Technische Universität München

TUM School of Computation, Information and Technology

Doctoral Thesis

Managing Dynamic Workloads in Relational Database Systems

Christian Michael Winter

Technische Universität München

TUM School of Computation, Information and Technology

Managing Dynamic Workloads in Relational Database Systems

Christian Michael Winter

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Alexander Pretschner

Prüfer*innen der Dissertation:

1. Prof. Alfons Kemper, Ph.D.
2. Prof. Dr. Thomas Neumann
3. Prof. Dr. Maximilian E. Schüle

Die Dissertation wurde am 06.03.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 25.09.2023 angenommen.

Abstract

The last decade has seen drastic changes in the data analytics landscape. For one, the perpetual growth of data has made it infeasible to materialize all data for processing. Instead, dedicated stream processing engines focus on processing vast amounts of data in a single pass without materializing the whole data set. Further, there has been a shift in the deployment of data analytics platforms from on-premise deployments to a cloud environment. These challenges threaten the leading position of relational database management systems in data analytics.

This thesis re-engineers several core components of database systems to overcome these challenges. In the first part, we devise three techniques for in-database stream processing. We demonstrate how streams can be analyzed in database systems as ring-buffered relations or using specialized materialized views. Further, we design a process to sample streams for later analysis efficiently. The second part of this thesis architects a technique allowing running queries to be suspended and migrated in dynamic cloud environments.

Zusammenfassung

Im letzten Jahrzehnt hat sich die Datenanalyselandschaft drastisch verändert. Zum einen ist es aufgrund des ständigen Datenwachstums nicht mehr möglich, alle Daten für die Verarbeitung zu materialisieren. Stattdessen konzentrieren sich spezielle Stream-Processing-Engines auf die Verarbeitung großer Datenmengen in einem einzigen Durchgang, ohne den gesamten Datensatz zu materialisieren. Darüber hinaus hat sich die Einsatzumgebung von Datenanalyseplattformen von lokalen Systemen zu Cloud-Umgebungen verlagert. Diese Herausforderungen bedrohen die führende Position von relationalen Datenbankmanagementsystemen in der Datenanalyse.

In dieser Arbeit werden mehrere Kernkomponenten von Datenbanksystemen überarbeitet, um diese Herausforderungen zu bewältigen. Im ersten Teil entwickeln wir drei Techniken für die datenbankinterne Streamverarbeitung. Wir zeigen, wie Datenströme in Datenbanksystemen als ringgepufferte Relationen oder mit Hilfe spezialisierter materialisierter Sichten analysiert werden können. Außerdem entwerfen wir einen Prozess, mit dem effizient Stichproben von Datenströmen für eine spätere Analyse genommen werden können. Im zweiten Teil dieser Arbeit wird eine Technik entwickelt, mit der laufende Abfragen in dynamischen Cloud-Umgebungen angehalten und migriert werden können.

To all my mentors, in academia and in life.

Preface

Excerpts of this thesis have been published in advance.

Chapter 4 has previously been published in:

> Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. "Meet Me Halfway: Split Maintenance of Continuous Views". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2620–2633

Chapter 5 has previously been published in:

> Christian Winter, Moritz Sichert, Altan Birler, Thomas Neumann, and Alfons Kemper. "Communication-Optimal Parallel Reservoir Sampling". In: *BTW*. vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 567–578

Chapter 6 has previously been published in:

> Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. "On-Demand State Separation for Cloud Data Warehousing". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2966–2979

In addition to these publications, the author of this thesis also co-authored the following related work, which is not part of this thesis:

> Christian Winter, Andreas Kipf, Christoph Anneser, Eleni Tzirita Zacharatou, Thomas Neumann, and Alfons Kemper. "GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons". In: *EDBT*. OpenProceedings.org, 2021, pp. 169–180

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Excerpts of this chapter have been published in [168, 170, 171].

For many years, on-premise relational database management systems have formed the sole backbone of the data analytics pipeline. From their inception over 50 years ago [34] to today, relational database systems have continuously evolved, constantly adapting to new hardware and workload trends. This evolution has allowed them to take advantage of many-core architectures [24, 62, 72, 82, 99], the growth of main memory capacity into the terabytes [42, 50, 79, 82, 125], and the availability of new storage mediums, such as flash storage [13, 56, 71, 100, 118, 162, 187]. Today, they remain at the forefront of data analytics and enjoy widespread success in many industries [13, 39, 68, 111, 158, 163].

However, recent years have seen the emergence of dynamic workloads and environments that traditional database management systems were not designed for. In particular, ephemeral data in the form of data streams [16, 148] and dynamic cloud environments [146, 183] are incompatible with the query execution model of current database systems. Consequently, it is unclear if and how relational database systems must be modified to efficiently handle these emerging trends. In this thesis, we address this question and devise four techniques that leverage the superior analytical capabilities of relational database systems and their optimizations for modern hardware to analyze ephemeral data in dynamic environments efficiently.

1.1 Challenges of Dynamic Workloads

Traditionally, relational database systems could rely on durable data and static deployment environments for analytical tasks and have optimized heavily for these circumstances. However, the assumptions that database systems have made no longer hold for cloud-based data analytics on ephemeral data streams.

To overcome the challenges arising from these new circumstances without sacrificing performance or reliability, it is important to understand their characteristics, which we discuss in this section.

1.1.1 Data Streams

The first major challenge we discuss stems from the growth in volume, veracity, and velocity of the data relevant for analysis generated by a wide range of sources, such as log streams of IoT and sensor data [36, 84], as well as the telecommunication [27, 85] and financial industry [30]. This growth has led to changes in how data is analyzed. Instead of fully materializing all data before analysis, as is required for database systems, it is increasingly processed in a streaming fashion in a single pass. Further, the desired time-to-insight has shrunken massively. While in the past, analytics were performed on nightly- or even weekly-updated data warehouses, users increasingly look for real-time insights into their data. Consequently, data needs to be analyzed in a real-time fashion.

Dedicated systems, often called stream processing engines (SPEs), have been developed to perform real-time analyses on data streams [3, 29, 31, 67, 115, 123, 154, 179, 181]. SPEs have evolved into capable analytical systems, allowing for both stateful [29, 31, 114, 123, 153, 179] and transactional [5, 25, 64, 110, 184] stream processing. However, while these systems provide all the required functionality to deal with data streams efficiently, they often lack the ability to manage and modify historic data internally to combine them with stream insights. Further, they are often unable to leverage the full potential of modern hardware [180]. Relational database systems, on the other hand, offer extensive functionality to manage and analyze historic data optimized for the underlying hardware. In turn, it is infeasible, or often even impossible, for current database systems to perform analytics over streamed data without fully materializing all input data. An ideal system should address this challenge by leveraging the performance of the database system's analytical query engine for data stream analytics without materializing the entire stream for processing.

1.1.2 Cloud-Based Data Analytics

The second challenge we investigate in this thesis is the gradual shift from on-premise deployments to a cloud setting for the data analytics pipeline. While the flexibility offered by cloud architectures offers benefits for customers and developers alike, allowing for easy scaling and pay-as-you-use server provisioning, leveraging the full potential of this architecture requires a redesign of many database components. A critical step in utilizing this new-gained flexibility is the

shift from a shared-nothing architecture, as implemented by the initial version of AWS Redshift [68], to a storage-separated architecture [39], where storage for relations can be scaled independent of the resources for query evaluation. The benefits of independently-scalable storage have led to the widespread adoption of this architecture in industry [28, 39, 157, 165]. Recently, even dominant shared-nothing database systems such as Redshift have shifted to this flexible architecture [13].

Decoupling storage and compute allows systems to quickly add and remove compute resources without copying durable data, such as database relations. However, more is needed to provide flexibility for individual queries. Even though new workers can access storage easily, the current state of a query still resides at the compute nodes. While cloud providers offer a wide range of worker machines, finding the right worker for a task remains challenging [102]. Without flexibility for running queries, methods to find the right worker for a task rely on speculative execution and have to restart the task in case of an adverse selection [10]. Not only end users but also service providers can benefit from flexibility for individual queries. For example, service providers often over-subscribe tenants to machines to achieve maximum resource utilization in a cluster [91]. When service providers cannot rely on elastic execution models to move queries between servers, this can lead to the cancellation of running queries when the resources required by all subscribed tenants exceed a machine's physical resources.

Separating state from compute in addition to storage, as proposed by the POLARIS systems [7], overcomes these issues by allowing to add, remove, and replace workers during runtime. However, this comes at a performance cost, as all states must be synchronized over the network. Ideally, this performance cost is only incurred when necessary, e.g., in case of changes to the worker pool by keeping the query state at the workers when not.

1.2 Contributions

The contributions made by this thesis can be split into two categories, each addressing one of the challenges outlined in Section 1.1. Before discussing our contributions, Chapter 2 provides the necessary background on the Umbra database system, which we extend with our techniques. Following, we discuss three techniques for in-database stream processing in Chapters 3 to 5. Each approach offers a different trade-off between ease of integration and analytical capabilities. Finally, Chapter 6 presents a technique to better leverage the flexibility of cloud environments by allowing to suspend and migrate queries on demand.

Relation-Based Stream Processing

Relations form the foundation for data management and processing in relational database systems. Naturally, this makes relations a prime candidate for integrating streams into database systems. In addition, database systems often support multiple relation types optimizing for different hardware and environments, such as dedicated in-memory and disk-based relations. For these relations, databases provide physical data independence, hiding the nature of the relation from upstream operators.

In Chapter 3, we describe a novel ring-buffered relation utilizing this physical data independence to provide in-database stream processing without materializing the entire stream or necessitating changes to other database components and functionality. This relation enables to combine finite or session-windowed streams with durable database relations in a single query. A relation-based integration further allows access to the full analytical capabilities of database systems and a fully SQL-based interface. We demonstrate the feasibility of our approach by integrating a stream relation into the Umbra database system and evaluate its performance in end-to-end benchmarks.

Continuous-View-Based Stream Processing

While relation-based stream processing provides excellent ease of integration for existing systems as it builds on the existing query model, it is limited in the streaming model it can support. For unbounded streams, relying on this query model is infeasible, as results are only available after all inputs have been fully processed. Consequently, enabling complex and stateful analytics for unbounded streams and queries involving multiple streams requires a less holistic view of query processing. Database systems already implement concepts similar to our desired query processing semantics of accessing current results for queries over dynamic inputs in the form of materialized views.

In Chapter 4, we propose a new type of materialized view that provides extensive capabilities for unbounded stream analytics within relational database systems, called *continuous view*. The key component of continuous views is a novel maintenance strategy, splitting the maintenance work between inserts and queries. By performing only the initial parts of a query for arriving tuples, we can sustain the insert rates necessary for high-velocity data processing. In turn, the resulting initial state reduces the views' memory footprint and aids view refreshes, allowing for low query latency and high view refresh rates. Our view-based approach to stream processing allows the database system to support stateful and long-running queries over multiple streams and durable relations. To demonstrate the practicality of this strategy, we integrate continuous views

into the Umbra database system. We show that split maintenance can outperform even dedicated stream processing engines on analytical workloads, all while still offering similar insert rates. Compared to modern materialized view maintenance approaches, such as deferred and incremental view maintenance, that often need to materialize expensive deltas, we achieve up to an order of magnitude higher insert throughput.

Communication-Optimal Sampling

The two strategies outlined above offer an excellent range of real-time stream processing capabilities. However, for some workloads, it is also desirable to perform analytics at a later time. Given the high volume of data streams, it is undesirable to materialize them for this purpose. Therefore, such analyzes are often performed on reduced versions of the streams, either in the form of query results as generated by the views of Chapter 4 or on representative samples. Reservoir sampling is an excellent choice for drawing samples from unbounded data such as streams, as it generates a fixed-size uniform random sample independent of the input cardinality. However, the collection of reservoir samples itself can already be a bottleneck when sampling from high-velocity data.

In Chapter 5, we introduce a technique that fully parallelizes reservoir sampling for many-core architectures. Our approach relies on the efficient combination of thread-local samples taken over chunks of the input without necessitating communication during the sampling phase and with minimal communication when merging. We show how our efficient merge guarantees uniform random samples while allowing data to be distributed over worker threads arbitrarily. Our analysis of this approach within the Umbra database system demonstrates linear scaling along the available threads and the ability to sustain high-velocity workloads.

On-Demand State Separation

Cloud providers offer unprecedented flexibility for application developers by enabling them to add and remove instances on demand from a large pool of workers, paying only for usage duration. One way to leverage this flexibility for database systems is to abstract away from worker machines using state-separated architectures and instead plan with abstract compute, state, and storage resources for queries. However, these architectures cause overhead when synchronizing state externally, degrading query performance. On the other hand, solutions without state-separated architectures resort to query restarts in the event of changes to the worker pool, negating progress made.

In Chapter 6, we propose a technique to suspend queries and migrate them between workers through on-demand state separation. By introducing state separation only when required, we retain maximum flexibility and performance while keeping already achieved progress in case of changes to the worker machines. To derive the requirements for state separation, we first analyze the query state of medium-sized workloads on the example of TPC-DS SF100. Using this, we analyze the cost and describe the constraints necessary for state separation on such a workload. Furthermore, we describe the design and implementation of on-demand state separation in the Umbra database system. Finally, using this implementation, we show the feasibility of our approach on the TPC-DS benchmark and give a detailed analysis of the cost of query migration and state separation.

CHAPTER 2

The Umbra Database System

Excerpts of this chapter have been published in [168, 170]

All contributions in this thesis were made in the framework of the Umbra database system [118]. Building upon the ideas of its predecessor HyPer [82], Umbra makes several improvements geared towards a more robust, adaptable, and flexible processing model. Most notably, Umbra no longer relies on a purely in-memory data representation and scales to workloads beyond main memory capacity [100]. While many of the improvements employed by Umbra are beyond the scope of this thesis, we want to briefly highlight the key differences to HyPer and describe the two components crucial to and modified by our contributions: the execution model and query state management.

