spark. The Spark MLlib[142] includes numerous machine learning algorithms that are implemented using Spark's operations. Similarly, Apache Mahout[10] provides a list of machine learning operations, but also a Scala-based DSL to express new operations. On the other hand, SystemML [25] (later renamed to SystemDS[12]) provides a declarative language to express the ML pipeline which is then optimized and compiled for platforms like Spark or Hadoop MapReduce.

Linear algebra extensions: Especially for ANNs implementations make heavy use of linear algebra operations. MLog [99] or LevelHeaded [3] extends the relational algebra with linear algebra to support machine learning algorithms.

Conclusion: We presented approaches to compile Python code or solve the Python global interpreter lock (GIL) issue. In our approach we integrate the compilation approaches into a interpreted query engine during runtime. Furthermore we introduce UDF engines that run separate Python interpreter instances to achieve parallelism and mitigate the GIL issue. The approaches on Machine Learning present either languages to express ML tasks or integrate these tasks into DBMSs using (Python) UDFs, SQL translations for simple classifiers or decision trees, or using ML runtime C-APIs. We extend this list of approaches with our approaches in which we express ANNs and the model inference step with basic relational primitives only (relations and

plain SQL queries) and a native operator without the usage of any external runtime. In [43] Du showed the applicability of this idea for Graph Convolutional Networks with each layer of an ANN expressed as its own table. During inference many tables that only store intermediate results are created, leading to many expensive update operations. While this paper shows that layers and activation functions can be expressed using relational features, our approaches follow a generic approach to import existing models into a single pre-defined relation and to execute the inference step using (nested) SQL queries that can be optimized by a database system's query optimizer. We thereby follow the goals of not including external runtime engines into the database engine and avoiding UDFs for performance reasons and compare our approaches against the baselines of using an external ML environment or integrating an ML runtime using UDFs or the respective C-API.

6.2 Accelerated Python UDFs

6.2.1 Introduction

Due to the broad functionality, Python UDFs can be used to offer support for advanced analytics as shown by the Grizzly framework in Chapter 5. In this context, typical UDFs may be complex and might include external modules. In order to accelerate these kind of analytics, the main goal of this section is to investigate the performance gap shown in Figure 2.4, aiming to reach native performance as much as possible. Figure 6.1 shows different levels of executing Python code from a database engine. While using the Python interpreter (1) is the most general approach, it also offers the worst performance and is therefore our upper baseline. On the other side, using plain C code (4) provides the best performance while restricting the support for Python modules and is therefore the lower baseline. Using the Python C-API (2) enables pre-compiling Python code in order to limit interpretation overhead while still providing full support of Python functionality. Using Numpy (3) is a special case here. While being fomulated as Python code, [129] shows that Numpy code can reach nearly native code speed as a consequence of a memory layout similar to C. However, UDFs are then limited to Numpy operations and fall back to (2) on the occurence of non-Numpy constructs.

We elaborate possibilities to move down the execution stack shown in Figure 6.1 for scalar Python UDFs, while also evaluating the following orthogonal features for performance improvement:

- **Compilation:** We evaluate compilation frameworks to transparently convert Python code into lower-level code. We show how to dynamically use compiled code during database runtime, which is a challenging task in interpreted query engines, and provide an open-source prototype engine for evaluation.
- **Vectorization:** We elaborate on the impact of Python code vectorization to reduce call overhead.
- **Parallelisation:** We investigate how to parallelise Python code in a parallel query engine, which requires mitigating the global interpreter lock (GIL) issue.

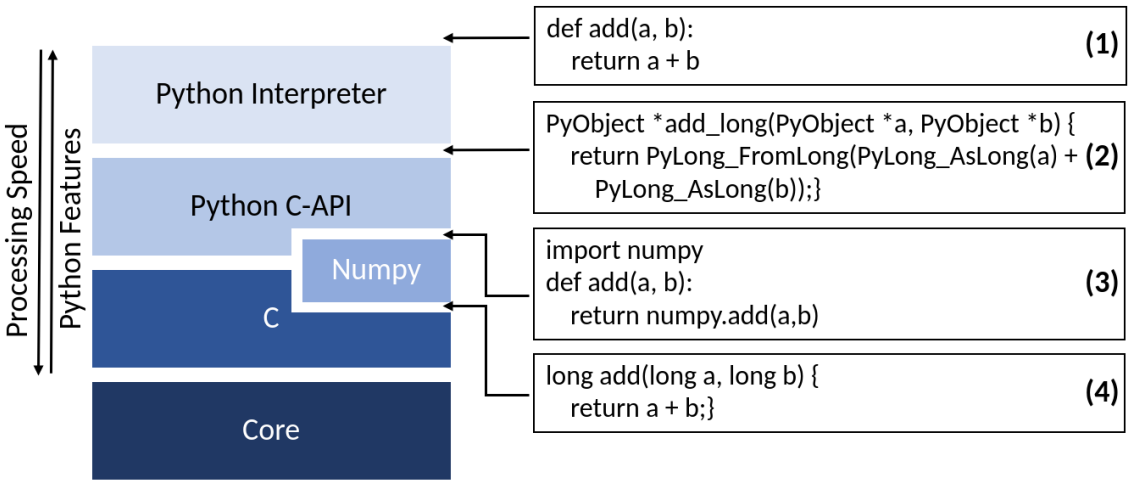


Figure 6.1: Levels of executing Python code

6.2.2 Compilation

As shown in Section 6.1, several frameworks target at increasing Python code performance by compiling Python code just-in-time or ahead-of-time. Similar to this approach of interpreted vs. compiled Python code, the dominant database design approaches of the last decades are also interpreted (vectorized) query engines vs. compiled query engines. While interpreted query engines like x100 built a query execution plan from a set of pre-defined operators and follow an iterator model like the Volcano iterator model [56], compiled query engines produce code for each specific query and compile it to an executable. In comparison to interpreted engines, there is a trade-off between the additional effort to produce and compile code for each query and the faster execution of optimized and compiled code. While using just-in-time compiled UDF code in queries of a compiled query engine involves linking of the query binary and is therefore straight forward, using compiled code in an interpreted query engine during runtime is more challenging. In this section, we investigate Python code compilation and integration by presenting a prototype engine that offers compiled UDF operators, which realize compilation, dynamic loading and UDF execution. The design of these operators considers the following challenges:

C1 **Transparency:** From a user perspective, there is no additional effort necessary to exploit compiled UDFs.

C2 **External code:** Importing external libraries in the UDFs must be possible to benefit from the wide range of Python frameworks.

C3 **Typing:** The operator needs to fill the gap between the strictly-typed database system and the weakly-typed Python code.

C4 **Safety:** The system must be secured from unauthorized access over Python UDFs.

Engine design

Our prototype engine[1] follows the common properties of modern analytical database systems and is written in C. It is designed as an in-memory engine that organizes data in a columnar layout. Column values are tightly packed as arrays in memory resulting in efficient access when iterating over this data. Variable-sized types like strings are stored as pointers to the memory region containing the data. Additionally, the engine follows the Volcano iterator model in a vectorized fashion as described in Section 2.2.2. As the intention of the prototype is to explore the design space of UDFs, it only contains a basic file scan operator, multiple different UDF operators, logic to "consume" the query result at the end of a query and a fine-grained profiling mechanism.

We integrated the most popular compilation/transpilation frameworks Numba, Nuitka and Cython into our prototype engine and created a separate operator encapsulating each of the frameworks. During the `open()` call, the UDF code is compiled using the respective framework, dynamically loaded into the running engine and initialised. As a baseline, we also designed a native Python UDF operator, which is based on the Python C-API. In order to achieve the transparency goal C1, the native Python operator can be used as a fallback in case of compilation or loading errors.

UDF compilation

The UDF compilation step is similar for each of the used compilation frameworks and only differs in minor, framework-specific details. However, all frameworks also share the same challenge related to the way Python handles imported modules. All frameworks are originally designed to produce a compiled module that is used in the Python interpreter again. When facing an `import` statement, the Python interpreter calls the module's initialisation method which initialises the module and provides an entry point to the module's functions. A subsequent function call is then resolved using this entry point. Consequently, a module's functions are not visible to code outside the module, especially not for our query engine which does not run in a Python context.

In order to make the UDF code visible, one could manually rewrite the generated code. However, this is fundamentally different for different frameworks and makes the compilation process highly dependent on the compilation framework version, as each change in the code generation would require changes in the manual code rewriting. The design of the compilation step follows the goal of abstracting as much as possible from the used framework. Therefore, we solve this issue using a rather technical trick. For the compilation, we configure the compilation frameworks to keep intermediate states of the compiled code. This way, we get access to the object files in the Executable and Linkable Format (ELF) before they are linked together to a shared library. ELF is a standard format for shared libraries, objects or executables and has a fixed structure in its header and data segments, being a flexible representation for code and data. As part of the ELF format, symbol tables are used to maintain symbolic references, e.g., functions, to the respective part in the

[1] `https://github.com/dbis-ilm/Compiled_UDF_engine`

code segment. Function symbols are visible outside of a library or executable when they belong to the dynamic symbol table, having global visibility and default binding properties. Using ELF manipulation tools like LIEF[100] we manipulate the visibility and binding of the function symbol that belongs to the UDF accordingly and repeat the last compilation step to produce the shared library. Consequently, we chose to accept duplicate work in favor of abstraction from the compilation framework, so we do not change a framework's compilation behaviour to manipulate the generated files in between, but simply repeat the last linking step.

In the compilation step, we also make the first step to solve challenge C3. When the UDF is created, the user needs to specify input and output data types of the function. This is not a restriction, as this metadata is also available in a later integration into a full-fledged database system, where a user needs to specify the types in a `CREATE FUNCTION` query. Using this metadata we annotate the UDF code when necessary, like in the case of Numba. Additionally, supporting external modules as described in challenge C2 is a property of the compilation framework and therefore automatically provided, except for the Numba framework which only supports the Numpy library as an import. In order to achieve safety as stated in challenge C4, we maintain a banlist of modules and scan the UDF code before the compilation for the items of the banlist. This way, we avoid that users can execute statements potentially dangerous for the system or the underlying environment, e.g., by using the `os` or the `sys` module.

The compilation process is performed using a Python script consisting of calling the framework, modifying symbol visibility and re-running the last compilation step. As this script is not changed during database runtime and consists of Python code, it can be compiled using the compilation script itself to produce a shared library containing the compilation logic and dynamically linked with the database engine. As a result, calling library functions is faster than invoking a Python script out of the database engine.

Dynamic loading and initialisation

After compilation, the library containing the UDF must be stored at a well-defined place that is accessible for the system. While this can be easily achieved in single-server database systems, we argue that even for distributed database systems this is not a limitation due to the presence of cloud storage in the cloud environment or shared storage like HDFS in the case of on-premise cluster solutions. In the next step, the library is now dynamically loaded into the running database application using `dlopen/dlsym`. The loading step is thereby driven by the goal to avoid as much effort as possible in the later execution, especially expensive branching.

Cython, Nuitka and Numba's object mode are based on Python's `PyObjects`, which are the type abstraction of Python's C-API and can contain arbitrary types. For these frameworks, we initialise input and output conversion functions based on the UDF signature and store them as function pointers to avoid branching during execution. The actual UDF's signature may however consist of arbitrary parameter type combinations. As this is not known during the compilation of the database engine, we define a placeholder function type and generate a wrapper function and a function definition for every possible UDF signature. In practical cases, the number

```
typedef void (*fplaceholder)(void);
typedef void (*WrapperFunc)(fplaceholder, char**, char**);
// For every parameter combination
typedef double (*dld)(long, double);
void func_38(fplaceholder f, char **params, char **out) {
  long* p0   = (long*) &params[0];
  double* p1 = (double*) &params[1];
  *((double*)out) = ((dld)f)(*p0, *p1);}
```

Listing 6.1: Example of the wrapping code

of possible UDF parameters must be reasonably limited. The wrapper is correctly set according to the UDF signature during the loading step and therefore introduces branching only once. As shown in an example function for a UDF that takes a long and a double parameter and returns a double in Listing 6.1, this wrapper casts the generic input parameters as well as the actual UDF and calls the UDF. The result is then stored at the given buffer location, which is a part of the result vector. By storing the wrapper code as a function pointer, we avoid branching during the UDF execution while also solving the typing challenge C3.

Besides the UDF, we also load the symbol for the library's initialisation function if required by the compilation framework. Again for Cython, Nuitka and Numba's object mode, we initialise the module in the Python enviroment afterwards potentially using Python's multi-phase initialisation. Note that this might happen after the Python interpreter was initialised during the engine startup, depending on a system's behaviour to use the Python interpreter, i.e., long-running vs. starting a (sub-)interpreter per query. This initialisation enables access to the module's internal data structures and helper functions, being the equivalent of Python's `import` statement.

UDF execution

As described in Section 6.2.2, our prototype engine follows the Volcano iterator model in a vector-at-a-time fashion and has a columnar storage layout. Consequently, each UDF operator receives a set of vectors (one for each column) from the child operator in the query tree. Keeping these vectors unchanged, the operator adds a result vector for the UDF output to the intermediate result. If required by the used compilation framework, the UDF input values are converted to `Py_Object` elements using the respective input conversion functions. The compiled UDF is called using the wrapper function and the results are converted back using the output conversion functions if necessary. These three steps are performed in a vectorized fashion. Note that there is no branching in this step, as function pointers were correctly set during the dynamic loading and initialisation as described in Section 6.2.2.

Another important aspect of an execution engine is NULL handling. In our approach, we basically have two options where to perform NULL handling: in the database engine or in the UDF code. The engine-internal handling would require to only execute the UDF on non-NULL values. In the vectorized variant, one would therefore need to track the offsets inside the vectors to be able to put results into the right position again after UDF execution. Performing the NULL handling in the

UDF on the other hand simply requires the input conversions to produce `Py_None` as Python's NULL indicator. In our engine prototype, we decided for the second option to perform NULL handling in the UDF, as this offers more flexibility to the user to handle NULLs. One could imagine use cases where UDFs explicitly need to handle cases where some parameters are NULL, which would not be possible to support when UDF calls are skipped by the engine-internal NULL handling.

6.2.3 Vectorization

The performance of the UDF execution is highly dependent on the way the UDF code is written. In commercial systems, user-written UDF code is likely to get wrapped into a code template. This way, customised error checking mechanisms, type conversion and interpretation for complex types and vector iteration mechanisms can be achieved. One can think about a user function that is called from a main function in different ways. If the user code is written in a tuple-at-a-time fashion, the user function is called for each element of the vector, introducing significant call overhead. If the user code is written in a vector-at-a-time fashion, the user function is only called once, significantly increasing speed. During our design we found the call overhead inside the Python code as one of the most limiting factor for performance. Using a vectorized UDF that pre-allocates the result vector and uses list comprehension (potentially zipping multiple input vectors before) showed the best performance.

In the current project state UDF vectorization is enabled manually by the user by (1) rewriting the UDF code to the vectorized version and (2) indicating vectorization using a prefix in the UDF naming. In order to further extend usability and reach the transparency goal C1, we aim at an automatic UDF rewriting mechanism in the future that embeds the UDF code into a vectorization template.