2.1 System Overview

We base our overview on the path of a SQL statement through Umbra's architecture, as shown in Figure 2.1. All SQL statements received by Umbra are first passed to the query compiler. The SQL parser transforms the received statement into an abstract syntax tree (AST) representation of the query using PostgreSQL-compatible grammar, which serves as a basis for Umbra's optimizer. In turn, the optimizer converts this AST into a physical query plan by employing several optimizer passes. For these passes, the optimizer has, e.g., access to gathered statistics and derived cardinality estimates [20, 21, 57] managed by the database runtime for join reordering [120] and a wide range of specialized physical operator and relation implementations [17, 45, 51, 55, 89, 128, 129, 134, 135, 142, 168]. Like HyPer, Umbra generates code for the resulting query plan [117]. However, instead of generating a monolithic piece of code, Umbra's execution plan comprises multiple steps. For each step, the code generator

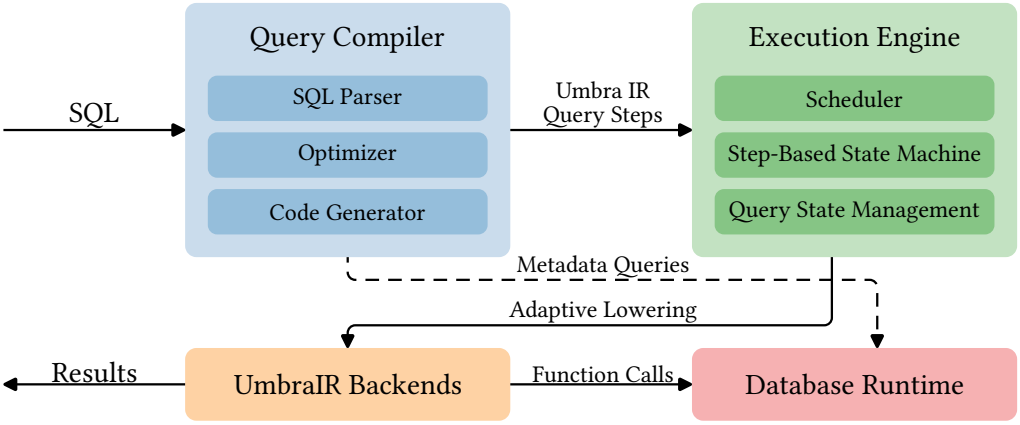


Figure 2.1: Overview of Umbra's architecture. Query compiler and execution engine show sub-components relevant to our contributions.

creates a static single-assignment [38, 130] intermediate representation called *UmbraIR* [83].

Steps represent an individual state in a conceptual state machine managing the execution. For execution, the UmbraIR corresponding to the current state is adaptively lowered to an executable form in one of the UmbraIR backends. These backends either directly interpret the UmbraIR code [88], emit x86 machine code [83], or compile it using the LLVM framework. Backends can be switched dynamically for running queries, enabling a low latency query startup and switching to more optimized executables as they become available [88]. During execution, backends can access complex functionality [26, 170] and allocate memory [46] via function calls to the database runtime.

2.2 Execution Model

As outlined in the previous subsection, Umbra's execution model is based on a state machine representation of a query. Each state machine's state corresponds to a logical part of the query plan, which we call *step*. To avoid ambiguity with the *query state* of analytical queries, we refer to the state-machine states as steps throughout this thesis. Relying on multiple steps instead of a code monolith allows for higher flexibility for the scheduler and the query's control flow.

First, steps form natural scheduling units because they are self-contained. This property allows the scheduler to reassign workers and suspend queries at step granularity without requiring extensive task preemption measures. Furthermore, in combination with the task-based interface of the scheduler [166], we can dynamically inject tasks in between steps, which we employ for both the

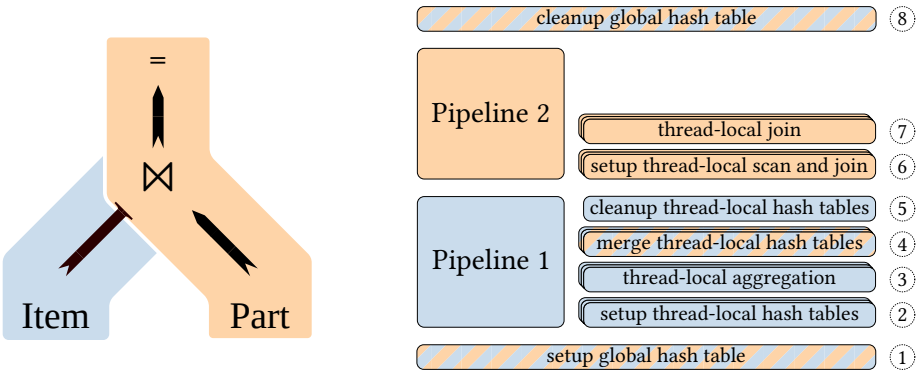


Figure 2.2: UmbraIR steps for an exemplifying query plan.

maintenance strategy of our view-based stream processing in Chapter 4 and the query live migration in Chapter 6. Finally, steps can also directly influence the control flow of the query plan, as the next step to be executed is determined by the return value of the previous step. This flexibility allows us to reorder or skip steps for execution, which we utilize for our maintenance strategy in Chapter 4.

All but two steps, the global setup (1) and cleanup (8), are associated with a *pipeline*. Pipelines comprise multiple operators that form a contiguous path in the query plan wherein a tuple does not have to be materialized for processing. For example, there are two pipelines in the plan of Figure 2.2. One comprises a scan of the *Item* relation and the join, materializing all tuples as the join's build side. The second pipeline comprises three operators, a scan of the *Part* relation, the join, and a result operator reporting all tuples to the user. We refer to the pipeline sink wherein tuples are materialized as the *pipeline breaker*. Operators, such as the join in Figure 2.2, can act as pipeline breakers for one pipeline and as intermediate operators for another. Further, pipeline breakers can be the source of multiple pipelines due to common subtree elimination optimizations. Each pipeline comprises up to four logical steps, local initialization, thread-local execution, a merge phase, and local cleanup. In regular queries, with the exception of recursive ones, each step is traversed once, one after the other. However, not all steps are required for all pipelines. Also, some steps, like the thread-local join in Figure 2.2, are executed multiple times for different parts of the input, but we still consider them one logical step. To better illustrate the role of steps in queries, we reference the steps of the query in Figure 2.2 in our description.

Global Initialization. Before starting per-pipeline processing, we set up all data structures to share results between pipelines. These include result buffers and hash tables for aggregation and joins. There is only one global initialization step per query, which is always executed as the first step (1).

Local Initialization. In the first step of each pipeline, we initialize all data structures relevant for in-pipeline processing. This initialization includes setting up temporary tuple buffers, simple data structures such as hash maps, and helpers for complex operators. In our example, this includes steps (2) and (6) to set up the structures needed for join porcessing.

Thread-Local Execution. During the execution phase, the actual tuple processing is performed. Processing can include, e.g., scanning tuples from disk or buffers, applying filters and functions, aggregation, and joins. Umbra employs morsel-driven parallelism [99] and, therefore, this step is run in parallel, processing a single morsel for each invocation. The results are stored in thread-local buffers or directly reported for the topmost pipeline. Step (3), e.g., stores the corresponding inputs in a local hash table to be later merged into a global join hash table. Step (7) probes the resulting hash table in parallel and reports the resulting join tuples directly without using a local buffer.

Merge. After thread-local processing is complete, the results are combined and made available for the following pipeline through shared storage regions. Examples of this step are merging the individual runs of the merge-sort for a sorting operator or combining local buffers and hash tables. In our example, this happens in step (4). Since pipeline 2 directly reports the final result, merging local results is unnecessary.

Local Cleanup. Finally, the auxiliary data structures used for intra-pipeline processing are cleaned up, and the memory used is freed ((5)).

Global Cleanup. After the query returns all results to the user, or if the query is canceled, we clean up all data structures initialized in the global initialization. This step is always executed, even if the query encounters errors, to free up all memory associated with a query ((8)).

2.3 Query State Management

We distinguish two categories of state for queries in Umbra, global and local state. The former is used for inter-pipeline communication and is available for the entire duration of a query. The latter, on the other hand, is only valid for one pipeline at a time. Each thread works on its thread-local copy of the local state to avoid contention between workers in the parallel execution phase of a pipeline. Both the global and the local state are of a fixed size, determined statically from the operator tree at query compile time. Consequently, Umbra requires a third memory region that materializes variable-size data accessible to both the local and global state for data generated during execution, such as the individual tuples. Separating a query's state into multiple state and storage

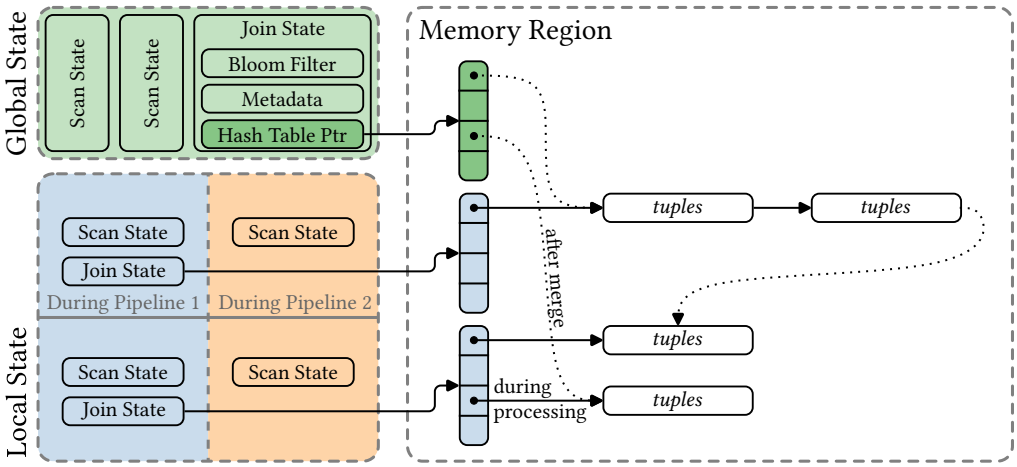


Figure 2.3: Overview of Umbra's query state management. Local state and associated structures in the memory region only exist during the local processing phase of the corresponding pipeline.

regions has several advantages. For one, it allows the allocation of the global and local state memory once at compile-time, re-using the local state memory for all pipelines. Further, a shared variable-size memory region allows a copy-free move of tuples from local to global state in the merge phase.

We show the use of the different storage areas for the exemplifying query of Figure 2.2 in Figure 2.3. While we need to keep thread-local copies of the metadata and index structures required for scans and the join, one can see that tuples are not copied at any point. During the merge phase, we can simply re-link tuple chains from thread-local hash tables to the hash table residing in the global state using pointers. Thread-local copies of the hash table can, thus, be destructed after the merge phase is complete, freeing the used memory and local state for the next pipeline.

CHAPTER 3

Relation-Based Stream Processing

Excerpts of this chapter have been published in [170].

As outlined in Chapter 1, there is an ever-increasing need for just-in-time analyses combining real-time data in the form of streams with information held in databases, such as user, customer, or billing data. Several solutions integrating durable data in stream processing engines (SPEs) have been proposed, either loading read-only data into the stream processing engine from local sources such as CSV files or offering interfaces to external databases [29, 31, 87, 179]. However, the reverse direction of integrating stream processing into relational databases has yet to receive much attention. While modern SPEs are capable tools for many workloads, they lack the functionality to manage historic data internally. We argue that the unmatched capabilities and performance of relational database systems for managing and analyzing relational data make them the ideal solution for processing durable relations and data streams.

In this chapter, we devise a technique to integrate streams into database systems through a specialized streaming relation. By relying on a ring-buffered specialization of regular database relations, we can utilize the database system's full type and query support and gain access to a wide range of pre-built functionality and operators, such as efficient joins and string operations. Our approach relies on regular SQL to interact with streams, necessitating no changes to the database grammar. Thus, streams can be used with all tools commonly used for database access, such as object-relation mapping (ORM) libraries available for many programming languages. Furthermore, the SQL-based interface allows users to easily express queries incorporating streams and regular tables. We demonstrate the applicability of our approach by implementing it in the state-of-the-art database system Umbra [118].

We outline the relation-based integration on an exemplifying workload, which we use as a running example throughout this chapter: Consider a microblogging service where users can share and like simple text posts. For reporting,

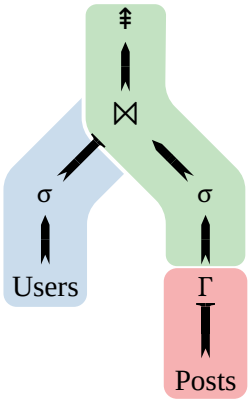
users				
id	username	contact_info	region	rate
1	Tom	+1 555 ...	US	200.0
2	Max	+1 555 ...	US	200.0
12	Chris	+49 89 289 ...	DE	300.0
...	...	...	...	...

posts			
uid	username	score	content
2	Max	54	Hello World!
12	Chris	9'153	It's great to process streams in a database system!
...	...	...	...

Figure 3.1: Schema for the microblogging workload of the running example.

```
with user_impact as (
  select     uid, avg(score) as score
  from       posts
  group by   uid
)
select       u.username, u.contact_info,
             u.rate, i.score
from         users u, user_impact i
where        u.id = i.uid and i.score >= 1000 and
             u.region = 'DE'
order by     u.rate / i.score
```

(a) SQL



(b) Queryplan

Figure 3.2: Exemplifying analytical query combining a stream and durable relations.

such a service might be interested in finding influential posters in a given region, e.g., users who achieved an average of at least 1'000 likes per post in the last month for promotional campaigns. Figure 3.2 depicts the corresponding query, with the schema shown in Figure 3.1. In many cases, business data, such as the contact info and payment details for paid bloggers, will be stored separately from the service data, such as posts. In the past, it was necessary to either find influential posters in the service database or materialize posts in the business database. The first option is undesirable as it involves analytical queries in a system likely optimized for simple lookup and update operations. In contrast, the second option would unnecessarily bloat the analytical database. On the other hand, our approach allows us to stream the last month's posts into the analytical database without materializing them, requiring no analytical functionality from the service database. This chapter covers the following key points:

- We describe the integration of stream processing in database systems as a specialized in-memory ring-buffered relation.
- We discuss the streaming model achieved by our integration.