6.2.4 Parallelisation

Using UDF compilation as described in the Section 6.2.2 is intended to increase single-core performance of the UDF execution in a vectorized, interpreted query engine. However, today's analytical database systems are typically massively parallel processing (MPP) systems to benefit from the increasing number of cores per socket and the possibility to scale systems in cluster environments. Therefore we want to describe possibilities to also scale the UDF execution, which is orthogonal to the compilation approach.

We assume a MPP system that runs multiple workers to execute query plans or subplans in parallel on independent data partitions. When integrating our compiled UDF approach in such a system, the frameworks that still use the Python environment (Cython, Nuitka, Numba object mode) would share the Python interpreter. As described in Section 6.1, this case would lead to high contention due to the GIL issue. Consequently, enabling parallelisation in Python leads to mitigating the GIL issue. Instead of using special transactional memory, we designed UDF engines that run as processes separately from the actual database workers as sketched in Figure 6.2 (Note: workers might be either processes or threads, but UDF engines

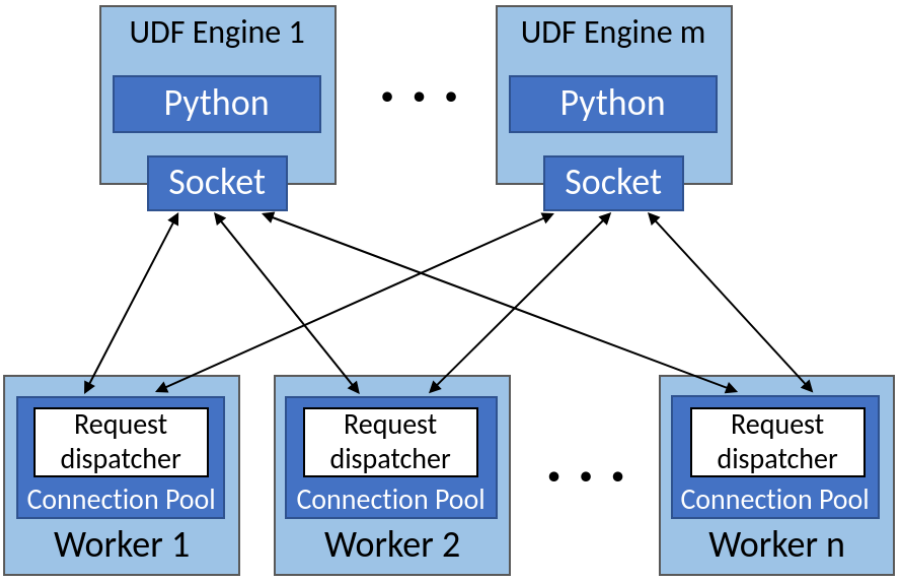


Figure 6.2: Overview over the UDF engine pool

are separate processes). These UDF engines start their own Python environments, which are therefore isolated by the process semantics. Database workers and UDF engines can be scaled independently and are connected over point-to-point socket connections. On the database engine side, these connections are maintained in a connection pool and requests (i.e., UDF execution requests) are dispatched over a scheduling strategy. For now, we use a simple round robin assignment strategy to assign requests to UDF engines. As vector sizes are fixed, this simple strategy already provides a balanced utilization of the UDF engines. Even when data is skewed, the unbalanced load in the database workers would be balanced among the UDF engines. After execution, results are sent back to the database workers over the socket connection. This "remote UDF" approach is again a tradeoff, as it introduces inter-process communication and data transfer while enabling parallel execution in the Python environment. As a side effect, moving the Python execution to separate processes offers possibilities to improve the security of running Python, e.g., by running the UDF engines in containers.

6.2.5 Evaluation

In the evaluation, we compare the code generated by the integrated frameworks as well as the impact of compilation, vectorization and parallelisation. The evaluation was performed on a machine consisting of a Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz and 24 GB RAM. We designed a set of experiments to represent different use cases:

- **Mod:** A simple modulo UDF, representing a mathematical operation.
- **Avg_word_len:** Splitting a string and computing the average word length, representing a string analytics function that consumes the whole string.
- **Roberta:** Using the Roberta ONNX model [102] to analyse text sentiments.

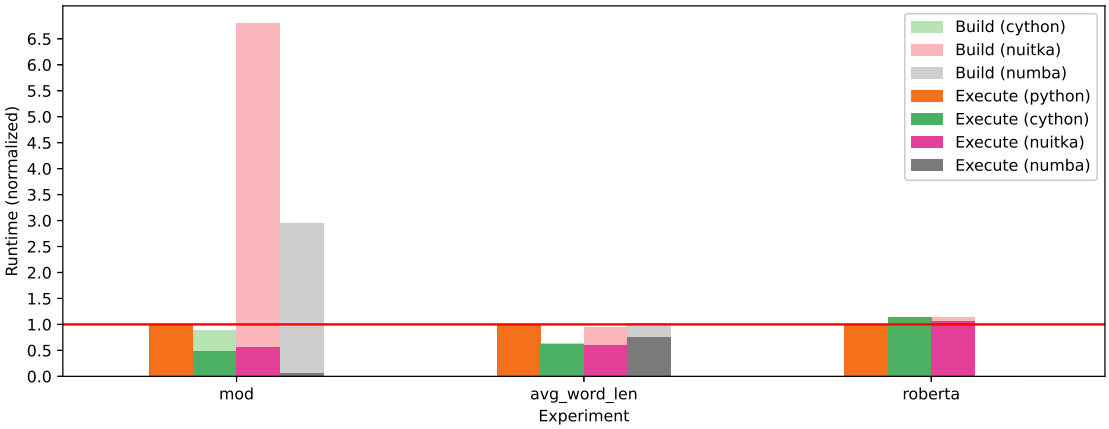


Figure 6.3: Compilation framework comparison in prototype engine

The experiments were performed on the TPC-H [27] partsupp table at SF10 and the IMDB movie review dataset [104]. In the first experiment we compare the behaviour of the integrated compilation frameworks using the prototype query engine. In this experiment, we manually write wrapped UDF code as explained in Section 6.2.3 in order to simulate the behaviour after database system integration. Figure 6.3 shows the results for the described experiments with the runtime being normalized against the native Python operator runtime, which acts as the reference. Build runtime including compilation is in a similar order of magnitude for each experiment, however taking a different part of the overall runtime depending on the execution runtime. For the modulo experiment we can observe a performance benefit for all compilation frameworks, with Numba being the best one as it produces native C code in this example equivalent to the lower baseline. It reaches stack level (4) in Figure 6.1 and runs around 20 times faster than the reference. For string analytics in the avg_word_len example, we can again see a performance benefit for all frameworks. However, Numba is slower than Nuitka and Cython here as it falls back to the Python object mode to handle strings. In the Roberta example, Numba is not applicable as the UDF uses an imported module. As the compilation frameworks obviously cannot influence the module code and the ONNX runtime environment, which is responsible for the major part of the runtime in this example, compilation does not lead to a significant benefit here. As we aim at a solution that is transparently applicable for all use cases, Numba is not an option. Nuitka and Cython both reach stack level (2) of Figure 6.1 and showed similar performance in all cases leading to a speedup of around 2 in mathematical operations and string analytics, while Cython has significantly lower compilation effort.

Based on the first experiment, we integrated the UDF approach using Cython as the compilation framework into the Actian Vector database system at version 6.1. In the second experiment we therefore operate on stack level (2) of Figure 6.1 and evaluate end-to-end query performance using combinations of compilation, vectorization and parallelisation. The queries thereby consist of a scan, the UDF and the aggregation of the result to reduce the result size. The runtime results are shown in Figure 6.4, again normalized against the reference runtime, which is

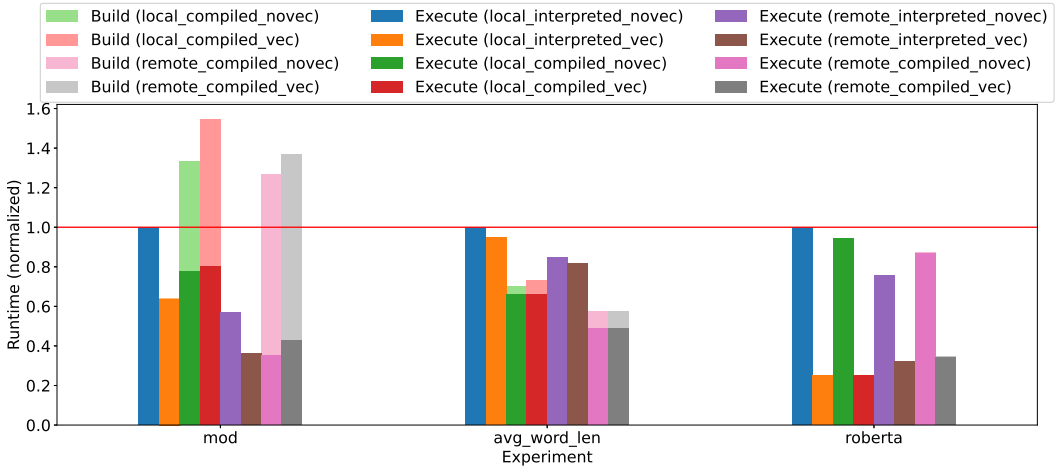


Figure 6.4: Actian Vector results

the interpreted, non-vectorized and non-parallelised version. We can observe that compilation has a significant benefit compared to the respective interpreted variants, with compilation expectedly being only slightly better than interpretation in the Roberta example, as major part of the runtime takes place in the ONNX runtime environment. Vectorization also leads to performance gains in nearly every case except the compiled variants, as the Cython framework seems to have problems to produce efficient code for list comprehension. Cython provides an advanced Python dialect, offering more efficient compilation for specially annotated code constructs. This could be exploited in the future by automatically rewriting the user defined python code into the Cython dialect in order to produce faster compiled code. However, Roberta especially benefits from vectorization, as the ONNX model needs to be loaded only once per vector instead once per tuple. Parallelisation using a parallelisation degree of 4 remote UDF engine processes additionally increases performance in our experiments except the Roberta example, as this experiment relies on the scalability of the ONNX runtime. Overall we can derive the following key findings from our evaluation:

- Only stack level (2) of Figure 6.1 is reachable when transparently supporting arbitrary python code.
- Compilation together with parallelisation leads to the best performance in most experiments with speedups between 2 and 3 compared to the Python execution.
- Compilation does not have an impact on functions that rely on external modules.
- Parallelisation using external UDF processes introduces overhead of inter-process data exchange, but leads to a speedup during execution by mitigating the GIL issue.
- Vectorization leads to performance gains by reducing Python call overhead.

Consequently, these features are a good choice for a broad range of use cases like user-specific calculations, scoring functions, analytics or ML pre- or post-processing steps. ML runtimes however do not benefit from the presented approaches. We therefore investigate approaches for a deeper integration into the database engine in the following.

6.3 Approaches for In-Database ML Inference

6.3.1 Introduction

In this section, we investigate the topic of in-DBMS batch inference from a performance perspective and evaluate several approaches to integrate model inference deeper into database systems by using different levels of abstraction. Performing ML inference on a table (i.e., joining the model with the table), we envision a high-level concept like

```
SELECT * from table MODEL JOIN m
```

which offers opportunities for representing the model m as a table, a tuple or a runtime object. We focus on neural networks as a subclass of ML models that can be useful for relational workloads. In general, we identify four classes of approaches for in-DBMS ML inference:

1. **Python UDFs:** A naive approach of pushing model inference into the database system is to exploit the recent upcome of Python user-defined functions (UDFs) in modern database systems, as shown in the case of the Grizzly ModelJoin in Section 5.3.3. However, this approach still moves data outside of the DBMS and performs expensive computations in the Python interpreter and therefore misses to make use of the powerful database engine.

2. **Native APIs of ML runtimes:** A basic approach of integrating ML runtimes over native APIs is given by Raven [76, 121], which requires the integration of external dependencies and conversions between the columnar data layout of an analytical database engine and the data layout of the runtime.

3. **SQL:** Model inference can be translated to SQL leading to a highly portable solution to any SQL compliant database. This also requires a relational model representation.

4. **Native Operator:** Model inference can be implemented directly as an operator inside the database engine without the need for external dependencies. The solution can be tailored to the engine's data layout.

The authors of [166] claim that "It is far too tedious for DBMS developers to reimplement DL algorithms. So, one must preserve the usability of DL tools such as TensorFlow for specifying complex DL workloads." While this might be true from a generalizability perspective, we scrutinize this claim from a performance perspective. Besides the existing approaches (1) and (2), we present additional approaches for

classes (3) and (4) that either reuse existing database capabilities or integrate ML inference natively into the engine. Our first approach translates trained models into a SQL representation using standard relational concepts. On a different abstraction level, we design a native ModelJoin operator, which relies on the relational model representation and performs the inference of neural networks directly on the data. Overall, this section provides the following core contributions:

- We classify existing neural network layer types and discuss their applicability to typical database workloads.
- We propose a generic relational representation of neural networks that is able to handle dense layers as well as recurrent layers.
- Based on the relational representation, we propose the ML-To-SQL Python framework, which generates standard SQL code for ML models. ML-To-SQL is a highly portable, extensible and easy-to-use way to perform model inference without any database engine changes, but still leverages the DBMS's capabilities for efficient query processing.
- As an alternative, we propose a native ModelJoin database operator, which is also based on the relational model representation and therefore performs the model inference independently from any ML runtime environment. The ModelJoin is implemented as a CPU and a GPU variant.
- We compare candidates for the presented classes of approaches, including the ML-To-SQL framework and the ModelJoin operator, against the baseline of moving data out of the database and using Python for inference in terms of performance, memory footprint, portability and generalizability.

6.3.2 ML Layer Classification

Various ML models have been developed to solve different kinds of problems. Artificial neural networks (ANNs), and especially deep neural networks have been proven to solve complex tasks, such as language translation, word or sentence completion and prediction, or image classification. There are various architectures for artificial neural networks (ANNs) that can be used to solve different tasks. In this following we discuss the most common architectures currently used and their typical application scenarios within database system. We thereby focus on relational data, as non-relational data is typically not stored directly in a database. Consequently, we only discuss the applicability of the model architectures for this kind of structured data, leaving out use cases of the models on unstructured or semi-structured data.

Perceptron and Feed Forward Networks The most basic form of an artificial neuron is the perceptron. It consists of a single activation function which is applied on the sum of the inputs. Single perceptrons are used to solve classification problems with two classes. In this most basic form, the single perceptron can easily be used within database systems: once the perceptron has been trained, the learned weights are multiplied with the corresponding input value and their sum in fed into the

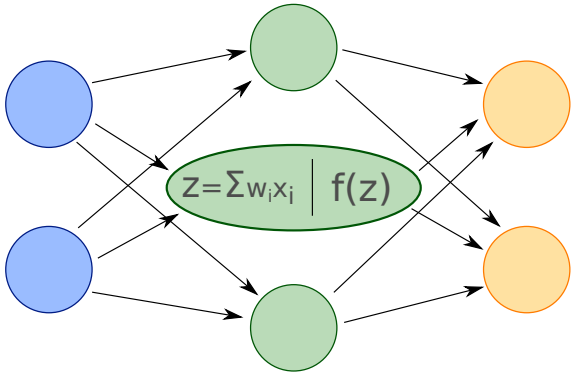


Figure 6.5: An example of a multi-layer perceptron showing the internal building blocks of an artificial neuron.

activation function. The result of the activation function is the classification result. Therefore, the concept of a perceptron can be easily expressed in SQL (for x and y being attributes/columns in *INPUT_TABLE* and b an optional bias):

```
SELECT w1*x + w2*y + b > 0 FROM INPUT_TABLE
```

The result of the perceptron is either *1* or *0* (or *TRUE*/*FALSE*), denoting the classification result of an object in a two dimensional input space. The concept of a perceptron can of course be applied to problems in higher dimensional spaces. To solve the limitations of the single perceptron (e.g., the XOR problem), the multi-layer perceptron (MLP) [135] additionally uses a hidden layer of neurons. The output of a neuron in the input layer are used as input for the neurons in the hidden layer and the outputs of the neurons in the hidden layer are used as inputs for the neurons in the output layer. Since neurons in one layer are connected only to neurons in the next layer and information is passed through the network only in one direction, this network architecture is called *feed forward networks*. As each neuron in one layer is connected to all neurons in the next layer, such layers are called *dense* layers. Artificial neurons use activation functions to filter irrelevant information. The activation function f takes the sum of the weighted inputs as an argument and produce an output. State of the art activation functions are, for example, *ReLU*, *sigmoid*, or *tanh*. Their characteristics and choice in application scenarios, however, are beyond the scope of this work as we mainly focus on performance independently of the model semantics.

Figure 6.5 shows an example MLP with the internal building blocks for one of the hidden neurons. Within a layer, all neurons use the same activation function, but different layers typically use different functions. Since single perceptrons and MLPs are mainly used for classification tasks, they mark an important candidate for pushing the inference step into the DBMS. Most recent success in machine learning comes from new advances in Deep Learning technologies, i.e., network architectures that use more than one hidden layer.

Recurrent Networks While in classic feed forward networks a neuron gets its input only from the previous layer, in recurrent neural networks (RNNs) [159] the

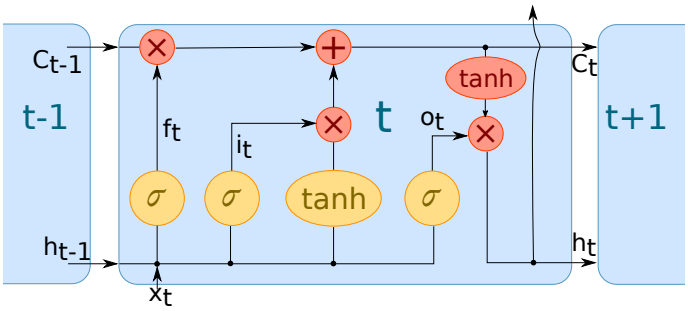


Figure 6.6: The internal gates of a LSTM module.

output of a neuron is also used as input in the next or any later iteration. This delay allows to include, e.g., in text processing, the context of a word by "remembering" previously encountered words, or "remembering" previous time steps in time series processing and forecasting. While neurons in classic RNNs are composed of a single activation function, advanced versions such as Gated Recurrent Units (GRUs) [33] and Long/Short Term Memory (LSTM) [73] networks make use of a more sophisticated structure to better include historical information in the learning process. The recurrent part in a LSTM includes a *cell state* and consists of actually four layers. Information is added or removed from the cell state by *gates*, that use the sigmoid function.

The recurrence in RNNs, GRUs, and LSTMs can be depicted by a chain of the LSTM module, as shown in Fig. 6.6. How often the module appears in the chain is determined by the number of time steps a network is supposed to look into the past. Besides transformer models [153], such recurrent networks and especially the LSTMs are used for time series analysis or various speech and text processing tasks, e.g., translation programs or text prediction algorithms. Since databases often store textual information, e.g., product reviews or user comments and gather time series data, e.g., from IoT applications, many use case scenarios would benefit from an in-DBMS execution of RNNs and LSTMs.

Convolutional Networks The classes of Convolutional Neural Networks (CNNs) [94] or Deep Convolutional Networks (DCNs) consist of another type of layers: kernels and pooling layers. The convolution kernels process the input data whereas in the pooling layers the output of the kernels is simplified. In contrast to the dense layers of MLPs, where output from one neuron is propagated to all neurons in the next layer, in CNNs a neuron only considers information within its *receptive field*. The receptive fields of all neurons finally cover the complete input space. Convolutional networks resemble the work of the visual cortex in our brain and are therefore also mainly used for image and audio processing. In databases, images and audio data are typically stored in BLOB fields. While we believe it would be possible to also represent CNNs in a relational DBMS, we argue that in most cases users do not store the actual image/audio files inside their database, but rather only the paths to the files. Consequently, we do not consider CNNs any further in this context and leave it for future work.

Conclusion From the above discussion we conclude that RNNs and especially the more modern LSTMs as well as classical feed forward networks with dense layers are used to solve problems based on data stored in databases. Therefore, in the following discussion of integrating model inference, we focus on these two ANN architectures.

6.3.3 ML-To-SQL Design

As the first approach to achieve our goal of pushing model inference into the database system we use SQL as the standard interface of a database system. We thereby achieve a highly-portable approach which might show non-optimal performance due to using generic query operators instead of a specialized ML environment. In this section we describe the design of the ML-To-SQL Python framework[2]. Using a pre-trained neural network model and a database connection, our ML-To-SQL framework offers a simple API to automatically generate model tables and SQL queries to perform model inference. We first present the generic relational model representation that handles dense layers as well as LSTM layers, which were shown in Section 6.3.2 to be the most important layer types for database workloads. Based on definitions for different function types, we then present the layer-specific implementations, which can be used as building blocks to construct SQL code for ML models. Note that based on stored parameters in the relational table representation and the approach of having extensible building blocks for SQL code generation, ML-To-SQL is also applicable for the existing approaches for decision trees or classifiers presented in Section 6.1. However, we focus on neural networks in this section.

We waive the topic of data encoding, as basic approaches like Min-Max-Encoding or One-Hot-Encoding can be implemented in SQL in an straight-forward way and are already covered by, e.g., MADlib [69]. Similarly, we assume that when having an LSTM model, the number of input columns is equal to the number of time steps the LSTM layer considers. Starting from a simple time series, this can be achieved by self-joining the table $n-1$ times for an LSTM layer considering n time steps, with a join predicate that lets tuples match with their predecessor in the series (e.g., by using a unique identifier or a timestamp).

Relational Model Representation

Conceptually, neural networks can be seen as directed graphs, as shown in the example model in Figure 6.7. Nodes perform a layer type specific computation that aggregate the weighted sum of its inputs as well as an activation function to produce the output. For two nodes i and j, an edge (i, j) is annotated with weight w_{ij} and all weights of a layer are collected in the kernel matrix. Additionally, a constant bias is connected to each node, with the weights forming a bias vector. Inputs are connected to the first layer and the output of the last layer is the model output. The example model would get two inputs and produce a single output. As described in Section 6.3.2, recurrent layers have a kernel as well as a recurrent kernel for each of the blocks in the block diagram. Recurrent kernels are used to combine the output of the last time step with the current output.

[2]Available as open source under `https://github.com/dbis-ilm/ML-To-SQL.git`

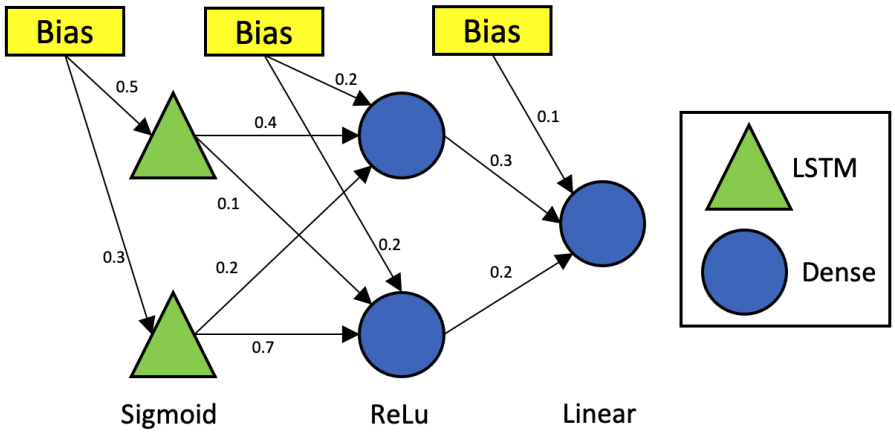


Figure 6.7: Example model graph

Table 6.1: ML-To-SQL function type definitions

Function type	Function signature
Input function	$R(ID, A_1, \cdots, A_n) \times Model \rightarrow R'(ID, Layer, Node, Output\_activated)$
Layer forward function	$R(ID, Layer, Node, Output\_activated) \times Model \rightarrow R'(ID, Layer, Node, Output)$
Activation function	$R(ID, Layer, Node, Output) \rightarrow R'(ID, Layer, Node, Output\_activated)$
Output function	$R(ID, Layer, Node, Output\_activated) \times S(ID, C_1, \cdots, C_i) \rightarrow S'(ID, C_1, \cdots, C_m, Prediction)$

Based on this, we define a relational representation of a model by holding information about edges as well as its weights. A node in the graph is identified by the unique pair $(Layer, Node)$ and an edge by the tuple $(Layer\_in, Node\_in, Layer, Node)$, all being integer values. For each edge we hold kernel weights W_i, W_f, W_c and W_o, recurrent kernel weights U_i, U_f, U_c and U_o as well as bias weights b_i, b_f, b_c and b_o, all being 4-Byte floating point values and representing the computation in the LSTM cell gates in Figure 6.6. Hence, the model table is defined to have 16 columns. Depending on the layer type, weights might be empty which is efficiently handled by X100's columnar storage layout, offering effective compression and the possibility to scan only necessary columns.

Our ML-To-SQL framework generates SQL code to automatically load a Python model object into the relational table representation by iterating over the model graph and producing layer type specific insert statements. The framework currently supports Keras models, but can be similarly extended to other ML frameworks in the future. As we represent models as tables holding a single tuple per edge in the neural network and not a single tuple per model, we denote the model inference as a specialized join operator in further discussions, combining a model table and a fact table.

Definitions

We denote a relation as $R(ID, A_1, \cdots, A_i)$, where R is the relation identifier and $A_1, \cdots, A_i$ is a list of attributes. We specify a column ID as a unique row identifier and assume its existence, may it be a primary key or a builtin row identifier column. We use the term Relation in the following to describe either tables or intermediate results, with the latter residing in memory and potentially not being materialized. A

```
ModelJoin :=
  Output(
    Activate(Layer_forward(
      ...
        Activate(Layer_forward(
          Input(R(ID,A_1,...,A_n), model),
        model)),
      ...
    model)),
  R(ID,C_1,...,C_m))
```

Listing 6.2: ModelJoin as nested ML-To-SQL function types

model is a special relation with the column definitions described in the relational model representation.

Additionally, we define a set of function types as shown in Table 6.1. Intuitively, we aim at keeping track of the model state for each tuple of a fact table on its way through the neural network graph in parallel. This is realized by combining the ID with the $(Layer, Node)$ identifier for each node in the graph. An input function takes a fact table and a model table and performs the initial join. For simplicity we assume that all columns $A_1, \cdots, A_n$ are input columns for the model, which means that the model input layer is of size n. With the input layer being a special model layer, each node i gets a single input value A_i and linearly propagates it to the output. On the contrary, an output function takes an intermediate model state (which is the result of the last model layer in most cases) and the fact table and joins the model inference result with the respective input tuple based on the unique tuple ID.

The initial assumption that all columns $A_1, \cdots, A_n$ are input columns for the model is not a restriction, as all "payload" columns are joined to the model result after the inference. In relational query execution this is similar to the commonly used optimization rule of "late projection", which avoids pulling a potentially large payload that is not used by query operators through a query tree, but joins it with the result just before returning it to the user.

The layer forward function is the main building block to traverse the model graph. It joins the current intermediate result with the model by joining the intermediate result's $(Layer, Node)$ pair with the models $(Layer\_in, Node\_in)$ pair and this way moving forward to the next layer in the graph. Afterwards it performs the layer type specific calculation and aggregates the results to produce the node output. Again, this is done per node and per tuple, keeping track of the model state for every tuple in parallel. An activation function takes the intermediate result of each node state per tuple and computes the activated output.

Finally, we can now define the ModelJoin as a nested construct of the above described function types. As shown in Listing 6.2 the ModelJoin between a relation R and a model can be described as a nesting of an input function into a series of layer and activate functions and a final output function. With these four basic building blocks defined in a generic way, ML-To-SQL can be easily extended with different functions for, e.g., different layer types or different activation functions. Basic approaches for realizing this building blocks are recursive queries or nested queries. We decided for the latter as we expect similar performance but better debugability and observability of the generated query.

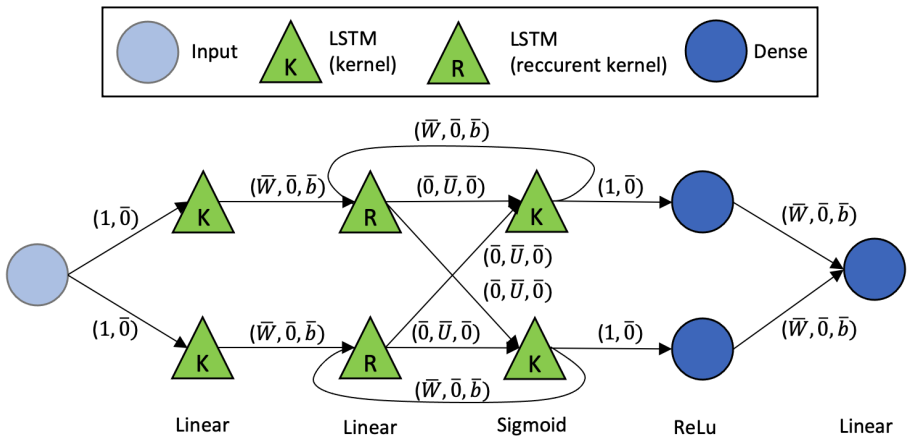


Figure 6.8: Internal representation of the example model graph

```
SELECT id, layer, node,
       value_1 as C1, ... value_n as CN
FROM INPUT_TABLE as data, MODEL_TABLE as model
WHERE model.layer_in = -1
```

Listing 6.3: LSTM input function template

Implementations

Internally, the ML-To-SQL framework transforms the model graph into a different representation, shown in Figure 6.8. Similar to the relational model representation a vector of weights is attached to each edge instead of a single weight. This vector holds 12 elements, being the concatenation of 3 parts: the kernel weights $\overline{W} = (W_i, W_f, W_c, W_o)$, the recurrent kernel weights $\overline{U} = (U_i, U_f, U_c, U_o)$ and the bias weights $\overline{b} = (b_i, b_f, b_c, b_o)$. In Figure 6.8 we use a variable sized vector of zeros $\overline{0}$ when its length is clear from the context. The bias nodes are dropped from the model graph and bias weights are placed into the weight vector. Although this replicates the same bias weight to every incoming edge of a node, it later avoids the need for an additional join. Furthermore, we introduce an artificial input layer consisting of a single node. In the following, we describe the layer-specific implementations of representing a layer in the relational model table and performing the layer-forward-function.