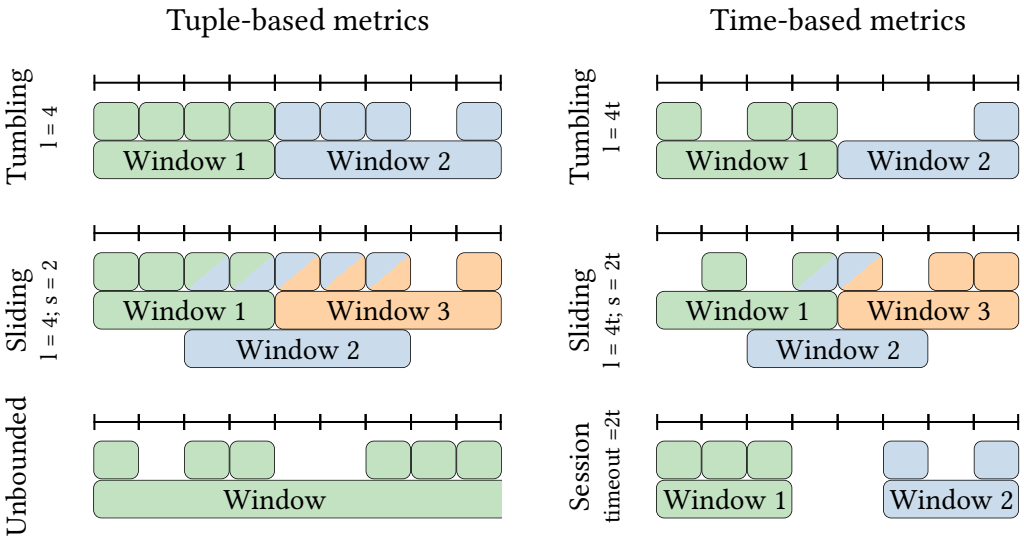


Figure 3.3: Overview of common stream windowing semantics.

- We evaluate our streaming relation against dedicated stream processing engines, focusing on end-to-end query and insert performance on a TPC-H-based workload.

The remainder of this chapter is structured as follows: In Section 3.1, we discuss background and related work in stream processing. Following, we discuss our approach to in-database stream processing in Section 3.2, which we evaluate in Section 3.3. Finally, we summarize the findings of this chapter in Section 3.4.

3.1 Background

Over the years, stream processing has evolved into a diverse area of research, spanning a wide variety of data stream models optimized for different applications. In this section, we outline the model underlying our work and discuss relevant related work in the intersection between stream processing and relational database systems.

3.1.1 Stream Model

While data stream models differ in many aspects, we focus on the two main categories most relevant to our work: windowing semantics and state management. Windowing semantics determines which subset of the stream qualifies for query evaluation at any given moment. Among the most common windowing semantics are sliding, tumbling, session, and unbounded windows, as shown

in Figure 3.3. The first two semantics are further subdivided into time- and tuple-based metrics. Tumbling and sliding windows define a fixed window size l, either in terms of the number of tuples or in a duration t. While tumbling windows always advance by the full window size, sliding windows advance by a set length s, which can result in overlapping windows. Session windows are separated from one another by periods of inactivity where no tuple arrives. Finally, unbounded windows consider the entire stream for a query and are, thus, unsuitable for infinite streams.

The window semantic closest to regular relation processing is the unbounded window. However, this would mean that queries over infinite streams will never report a result as database queries only advance to the next step, i.e., operator, once an input is fully depleted. Therefore, we instead follow the session window semantics, which is still close to relation scan semantics, and assume a scan as depleted when no new stream tuples have arrived for the specified inactivity duration. Note that sliding and tumbling windows can still be achieved on top of this session window using the SQL `WINDOW` operator.

The second category, state management, determines where and how systems manage the state of streaming queries. This state mainly comprises intermediate query results, such as aggregates, but also includes routing and meta information, e.g., for worker and checkpoint management. In their survey, To et al. [153] identify four different state models for stream processing. Of those, we most closely follow the *operator view* of Fernandez et al. [53] wherein query progress and state are materialized within operators. However, due to Umbras pipeline-based query execution model, query state only occurs at pipeline breakers, not at all individual operators. Further, distributing tuples to workers is handled through morsel-driven parallelism [99] in our approach. Therefore, routing decisions for tuples are made by downstream operators pulling new morsels, not actively and push-based by upstream operators.

3.1.2 Related Work

Our approach overlaps with two primary research areas in data analytics: relational database systems and stream processing. Both have seen vast amounts of research, and we, therefore, focus our discussion of related work on their intersection.

Durable data in stream processing engines. Recent years have seen increased demand for analytics combining historic and streamed data. Consequently, stream processing engines such as Apache Flink [29] and Apache Spark [179] enable the use of historic data in analytical queries over data streams. However, they do not support managing historic data internally and instead rely on

external sources. These external sources can be file formats like Parquet and CSV or database systems through connectors such as JDBC.

While stream processing engines do not offer capabilities for managing historic data, modern SPEs manage state for long-running and complex queries internally [114, 123, 153]. To prevent conflicts between multiple queries on a shared state, some SPEs rely on transaction semantics commonly used by database systems [25, 64, 184]. TSpoon [5] extends Apache Flink [29] with a transaction model, thereby enabling a queryable state for data stream analytics at configurable isolation levels. In addition, Meehan et al. [110] build upon the OLTP database system H-Store [79] and utilize H-Store's transactional processing model for data streams, enabling the ACID-compliant execution of streaming and transactional database queries in a single system.

In-database stream processing. Combining streams and relational data in a single system has been proposed in the context of data warehouse architectures [116, 167]. However, these architectures rely on two separate engines for internal relational query and stream processing. Some works propose a unified SQL-based query language to express queries over both streams and durable relations easily [18, 77]. Past research integrating stream processing and durable data in a single engine often rely on materialized views [69, 141] to realize continuous queries [16, 106, 148]. DBToaster [87, 122] implements higher-order incremental view maintenance in a standalone engine to enable high insert and query throughput for views combining both static and dynamic data. In addition, PipelineDB [124] supports stream processing in the full-fledged database system PostgreSQL using dedicated streaming views.

3.2 Approach

Having defined the theoretical streaming model of our ring-buffer-based in-database stream processing, we can describe its design and implementation. The core difference between our in-database stream processing to processing relational data is that streamed data is not fully and permanently materialized within the database at the start of a query. For regular queries, the data that a query is working on is determined by its transaction. Each transaction has a single state of the database that it will evaluate all queries on. To achieve this, all data must be fully materialized at the start of a query, and any parallel changes must be handled transparently. Streams, on the other hand, are not transactional. Stream entries are ephemeral and are only cached in the database for a short time. Queries, therefore, cannot rely on the availability of all stream elements for query evaluation. Furthermore, queries involving streams have to handle stream arrivals during a query. In this chapter, we will only deal with

queries over bounded streams and discuss queries for infinite streams in later chapters.

3.2.1 Interface

We want to rely purely on regular SQL grammar to interact with data streams. However, we must also ensure the database can infer which relations to treat as a stream and which to persist. For this, we follow a syntax similar to that of PipelineDB [124] and create streams as foreign tables with a reserved server name *stream*. For our example stream *posts*, this will result in the SQL statement:

```
create foreign table posts (
  uid       integer,
  username  varchar,
  score     integer,
  content   varchar
) server stream
```

Following the creation of this foreign table, all operations can be kept oblivious to the streaming nature of the relation. Inserts, therefore, can use regular SQL insert semantics, e.g.,

```
insert into posts values (
  12, 'Chris', 9153,
  'It's great to process streams in a database system!'
)
```

to insert the tuple displayed in Figure 3.4.

3.2.2 Caching Layer

Before discussing inserts and queries to data streams, we must establish how streaming data is stored within the database. Conceptually, ephemeral streams do not need to be materialized outside of queries and instead can be processed fully and directly on arrival. However, it is beneficial to cache chunks of the stream before processing. For one, a cache can compensate load spikes in the input stream where inserts occur too frequently for queries to keep up. Using a cache can help alleviate such short spikes by accepting tuples to be scanned by queries later on. Furthermore, a cache allows for morsel-wise processing of queries scanning the stream, eliminating expensive input synchronization operations for individual tuples.

For our approach, we implement a caching layer based on a ring buffer. Our buffer has two main components, displayed in Figure 3.4. The first component, the tuple buffer, stores fixed-size tuple data. This data includes all fixed-size

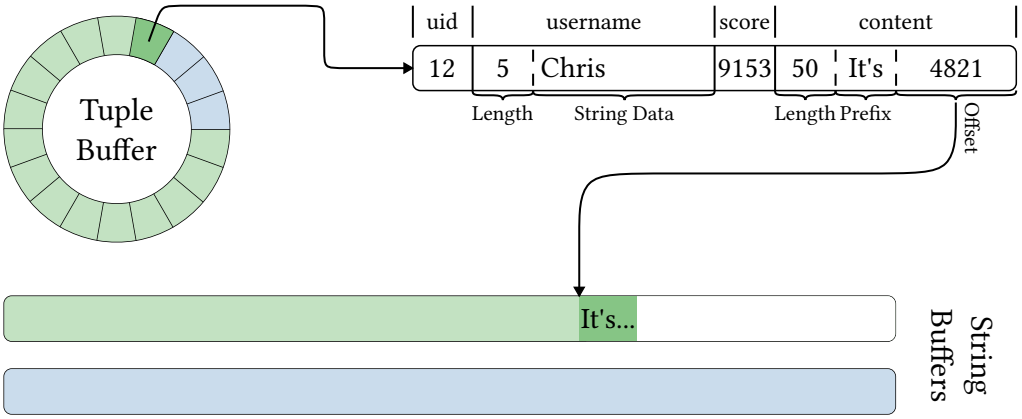


Figure 3.4: Overview of the caching layer consisting of a ring buffer for fixed size tuple parts and two string buffers used in alternation.

columns, such as integer, numeric, and floating-point values, as well as metadata for variable-sized types, such as strings. In our example *posts* stream of Figure 3.1, these are the values for the integer columns *uid* and *score* and the metadata for the string columns *username* and *content*. The number of tuples to be held in this buffer can be configured to fit the expected load. By only storing metadata for variable-sized data and not the data itself [118], we ensure that all tuples have the same size, allowing us to easily re-use slots without checking for overlaps with still valid data.

The data for variable-sized types are instead stored in resizable buffers pointed at by the metadata. In the exemplifying tuple in Figure 3.4, one can see the two different storage formats for strings used by Umbra. Both formats first store the 4 byte length of the string. Short strings up to a length of 12 characters, such as this *username*, are stored inline in the remaining 12 bytes, requiring no additional buffer storage. For longer strings, here for the *content*, we store a 4 byte prefix and an offset into a separate storage region. For streams, this region is one of the string buffers. The prefix helps quickly answer some comparisons and filter predicates without loading the full string. Note that Umbra picks the string format for each individual string value, meaning a longer username for another tuple might be stored externally.

In contrast to the fixed-size tuple data, we cannot overwrite a single ring buffer for strings. When inserting a string at slot n in the ring buffer longer than the string of the tuple previously cached at slot n, it would at least partially overwrite the string data of the tuple still held in slot $n + 1$. This tuple is,

Algorithm 3.1 Stream caching layer insert processing

```
1:  function processInsert(Tuple t)
2:      tid ← writeTid.fetchAdd(1)
3:      bufferSlot ← tid  mod bufferSize
4:      odd ← ⌊tid / bufferSize⌋  mod 2
5:      stringBuffer ← stringBuffers[odd]
6:      if slot = 0 then
7:          stringBuffer.clear()
8:      for value ∈ tuple do
9:          if value.isString() then
10:             if stringLength(value) > 12 then
11:                 buffer.storeExternalString(slot, stringBuffer.store(value))
12:             else
13:                 buffer.storeStringInPlace(slot, value)
14:         else
15:             buffer.store(slot, value)
16:     /* Delay scan visibility until all previous tuples are visible */
17:     while validTuples < tid - 1 do
18:         wait()
19:     validTuples ← tid
```

however, still accessible through the cache. To prevent overwriting still valid and accessible tuples, we alternate between two string buffers for different runs through the ring buffer, using one buffer for even and one for uneven runs. Alternating between buffers guarantees that string offsets are valid at least until the tuple is overwritten in the tuple buffer while still avoiding unnecessary allocations.

3.2.3 Insert Processing

Having outlined the layout of our caching layer, we can describe the insert process for stream tuples. While tuple-at-a-time inserts are also possible, we optimize for bulk inserts from an external streaming broker like Kafka [96] or SQL insert statements, e.g., reading from CSV files. The process is outlined in Algorithm 3.1 conceptually for a single tuple. In our implementation, we perform such inserts at morsel-granularity instead, collecting inserted tuples thread-locally before merging them into the relation in bulk for performance reasons. For each insert, we acquire a tuple identifier for the new tuple (Line 2). This id determines the ring buffer slot to store the tuple in. Furthermore, it

determines the string buffer to be used for external strings. Before writing the first slot of the ring buffer, we additionally mark the corresponding string buffer for cleanup (Line 7). Note that while there are no longer any direct references into the string buffer from the ring buffer, we still do not free the memory region to not interfere with queries still processing buffer entries that were just overwritten. Instead, the last scan to finish on this string buffer will free the associated memory when it is completed. We will discuss string buffer memory management when discussing queries in Section 3.2.4.

Following, we write the tuple data into the ring buffer slot. For strings, we decide between the two storage layouts outlined in Section 3.2.2 based on the string length. All other values are stored in-place in the ring buffer. Finally, we mark the tuple as valid to make it visible for scans (Line 19). To prevent partially-written cache entries of parallel inserts from being accessible for scans, we only mark new tuples as visible once all previous tuples are visible.

3.2.4 Query Processing

Through our specialized relation, the only operator in a query plan aware of an input's streaming nature is the table scan. All other operators can be kept oblivious about the nature of their input. While this integration is minimally invasive, it requires a careful design of the scan operator. Scans of regular relations rely on table metadata to determine the range of tuples to scan at query planning time which further determines the boundaries for the scan at execution time. For streams, however, we cannot rely on the scan boundaries to be known as tuples will still arrive during the query execution. Even cardinality estimates for query optimization can be unreliable as past stream behavior does not necessarily reflect future behavior. Therefore, we need to adapt query processing in two areas to handle streams efficiently: query planning and scan operator design.