Input Layer The input layer is responsible for joining the fact table with the model table, realizing the input function from Table 6.1. The layer consists of a single node which is connected to each node of the first layer. Each of these edges has a weight $W_i = 1$ and subsequent zeros in the weight vector. The way the input join is performed depends on the type of the first layer. For LSTM layers, we assume that a tuple consists of n columns for n timesteps in the time series. All input columns are passed to the LSTM layer, leading to a simple SQL template cross-joining the inputs and generically renaming columns like shown in Listing 6.3. The activated output is here a list of columns instead of a single column.

For dense layers, performing the inference for an input tuple conceptually requires transposing the tuple and attaching the i-th input column to node i of the first layer. As transposition is difficult in SQL, the fact table is cross-joined with the input layer instead and input columns are generically renamed. As now all input columns are

```
SELECT id, layer, node, CASE
  WHEN node=0 then C0
  ...
  WHEN node=N then CN
END as output_activated FROM
  (SELECT data.id as id, COL0 as C0, ..., COLN as CN,
          layer, node
  FROM INPUT_TABLE as data, MODEL_TABLE as model
  WHERE model.node_in = -1) as t
```

Listing 6.4: Dense input function template

```
SELECT id, node, layer, s + bias as output FROM
  (SELECT id, model.node as node,
      model.layer as layer,
      SUM(input.output_activated * model.W_i) as s,
      model.b_i as bias
  FROM (QUERY) as input, MODEL_TABLE as model
  WHERE input.node = model.node_in
        and input.layer = model.layer_in
  GROUP BY id, model.node, model.layer, model.b_i) t
```

Listing 6.5: Dense layer forward template

attached to each node of the first layer, a switch statement selects the i-th column as an input for the i-th node, shown in Listing 6.4. Due to the weight $W_i = 1$, the input is equal to the output of the artificial input layer.

Dense Layer Transforming a dense layer into the relational representation inserts a tuple for each incoming edge into the model table. Dense layers only have a single weight and bias for their layer forward calculation, so we set the weight at the position of W_i and the bias at the position of b_i in the weight vector, with the remaining positions being zeros. Thus, for the last layer of the example model in Figure 6.8, we insert two tuples into the model table, and each weight vector only contains two non-zero values.

In order to realize the layer forward function from Table 6.1 for dense layers, we need to join the intermediate result with the model, multiply the inputs with their weights, add their biases and aggregate the results for each node and each tuple, as we keep track of the model state for each input tuple. This can be performed by the SQL template shown in Listing 6.5. we traverse the model and follow the edges by joining on $(Layer, Node)/(Layer\_in, Node\_in)$ pairs and nest the query for layer i into the generated query for layer $i + 1$.

LSTM Layer In order to transform an LSTM layer to the relational model representation, we need to consider the different shapes of the kernel matrix and the recurrent kernel matrix. With n being the dimension of the LSTM layer and m being the dimension of the preceding layer, the kernel matrix is of size $m \times n$, while the recurrent kernel is of size $n \times n$. To represent this behaviour in the relational representation, we split the LSTM layer into two separate types of sublayers. The computation of the layer forward function is also separated into two building blocks and based on the Keras implementation. The "kernel" sublayers take the (initial empty) inner cell state and the series of inputs. If there is an input in the series left, it

performs the kernel multiplication and bias addition on the first input value and drops it from the series. Then, it passes the result back to the previous "recurrent kernel" sublayer, which is the equivalent of the recurrence in the LSTM computation. If no input is left, the output is passed to the next layer. The "recurrent kernel" sublayer takes the the output from the "kernel" sublayer and performs the recurrent kernel multiplication. In the example in Figure 6.8, the backward edge would not exist if the LSTM layer only considered two timesteps, as each element of the input series is consumed by a "kernel" sublayer. Additionally, only the last "kernel" sublayer has an activation function. Each "kernel" sublayer produces the output type of the layer forward function in Table 6.1, potentially with an additional cell state that does not exist in the result of the last sublayer.

In the relational model representation, we also consider both, kernel and recurrent kernel sublayers with their respective edges. However, we store each of them only once, because weight matrices are equal for every time step. In the example in Figure 6.8, we would drop the second "kernel" sublayer with its outgoing (backward) edges. When initially passing a model object to ML-To-SQL, the number of time steps the LSTM layer considers is determined. This allows us to automatically generate the backward edges in the layer forward function by joining the $(Layer - 2, Node)$ key with $(Layer\_in, Node\_in)$ and this way stepping two layers back.

Output layer In the relational model representation we do not hold an explicit output layer. When the last layer is reached during query generation, we apply the output function as described in Table 6.1. For each ouput layer node, the original fact table is joined with the inference result on the unique identifier column in order to add the prediction result to the respective tuples. This way we perform the "late projection" as described in the function definitions. If there is a single output node, meaning that we have only one result per ID value ($(Layer, Node)$ pairs are equal for all results), a single join and column renaming is sufficient. Otherwise we perform multiple joins, each with a filter on the $Node$ column of the inference result.

Activation Functions As implied by the function signatures in Table 6.1, an activation function can be applied after every layer forward function. The activation function thereby only consists of a projection on the ID, $Layer$ and $Node$ column and a function call applied on the *output* column, with the result being renamed to *output_activated*. ML-To-SQL currently supports the basic activation functions *linear*, *ReLU*, *sigmoid* and *tanh*.

Optimizations

Advancing from the basic ML-To-SQL workflow described so far, we made a set of optimizations in order to improve performance of the generated ModelJoin queries.

First, we replace the $(Layer, Node)$ pair that uniquely identifies a node in the model graph with a unique node identifier. This node ID is an incrementing integer value that is assigned by traversing the model graph, i.e., first layer of dimension n_1 has IDs 0 to $n_1 - 1$, second layer of dimension n_2 gets IDs from n_1 to $n_1 + n_2 - 1$, and so on. The artificial input node gets a node ID of -1. As a result, the storage size

of the model table decreases and the join predicate in the layer forward functions reduce from two columns to one column and a offset calculation, i.e.:

```
WHERE intermediate.node = model.node - offset
```

and the offset being a running sum over the layer dimensions. As layer dimensions are maintained by ML-To-SQL, the offset can be determined during query generation.

Second, we introduce filter predicates on the *Layer* column of the model table for each join in the layer forward function. Here, we want to traverse the model graph from the current layer to the next layer. Thus, we only need to join with tuples of that subsequent layer. Applying the filter before joining reduces the hash table size of the hash join while also enabling block pruning of the model table and therefore reducing I/O effort. The latter is achieved in X100 by the use of Small Materialized Aggregates as described in Section 2.2.1 but can also be realized by the use of any index structure in other database systems. With the optimization of a unique node ID, the filter predicate on the *Layer* column is replaced by a range predicate on the *Node* column.

X100 exploits partitioning for parallelism. Model inference is a task that is independent between tuples, as it results in a prediction for each tuple. To achieve parallelism for inference, we can partition the fact table on the unique identifier, resulting the parallel execution of the ModelJoin. The model table is shared between the execution threads. Similarly, in a distributed environment this could be achieved by replicating the model table between nodes. A unique partition key will lead to a balanced partitioning, and as the grouping key $(ID, Node)$ for summing up inputs of a node per tuple can be derived from a partitioning based on ID, no repartitioning is necessary.

Besides partitioning and parallelism, another key to achieve scalability is exploiting vectorized execution and pipelining, meaning that it is not necessary to collect a whole intermediate result at some point during query execution. The main problem with pipelining are the aggregations. However, defining a sort order on both the model table and the fact table will lead to a fully pipelined execution. The cross join as well as the join with a sorted model table is order-preserving, leading to an aggregation input flow that is sorted on the grouping keys. Thus, a hash-based aggregation can be replaced by an order-based aggregation, generating an output tuple whenever a value in the grouping key changes since it is ensured by the order that no additional tuples will occur for the group. Consequently, the aggregation does not need the full dataset, leading to a low memory footprint and pipelined execution. Additionally, the output of an order-based aggregation is still ordered, so the criterion remains valid for subsequent operations. This also holds when having a partitioned fact table, as we stated above that no sort order destroying repartitioning is necessary.

6.3.4 Native ModelJoin Operator

The ML-To-SQL framework presented in Section 6.3.3 offers portability by generating plain SQL queries to realize the ModelJoin. However, mapping calculations to generic relational operations leads to runtime overhead, e.g., by the need to realize a sum

by a hash aggregation. In this section we describe the design of a native ModelJoin database engine operator[3]. Compared to ML-To-SQL, this requires changes in the database engine, offering better performance due to the direct execution of operations needed for model inference. The operator is independent from framework-specific libraries like Tensorflow and the need for a respective C-API, but rather works on the generic model representation. Similar to ML-To-SQL this decision limits generizability to only the implemented functionalities. However, we assume to achieve better performance by keeping data in their columnar layout in CPU caches instead of converting them to the input format of an ML runtime integrated over its C-API.

The ModelJoin operator is integrated into X100. Being based on relational algebra, it does not offer support for linear algebra operations and consequently the ModelJoin can not be composed of these. The major challenge to consider in the operator design is the fact that model inference follows a row-wise access pattern. For a column store this means that conceptually each column needs to be touched once for every tuple inference, which contradicts to the cache-friendly vectorized execution model. In our operator design we carefully consider this fact by providing a vectorized model inference that works on vector of column values and performs the inference once for a set of column vectors.

Operator Design

The ModelJoin operator is based on the relational model representation described in Section 6.3.3. Conceptually, it follows a typical two-phase-join pattern shown in Figure 6.9, which is similar to, e.g., a hash join. The build phase is necessary here as we do not want to hold each model in memory but rather read it from storage (if not cached). Additionally, it is based on the Volcano iterator model as described in Section 2.2.2. While *open()* and *close()* are responsible for allocating and freeing memory, especially for weight and bias matrices of the model, *next()* triggers the execution and returns a set of vectors of input columns and additional vectors for the inference results. On the first *next()* call, the ModelJoin starts the build phase by repeatedly calling *next* on the model side until it is exhausted and performing the parallel model build. After finishing the build phase, we call *next()* on the input flow side and perform the model inference for every *next()* call on the ModelJoin operator. As the ModelJoin is designed as a regular operator, it can be used in arbitrary queries also further processing the inference result.

The operations required in the inference phase are based on the Basic Linear Algebra Subprograms (BLAS) library[24], offering the necessary matrix representations and operations. We utilize the Intel Math Kernel Library (MKL) to realize the BLAS interface. Naturally database operators are executed on the CPU. Some modern (server-) CPUs comprise more than 100 logical threads allowing massive intra- and inter-operator execution. As especially the linear algebra operations for the inference can be parallelized, the ModelJoin operator will benefit from the many cores and threads.

Although the degree of parallelization achievable with these modern CPUs is enormous, GPUs provide even more parallel units. Thus, besides the CPU variant of

[3]Available standalone under `https://github.com/dbis-ilm/ModelJoin_Operator.git`

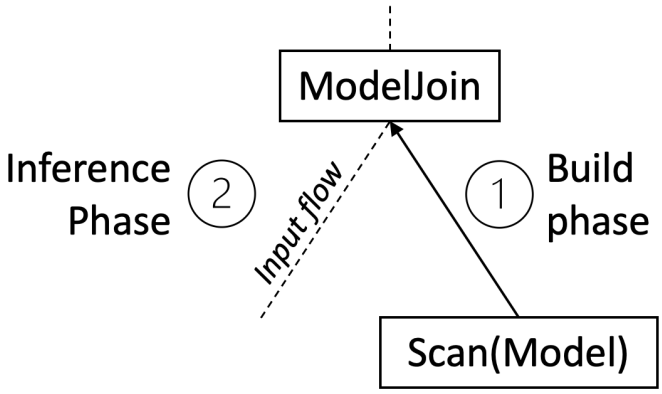


Figure 6.9: Conceptual view of the ModelJoin operator

the ModelJoin operator, the operator can also be implemented for GPUs. In order to perform the linear algebra operations on the GPU, the cuBLAS[38] library can be used. Although a GPU implementation requires to move data from the host RAM to the GPU's device memory, we expect the GPU variant to be superior to the CPU variant for large models where many matrix multiplications are required.

Parallel Model Building

Achieving parallelism in x100 is based on data partitioning, which can be done explicitly during table creation or implicitly during runtime by the scan operator that splits tables on-the-fly. Thus, we can assume that our model table is arbitrarily partitioned. Additionally, each execution thread gets a private query-plan, which leads to separate and independent physical operator instances. With our parallel model building phase we follow the aim of avoiding independent model instances for each thread, which would lead to a huge memory overhead depending on the model size. Instead, all threads build a shared model in parallel. As the actual model inference only needs read access to the model, synchronization only needs to be performed in the building phase.

Conceptually, the model building phase is equal for the CPU and GPU variant, as described below: First, memory allocation for layer weight matrices and bias vectors is performed single-threaded to a shared memory location known by all execution threads. Depending on the implementation variant, this memory is either allocated on the host memory (RAM) or device memory (GPU). Afterwards, each thread parses the relational model representation and fills the weight matrices indicated by the *Layer* column at the position indicated by the $(Node\_in, Node)$ pair, as shown in Figure 6.10. In the GPU variant, this invokes data transport to the device. As partitioning is arbitrary but distinct, it is guaranteed that there is no concurrent access to memory during this phase, making synchronization obsolete and providing true parallelism. Depending on the layer type, we have a different set of weight matrices. While we hold a single weight matrix and a bias vector for dense layers, we maintain a set of kernel weight matrices, recurrent kernel matrices and biases for each block of the LSTM layer as shown in Figure 6.6. Before the build phase can be finished and the inference phase is started, we need to introduce a single

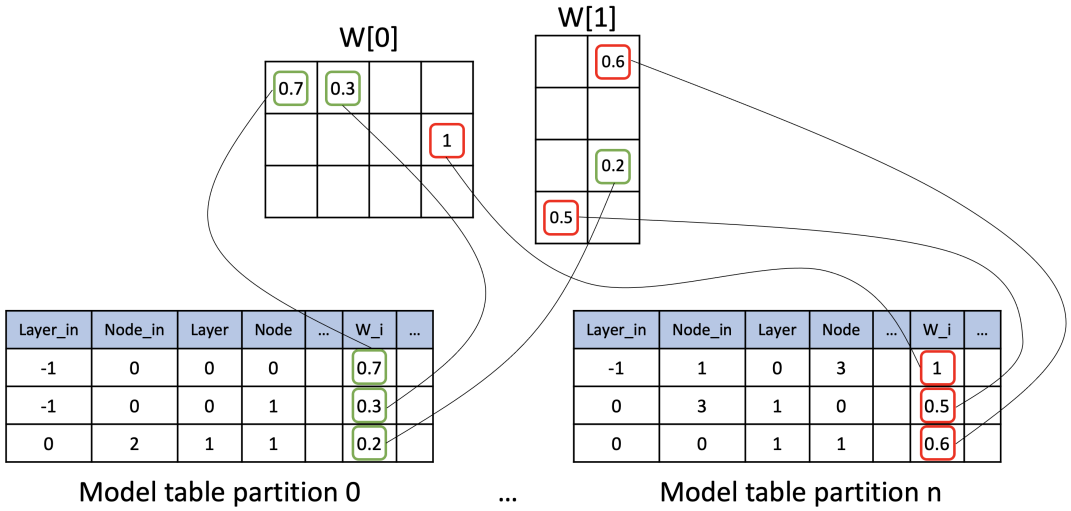


Figure 6.10: Parallel ModelJoin build phase

synchronization point to ensure that the whole model table is consumed. This is realized by a barrier before leaving the build phase.