Query Planning

For query planning, especially for join ordering, database systems rely on cardinality estimates for scans and filter predicates. These estimates are sourced from statistics maintained by inserts and updates to the relation. We cannot assume that previous statistics are available and reliable for streams. However, we still want the optimizer to produce an optimized query plan, especially to reduce materialized intermediate result sizes in the case of high-volume data streams. We assume streams are of the largest cardinality and, therefore, want to only materialize them if necessary. While the optimized query plan of Figure 3.2b only materializes the aggregated scores per user, an unoptimized plan without

Algorithm 3.2 Stream scan operator morsel picking

```
1:  function selectScanRange
2:      morsel ← {}
3:      while now() − lastPick.load() < timeout do
4:          position ← validTuples.load()
5:          limit ← lastScanned.load()
6:          loop
7:              if position ≤ limit then
8:                  updateLimit()
9:                  break
10:             lastPick.exchange(now())
11:             if pickRange(morsel, position, limit) then
12:                 return morsel
13:     return ScanDone
```

statistics might first join the *users* and *posts* relations, potentially materializing the *posts* stream in the join hash table. To avoid this, we hint to the optimizer that streams will always comprise the most data, thus ordering them to the probe side of joins.

Scan Operator

In contrast to scan operators for durable relation, our stream scan operator has to mask two things: unknown input bounds and ephemerality of tuples. As we, apart from the scan, entirely rely on existing database operators for query processing, we also have to adhere to the execution model of the database system. In our system Umbra, this is the producer-consumer model [117]. Generally, database execution models will process an operator, or, in our case, a pipeline, entirely before starting working on the next. Consequently, they cannot handle late arrivals of tuples. Therefore, we must determine when a stream has been fully processed at the scan before signaling that query processing can move to the next task. There are two different possibilities to achieve this: Sending a dedicated end-of-stream tuple or message signaling that the input is depleted or detecting input depletion from metadata, such as arrival rate. We focus on the latter option, which is most consistent with our approach of handling streams transparently using an SQL interface.

For simplicity, we rely on session window semantics to achieve this, advancing to the next step of query processing when no new tuples have arrived for a predefined timeout. For our integration into Umbra, we integrate this session window semantic into the morsel selection [99] where range-based metrics

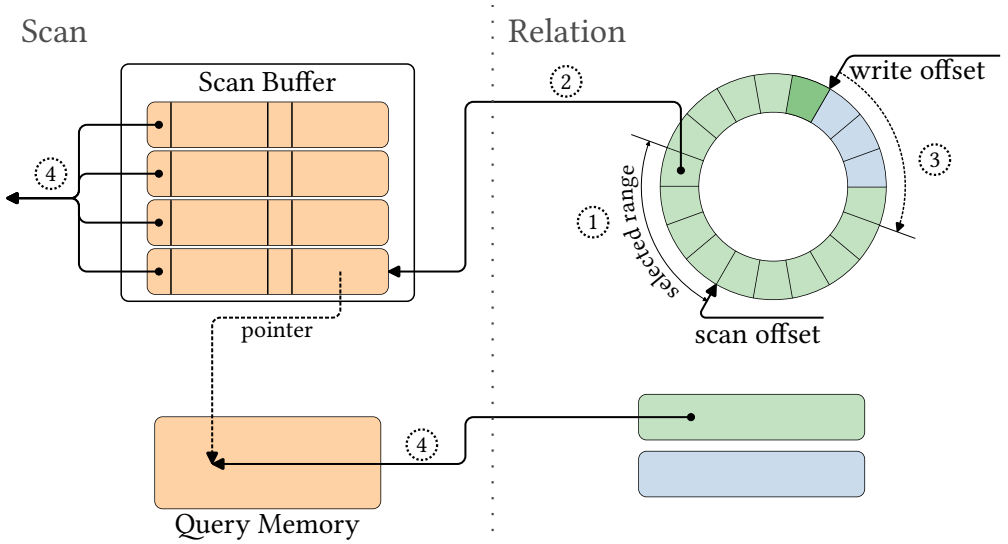


Figure 3.5: Overview of the four phases of the stream scan operator. Locating the desired range in the ring buffer (1), copying the range to the scan-local buffer (2), checking for potential conflicts (3) and reporting the tuples to the downstream operator (4).

reside for regular scans. Algorithm 3.2 outlines the resulting strategy. Before obtaining a scan range, we check the timeout condition (Line 3). If no thread detected new arrivals during this timeout, we continue with the next query processing task as we consider the stream input depleted. Following, we fetch the latest range information and check if tuples are available for processing (Line 7). If not, we return to the timeout check of line 3. Once tuples are available, we update the timeout condition (Line 10) and try to pick a morsel. Note that the unchecked change to the timeout condition can lead to a slight imprecision for the timeout, as it might lead to the loss of a more recent timestamp. However, we deem this acceptable as it allows us to reduce synchronization overhead.

We still must mask the second property of streams, their ephemerality. The ephemeral nature of streaming data does not impact the range selection outlined above, which is performed entirely on tuple ids. However, once the scan tries to access the corresponding values in the tuple buffer, we must ensure that the scan can never encounter values overwritten by concurrent inserts. This is especially important for string values with externally stored data, such as the *content* value of Figure 3.4, where trying to access an invalid value could lead to a segmentation fault. Figure 3.5 depicts our scan strategy.

First (1), we find the selected range in the buffer based on the tuple ids and prevent the deletion of the corresponding string region through reference counting. Following (2), we copy all fixed-size tuple data in the range into

a separate buffer residing in the scan operator. After copying all tuples in the range, we check to see if concurrent writes have overwritten any tuples that we have scanned (3). In case of an overlap, we cannot guarantee that all tuples in the scan buffer are the desired tuples and, thus, have to abort the scan. If all scanned tuples are still valid, we report them to the downstream operator of the pipeline (4). At this stage, the first downstream operator materializing the tuples relocates the out-of-place content for strings into query memory, thereby protecting them from deletion and preventing segmentation faults in case of concurrent inserts. After all tuples of the scan range were processed by the downstream operators of the scan's pipeline, we release our reference to the relation's string buffer, freeing the corresponding memory region if we held the last reference.

3.3 Evaluation

Having outlined our relation-based approach to in-database stream processing, we evaluate its performance against two popular dedicated stream processing engines, Apache Flink [29] and Apache Spark [179]. We focus our evaluation on data ingestion rates, scalability, and performance on a mix of simple stream aggregation and complex analytical queries.

3.3.1 Setup

We perform all experiments in this section on a server equipped with 256 GB DDR3 main memory and an Intel Xeon E5-2660 v2 CPU with 28 physical cores. All data for the experiments is stored on a Samsung 970evo NVME SSD. Results reported in this section are based on the geometric mean of 5 runs taken after 2 warmup runs.

Workload. We base our experiments on the TPC-H benchmark [2] at scale factor 100. To transform TPC-H to a stream analytics workload, we consider the largest relation by far, *lineitem*, to be a stream. All other relations are considered durable and materialized at the start of a query. Consequently, we only include queries with exactly one scan of the lineitem relation in our experiments. Due to issues with Flink, we had to remove queries 5 and 8 from our benchmark. We do not print the query result in any of the systems.

Flink. We implement a standalone Apache Flink executable based on Flink version 1.6.1 and express all relations using the batch table API based on external CSV data. Further, we allow Flink's optimizer to re-order joins for query processing using the `TABLE_OPTIMIZER_JOIN_REORDER_ENABLED` flag. We submit

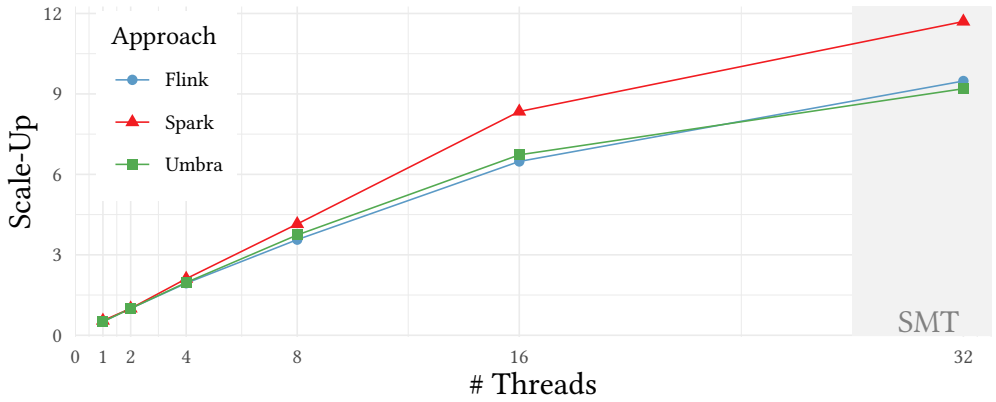


Figure 3.6: Scalability of stream insert processing for all three systems, normalized to each system's single-threaded performance.

all queries to Flink using its SQL interface. All intermediate results that Flink requires for processing are located in an in-memory file system.

Spark. We implement our TPC-H-based workload in Spark version 3.3.2. All queries are expressed using the Spark SQL API on external CSV data frames. Jobs are submitted to a local standalone spark cluster using `spark-submit`.

Umbra. We implement a streaming relation in Umbra as outlined in the previous section. Further, we create all relations except for *lineitem* as durable relations in Umbra. Lineitem is created as a stream using the interface described in Section 3.2.1. We subtract the session window timeout from Umbra's query runtimes as the system is idle during this period. Both Flink and Spark are run in non-windowed configuration and are, thus, not introducing similar delays.

3.3.2 Stream Ingestion

As a first experiment, we examine the data ingestion rate offered by the three systems. For this, we fully insert the lineitem relation once into each system. As Spark and Flink rely on pull-based semantics for efficient analytical queries and do not offer full support for push-based inserts, we express inserts to them as `SELECT COUNT(*) FROM lineitem` queries. For Umbra, we use the bulk insert command `COPY lineitem FROM CSV`.

Scalability. First, we compare the scalability of the systems. Scalability is crucial for data stream processing, as the volume of streams can only be handled efficiently on many-core machines and in clusters. Figure 3.6 depicts the scalability along the number of threads for each system, normalized to the respective single-threaded performance. One can see that all systems scale well to the available physical cores. However, once the systems reach simultaneous multi-

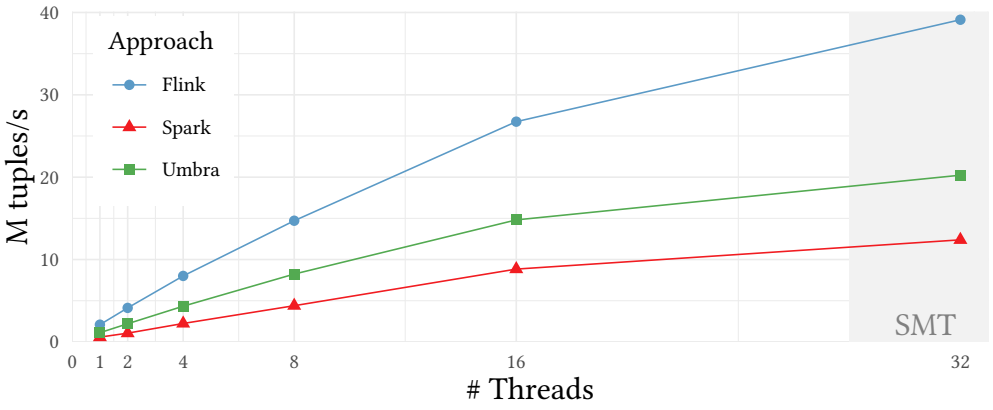


Figure 3.7: Insert throughput for processing the lineitem stream in Umbra, Flink and Spark in million tuples per second.

threading (SMT), scalability reduces as threads compete for the same physical resources. While Flink and Umbra scale almost identically, Spark manages to outperform them in this regard consistently.

Insert Throughput. In addition to scalability, we investigate the insert throughput for all approaches. Figure 3.7 shows the insert performance in millions of tuples per second along the number of insert threads. While Spark offered the best scale-up of all systems, it is consistently outperformed by both Flink and Umbra when considering the achieved insert throughput. Overall, Flink offers the best insert performance, outperforming Umbra by a factor of 1.9 for 32 threads. This advantage can be attributed to the slight difference in the semantics of our insert queries. Our approach has to fully process all columns to materialize them in the ring buffer. In contrast, Flink can only count the number of rows without parsing them entirely. While we expect this advantage to disappear for more complex queries where multiple columns must be parsed, it is very beneficial for such simple workloads.

3.3.3 Query Performance

Having analyzed the data ingestion capabilities of all approaches, we now focus on their analytical capabilities. For this, we run a combined workload of the twelve TPC-H queries selected for our benchmark, running each query once. As Umbra has to process inserts and queries in parallel, we require at least two threads per run. For Umbra, the specified number of threads is the total number available to the system, which must be shared between query and insert processing. We concurrently schedule two queries in Umbra, one inserting

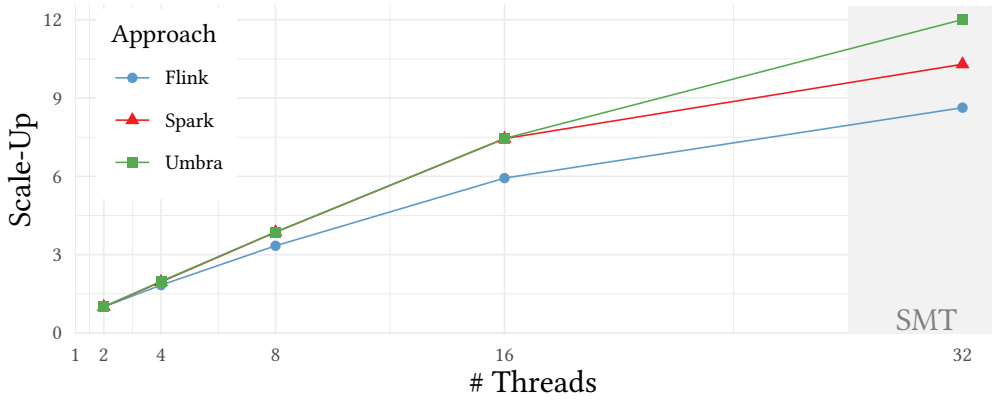


Figure 3.8: Scalability of stream analytics processing for all three systems, normalized to each system's performance for two threads.

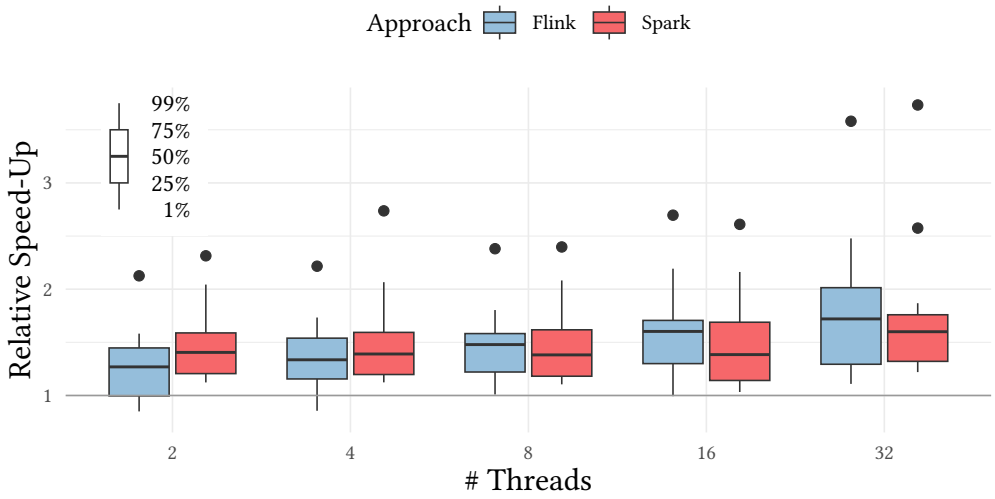


Figure 3.9: Relative speed-up of Umbra over Flink and Spark for analytical query processing along the number of worker threads.

the lineitem stream from CSV and another evaluating the TPC-H query on the inserted stream.

Query Scalability. We first investigate the scalability of the three systems and show the results in Figure 3.8. Scalability is reported over the geometric mean of all queries. Again, all systems scale well to the number of physical cores, with Umbra and Spark scaling slightly better than Flink. Umbra demonstrates the best overall scalability, outperforming Spark once the approaches employ simultaneous multithreading.

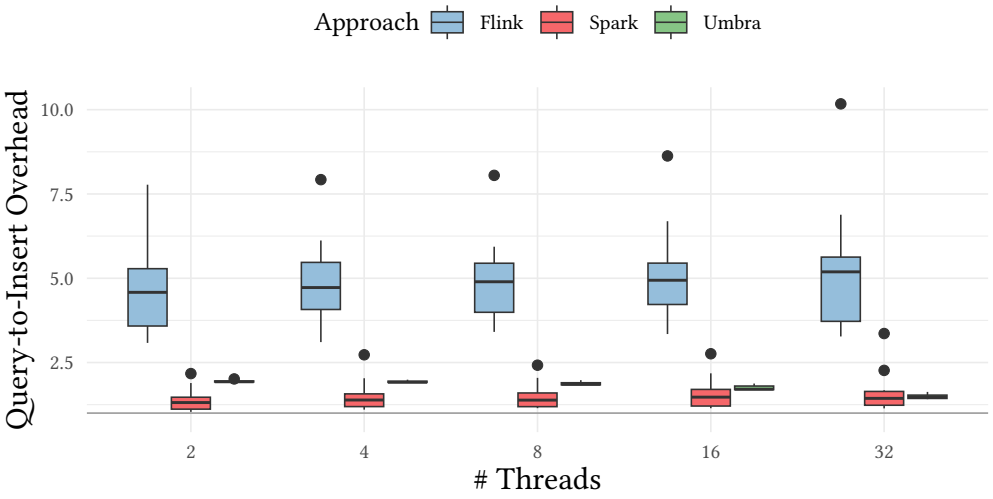


Figure 3.10: Overhead of queries to data ingestion for Flink, Spark and Umbra along increasing thread count.

Relative Query Performance. Focusing on query performance, we examine the relative performance of Spark and Flink compared to Umbra. Figure 3.9 depicts the resulting speed-up of Umbra for an increasing number of threads. Even though Umbra has to split the available workers between insert and query processing queries, it consistently outperforms both Flink and Spark, independent of the number of available worker threads. Furthermore, we see that the advantage Flink had when ingesting data into the system does not transfer to query analytics, where multiple columns have to be parsed. Furthermore, we can see the trends from the query scale-up: The relative speed-up of Umbra over Spark stays constant until SMT is reached, whereas Flink's relative performance degrades with increasing thread count.

Overhead over Stream Ingestion. Finally, we want to investigate the overhead that queries introduce in the system on top of the work for data ingestion. Figure 3.10 shows the relative overhead that evaluating our TPC-H-based benchmark creates for each system. The overhead of Spark is nearly constant. However, we see two contrary developments for Flink and Umbra. While the overhead of Umbra decreases with increasing thread count, Flink's queries scale worse than its inserts. The drastically higher overhead for Flink confirms our assumption of Section 3.3.2 and indicates that Flink heavily optimizes for the `count (*)` query that we used to emulate inserts. For Umbra, we expect the observed overhead as threads are divided between queries and inserts. However, more fine-grained scheduling could further reduce the overhead.

3.3.4 Discussion

All three systems demonstrate excellent scalability for both insert and query processing, which is necessary to analyze large-volume data streams on modern hardware and in cluster settings efficiently. While Umbra consistently outperforms dedicated stream processing engines for complex analytical queries, Flink demonstrates an advantage for simple stream aggregations such as count star. This performance advantage can be expected, as Umbra has to fully parse tuples to store them in the ring buffer for analysis. However, it also indicates the need for full on-arrival processing of such queries, which we investigate in the next chapter of this thesis. Once queries require filter predicates and aggregates on individual columns or access to durable relations, the advantages of using a database system for stream processing become amply clear. In addition, the overhead of insert processing can, in theory, be amortized by attaching multiple concurrent queries to a single data stream. However, this requires careful scheduling to avoid starving either insert or query threads.

3.4 Summary

In this chapter, we design a technique for relation-based stream processing in relational database systems. Relying on a ring-buffered relation for stream processing provides high ease of integration for existing database systems, enabling database systems to handle stream-enrichment queries combining transient with durable data. To demonstrate the applicability of this relation, we integrate it into the code-generating Umbra database system.

Using the implementation within Umbra, we show the performance of our streaming relation in a number of end-to-end benchmarks against dedicated stream-processing engines. Our approach consistently outperforms dedicated stream processing engines on analytical streaming workloads while requiring only minimal changes to the database system's execution model.

CHAPTER 4

Continuous-View-Based Stream Processing

Excerpts of this chapter have been published in [170].
With contributions from Tobias Schmidt.

Relation-based stream processing, as outlined in the previous chapter, is an excellent solution for stream enrichment in database systems. However, while a relation-based stream integration offers a high ease of integration, its applications are limited to single finite streams with session-window semantics, lacking functionality for more complex analytical workloads. One way databases support long-running queries over multiple inputs is through materialized views. These views abstract from the changes to the database contents, such as inserts, through maintenance strategies, either directly propagating changes for newly arriving tuples or updating the materialized query result on demand in a deferred fashion. These properties make materialized views a great conceptual fit for complex stream analytics in database systems.

Consequently, we propose a new type of view, which we call *continuous view*, for the emerging workload of complex analytics of high-velocity data. These views are split for maintenance between inserts and queries, as outlined in Figure 4.1. By splitting the maintenance work, we achieve the insert rates required for real-time stream processing, while simultaneously supporting fast analytical queries. We integrate continuous views into our newly developed Umbra database system [118]. Continuous views exploit Umbra's state-machine-based approach to query execution to split the query efficiently for maintenance. Inserts perform the first logical steps for each arriving tuple, i.e., the necessary calculations for the first logical query pipeline (red). Only at times when the result of the view query is required are the remaining processing steps for the changes performed (purple).

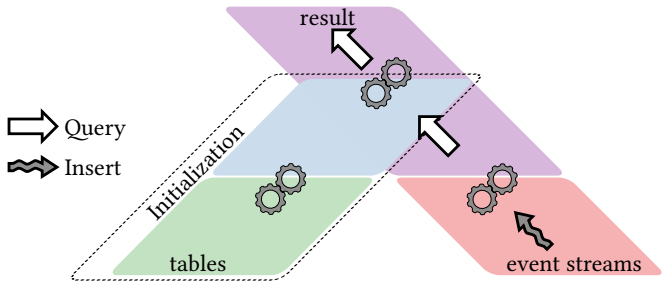


Figure 4.1: Exemplifying split of a continuous view query plan for maintenance. Parts that are evaluated for each insert are marked with a grey arrow. Parts with a white arrow are evaluated for each query to the view. The framed part is evaluated once at view creation time.

In contrast, specialized systems for high-velocity data, often called Stream Processing Engines (SPEs), do not offer functionality to internally manage and modify historic data. Besides, many state-of-the-art SPEs such as Flink [29], Spark [179], and Storm [154] fail to properly utilize the underlying modern hardware, as shown by Zeuch et al. [180].

Modern view maintenance strategies such as higher-order incremental view maintenance (IVM) [87], on the other hand, are optimized for query-heavy workloads. While they offer high view refresh rates to better cope with high-velocity data, they refresh views on each insert and thereby optimize the query performance. By always having the current state of the view available, they trade insert for query performance. However, we argue that these constant full refreshes are not required for many insert-heavy streaming use cases. We will support this argument by using a small contrived example of an Industry 4.0 manufacturing plant, which we will also use as a running example throughout this chapter:

Example: Consider a modern just-in-time car manufacturing plant that aims at keeping parts stored for as little time as possible. Parts are delivered to workstations just when they are needed for production. To avoid costly production halts, a car company might monitor the resupply needs with a query such as shown in Figure 4.2. Usage and resupply of parts at workstations are tracked automatically using the stream relations *part_usage* and *part_restock* for which exemplifying entries can be found in Figure 4.3. The query reports those workstations where parts are running low, indicating problems in the supply chain. It is sufficient to check the current query result each second, or even only a few times per minute, to detect problems in time. Inserts, on the other hand, happen regularly, continuously, and with high velocity and can easily reach the tens or even hundreds of thousands per second.

```
with used as (
  select     part, station, count(*)
  from       part_usage
  group by   part, station
),
restocked as (
  select     part, station, count(*)
  from       part_restock
  group by   part, station
)
select       r.part, s.location, s.supplier
from         station s, used u, restocked r
where        s.id = u.station and
             u.station = r.station and
             u.part = r.part and
             r.count - u.count < 5
```

Figure 4.2: Continuous query monitoring part supply on workstations within a manufacturing plant.

part_usage			
part_id	part	station	worker
1	Wheel	5	4
2	Seat	2	2
3	Seat	2	2
...	...	...	...

station			
id	plant	location	supplier
1	Spartanburg	Gate 2 - EN	+1 555 ...
2	Spartanburg	Gate 1 - SE	+1 555 ...
3	Regensburg	Tor 2	+49 555 ...
...	...	...	...

part_restock			
part_id	part	station	worker
1	Wheel	5	1
2	Seat	2	6
6	Engine	1	3
...	...	...	...

Figure 4.3: Exemplifying *part_usage*, *part_restock* and *station* relations. *station* is considered a static table, *part_usage* and *part_restock* are streams.

Traditional view maintenance algorithms would update the at times expensive deltas for each tuple and refresh the view fully, even when the current result is not required. SPEs, on the other hand, are able to monitor the difference in parts for high insert rates. However, they lack the functionality to automatically combine the result with the necessary information held in databases without any additional overhead. In our example, SPEs would miss details about the station and the supplier in charge which are crucial for the query.

Our integration allows the user to access continuous views in regular database queries, and the underlying continuous queries have access to the database state just like regular views. Further, continuous views can be created using standard SQL. Umbra uses data-centric processing and the producer-consumer model [117] for compilation. Using the same concept for our continuous views, we keep the overhead of stream inserts low to compete with SPEs while simulta-

neously eliminating the need for a full stream or table scan at query time. In the categories defined by Babu et al. [16], continuous views are result-materializing queries with updates and deletes. While some research has been conducted on IVM maintained stream views in database systems [176], and continuous views also exist in open-source projects [124], we are not aware of other work proposing specialized view maintenance strategies for high-velocity stream workloads. In this chapter, we address the following key points:

- We present a novel strategy for view maintenance, especially for views on stream inputs, that divides the work between inserts and queries. Our approach can support extensive analytical queries on streams without having to materialize the full stream input.
- We integrate stream processing using continuous views into our general-purpose database system Umbra [118], using the query optimizer as well as specialized code generation to fully utilize the underlying hardware.
- We describe how continuous views are created from regular SQL statements using only standard operators, thus allowing easy integration into existing systems.
- We demonstrate the capabilities of our system using the AIM benchmark [27], comparing it to state-of-the-art SPEs. We evaluate the performance limits using microbenchmarks and offer a comparison to the scale-up SPE Trill [31] and a database engine implementing higher-order IVM, DBToaster [87], on TPC-H.

The rest of this chapter is structured as follows: In Section 4.1 we give a brief overview of the Umbra system developed by our group, focusing on the aspects relevant for this work. Afterward, we describe our novel split maintenance strategy for continuous views in Section 4.2. Section 4.3 shows the capabilities of our approach against state-of-the-art baselines and on a microbenchmark. In Section 4.4, we discuss relevant related work before summarizing in Section 4.5.