We experienced that fine-grained data movement to GPU memory introduces a large overhead. As an optimization, we therefore always perform the parallel model build phase on the host memory and move the model to GPU memory once building is finished.

Input and Output Data Conversion

When calling *next()* on the input flow join side, we get a set of vectors with column values of the intermediate result as the input for model inference. In contrary to ML-To-SQL, we can use a subset of the input columns as prediction columns if necessary and avoid the "late projection" of payload columns. This is possible because we can leave columns untouched in a native operator introducing no overhead, while in ML-To-SQL columns need to be processed through multiple joins and aggregations. Similar to joins, prediction columns are listed in the ModelJoin call.

Before performing the model inference, we need to convert the input into a layout that fits the model. While we have a set of vectors each of length $vectorsize$, the model accepts an input vector of the length n, which is the number of input columns. This is caused by the fact that the weight matrix of the first layer in the model is of size $n \times m$ and dimensions must match to perform multiplication. In order to avoid iterating over the vectors and forming vectors of size n by copying a single value from each column vector, we allocate a matrix of size $vectorsize \times n$ (either on host or device memory) and copy the input column vectors into the matrix as shown in the first step of Figure 6.11, touching them only once. If input vectors would be placed in consecutive memory regions, one could only cast the pointer and avoid copying for the CPU variant. We do not consider this optimization as we can not guarantee this requirement.

After the model inference finished, the result matrix is broken up into vectors again and moved to the result column vectors. For p prediction columns, this matrix

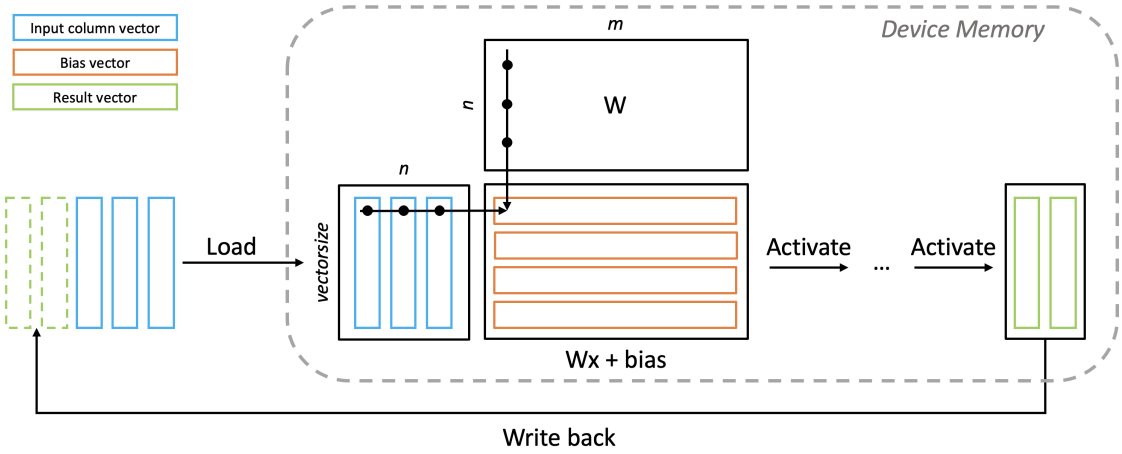


Figure 6.11: ModelJoin inference phase for dense layer network

would be of size $p \times vectorsize$, holding the p prediction results for each of the input tuples.

Vectorized Model Inference

Converting the columnar input data into an input matrix as described in the previous Section enables vectorized model inference, performing a single inference for a vector of inputs. For the inference, we now iterate over the model layers and perform the model specific layer forward function using the BLAS interface.

The dense layer forward function consists of a single matrix multiplication and bias addition. The LSTM layer forward function is again based on the Keras implementation, translated to a BLAS implementation and shown in Listing 6.6. Note that as we rely on the BLAS interface, we can find implementations of each function either in Intel MKL for the CPU variant or in cuBLAS for the GPU variant, making Listing 6.6 generically applicable for both when assuming the existence of respective memory manipulation functions. The LSTM layer forward function additionally reuses the generic activation functions that are also used for the activation of layer outputs.

One part of each layer forward function is the addition of the bias vector. Having a weight matrix of size $n \times m$, the bias vector has length m. As shown in Figure 6.11, an input matrix of size $vectorsize \times n$ would result in a intermediate result of size $vectorsize \times m$, so we would need to add the bias vector once for each of the $vectorsize$ input tuples on the intermediate result (with a respective offset). In order to avoid these fine-grained additions, we invest additional memory in the ModelJoin build phase after building finished to reallocate each bias vector once to the size $vectorsize \times m$ and replicate the vector in the matrix. With this one time effort, bias addition now incorporates a single, large addition that can be efficiently parallelized by MKL/cuBLAS. As an implementation specific detail, the BLAS matrix multiplication *sgemm* calculates $y := Ax + y$, which has two consequences. First, we can copy the bias matrix containing the repeated bias vectors once to the result matrix and get the bias addition automatically. Second, we can not perform

```
Input:  vectorsize, layer_dim, num_recurrence, data,
        weight matrices of layer (W_x, U_x, b_x)
Output: Layer output
float *h = NULL, *c = NULL; // LSTM cell states
int mat_size = vectorsize*layer_dim*sizeof(float);
float *z_x = ALLOCATE(mat_size);

for (int round = 0; round < num_recurrrence, round++) {
  COPY(z_x, bias_x);
  z_x = sger(W_x, data, z_x); // Dot product + z_x
  if (h) {
    z_x = sgemm(U_x, h, z_x); // Matrix multiplication + z_x
  }
  SIGMOID(z_i);
  SIGMOID(z_f);
  TANH(z_c);
  z_c = vsMul(z_i, z_c); // Elementwise multiplication

  if(c) {
    c = vsMul(z_f, c); // Elementwise multiplication
    c = vsAdd(z_c, c); // Elementwise addition
  } else {
    c = ALLOCATE(mat_size);
    COPY(c, z_c);
  }
  SIGMOID(z_c);

  if (!h) h = ALLOCATE(mat_size);
  COPY(h, c);
  TANH(h);
  h = vsMul(z_o, h); // Elementwise product
}
FREE(z_x);
if (c) FREE(c);

return h;
```

Listing 6.6: LSTM layer forward BLAS implementation. Lines containing an "x" are replicated for x in {i,f,c,o}

$xA = y$, but need to perform $A^T x^T = y^T$. We therefore fill the weight matrices already in a transposed way in the build phase and transpose the input matrix once before the first layer forward function. All intermediate results are then automatically transposed and we also consider this when converting the model inference result back to the column vectors. After performing a layer forward function, the layer activation function is applied on the intermediate result matrix. This can be done in parallel by, e.g., using handcrafted CUDA kernel implementations for different types of activation functions. Our implementation features a set of commonly used activation functions as CPU and GPU implementations.

Model inference is an independent operation for every input tuple. Consequently, we can again exploit partitioning to perform the inference phase of the ModelJoin in a parallel way on partitions of the input flow. Furthermore, the inference phase supports pipelined execution, which means that we return an inference result for each set of input column vectors without the need to touch the whole dataset. Thus, the ModelJoin operator is not a pipeline breaker like sorts or aggregations, leading to a low memory footprint, besides the fact that the model needs to fit into memory. Especially in the GPU variant, where we offload the execution to a different device, this maintains the cache efficiency of a vectorized execution engine. In summary, pipelining in combination with partition support leads to a highly scalable operator.

Using semantics in the table creation

Currently, calling the ModelJoin requires passing meta information about the model, i.e., the layer dimensions, the layer types and the layer activation functions. In the future, one could think about introducing semantics in the model table definition similar to dimension tables in Oracle. This way, one could fix the model table schema and maintain a model's meta information in the database catalog. Making the DBMS aware that a table is a model additionally enables custom query optimizations, sanity checks and also potential model lifetime cycle management.

6.3.5 Evaluation

We compare the approaches of the ML-To-SQL framework as a portable frontend ModelJoin solution and a native ModelJoin database operator against the baselines of using Tensorflow in the Python environment, using a Python UDF or integrating a ML runtime over native APIs. We follow the goal to show scalability for different data sizes as well as model sizes in terms of runtime and memory consumption. Please note that we forego a comparison to MADlib, as this would lead to a comparsion between DBMS performance. We investigate the performance of different general concepts while keeping the underlying database system fixed.

Setup

The experiments were run on a server consisting of an AMD EPYC 7272 2,9 GHz CPU, 512 GB RAM and a NVIDIA A100 GPU with 40 GB device memory connected over PCIe. The server runs Actian Vector 6.2, in which we integrated our ModelJoin operator implementation described in Section 6.3.4 and a Raven-like operator that

relies on the Tensorflow C-API. We evaluate the approaches on a low abstraction level, i.e., we evaluate performance for different shapes of models instead of focussing on use cases. Consequently, we evaluate the building blocks to build different applications like classifications, regressions and similar.

We designed separate experiments for dense layer networks and LSTM networks. The dense layer experiment is based on the Iris dataset [49] that is replicated to mimic varying fact table sizes. The dataset consists of four feature columns that are used to predict a class attribute and is a commonly used real-world example for machine learning. In order to evaluate the scalability for different ML model sizes, we use dense layer networks with all combinations of $model\_widths \in \{32, 128, 512\}$ and $model\_depths \in \{2, 4, 8\}$, i.e., a model of width 128 and depth 4 has 4 dense layers of width 32 and an output layer of size 1. For the LSTM layer experiment we generated a time series based on a sinus function and used 3 time steps for each forecast. In our evaluation we focus on prediction runtime, which is independent from the actual mathematical function the neural network represents. Consequently, a generated sinus function leads to the same runtime results as real-world examples, but is easier understandable and reproducible. As typically a single LSTM layer is used, we do not use different $model\_depths$ in this experiment, but varied the LSTM layer width, followed by a single neuron output layer. For all experiments the batch size is equal to the database engine's vector size of 1024. Tables are partitioned into 12 partitions and the engine runs with a parallelism level of 12.

Based on these setups we measured the runtime to apply the respective model on a varying number of tuples. We thereby compare our approach of a native ModelJoin operator, both as a CPU and GPU variant, and the ML-To-SQL framework against Tensorflow running either in Python, integrated over the C-API both on CPU and GPU or by using a Python UDF. For Tensorflow in Python, data is moved from the database to the Python environment using ODBC and classified using Tensorflow. Here measurements include data movement and classification runtime. Using the C-API, data does not need to be moved, but converted to the expected input format of the Tensorflow API. This requires moving data from a columnar format into a row-major matrix, and results back to columnar layout. In the Python UDF, we load the saved model, apply it to the data using Tensorflow on the CPU and return the predictions. Additionally, we optimize the UDF by using Actian Vector's parallel and vectorized UDFs as described in Section 6.2, i.e., calling the UDF once per vector instead of once per tuple. In order to provide a fair comparison, we assume the model to be already loaded, which is automatically done for ML-To-SQL and the ModelJoin operator by storing the model as a table. For Tensorflow, we start timing the inference runtime after the model was loaded from disk and for UDF we use a cached model realized by storing a loaded model into a global state. Note that in the experiments we do not examine typical ML metrics like model precision, but purely focus on prediction runtime. We use the same model for each implementation variant and ensure consistent results.

Results

Model inference runtimes The model inference runtimes for the dense experiment are shown in Figure 6.12. As expected, the ML-To-SQL variant scales worse

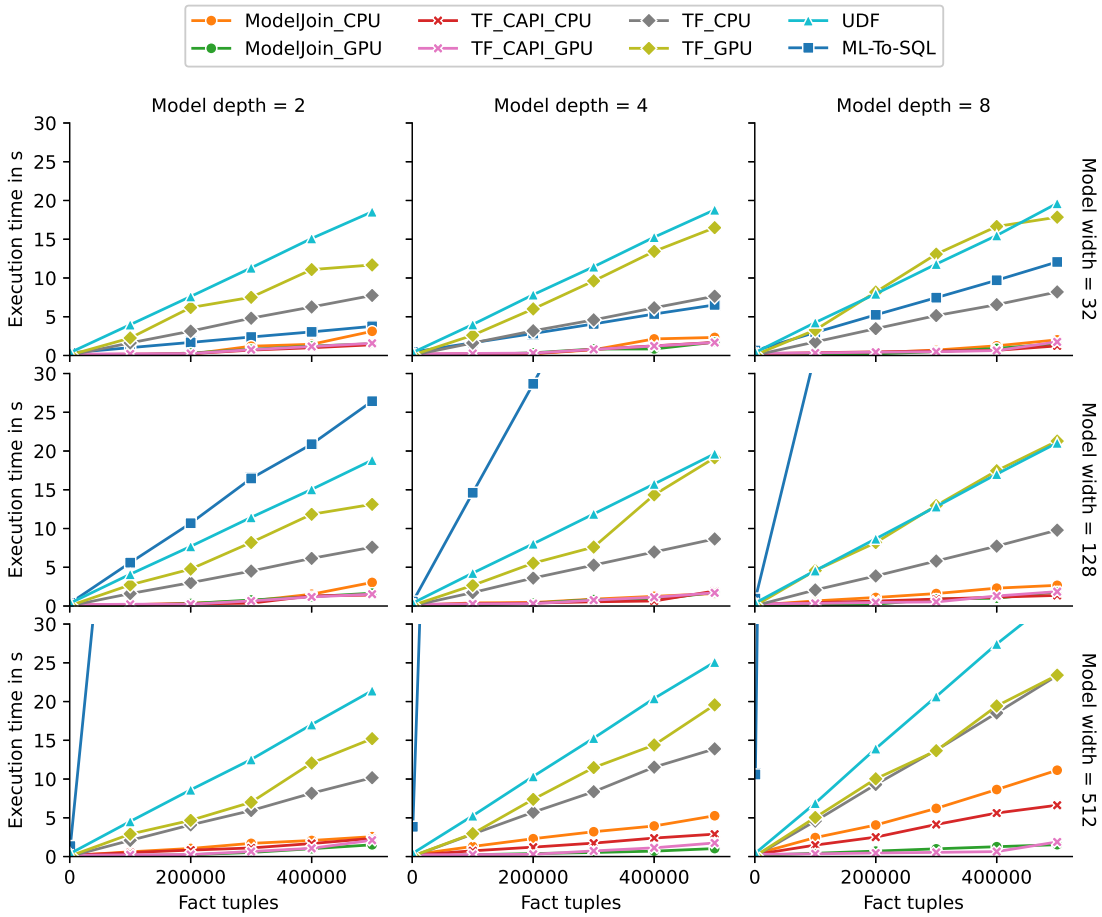


Figure 6.12: Runtime results for dense layer networks

than the other approaches as it uses generic constructs query operators for very specific computations. Using the native C-API of Tensorflow is the best alternative in terms of runtime and scalability for all evaluated cases, being either on-par with the native ModelJoin operator in the GPU variant or even better in the CPU variant. Consequently having a native integration of the ModelJoin operator does not bring any benefits compared to integrating Tensorflow over the C-API. Both approaches are about an order of magnitude faster than the Tensorflow variant in Python and the UDF variant. Comparing the latter two, it shows that the native Python variant using Tensorflow is slightly better than the UDF variant, which is caused by context switches between the database engine and the Python environment as well as data conversions and data transport between the engine and the Python environment. However, this is also an effect of the used queries, as only performing the ModelJoin is in favor of the Tensorflow variant. In more complex analytic pipelines, the inference results are typically further processed by, e.g., aggregations. As using UDFs data remains inside the database kernel, these further operations are significantly faster inside the DBMS compared to Python. Tensorflow in Python mainly suffers from the overhead of data transport over ODBC. As Tensorflow inference takes place on the same machine that runs the database instance, the setup is still in favor of Tensorflow by minimizing data transport costs. Moving large datasets from