4.1 Background

As stated in the introduction of this chapter, the continuous views are integrated into our Umbra database system [118] and exploit some of its concepts to achieve fast and split view maintenance. While these concepts are outside the scope of this chapter, we briefly describe them in this section to help the reader better understand our approach. We provide a more detailed description of the relevant

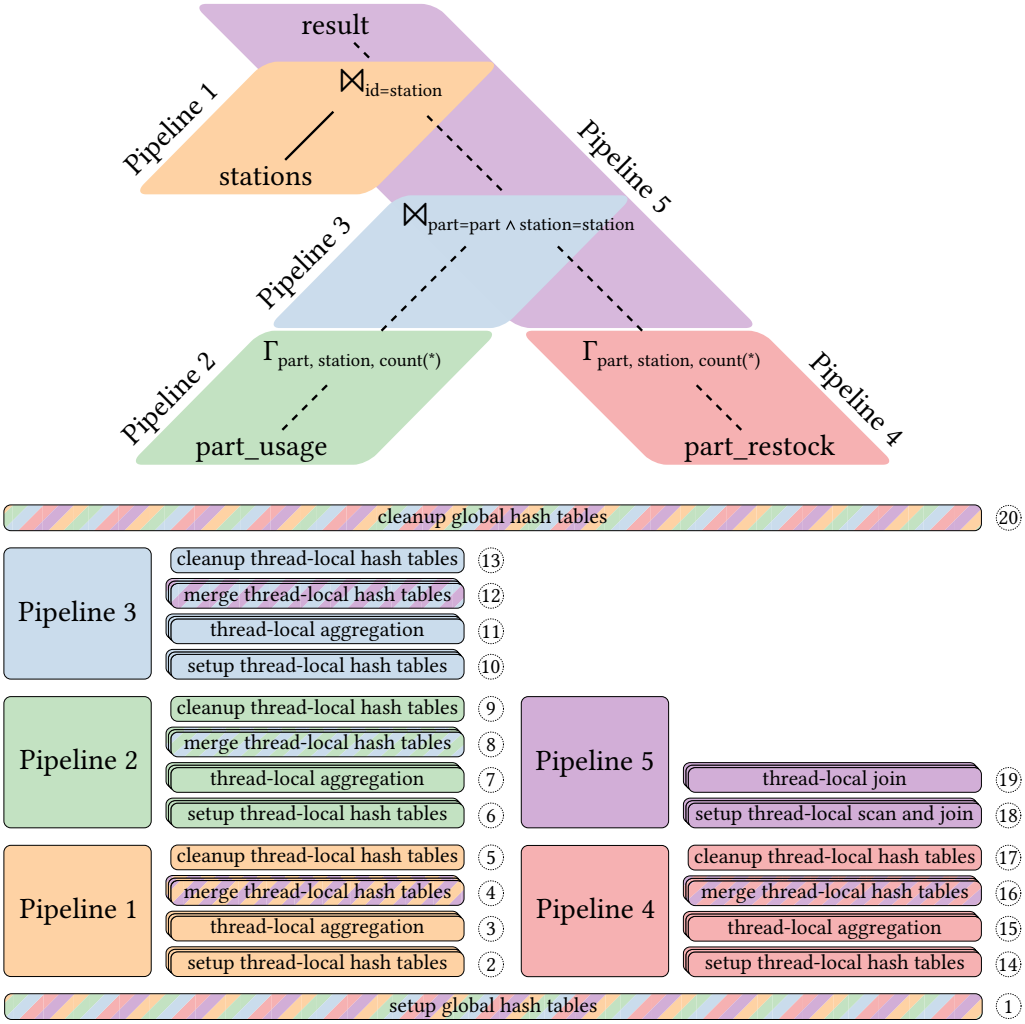


Figure 4.4: Pipelines and corresponding logical steps for the query of Figure 4.2, joining two grouped streams and a durable table. Pipelines containing streams are dashed. Striped coloring indicates state shared between multiple pipelines is used. Colors indicate involved pipelines.

Table 4.1: Steps of the exemplifying query of Figure 4.4, categorized into the step categories defined in Section 2.2.

Category	Steps
Global Initialization	1
Local Initialization	2, 6, 10, 14, 18
Thread-Local Execution	3, 7, 11, 15, 19
Merge	4, 8, 12, 16
Local Cleanup	5, 9, 13, 17
Global Cleanup	2

components in Chapter 2. Like its predecessor HyPer [117], Umbra compiles query plans into machine code. However, in contrast to HyPer, Umbra does not produce a code monolith and instead relies on multiple code fragments to represent a query as a conceptual state machine. Each state of the state machine corresponds to a code fragment. Transitions switch between code fragments and, thus, between different phases of query execution.

We refer to the states and their associated code fragments as *steps*. These steps form the building blocks we use for our split maintenance strategy. All but two steps of a query correspond to a pipeline, a connected path in the logical query plan wherein a tuple does not have to be materialized between operators. An example of such a path can be seen between the scan of *part_usage* and the grouping in pipeline 2 of Figure 4.4. Pipeline 2 comprises the logical steps for evaluating the *used* CTE of Figure 4.2, a simple aggregating group by. Umbra splits these steps into even finer-grained steps and might execute them in a slightly different order. To better illustrate the role of the steps in the execution of the query in Figure 4.2, we show the individual steps for all pipelines in Figure 4.4 and their categories in Table 4.1.

During execution, steps have access to two categories of query state: local and global state. Local state is used for intra-pipeline communication and state-keeping within steps and is not shared between pipelines. It is, however, possible for multiple steps of the same pipeline to access the same local state. Consequently, pipeline steps must be executed in the pre-defined order. Each worker thread has its own copy of the local state, avoiding contention between workers. The local state is designed to be inherently thread-local. In contrast to the state model for regular queries outlined in Section 2.3, we do not re-use the memory for local state for different pipelines and instead allocate memory for each local state. Keeping the local states of all pipelines separate allows us to have multiple active pipelines per query, which we utilize for our maintenance strategy. The global state is used to share information between pipelines, and

all threads working on a query have access to the same instance of the global state. Therefore, modification of the global state within parallel steps, e.g., when merging thread-local hash tables, requires data-dependent synchronization.

4.2 Approach

Traditional approaches to materialized view maintenance fully process tuples. In deferred maintenance, all tuples are either buffered in memory or stored on disk. When the view result is requested, the database will reevaluate the query from scratch, requiring it to scan the materialized relation. All tuples, therefore, need to be materialized and kept accessible at a moment's notice. This is problematic for unbounded high-velocity data streams, which are often desired to be processed in an exactly-once non-materializing fashion.

For our system, a full materialization of the stream would either mean writing it to disk, or keeping it in memory. The first option would increase the disk IO tremendously, affecting other disk operations for other queries. Keeping the tuples in memory, on the other hand, reduces the memory available for query processing. Both could dampen the overall system performance, which means that we therefore have to rely on the inserter to handle the tuple.

Other approaches processing new tuples at insert time, like eager and incremental view maintenance, also avoid the high storage cost of stream data. However, they propagate the full change immediately. This is not trivial at the high frequency required for stream processing, and most systems require hand-written queries to handle the updates. Even modern high-velocity approaches tackling this problem without manual user input, like DBToaster, require specialized operators with support of deletes and updates at any part of the query. This makes it hard to integrate this approach into an existing database system efficiently. For example, DBToaster only exists as a stand-alone solution or as an external library, not fully integrated into a database system.

4.2.1 Split Maintenance Strategy

Our approach can be seen as a combination of eager and deferred view maintenance, providing the best of both worlds. To keep the introduced overhead low, we propose processing inserts only as far as needed. In general, this means we want to process the input until the point where it has to be materialized for the first time, that is, the first pipeline breaker. This allows us to perform initial processing steps at insert time while reducing the memory consumption compared to deferred maintenance. Using pipeline breakers as a natural storage point also allows for easy integration. Tuples are never invalidated at pipeline

breakers, e.g., join build sides. After materialization in pipeline breakers, the remainder of the query is oblivious to the nature of the input. Therefore, we do not require specialized operators that support removing or updating tuples at any given time, as is needed for incremental view maintenance.

Further, in contrast to deferred view maintenance, we never need to materialize the full stream inputs. This greatly reduces the storage cost of stream processing. In the *used* CTE of Figure 4.2 in pipeline 2 of Figure 4.4, e.g., we would insert the tuple into the hash table of the grouping operator and update the aggregate accordingly. While in this simple query the overhead of updating the result tuple is negligible, this is not the case for more complex queries.

Consider, e.g., the query plan for the full query displayed in Figure 4.4. The query still is rather simple; however, fully processing new tuples in this query is not. Inserts into pipeline 2 would trigger a recalculation of the join in pipeline 5. Since we do not inherently support updates in and deletes from join build sides in our system, the other event stream would have to be fully reprocessed. This basically requires us to run steps (7) to (19) for every insert if we process all tuples fully. Once the tuple is materialized within the first pipeline breaker, finalizing query processing when the query result is required is possible by running the remaining steps for the other pipelines. We will use the query in Figure 4.4 as a running example throughout this chapter, assuming *stations* to be a regular database table.

This approach benefits from Umbra's novel state machine approach to query processing wherein we can stop and delay processing after any of the steps enumerated in Figure 4.4, and even run multiple input pipelines independent from one another. We use the remainder of this section to outline the supported queries, the different phases of the maintenance, and the integration into regular database query processing.

4.2.2 Supported Queries

Similar to SPEs, our system restricts the queries it supports to protect the user from undesired side effects, such as state explosion and ambiguous query results. In the following section, we describe the rules for queries in Umbra. Moreover, we offer an overview of the supported queries in other systems.

Umbra

For Umbra, Figure 4.5 visualizes and summarizes the most important rules that we motivate in the following section.

Input Pipelines. We require the first pipeline breaker of every stream input to be a grouping operator, as shown in the first row of Figure 4.5. This reduces

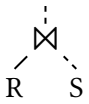
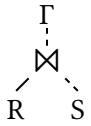
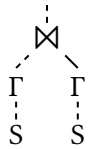
	Infeasible	Supported
Stream-Relation	 Unaggregated stream in output	 All streams are aggregated
Stream-Stream	 Unaggregated stream-stream join	 Join of aggregated streams

Figure 4.5: Supported and infeasible queries for continuous views. Streams are marked *S* and regular relations *R*. Pipelines containing streams are dashed. Subtrees containing no streams are not restricted.

the risk of state explosion compared to allowing arbitrary pipeline breakers. Most pipeline breakers, like sorting or join build-sides, would still materialize at least parts of the stream leading to memory shortages that would impact the performance of the system. As the grouping itself is mostly oblivious about its input, it is still possible to query the entire stream by grouping for a key if this is desired by the user. This would, however, lead to the entire stream being materialized.

Transactions. For us, streams are inherently not transactional. Keeping streams in a transaction would mean that once an analytical query is started within another transaction, newly arriving stream tuples would not qualify for this query. The user, however, will always expect to see the current state of the stream, independently of the current transaction. We still want to isolate the continuous view from changes to regular tables. This isolation is necessary to guarantee that previously reported join results are not invalidated by later arriving tuples. Consider, e.g., a join as in the first row of Figure 4.5. Removing a row from the relation on the left-hand side would mean that results reported for that tuple as join partner are no longer valid. This would lead to ambiguous and even inconclusive results for the user, which we want to prevent. To achieve this isolation without requiring transaction handling, a continuous view reads all committed changes at creation time and uses exactly this state of the database throughout its lifetime. This isolation furthermore allows us to run pipelines not involving streams only once and keeping the results cached. In essence, this means views read the committed state of all relations once at creation time [119].

Joins. Streams are often problematic for joins, and most SPEs, therefore, restrict joins in some way. Joins of unbounded streams without windowing can lead to a state explosion as both sides would need to be fully materialized to ensure join correctness. While there are viable solutions for this problem in many SPEs, in the form of windowed joins, we want to allow queries spanning entire streams.

We plan to enable windowed processing based on algorithms for persisted data [101] in a future version. Since even non-blocking joins like the XJoin [155] are not designed to handle unbounded inputs, we ensure that streams are on the probe side if only one input depends on a stream. This way, we never have to materialize the stream and can simply probe the pre-calculated build side. For stream-stream joins we utilize the restrictions mentioned above, forcing streams to be grouped. We further extend this restriction and require both inputs of a stream-stream join to be grouped as seen on the bottom of Figure 4.5. When joining only previously grouped streams, we can ensure that we do not report join results that would later be invalidated by new tuples arriving on either side, again preventing inconclusive results in the view. We do not restrict joins between regular tables.

Other Approaches

To better illustrate how the rules motivated above compare to existing systems, Table 4.2 provides an overview of natively supported features of similar stream processing approaches. We group the approaches into specialized SPEs and stream processing in the context of databases. The table is based on features described in the documentation of the specified version. While some of the described systems have additional restrictions and features, we believe the table provides a good overview of those most important.

One can see that Umbra offers an extensive functionality, especially for joins, second only to DBToaster. Further, there is a notable difference in the focus of the systems. While the SPEs focus on windowed queries with limited join options, the in-database approaches offer only basic or manual windowing. However, they support more complex queries and even manage historic data internally. Umbra's restriction of joins only applies to ungrouped streams and can be bypassed, e.g., by grouping for a key. On the downside, this will likely lead to a performance decrease. Information in streams has to be condensed for analysis, and grouping is a common way to achieve that. Therefore, we argue that requiring the grouping of streams does not gravely limit the applicability of our approach. Queries without grouping or aggregation, i.e., map-like or filtering queries popular for SPEs, are also possible within Umbra through the relation-based stream processing technique outlined in Chapter 3.

Table 4.2: Comparison of stream processing approaches. Dashes indicate the system's documentation either directly states a feature is not supported, or it does not contain enough information to indicate support.

Aspect		Stream Processing Engines					In-Database Stream Processing		
		Flink [29]	**Spark Structured** [179]	**Storm Trident** [154]	**Trill** [31]	**Saber** [90]	**PipelineDB** [124]	**DBToaster** [87]	**Umbra**
Version		1.10	3.0.0	2.2.0	2019.9.25.1	7be036c	1.0.0	2.3	-
Stream Deletes and Updates		-	-	-	-	-	-	Yes	-
Early Results		Limited	Limited	Windowed	Yes	Windowed	Yes	Yes	Yes
Historic Data		External	External read-only	External read-only	External read-only	-	Internal w/ Postgres	External read-only	Internal
Scale-Up		Yes	Yes	-	Yes	Yes	Limited	-	Yes
Scale-Out		Yes	Yes	Yes	-	-	-	-	-
Aggregates		Extensive	Basic	Extensive	Extensive	Basic	Extensive	Basic	Extensive
Windowing		Built-in support	Built-in support	Built-in support	Built-in support	Built-in support	Built-in support	Using GROUP BY	Using GROUP BY
Stream-Joins	Equi	Yes	Yes*	Batchwise	Yes	Windowed	Yes‡	Yes	Yes‡§
	Theta	-	Yes*	-	-	Windowed	Yes‡	Yes	Yes‡§
	Outer	Equi	Windowed*†	Batchwise†	Yes†	-	Yes†‡	-	Yes†‡§
	Semi	Yes	-	-	-	-	Yes†‡	Yes	Yes†‡§
	Anti	Limited	-	-	Yes†	-	Limited†‡	Yes	Yes†‡§

*Only unaggregated streams †Single direction ‡No stream-stream joins §Restricted only for ungrouped streams

4.2.3 Query Planning

Like all other tables and views, continuous views are created using the regular SQL interface. This means we have to enforce the aforementioned restrictions in the semantic analysis of the statement and reshape the query plan according to our needs. As a first step, we translate the query representing our continuous view into an unoptimized logical query plan. Performing a depth-first traversal of the query plan, we remember for each operator whether its input contains a stream, and if so, whether there is a grouping in between. Given this mapping we modify the query plan in a second traversal, taking steps for three operator types:

Group By. We again keep track of groupings, but this time we remember if there is a grouping higher up in the tree. Each time we encounter a stream input we ensure that its parents contain a grouping, thus verifying we do not store ungrouped streams.

Order By with Limit. Contrary to stream inputs, we ensure that the parents of sorting operators do not include a grouping. As we require stream inputs to be grouped somewhere, we know that there is a grouping below. By doing so, we avoid sorting ungrouped streams. Sorting operators without a `LIMIT` clause are simply dropped per the SQL standard. As Umbra, like many other systems, does not guarantee tuples to be processed in scan order, scanning the materialized result at query time will anyway lose the order. Therefore, if a sorted result is desired, it has to be requested when querying the materialized view.

Join. Joins require the most intrusive modification of the query plan. These depend on the order and stream containment of the join inputs. We say an input pipeline *contains* a stream if there is a stream input somewhere upstream of the pipeline. If no input contains a stream, we leave the join unmodified. When both inputs have a stream, we ensure that both streams have a grouping operator between the join and the stream input using the pre-calculated containment map.

In cases where only one input contains a stream, we try to modify the query plan to have the stream on the probe side of the join, independent of grouping. This allows us to later pre-calculate the build side and rerun the join for changed input with little overhead. While this is easy for simple uncorrelated joins independent of the join type, switching the input is not always possible. For correlated joins, e.g., only some cases can be unnested in a way that supports switching the inputs. When switching is not possible, we again simply reject the view.

As the query of our running example is too simple to show all these modifications, we show them in a more complex plan in Figure 4.6. As a first step, the order by without a limit is removed (1). Afterward, we modify both joins

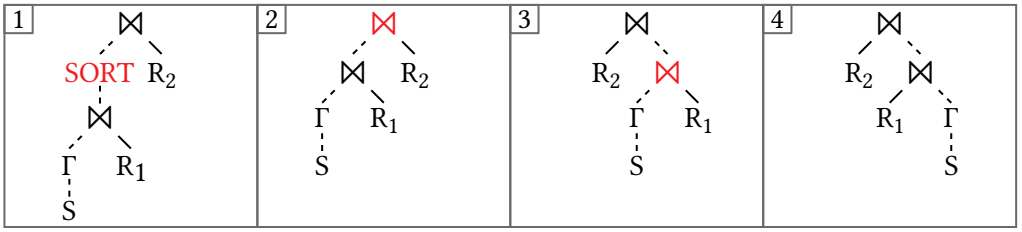


Figure 4.6: Modifications performed on a query plan prior to compilation. Streams are marked *S* and regular relations *R*. Pipelines containing streams are dashed. Red color marks the currently modified operator. Removing unused sort [1], moving streams to the probe side of joins ([2] and [3]) top down, and finished plan [4].

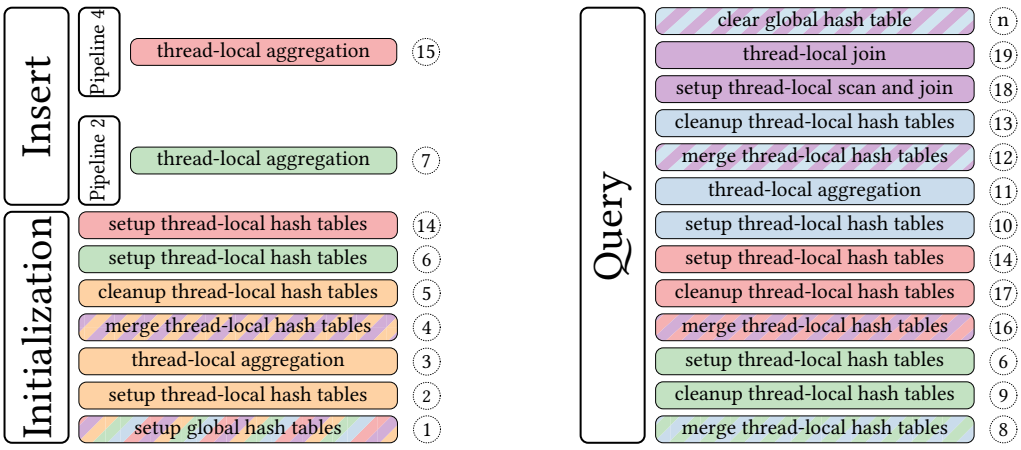


Figure 4.7: Generated callbacks for initialization, insert handling, and query evaluation for the running example. Colors, steps and step numbers are consistent with the query plan shown in Figure 4.4. (n) denotes a new step required for rebind.

to have the streams on the probe side ([2] and [3]). Note that in [3], the join is modified even though there is no direct stream input, only one upstream through the other join. After enforcing the aforementioned restrictions, we optimize the query plan using Umbra's optimizer, ensuring the restrictions are not violated by optimizations.

4.2.4 Code Generation

After adapting the logical query plan for a continuous view, we look at the code generation for the identified phases of maintenance. As outlined in Figure 4.1, we distinguish three different tasks: The view initialization, handling stream

inserts, and query evaluation. Each of these tasks consists of individual steps of pipelines, like those described in Figure 4.4. In general, each pipeline of a query plan is mapped to exactly one of these tasks. In the following sections, we will refer to the pipelines as follows:
Static Pipeline: A pipeline that has no stream input, either directly or through one of its child pipelines (pipeline 1 in our example). These pipelines' steps are handled by the view initialization.
Stream Pipeline: A pipeline that directly processes a stream input (pipelines 2 and 4).
Upper Pipeline: A pipeline that has an indirect stream input, used for query evaluation (pipelines 3 and 5).

To better illustrate the generated callbacks discussed within this section, we summarize the steps involved in Figure 4.7.

Initialization

After having established the fundamental logical components of each pipeline, we describe which steps have to be run at view initialization, i.e., at view creation time. Based on a query plan modified according to the rules described in Section 4.2.3, we can prepare the view. The initialization consists of two main tasks: **(1)** processing static pipelines, and **(2)** preparing all stream pipelines for inserts. Below, we describe these steps in theory and using our running example.
(1) As discussed before, we do not want changes to the view's table inputs to be propagated to the view. Therefore, we can calculate the results for all pipelines affected only by non-stream inputs exactly once. By doing this at view creation time, we immediately gain independence from the underlying database tables as we cache the results in memory. As a first step, we generate the code for the global setup (1). In the following step, we handle all steps of static pipelines. One can easily see that static pipelines always form subtrees in the views query plan, with the root being the operator where the pipeline intersects with a stream or upper pipeline (pipeline 1). Each of these subtrees is handled as it would be in regular query processing, in a left-to-right bottom-up fashion. For our example, this means generating code for steps (2) to (5).
(2) In addition to executing static pipelines, we also initialize all stream pipelines. Delaying pipeline initialization to the insert callbacks would drastically increase the insert time for a few tuples. For regular queries, Umbra ensures that there is exactly one worker thread accessing a local state instance. Currently, we cannot keep this strict worker-to-state mapping for performance reasons, and therefore have to be aware of possible race conditions. Consider, for example, a second insert callback arriving at a pipeline that is currently being initialized. At this

point, an insert might be performed for a not fully allocated hash table, resulting in a lost tuple, or worse, a corrupt memory access. In our example we have two stream pipelines, pipelines 2 and 4, and therefore generate code for steps (6) and (14).

Insert Handling

Handling inserts is the most performance-critical part of the view's maintenance, as this part is performed for every single tuple upon arrival. Therefore, we want this step to execute as little code as possible by only running the thread-local execution. We modify the local execution code slightly to process single tuples instead of morsels. This way, we can later call this step in a callback fashion for each inserted tuple. However, as inserts in Umbra are processed morsel-wise, this will still lead to morsel-wise processing of updates for larger inserts. The callback will then be triggered for each tuple within the morsel. These callbacks are generated for each stream pipeline independently. As we have two stream inputs for our example in Figure 4.4, we also generate two callbacks consisting of steps (7) and (15) respectively.

Query Evaluation

Finally, we need to generate the code to combine the cached static pipelines and the dynamic stream pipelines to obtain the query result. This step is again composed of two main parts: **(1)** processing all upper pipelines, and **(2)** resetting the view's internal state to be ready for the next query.
(1) As a first step, we want to execute the merge phase of all stream pipelines. This makes all locally held results available for the upper pipelines using the global state. Additionally, we reset the stream pipelines by running the local cleanup and initialization. This way we make the local state available for new inserts as fast as possible. For our running example this translates to steps (8), (9), and (6), as well as (16), (17), and (14), in that order. After these steps have been completed, we can run all upper pipelines as if they were a separate query. In our running example, this includes pipelines 3 and 5, and therefore, steps (10) to (13), as well as (18) and (19). Finally, we generate code to store the view's query result, similar to a materialized view. For one thing, this allows us to compare different versions of the view with one another. Furthermore, slightly outdated versions of the view can then be read without having to run the query evaluation if requested.
(2) After materializing the view's current result, we have to reset the internal state. We cannot, however, clean up and reinitialize the global state as we did for the local state. The global state still holds all cached results for static and

Algorithm 4.3 Continuous view insert handling

```
1: function processInsert(ContinuousView v, Queue q, Mutex m, Tuple t)
2:     if m.lock() successful then
3:         if t2 ← q.dequeue sucessful then
4:             /* execute matching insert callback of Figure 4.7 */
5:             processTuple(t)
6:             v.hasNewInput ← true
7:             m.unlock()
8:     else
9:         q.dequeue(t)
```

stream pipelines, which would be lost at a full reset. For our example, we would want to avoid rebuilding the build side of the join between pipelines 1 and 5. Most operators already provide some functionality for such a reset in the form of an operator rebind, normally used for recursive queries. We can safely rebind all operators that exclusively belong to upper pipelines, i.e., all except those that are top-level operators of stream pipelines and static pipeline trees. Further, we reset all join operators that have a stream pipeline on the build side, e.g., the join between pipelines 3 and 5 (n in Figure 4.7).

Resetting states is necessary to prevent tuples from being processed multiple times for the same query: In between queries, all tuples are held in local hash-tables, such as the grouping of the *used* CTE of our example. If we do not clear the local hash table after a query, the tuples would be included in both the local and global state at the same time. The next query would then again merge the local state into the global state and, thereby, include the tuples twice.

4.2.5 Runtime Integration & Optimizations

Now that we have the required code fragments to handle continuous view maintenance, we have to integrate them into our database runtime. The initialization is executed once at view creation time. Next, the view is prepared to handle inserts through the aforementioned callbacks. We register the callbacks with each stream input. From there on out, each insert into the stream triggers the callback and handles the local processing of the tuple. Queries to the view work in a similar fashion: Each query triggers the materialization callback and writes the results to a temporary table. Materializing the results is necessary to have a consistent state of the view within a query, e.g., for self-joins. This materialized result is scanned for the actual query processing of the database as if it were a regular table.

The described integration is limited in two ways: First, local processing is not thread-safe as multithreaded operations on the same instance of the local state were not intended for Umbra. Without controlled access to the local processing, i.e., through a lock, we could experience race conditions for some operators. Second, query processing resets the local state of all stream pipelines. Hence, querying requires an exclusive lock to prevent inserts during query processing. This lock, however, can be released as soon as the local states have been reset. Preliminary experiments showed that acquiring the lock is often costlier than the insert itself, and delays to inserts are mainly introduced by queries. To reduce these delays, we introduce a lock-free queue in front of the local state. Newly arriving tuples are buffered in this queue when either a query or another insert is blocking the local state.

Insert Handling. Algorithm 4.3 describes the modified insert handling: We again first try to obtain the lock for the view. If another thread holds the lock, we enqueue the tuple in the buffer queue and retry for the next tuple to be inserted. As there can be arbitrarily large gaps between queries, the queue could grow indefinitely when we empty it only at query-time. Therefore, we need to empty the queue between queries. As we do not want to integrate a dedicated background worker within our system, which would introduce scheduling overhead, the queue is emptied by inserts. Once an insert has acquired the lock, it additionally dequeues tuples from the queue and processes them. While dequeuing a single tuple proved sufficient in our experiments, dequeuing multiple tuples or the whole queue is possible as well.

Query Evaluation. We consider all previously inserted tuples for queries. Therefore, we empty the queue whenever a query arrives. To prevent race conditions from inserts, we redirect all tuples to the queue until the local states of stream pipelines have been reset. If no tuples have arrived since the last materialization, the processing is skipped, and the last materialized result is used instead.

4.2.6 SQL Interface

After having described the inner workings of our continuous views, we briefly show how they are created using the SQL interface of our database system. To minimize the necessary changes in the SQL dialect all required operations can be performed using regular statements.

Creating Streams. As a first step, the user needs to create the streams that will be evaluated in a continuous view. Streams can be created as *foreign tables*:

```
CREATE FOREIGN TABLE stream_name (...)
SERVER stream
```

We restrict the keyword *stream* for all foreign tables so that no server named *stream* can be created.

Creating Continuous Views. After the required streams have been created, users can create continuous views using simple `CREATE VIEW` statements:

```
CREATE VIEW continuous_view_name AS query
```

During the semantic analysis of the query, we check whether any of the queried relations is a stream. If so, we automatically create a continuous view, else we create a regular view. All modifications and checks described in Sections 4.2.2 and 4.2.3 are only performed for continuous views.