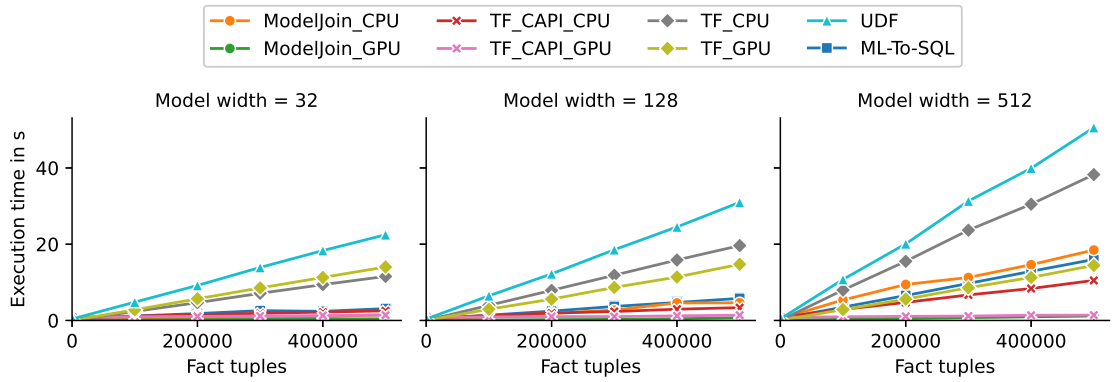


Figure 6.13: Runtime results for LSTM layer networks

a database server to a separate machine for running the inference would further decrease the performance of the Tensorflow variant. For large models we can observe that GPU variants perform at least equal (for Tensorflow in Python) or even better (for ModelJoin and Tensorflow using the C-API) than their respective CPU implementations. Comparing different model sizes, inference runtime slightly increases in both directions of model width and model depth. However, the dominating part of the execution time for the small models is data transport, so inference runtime does not double when doubling the model size. The number of model parameters does not scale linearly but quadratically, i.e., the model with width 512 and depth 8 having $4 \cdot 512 + 7 \cdot 512^2 + 512 \approx 1,8 \cdot 10^6$ parameters, while a model of same depth but 128 neurons per layer only has around 115.000 parameters. Consequently, the size of intermediate results in the ML-To-SQL variant also increases quadratically, leading to bad scalability.

The inference runtimes for the LSTM experiment are shown in Figure 6.13. Compared to the dense experiment, increasing the width of the LSTM layer has a higher impact on runtime for all ModelJoin alternatives, as the computation of a LSTM layer is more complex than a dense layer. Using the ML-To-SQL framework for performing the ModelJoin performs significantly better in this case compared to the dense experiment due to only a single layer being processed, which leads to significantly smaller intermediate results. However, the ML-To-SQL variant scales worse than the other alternatives with increasing model size, caused by the quadratic increase in intermediate result size. Again, the native ModelJoin operator and using Tensorflow over the C-API show the best runtimes and scalability both comparing CPU or GPU variants, with the GPU variant being the better choice due to the increased complexity of the LSTM layer compared to the dense layer.

For both experiments we can observe that the size of the fact table has the highest impact on the model inference runtime. For use cases where the fact table is large, a native integration either using a native ModelJoin operator or a ML runtime's API is the best choice, with the GPU implementations showing better scalability and should therefore be used whenever possible. Especially in cloud environments instance types can almost arbitrarily be chosen, so we argue that the existence of a GPU on a database deployment can be easily realized.

Table 6.2: Peak memory for model inference of 100K tuples

Model	ModelJoin	TF(C-API)	TF(Python)	ML-To-SQL
Dense(32,4)	109.5 MB	153.9 MB	2.8 GB	1.4 GB
Dense(128,4)	121.7 MB	154.8 MB	3.4 GB	2.4 GB
Dense(512,4)	194.9 MB	185.4 MB	10.5 GB	6.6 GB
LSTM(128)	123.5 MB	192.2 MB	3.4 GB	411 MB

Memory usage Besides performance, we compared the approaches in terms of memory consumption. We measured peak memory of the database engine for the ModelJoin operator, the Tensorflow C-API approach and ML-To-SQL while measuring peak memory of the Python process for Tensorflow using Python. This way we captured the model memory itself and the size of intermediate results. The results for a representative subset of the models and a fact table of 100K tuples are shown in Table 6.2. Due to the models themselves being quite small in size and being shared by all threads, the ModelJoin operator has a very low memory consumption. Tensorflow using the C-API has a similar memory consumption with a slightly higher fixed memory. Tensorflow in Python shows a significantly higher memory usage due to the Python environment and parallelism. ML-To-SQL's memory consumption is above the ModelJoin operator, caused by the use of more generic operators. However, it is below Tensorflow(Python)'s memory consumption due to the pipelined execution described in Section 6.3.3. For the LSTM experiment, memory consumption is only slightly higher than the ModelJoin operator, again caused by more generic operators. It is also lower than in the dense experiments, as the model only consists of a single layer whose computation is however more complex. The memory consumption of the UDF variant is not included in Table 6.2, as it can be seen as a wrapper around the Tensorflow variant and therefore has similar memory requirements. During the design of the ModelJoin operator's GPU implementation we assumed that the model fits into GPU memory. Comparing the memory consumption of the ModelJoin operator from Table 6.2 to the device memory of our GPU (40 GB) or similar devices, we argue that this is no limitation for typical model sizes.

Resume We evaluated model inference using candidates of the classes of approaches introduced in Section 6.3.1 for models of different widths and depths. A qualitative comparison of the results is given in Table 6.3. We observed that the native integration using a specialized ModelJoin operator or using the native API of ML runtimes significantly outperforms the alternatives for all models and data sizes by an order of magnitude and has a low memory footprint. Enabling the GPU variant further leads to better scalability, which is especially useful for large models or large fact tables and should be used if respective hardware is available. ML-To-SQL showed to scale worse, but is still a reasonable choice for small models or small data, e.g., to classify periodical inserts or IoT data, as here the advantages of a portable solution predominate the scalability issues. ML-To-SQL can be ported to any SQL-compliant database system without requiring any changes in the engine code, neither any UDF support or a ML runtime, but uses the existing capabilities of database engines for

Table 6.3: Qualitative comparison of ML inference approaches

	ML-To-SQL	Native ModelJoin	TF (Python)	TF (C-API)	UDF
Performance (Small Models)	Good	Good	Medium	Good	Medium
Performance (Large Models)	Bad	Good	Bad	Good	Bad
Memory Consumption	Medium	Good	Bad	Good	Bad
Portability	Good	Bad	Good	Bad	Medium
Generalizability	Bad	Bad	Good	Good	Good

efficient query processing. Comparing the native integration approaches, the native operator performes nearly as good as the integration of a highly specialized ML runtime using the C-API.

Coming back to the quote of [166] questioned in Section 6.3.1, reimplementing ML algorithms is consequently not reasonable and the drawbacks of limited generalizability to only the reimplemented layer types dominate.

6.4 Conclusion

Based on our findings with the Grizzly frontend solution to push modern analytics towards the database system in Section 5.5 we explored how to integrate Python UDFs and Machine Learning model inference deeper into the database engine, with the goal of achieving better performance.

User-defined Python functions are easy-to-use extensions to database engines and provide support for modern analytical tasks. Motivated by the fact that the execution of Python code suffers from bad performance and scalability we explored the design space of accelerating Python UDFs in vectorized query engines. We evaluated the impact of vectorization, parallelisation and compilation, with the latter being realized by different existing compilation frameworks. We show how UDFs can be compiled just-in-time during database runtime, dynamically loaded and used in query execution and implemented this feature as an open-source prototype engine. Furthermore we presented an approach to mitigate Python's GIL issue using separate UDF engine processes to enable parallel UDF execution. In our evaluation we integrated the Cython compilation framework into the X100 query engine and evaluated end-to-end query performance for representative use cases. We thereby achieved significant speedups for different use cases with compilation and parallelisation used together. With transparency as a main design goal, the presented approaches are easy-to-use and work for arbitrary Python code constructs.

Pushing ML model inference into database systems offers great possibilities for performance, scalability and data privacy. We evaluated different ways to exploit the capabilities of database engines for ML model inference and focused on neural networks as a subclass of ML models. Based on a generic relational model representation that is able to hold dense layers as well as LSTM layers we first described the design of the ML-To-SQL framework. ML-To-SQL offers an easy-to-use API to transform ML models into tables and to perform the model inference using standard SQL. As the opposite direction we discussed the design

of a native ModelJoin database operator, which offers a parallel model building phase and a vectorized inference phase. The ModelJoin operator is realized as a CPU and a GPU variant. Our evaluation showed that ML-To-SQL is a reasonable choice for small data sizes or small model sizes, but shows poor scalability as the price for using generic database operators and therefore being highly portable. The native ModelJoin operator integrated into the X100 engine however outperforms the alternatives of using Python/Tensorflow or a Python UDF by an order of magnitude and shows a significantly lower memory footprint for different model and data sizes. The GPU variant further increases performance and scalability and is therefore the best choice if respective hardware is available. However, it did not outperform a native operator built on the C-API of existing ML runtimes both in CPU and GPU variants. Consequently, reimplementing functionality for ML inference does not amortize for the lack in generalizability and we conclude that using native APIs to integrate existing ML runtimes is the best option for more native ML support.

Chapter 7
Conclusion and Open Research Questions

7.1 Conclusion

Cloud computing is one of the most successful technologies of the last decade and will be an important, business-driving technology in the future. It significantly changes the way developers and users design, deploy and use systems. In this thesis, we focused on analytical database systems in the cloud and started by identifying three interaction points of database systems with their environment that show a shift in requirements compared to traditional, on-premise data warehouse solutions: interaction with the elastic hardware environment, interaction with various endpoints for data ingestion and interaction with data science and machine learning (ML) environments for modern workloads. Based on these, this thesis provides three major contributions.

Elastic scaling We presented an elastic scaling feature that allows a distributed database engine built on a shared-nothing architecture to scale during system runtime, thereby avoiding a system restart, a long running log replay and consequently a potentially long offline time. The approach relies on the MPI group and communicator management. As load distribution in these kind of systems is typically based on data partitioning, we introduce a partition manager that supports elastic scaling. The partition manager explicitly handles partition assignments for equivalence classes of partitionings, which allows to minimize partition reassignment in the case of scaling while still providing load balancing and preserving the co-location of partitions for efficient partitioned query execution. In order to mitigate a cold start problem caused by empty buffers after scaling we introduce the buffer matching and filling mechanism, which makes use of buffer eviction priorities to find suitable datasets to pre-heat buffers and benefit from an immediate performance gain. The evaluation showed that the elastic scaling method enables fast scaling within a few seconds on modern cloud hardware.

PatchIndexes to increase schema quality A cloud database is highly available and accessible from nearly everywhere, so data can be ingested from many devices. Additionally, data freshness for real-time analytics is of major importance. Supporting frequent data ingestions from many endpoints fundamentally differs from bulk loads or ETL processes in traditional on-premise data warehouse solutions. In many cases users avoid defining database constraints to prevent aborts due to constraint conflicts or to accelerate ingestion by skipping expensive constraint checks. We introduce the PatchIndex concept that allows the definition of approximate constraints in database systems, i.e., constraints that hold for all tuples except a set

of exceptions. The PatchIndex maintains these exceptions in a sharded bitmap data structure, which offers efficient scan and update support. The information about approximate constraints can be used in query optimization and execution. Typically the datastream is splitted into data fulfilling a constraint and exceptions, and expensive operators can be skipped on the first one. While PatchIndexes are designed as a generic data structure that can be used for arbitrary constraints by realizing a simple interface, we showed how to apply the approach to traditional constraints with approximate sorting and approximate uniqueness. The evaluation showed that we can achieve similar performance as traditional materialization approaches in sort queries and distinct queries even for large exception rates, while PatchIndexes can be updated significantly more efficiently. As a second example we showed how PatchIndexes can be used to achieve a multi-key data partitioning, i.e., finding a data partitioning where the constraint that equal values are assigned to the same partition holds for multiple partitioning keys. With PatchIndexes it is possible to relax this problem to a patched multi-key partitioning. We show that this problem can be mapped to a graph partitioning problem and present a workflow to find a solution using arbitrary graph partitioning algorithms. The evaluation here showed that we can achieve robust query performance for workloads that require multiple different partition keys without replicating data or repartitioning full tables during query execution.

In-database support for modern analytics In parallel to the success of cloud computing, the shape of typical analytic workload changed in the recent years. While traditional SQL based workloads used to be the common case for business intelligence and had been researched over decades to run efficiently, data science and machine learning workloads gained significant importance. These workloads often run in Python in different environments like Python Pandas for data processing or Tensorflow for machine learning. As database systems do not support these workloads data is simply downloaded from the database to run the workloads outside of the database engine, wasting its great capabilities for efficient query processing. As this has major drawbacks we aimed at pushing modern workloads towards the database engine. We started with introducing the Grizzly framework, a DataFrame-to-SQL transpiler that is intended to work as a drop-in replacement for Pandas. Instead of eagerly executing operations, the framework lazily collects DataFrame operations in a query tree and transpiles it to a SQL query when a result is needed. This way it moved the query to where the data resides to exploit the database engine for efficient processing, leading to a speedup of several orders of magnitude compared to Pandas. Grizzly can work as a totally different frontend to the database system, enabling a whole new user group that is used to data processing in Python to use database systems. We extended Grizzly with API extensions for external file support, user-defined functions (UDFs) and ML model inference. Based on this we identified Python UDFs and ML model inference as typical tasks that can greatly benefit from a deeper database engine integration. We therefore investigated opportunities to integrate Python UDFs optimized performance using just-in-time compilation of UDF code, vectorization and parallelization. Furthermore we compared different approaches to push ML model inference towards the engine, starting with a naive

Python UDF approach, followed by a mapping of model inference to SQL, a native integration using Tensorflow's C-API or a native integration using a designated ModelJoin database operator. The evaluation showed that while a SQL mapping is very portable and can be useful for small models or small data, a native integration using the runtime's C-API is the most efficient and generic solution.

7.2 Open Research Questions

In this thesis we overall introduced building blocks for a database engine to more natively support the challenges and opportunities the cloud environment offers. In this environment we believe that a database system should mature towards a data platform that is most importantly easy-to-use, flexible, self-managing as much as possible, feature-rich and offers support for many use cases in an integrated way. While engine performance has been the major differentiator over the last decades, we believe that user experience becomes even more important for the success of a database system. Based on our contributions we propose the following open questions for the future that would help to come closer to this vision.

Elasticity Our proposed scaling mechanism assumes that the system can be on hold for a short amount of time in order to scale up. As an improvement online scaling could at least allow read queries during the scaling process. The logical next step however would be a more flexible serverless approach (similar to a Function as a Service approach), i.e., managing a pool of stateless workers that can execute arbitrary (sub-)queries. While this leads to more elasticity and reduces the effort to start and configure a compute cluster, it also introduces new challenges like isolation in a multi-tenant environment. Furthermore it requires to separate state from the workers. Metadata and catalog information need to be stored separately in a scalable service to be queried by workers on demand. For both approaches, a stateful online scalable system or a serverless engine, system monitoring is a key to automatically scale the system in order to match the workload demand in the most cost-effective way. Several workload-driven scaling approaches have been proposed like [134] that can be realized on top of a system monitoring component. At the time of writing this thesis such a system monitoring feature is under development at Actian to be integrated with Actian Avalanche.

PatchIndexes PatchIndexes offer the possibility to maintain and benefit from constraints even if these constraints do not hold for all tuples. To reach the goal of a self-managing database engine, columns that could benefit from PatchIndexes should be automatically discovered. In a workload-driven approach possible candidates are columns that are often queried in a certain manner, e.g., aggregated when using a uniqueness constraint or sorted when using a sorting constraint. Runtime metrics like the number of aggregation groups can give indications on the expected exception rates. Additionally, PatchIndexes are designed in a generic way to be applied to arbitrary constraints. More candidates for constraints could be evaluated and even user-defined constraints could be made available.

DataFrame Frontend - Grizzly Our Grizzly framework offers a new way to use a database system from the Python environment. While Grizzly works easily with Jupyter notebooks, the most common way to work with Python nowadays, this setup offers room for improvement. In Jupyter execution can be done cell-by-cell in a stepwise manner. This can be exploited in Grizzly by materializing intermediate results as, e.g., materialized views to be reused when executing code cells again. Moreover, a logical next step for Grizzly is the support for heterogeneous data sources, i.e., holding connections to multiple databases. In the query graphs that Grizzly is based on join operations between different database instances can then be seen as "pipeline breakers", where data needs to be fetched from both join sides and combined locally. Using a query optimizer, computation effort needs to be pushed into the database instances as much as possible to reduce intermediate result sizes, transfer costs and the client-side processing costs. In order to combine intermediate results locally Grizzly could be extended with an embedded processing engine like DuckDB [128] or MonetDBLite [127]. With this, Grizzly could work as a frontend for hybrid cloud setups, e.g., combining data from a public cloud instance and a private instance.

Modern workloads We presented approaches to push Python UDFs and ML model inference towards the database engine. Estimating execution costs for these operations is crucial for query optimization and achieving robust and efficient query plans. These costs are impacted by many properties that are hard to quantify, particularly because Python UDF code can be of arbitrary shape. Such a cost estimator could be constructed out of explicit costs of using a learning-based approach. To the best of our knowledge related work about integrating Python UDFs or ML workloads has not covered this topic yet. In addition to that, support for data preprocessing is needed in the desired feature-rich data platform in order to support the construction and execution of whole pipelines. These preprocessing steps include feature encoding, scaling or normalization and are already investigated in MADlib [69] or in [137]. Last, we waived the topic of ML model training in our investigations. While there are several works on in-database model training, we believe that the update-heavy training process is conceptually not a very good fit for a database engine optimized for read performance on large data. Additionally, parallelizing the training in a distributed database engine is challenging, as multiple model instances are trained simultaneously and independently. Merging these models is a difficult problem and synchronizing the parallel training would significantly reduce parallelism. For feature encoding this might even lead to inconsistent results, e.g., if similar values are differently mapped by independent dictionary instances.

Bibliography

[1] Daniel Abadi et al. “The Design and Implementation of Modern Column-Oriented Database Systems”. In: *Found. Trends Databases* 5.3 (2013), pp. 197–280.

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. “Profiling relational data: a survey”. In: *VLDB J.* 24.4 (2015), pp. 557–581.

[3] Christopher R. Aberger et al. “LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying”. In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 449–460.

[4] Wilfried Yves Hamilton Adoni et al. “A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems”. In: *Distributed Parallel Databases* 38.2 (2020), pp. 495–530.

[5] Divyakant Agrawal et al. “Database Scalability, Elasticity, and Autonomy in the Cloud - (Extended Abstract)”. In: *Database Systems for Advanced Applications - 16th International Conference, DASFAA 2011, Hong Kong, China, April 22-25, 2011, Proceedings, Part I*. Ed. by Jeffrey Xu Yu, Myoung-Ho Kim, and Rainer Unland. Vol. 6587. Lecture Notes in Computer Science. Springer, 2011, pp. 2–15.

[6] Anastasia Ailamaki, David J. DeWitt, and Mark D. Hill. “Data page layouts for relational databases on deep memory hierarchies”. In: *VLDB J.* 11.3 (2002), pp. 198–215.

[7] Vladimir E. Alekseev et al. “NP-hard graph problems and boundary classes of graphs”. In: *Theor. Comput. Sci.* 389.1-2 (2007), pp. 219–236.

[8] Amazon. *Amazon Redshift ML*. `https://aws.amazon.com/de/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml/`. [Online; accessed 03-February-2022]. 2022.

[9] *Apache Hadoop*. en. `https://hadoop.apache.org/`. [Online; accessed 21-November-2022].

[10] *Apache Mahout*. `https://mahout.apache.org/`. [Online; accessed 21-February-2022]. 2022.

[11] *Apache Spark - Unified Engine for large-scale data analytics*. en. `https://spark.apache.org/`. [Online; accessed 21-November-2022].

[12] *Apache SystemDS*. `http://systemds.apache.org/`. [Online; accessed 21-February-2022]. 2022.

[13] Nikos Armenatzoglou et al. "Amazon Redshift Re-invented". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 2205–2217.

[14] Manos Athanassoulis, Zheng Yan, and Stratos Idreos. "UpBit: Scalable In-Memory Updatable Bitmap Indexing". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 1319–1332.

[15] *Azure Synapse Analytics*. en. `https://azure.microsoft.com/en-us/products/synapse-analytics/`. [Online; accessed 10-October-2022].

[16] Brian Babcock and Surajit Chaudhuri. "Towards a Robust Query Optimizer: A Principled and Practical Approach". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. Ed. by Fatma Özcan. ACM, 2005, pp. 119–130.

[17] Stephan Baumann, Peter A. Boncz, and Kai-Uwe Sattler. "Bitwise dimensional co-clustering for analytical workloads". In: *VLDB J.* 25.3 (2016), pp. 291–316.

[18] Stefan Behnel et al. "Cython: The Best of Both Worlds". In: *Comput. Sci. Eng.* 13.2 (2011), pp. 31–39.

[19] Laszlo A. Belady. "A Study of Replacement Algorithms for Virtual-Storage Computer". In: *IBM Syst. J.* 5.2 (1966), pp. 78–101.

[20] Bishwaranjan Bhattacharjee et al. "Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2". In: *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*. Ed. by Johann Christoph Freytag et al. Morgan Kaufmann, 2003, pp. 963–974.

[21] José A. Blakeley, Neil Coburn, and Per-Åke Larson. "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates". In: *ACM Trans. Database Syst.* 14.3 (1989), pp. 369–400.

[22] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. "Efficiently Updating Materialized Views". In: (1986). Ed. by Carlo Zaniolo, pp. 61–71.

[23] José A. Blakeley and Nancy L. Martin. "Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis". In: *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*. IEEE Computer Society, 1990, pp. 256–263.

[24] *BLAS (Basic Linear Algebra Subprograms)*. `http://www.netlib.org/blas/`. [Online; accessed 02-December-2022].

[25] Matthias Boehm et al. "SystemML: Declarative Machine Learning on Spark". In: *Proc. VLDB Endow.* 9.13 (2016), pp. 1425–1436.

[26] Robert B. Bohn et al. “NIST Cloud Computing Reference Architecture”. In: *World Congress on Services, SERVICES 2011, Washington, DC, USA, July 4-9, 2011*. IEEE Computer Society, 2011, pp. 594–596.

[27] Peter A. Boncz, Thomas Neumann, and Orri Erling. “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”. In: *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*. Ed. by Raghunath Nambiar and Meikel Poess. Vol. 8391. Lecture Notes in Computer Science. Springer, 2013, pp. 61–76.

[28] Peter A. Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution”. In: *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 225–237.

[29] Aydin Buluç et al. “Recent Advances in Graph Partitioning”. In: *Algorithm Engineering - Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Vol. 9220. Lecture Notes in Computer Science. 2016, pp. 117–158.

[30] Francesco Del Buono et al. “Transforming ML Predictive Pipelines into SQL with MASQ”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li et al. ACM, 2021, pp. 2696–2700.

[31] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. “Serializable isolation for snapshot databases”. In: *ACM Trans. Database Syst.* 34.4 (2009), 20:1–20:42.

[32] Guadalupe Canahuate, Michael Gibas, and Hakan Ferhatosmanoglu. “Update Conscious Bitmap Indices”. In: *19th International Conference on Scientific and Statistical Database Management, SSDBM 2007, 9-11 July 2007, Banff, Canada, Proceedings*. IEEE Computer Society, 2007, p. 15.

[33] Kyunghyun Cho et al. “On the Properties of Neural Machine Translation: Encoder-Decoder Approaches”. In: *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014*. Ed. by Dekai Wu et al. Association for Computational Linguistics, 2014, pp. 103–111.

[34] *Cloud Computing Market Size Report, 2022-2030*. en. `https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry`. [Online; accessed 10-October-2022].

[35] Andrei Costea and Adrian Ionescu. *Query Optimization and Execution in Vectorwise MPP*. en. master thesis.

[36] Andrei Costea et al. “VectorH: Taking SQL-on-Hadoop to the Next Level”. In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 1105–1117.

[37] Harald Cramér. *Mathematical Methods of Statistics*. Princeton University Press, 1946. ISBN: 9780691080048.

[38] *cuBLAS*. `https://developer.nvidia.com/cublas`. [Online; accessed 02-December-2022].

[39] Carlo Curino et al. "Schism: a Workload-Driven Approach to Database Replication and Partitioning". In: *Proc. VLDB Endow.* 3.1 (2010), pp. 48–57.

[40] Joseph Vinish D'silva, Florestan De Moor, and Bettina Kemme. "AIDA - Abstraction for Advanced In-Database Analytics". In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1400–1413.

[41] Benoit Dageville et al. "The Snowflake Elastic Data Warehouse". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 215–226.

[42] *Dask*. en. `https://www.dask.org/`. [Online; accessed 17-October-2022].

[43] Len Du. "In-Machine-Learning Database: Reimagining Deep Learning with Old-School SQL". In: *CoRR* abs/2004.05366 (2020).

[44] Christian Duta, Denis Hirn, and Torsten Grust. "Compiling PL/SQL Away". In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[45] Wenfei Fan et al. "Incrementalization of Graph Partitioning Algorithms". In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1261–1274.

[46] Arash Fard et al. "Vertica-ML: Distributed Machine Learning in Vertica Database". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020, pp. 755–768.

[47] Xixuan Feng et al. "Towards a unified architecture for in-RDBMS analytics". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan et al. ACM, 2012, pp. 325–336.

[48] Tim Fischer, Denis Hirn, and Torsten Grust. "Snakes on a Plan: Compiling Python Functions into Plain SQL Queries". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 2389–2392.

[49] Rory A. Fisher. "THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS". In: *Annals of Human Genetics* 7 (1936), pp. 179–188.

[50] Michael L. Fredman. "On computing the length of longest increasing subsequences". In: *Discret. Math.* 11.1 (1975), pp. 29–35.

[51] Carlos Garcia-Alvarado et al. "Automatic Data Placement in MPP Databases". In: *Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012*. Ed. by Anastasios Kementsietsidis and Marcos Antonio Vaz Salles. IEEE Computer Society, 2012, pp. 322–327.

[52] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. “White-box Compression: Learning and Exploiting Compact Table Representations”. In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[53] *Google BigQuery ML*. `https://cloud.google.com/bigquery-ml/docs`. [Online; accessed 03-February-2022]. 2022.

[54] *Google BigQuery ML Tensorflow support*. `https://cloud.google.com/bigquery-ml/docs`. [Online; accessed 03-February-2022]. 2022.

[55] *GraalVM*. `https://www.graalvm.org/`. [Online; accessed 02-December-2022].

[56] Goetz Graefe. “Volcano - An Extensible and Parallel Query Evaluation System”. In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135.

[57] Goetz Graefe et al. “Robust Query Processing (Dagstuhl Seminar 12321)”. In: *Dagstuhl Reports* 2.8 (2012), pp. 1–15.

[58] Alex Guazzelli et al. “PMML: An Open Standard for Sharing Models”. In: *R J.* 1.1 (2009), p. 60.

[59] Alessio Guerrieri. “Distributed Computing for Large-scale Graphs”. In: (2015).

[60] Anurag Gupta et al. “Amazon Redshift and the Case for Simpler Data Warehouses”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1917–1923.

[61] Ashish Gupta and José A. Blakeley. “Using Partial Information to Update Materialized Views”. In: *Inf. Syst.* 20.8 (1995), pp. 641–662.

[62] Surabhi Gupta and Karthik Ramachandra. “Procedural Extensions of SQL: Understanding their usage in the wild”. In: *Proc. VLDB Endow.* 14.8 (2021), pp. 1378–1391.

[63] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5 (1988), pp. 532–533.

[64] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. “Conquering a Panda’s weaker self - Fighting laziness with laziness”. In: *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*. Ed. by Yannis Velegrakis et al. OpenProceedings.org, 2021, pp. 670–673.

[65] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. “Putting Pandas in a Box”. In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[66] Stefan Hagedorn and Kai-Uwe Sattler. “Cost-Based Sharing and Recycling of (Intermediate) Results in Dataflow Programs”. In: *Advances in Databases and Information Systems - 22nd European Conference, ADBIS 2018, Budapest, Hungary, September 2-5, 2018, Proceedings*. Ed. by András Benczúr, Bernhard Thalheim, and Tomás Horváth. Vol. 11019. Lecture Notes in Computer Science. Springer, 2018, pp. 185–199.

[67] Masatoshi Hanai et al. “Distributed Edge Partitioning for Trillion-edge Graphs”. In: *Proc. VLDB Endow.* 12.13 (2019), pp. 2379–2392.

[68] Arvid Heise et al. “Scalable Discovery of Unique Column Combinations”. In: *Proc. VLDB Endow.* 7.4 (2013), pp. 301–312.

[69] Joseph M. Hellerstein et al. “The MADlib Analytics Library or MAD Skills, the SQL”. In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1700–1711.

[70] Sándor Héman et al. “Positional update handling in column stores”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 543–554.

[71] Nikolas Roman Herbst, Samuel Kounev, and Ralf H. Reussner. “Elasticity in Cloud Computing: What It Is, and What It Is Not”. In: (2013). Ed. by Jeffrey O. Kephart, Calton Pu, and Xiaoyun Zhu, pp. 23–27.

[72] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. “Learning a Partitioning Advisor for Cloud Databases”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier et al. ACM, 2020, pp. 143–157.

[73] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (1997), pp. 1735–1780.

[74] *Ibis Project*. en. `https://ibis-project.org/docs/3.2.0/`. [Online; accessed 17-October-2022].

[75] Doug Inkster, Marcin Zukowski, and Peter A. Boncz. “Integration of vectorwise with ingres”. In: *SIGMOD Rec.* 40.3 (2011), pp. 45–53.

[76] Konstantinos Karanasos et al. “Extending Relational Query Processing with ML Inference”. In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[77] Manos Karpathiotakis et al. “Adaptive Query Processing on RAW Data”. In: *Proc. VLDB Endow.* 7.12 (2014), pp. 1119–1130.

[78] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM J. Sci. Comput.* 20.1 (1998), pp. 359–392.

[79] Alfons Kemper and Thomas Neumann. “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots”. In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. Ed. by Serge Abiteboul et al. IEEE Computer Society, 2011, pp. 195–206.

[80] Steffen Kläbe and Stefan Hagedorn. “Applying Machine Learning Models to Scalable DataFrames with Grizzly”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme” (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. Ed. by Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 195–214.

[81] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. “Exploration of Approaches for In-Database ML”. In: *Proceedings of the 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28 - March 31, 2023*. To appear. 2023.

[82] Steffen Kläbe and Kai-Uwe Sattler. “Patched Multi-Key Partitioning for Robust Query Performance”. In: *Proceedings of the 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28 - March 31, 2023*. To appear. 2023.

[83] Steffen Kläbe, Kai-Uwe Sattler, and Stephan Baumann. “PatchIndex - Exploiting Approximate Constraints in Self-managing Databases”. In: *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 139–146.

[84] Steffen Kläbe, Kai-Uwe Sattler, and Stephan Baumann. “PatchIndex: exploiting approximate constraints in distributed databases”. In: *Distributed Parallel Databases* 39.3 (2021), pp. 833–853.

[85] Steffen Kläbe, Kai-Uwe Sattler, and Stephan Baumann. “Updatable Materialization of Approximate Constraints”. In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 1991–1996.

[86] Steffen Kläbe, Kai-Uwe Sattler, and Stephan Baumann. “Updatable Materialization of Approximate Constraints”. In: *CoRR* abs/2102.06557 (2021).

[87] Steffen Kläbe et al. “Accelerating Python UDFs in Vectorized Query Execution”. In: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[88] Steffen Kläbe et al. “Elastic Scaling in VectorH”. In: *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*. Ed. by Angela Bonifati et al. OpenProceedings.org, 2020, pp. 498–509.

[89] *Koalas*. en. `https://github.com/databricks/koalas`. [Online; accessed 17-October-2022].

[90] Henning Köhler, Sebastian Link, and Xiaofang Zhou. "Possible and Certain SQL Key". In: *Proc. VLDB Endow.* 8.11 (2015), pp. 1118–1129.

[91] Donald Kossmann. "The State of the art in distributed query processing". In: *ACM Comput. Surv.* 32.4 (2000), pp. 422–469.

[92] Tim Kraska et al. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 489–504.

[93] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*. Ed. by Hal Finkel. ACM, 2015, 7:1–7:6.

[94] Yann LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Comput.* 1.4 (1989), pp. 541–551.

[95] Mun-Kyu Lee et al. "A simple proof of optimality for the MIN cache replacement policy". In: *Inf. Process. Lett.* 116.2 (2016), pp. 168–170.

[96] Wolfgang Lehner and Kai-Uwe Sattler. *Web-Scale Data Management for the Cloud*. Springer, 2013. ISBN: 978-1-4614-6855-4.

[97] Guoliang Li, Xuanhe Zhou, and Lei Cao. "AI Meets Database: AI4DB and DB4AI". In: (2021). Ed. by Guoliang Li et al., pp. 2859–2866.

[98] Guoliang Li, Xuanhe Zhou, and Lei Cao. "AI Meets Database: AI4DB and DB4AI". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li et al. ACM, 2021, pp. 2859–2866.

[99] Xupeng Li et al. "MLog: Towards Declarative In-Database Machine Learning". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1933–1936.

[100] *LIEF*. `https://github.com/lief-project/LIEF`. [Online; accessed 02-December-2022].

[101] Dong Liming, Liu Weidong, and Shao Jie. "Coexistence of Multiple Partition Plan Based Physical Database Design". In: *Proceedings of the 5th International Conference on Communications and Broadband Networking*. ICCBN '17. event-place: Bali, Indonesia. New York, NY, USA: Association for Computing Machinery, 2017, pp. 41–46. ISBN: 978-1-4503-4861-4.

[102] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019).

[103] Ester Livshits et al. "Approximate Denial Constraints". In: *Proc. VLDB Endow.* 13.10 (2020), pp. 1682–1695.

[104] Andrew L. Maas et al. "Learning Word Vectors for Sentiment Analysis". In: *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*. Ed. by Dekang Lin, Yuji Matsumoto, and Rada Mihalcea. The Association for Computer Linguistics, 2011, pp. 142–150.

[105] Remigius Meier and Armin Rigo. “A way forward in parallelising dynamic languages”. In: *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE, ICOOOLPS@ECOOP 2014, Uppsala, Sweden, July 28, 2014*. ACM, 2014, 4:1–4:4.

[106] Remigius Meier, Armin Rigo, and Thomas R. Gross. “Virtual machine design for parallel dynamic programming languages”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 109:1–109:25.

[107] Sergey Melnik et al. “Dremel: A Decade of Interactive SQL Analysis at Web Scale”. In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3461–3472.

[108] Jim Melton et al. “SQL/MED - A Status Report”. In: *SIGMOD Rec.* 31.3 (2002), pp. 81–89.

[109] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`. [Online; accessed 02-December-2022].

[110] Microsoft. *SQL Server Machine Learning Services*. `https://docs.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-2017`. [Online; accessed 03-February-2022]. 2022.

[111] Guido Moerkotte. “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *VLDB’98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Ed. by Ashish Gupta, Oded Shmueli, and Jennifer Widom. Morgan Kaufmann, 1998, pp. 476–487.

[112] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 561–577.

[113] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. “Recycling in pipelined query evaluation”. In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, 2013, pp. 338–349.

[114] Rimma V. Nehme and Nicolas Bruno. “Automated partitioning design in parallel database systems”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. Ed. by Timos K. Sellis et al. ACM, 2011, pp. 1137–1148.

[115] *Nuitka the Python Compiler*. `http://nuitka.net/`. [Online; accessed 02-December-2022].

[116] Rei Odaira, José G. Castaños, and Hisanobu Tomari. "Eliminating global interpreter locks in ruby through hardware transactional memory". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. Ed. by José E. Moreira and James R. Larus. ACM, 2014, pp. 131–142.

[117] Dan Olteanu. "The Relational Data Borg is Learning". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3502–3515.

[118] *ONNX Model Zoo*. `https://www.github.com/onnx/models`. [Online; accessed 02-December-2022].

[119] Thorsten Papenbrock and Felix Naumann. "A Hybrid Approach for Efficient Unique Column Combination Discovery". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. Ed. by Bernhard Mitschang et al. Vol. P-265. LNI. GI, 2017, pp. 195–204.

[120] Panos Parchas et al. "Fast and Effective Distribution-Key Recommendation for Amazon Redshift". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2411–2423.

[121] Kwanghyun Park et al. "End-to-end Optimization of Machine Learning Prediction Queries". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 587–601.

[122] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan et al. ACM, 2012, pp. 61–72.

[123] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. "Discovery of Approximate (and Exact) Denial Constraints". In: *Proc. VLDB Endow.* 13.3 (2019), pp. 266–278.

[124] *PEP 249 - Python Database API Specification v2.0*. `https://www.python.org/dev/peps/pep-0249/`. [Online; accessed 02-December-2022].

[125] Devin Petersohn et al. "Towards Scalable Dataframe Systems". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2033–2046.

[126] Meikel Pöss, Raghunath Othayoth Nambiar, and David Walrath. "Why You Should Run TPC-DS: A Workload Analysis". In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Ed. by Christoph Koch et al. ACM, 2007, pp. 1138–1149.

[127] Mark Raasveldt. "MonetDBLite: An Embedded Analytical Database". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1837–1838.

[128] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz et al. ACM, 2019, pp. 1981–1984.

[129] Mark Raasveldt and Hannes Mühleisen. “Vectorized UDFs in Column-Stores”. In: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20, 2016*. Ed. by Peter Baumann et al. ACM, 2016, 16:1–16:12.

[130] Mark Raasveldt et al. “Deep Integration of Machine Learning Into Column Stores”. In: *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. Ed. by Michael H. Böhlen et al. OpenProceedings.org, 2018, pp. 473–476.

[131] Fatemeh Rahimian et al. “A Distributed Algorithm for Large-Scale Graph Partitioning”. In: *ACM Trans. Auton. Adapt. Syst.* 10.2 (2015), 12:1–12:24.

[132] Fatemeh Rahimian et al. “Distributed Vertex-Cut Partitioning”. en. In: *Distributed Applications and Interoperable Systems*. Ed. by Kostas Magoutis and Peter Pietzuch. Vol. 8460. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 186–200.

[133] Alexander Ratner et al. “SysML: The New Frontier of Machine Learning Systems”. In: *CoRR* abs/1904.03257 (2019).

[134] Quentin Rebjock et al. “A Simple and Effective Predictive Resource Scaling Heuristic for Large-scale Cloud Applications”. In: *AIDB@VLDB 2020, 2nd International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020, Monday, August 31, 2020, Online Event / Tokyo, Japan*. Ed. by Bingsheng He, Berthold Reinwald, and Yingjun Wu. 2020.

[135] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[136] Kai-Uwe Sattler and Oliver Dunemann. “SQL Database Primitives for Decision Tree Classifiers”. In: *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, November 5-10, 2001*. ACM, 2001, pp. 379–386.

[137] Maximilian E. Schüle et al. “Blue Elephants Inspecting Pandas: Inspection and Execution of Machine Learning Pipelines in SQL”. In: *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*. Ed. by Julia Stoyanovich et al. OpenProceedings.org, 2023, pp. 40–52.

[138] Maximilian E. Schüle et al. “Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL”. In: *SSDBM 2020: 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, July 7-9, 2020*. Ed. by Elaheh Pourabbas et al. ACM, 2020, 6:1–6:12.

[139] Maximilian E. Schüle et al. “In-Database Machine Learning with SQL on GPUs”. In: *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*. Ed. by Qiang Zhu et al. ACM, 2021, pp. 25–36.

[140] Maximilian E. Schüle et al. “ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python”. In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Ed. by Melanie Herschel et al. OpenProceedings.org, 2019, pp. 562–565.

[141] Phanwadee Sinthong and Michael J. Carey. “AFrame: Extending DataFrames for Large-Scale Modern Data Analysis”. In: *2019 IEEE International Conference on Big Data (IEEE BigData), Los Angeles, CA, USA, December 9-12, 2019*. Ed. by Chaitanya K. Baru et al. IEEE, 2019, pp. 359–371.

[142] Apache Spark. *MLlib*. `https://spark.apache.org/mllib/`. [Online; accessed 21-February-2022]. 2022.

[143] *Spark-Vector Connector*. `https://github.com/ActianCorp/spark-vector`. [Online; accessed 02-December-2022].

[144] Leonhard F. Spiegelberg et al. “Tuplex: Data Science in Python at Native Code Speed”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li et al. ACM, 2021, pp. 1718–1731.

[145] Thomas Stöhr, Holger Märtens, and Erhard Rahm. “Multi-Dimensional Database Allocation for Parallel Data Warehouses”. In: *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Ed. by Amr El Abbadi et al. Morgan Kaufmann, 2000, pp. 273–284.

[146] Michael Stonebraker et al. “C-Store: A Column-oriented DBMS”. In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm et al. ACM, 2005, pp. 553–564.

[147] Michal Switakowski, Peter A. Boncz, and Marcin Zukowski. “From Cooperative Scans to Predictive Buffer Management”. In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1759–1770.

[148] James Thorne et al. “From Natural Language Processing to Neural Databases”. In: *Proc. VLDB Endow.* 14.6 (2021), pp. 1033–1039.

[149] *TIOBE-Index*. en. `https://www.tiobe.com/tiobe-index/python/`. [Online; accessed 17-October-2022].

[150] Frank Wm. Tompa and José A. Blakeley. “Maintaining materialized views without accessing base data”. In: *Inf. Syst.* 13.4 (1988), pp. 393–406.

[151] Patrick Valduriez. “Join Indices”. In: *ACM Trans. Database Syst.* 12.2 (1987), pp. 218–246.

[152] Ramakrishna Varadarajan et al. “DBDesigner: A customizable physical design tool for Vertica Analytic Database”. In: *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. Ed. by Isabel F. Cruz et al. IEEE Computer Society, 2014, pp. 1084–1095.

[153] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 5998–6008.

[154] Adrian Vogelsgesang et al. “Get Real: How Benchmarks Fail to Represent the Real World”. In: *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*. Ed. by Alexander Böhm and Tilmann Rabl. ACM, 2018, 1:1–1:6.

[155] Midhul Vuppalapati et al. “Building An Elastic Query Engine on Disaggregated Storage”. In: *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. Ed. by Ranjita Bhagwan and George Porter. USENIX Association, 2020, pp. 449–462.

[156] Bo Wang, Heiner Litz, and David R. Cheriton. “HICAMP bitmap: space-efficient updatable bitmap index for in-memory databases”. In: *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*. Ed. by Alfons Kemper and Ippokratis Pandis. ACM, 2014, 7:1–7:7.

[157] Kewen Wang and Mohammad Maifi Hasan Khan. “Performance Prediction for Apache Spark Platform”. In: *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*. IEEE, 2015, pp. 166–173.

[158] Ziheng Wei, Uwe Leck, and Sebastian Link. “Discovery and Ranking of Embedded Uniqueness Constraints”. In: *Proc. VLDB Endow.* 12.13 (2019), pp. 2339–2352.

[159] Paul J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proc. IEEE* 78.10 (1990), pp. 1550–1560.

[160] Marianne Winslett and Vanessa Braganholo. “Goetz Graefe Speaks Out on (Not Only) Query Optimization”. In: *SIGMOD Rec.* 49.3 (2020), pp. 30–36.

[161] Lucas Woltmann et al. “PostCENN: PostgreSQL with Machine Learning Models for Cardinality Estimation”. In: *Proc. VLDB Endow.* 14.12 (2021), pp. 2715–2718.

[162] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. “Robust Query Optimization Methods With Respect to Estimation Errors: A Survey”. In: *SIGMOD Rec.* 44.3 (2015), pp. 25–36.

[163] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. Ed. by Steven D. Gribble and Dina Katabi. USENIX Association, 2012, pp. 15–28.

[164] 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, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 17–30.

[165] Yi Zhang, Herodotos Herodotou, and Jun Yang. “RIOT: I/O-Efficient Numerical Computing without SQL”. In: *Fourth Biennial Conference on Innovative Data Systems Research, CIDR 2009, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. www.cidrdb.org, 2009.

[166] Yuhao Zhang et al. “Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches”. In: *Proc. VLDB Endow.* 14.10 (2021), pp. 1769–1782.

[167] Marcin Zukowski et al. “Super-Scalar RAM-CPU Cache Compression”. In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. Ed. by Ling Liu et al. IEEE Computer Society, 2006, p. 59.