Inserting Data. Inserts can be expressed as regular `INSERT` statements, and streams can be copied from CSV using `COPY`:

```
INSERT INTO stream_name {VALUES (...)| query}
```

```
COPY stream_name FROM 'filename' CSV
```

`query` can be an arbitrary SQL query that does not contain the stream itself. It is possible to attach new continuous views and queries to streams that are currently fed with data, but those only see tuples arriving after their creation.

4.2.7 Updates

Up until this point, we only discussed inserts into streams, not changes to the static tables involved. Both updates and deletes to tables exclusively used in upper pipelines, like *stations*, can be realized at any time. To incorporate changes to these tables, we simply reevaluate all static pipelines affected. For our running example, this means rerunning all steps in the initialization callback that correspond to pipeline 1 (2 to 5) and replacing the build side of the topmost join with the new stations. The next time the view is queried, *all* stream tuples are then evaluated using the changed tables, guaranteeing consistent results for every materialized state.

Changes to static tables and subtrees joined directly with stream pipelines (e.g., top right of Figure 4.5) are not supported. In order to support changes to such tables, we would have to either (a) keep all stream tuples materialized to reevaluate the join correctly, or (b) join stream tuples with the state of tables at arrival time. Case (a) would lead to extensive memory consumption and high refresh times, which is exactly what we aim to avoid with our continuous views. While case (b) is used in some systems, like PipelineDB, we refrain from using this approach. The results of the view query would otherwise be dependent on the processing order of stream events and table updates, leading to inconclusive

and inconsistent results. Like many other SPEs (c.f. Table 4.2), we consider streams to be append-only. Therefore, we do not support deletes or updates of stream tuples.

4.2.8 Fault Tolerance

While it is not the focus of this chapter, we want to briefly address fault tolerance. For continuous views, we can utilize the integration into Umbra, and the fact that we use the SQL interface to interact with the database. This way, we have access to Umbra's logging mechanism and can replay not fully processed tuples in case of an error. However, replaying the full stream in case of an error can lead to a considerable delay during recovery. To reduce the number of tuples to be replayed, we can combine this with checkpoints which are common in SPEs for recovery, e.g., in Flink. Here, we can utilize our custom memory management and the separation of global and local state. Global state is only modified during materialization and, therefore, captures a consistent state of all operators between queries. We can take regular snapshots of the global state as checkpoints and restore the last snapshot in case of a failure. This way, only tuples that arrived since the last materialization have to be replayed.

Another aspect we want to mention is how our strategy can deal with a high load. Many SPEs utilize publish-subscribe-like asynchronous interfaces to accept data, allowing them to delay processing in case of a high load without influencing the inserter. The interface of our continuous views, on the other hand, is SQL-based and, therefore, synchronous. We offer some load balancing in the form of the lock-free queue described in Section 4.2.5. The queue can help overcome short spikes, but, for a prolonged high load, it can grow indefinitely. This can lead to a decrease in the overall system performance. As we do not consider load shedding, which would mean losing information, Umbra should not be used in a completely standalone fashion when an extremely high load is expected for a long time. Instead, we envision it integrated into an ETL scenario with an external data collection engine, as described by Meehan et al. [109].

4.2.9 Portability

While our described implementation of continuous views is optimized for Umbra's execution engine, our approach is in no way limited to Umbra. Continuous views, as described above, can be realized in many database systems using stored procedures, auxiliary tables, and materialized views, albeit less optimized than our fully integrated approach. To demonstrate the feasibility of such an integration, we implemented continuous views in PostgreSQL using its procedural

languages.[1] Our implementation parses view queries and generates SQL statements for keeping continuous views up-to-date. When inserting new tuples, designated insert functions update the aggregates instead of materializing the stream.

As in Umbra, we distinguish between static, stream, and upper pipelines. Static pipelines are evaluated and cached during initialization. For stream pipelines, we generate insert functions that incrementally update the previous results in an auxiliary table. A single view combines all upper pipelines and makes the query result available to users on refresh. In contrast to eager maintenance, the update logic for our continuous views can be generated automatically.

4.3 Evaluation

First, we evaluate our approach for multiple parallel streaming queries using the AIM benchmark, comparing it to an SPE and to an in-database solution. To show real-world performance, we use full query round-trip times on the AIM benchmark. To highlight the raw analytical performance, we further evaluate it against competitors on a TPC-H workload modified for streaming in Umbra and in PostgreSQL on isolated queries. Subsequently, we study internals of our maintenance strategy, such as the load balancing capabilities or the memory consumption, using microbenchmarks. We will refer to continuous views within Umbra as *Umbra* throughout this section. All systems, approaches, and experiments in this section are run on a machine equipped with an Intel Xeon E5-2660 v2 CPU (2.20 GHz) and 256 GB DDR3 RAM.

4.3.1 AIM Benchmark

The AIM telecommunication workload, as described in [27], has been previously used to evaluate the performance of modern database systems against SPEs and specialized solutions [85]. We want to extend this evaluation with the comparison of stream processing in databases, in the form of our continuous views, with SPEs. As a second approach to stream processing using continuous views, we choose the open-source PostgreSQL extension PipelineDB [124].

In the AIM Benchmark, calls for a number of customers are tracked as marketing information. On these calls, analytical queries are run to make offers or suggest different plans to certain customers. In contrast to [85], we do not want to store all possible aggregates for ad-hoc queries and instead only aggregate what is necessary for the AIM queries specified in [27]. Table 4.3

[1] Available at: `https://github.com/tum-db/pg-cv`

Table 4.3: Continuous view AIM queries, slightly simplified for readability. $\alpha \in [0, 2]$, $\beta \in [2, 5]$, $\gamma \in [2, 10]$, $\delta \in [20, 150]$, $t \in$ SubscriptionTypes, $cat \in$ Categories, $v \in$ CellValueTypes

In evaluation: $\alpha = 2$, $\beta = 2$, $\gamma = 4$, $\delta = 25$, $t = 2$, $cat = 1$, $v = 2$	
Q1	SELECT avg(t_duration) FROM (SELECT sum(duration) AS t_duration FROM events WHERE week = current_week() GROUP BY entity_id HAVING count(local_calls) > α)
Q2	SELECT max(max_cost) FROM (SELECT max(cost) AS max_cost FROM events WHERE week = current_week() GROUP BY entity_id HAVING count(*) > β)
Q3	SELECT sum(t_cost) / sum(t_duration) AS cost_ratio FROM (SELECT sum(cost) AS t_cost, sum(duration) AS t_duration, count(*) AS num_calls FROM events WHERE week = current_week() GROUP BY entity_id) GROUP BY num_calls LIMIT 100
Q4	SELECT city_zip, avg(num_calls), sum(duration_calls) FROM (SELECT entity_id, count(*) AS num_calls, sum(duration) AS duration_calls FROM events WHERE NOT long_distance AND week = current_week() GROUP BY entity_id HAVING count(*) > γ AND sum(duration) > δ) e, customers c WHERE e.entity_id = c.id GROUP BY city_zip
Q5	SELECT c.region_id, sum(long_distance_cost) AS cost_long, sum(local_cost) AS cost_local FROM events e, customers c WHERE e.entity_id = c.id AND week = current_week() AND c.type = t AND c.category = cat GROUP BY c.region_id
Q7	SELECT sum(cost) / sum(duration) FROM events e, customers c WHERE e.entity_id = c.id AND week = current_week() AND c.value_type = v

shows the AIM queries modified for use as a continuous view. As we cannot express the original query 6 as a single view we omitted it from all experiments. Furthermore, as neither PipelineDB nor our continuous views support changing view queries at runtime, we select one value by random for each query parameter ($\alpha, \beta, \dots$) from the range specified in Table 4.3.

Configuration

Both the client and the server run on the same machine and, to recreate the setup of [85], the client generates events using one thread and queries using another thread. For the experiments, we vary the number of threads of the server. When speaking of *number of threads* in the remainder of this section, we always mean the number of server threads. The number of client threads remains untouched. The database is initialized with 10M customers unless stated otherwise. We implement the approaches as follows:

Flink. To represent classical SPEs, we use Apache Flink [29]. As the internal state of Flink can only be accessed by point lookups, and to recreate the setup of [85] as closely as possible, we implement a custom operator for the workload. Our custom operator keeps track of the required aggregates per customer (e.g., number of local calls this week), and queries are run exclusively on these aggregates. While Flink can handle all individual queries on its own, we use the custom operator to process all queries at once. The operator further allows us to share aggregates between queries. All experiments are performed on Flink 1.9.2.

PipelineDB. We use the latest available version of PipelineDB (short *PDB*), 1.0.0, and integrate it into PostgreSQL 11.1, both in default configuration. We slightly adapt the queries in Table 4.3 to fit the syntax of PipelineDB. PipelineDB requires at least 18 worker threads, thus we cannot limit the total number of threads. Instead, we vary the number of threads per query (*max_parallel_workers_per_gather*). In our experiments PipelineDB inserts timed out occasionally, but this did not limit the reported query throughput. However, because of this, we consider the numbers reported for PipelineDB to be an upper bound.

Umbra. For Umbra we create the continuous views as specified in Table 4.3. The number of threads maps to Umbra's worker threads. In addition, we run a single thread that handles all connections to the database and is not involved in query processing.

Unless stated otherwise, all experiments for both Umbra and PipelineDB are measured using the pqxx library to connect to the systems. Events are generated inside the database. Throughput averages are calculated over three minutes of execution based on full query round-trip times.

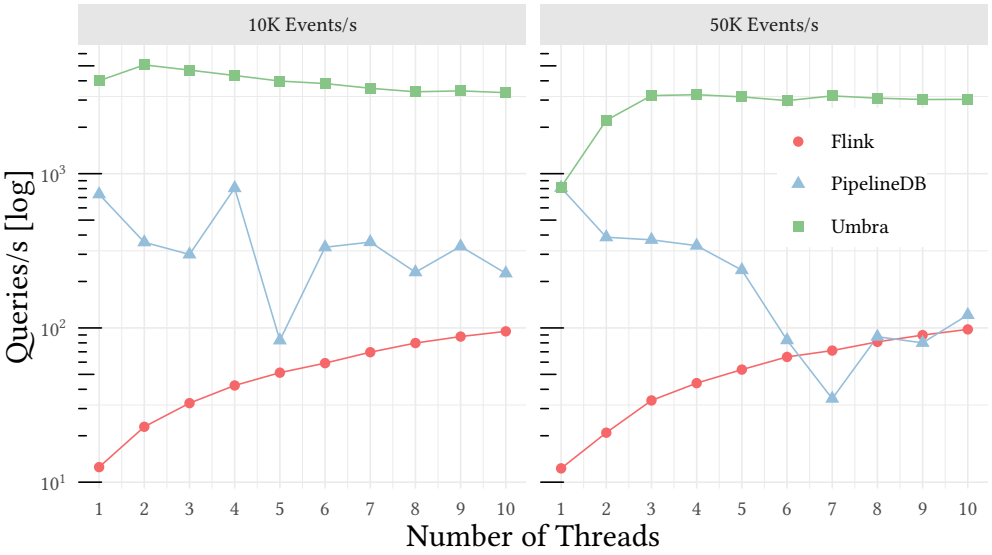


Figure 4.8: Concurrent throughput of queries and inserts with increasing number of threads.

Experiments

Concurrent Access. First, we look at the overall performance of the systems under concurrent write and read accesses with an increasing number of threads. Each query in Table 4.3 is executed with equal probability, and inserts are performed at 10K and 50K events per second respectively. We report the results in Figure 4.8. Flink scales nicely with an increasing number of threads but keeps behind both PipelineDB and Umbra. For PipelineDB we expected to see little scale-up as most of the maintenance is handled by background workers, which we could not limit to the given thread counts. However, the throughput is still very unstable, most likely attributable to the aforementioned problems with inserts. For 50K inserts per second PipelineDB performance degrades with increasing thread count. This is likely caused by interference between insert and query threads.

Since Umbra can handle most of the queries' workload single-threaded, we do not notice a scale-up beyond three threads. Despite not utilizing all threads, we are still able to outperform the competitors consistently, in parts by over an order of magnitude. For higher insert rates, initial scaling is better, since the inserts take more time, and thus parallel execution of queries has a larger impact.

Insert Throughput. As the goal of our continuous views is to handle insert-heavy workloads, we further investigate the isolated insert throughput without

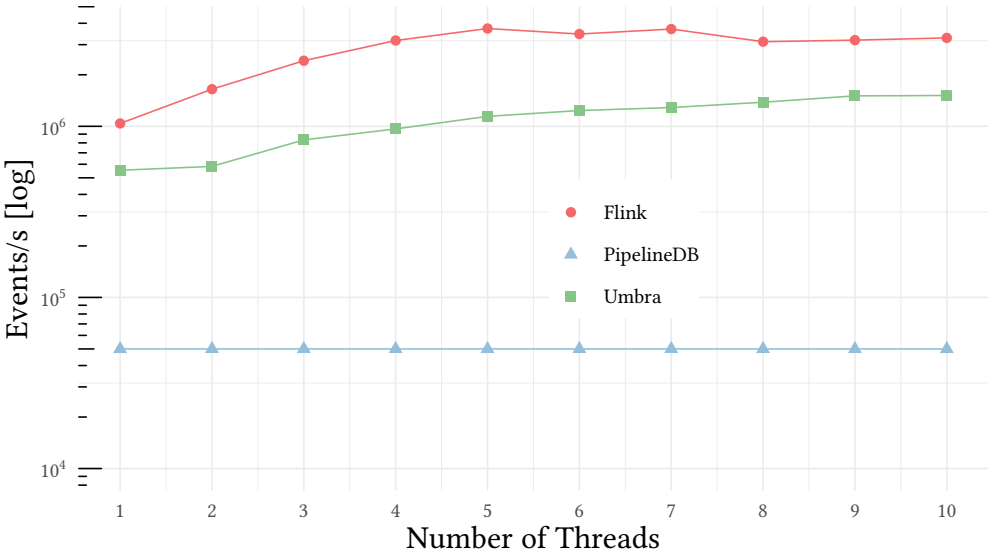


Figure 4.9: Isolated insert throughput with increasing number of threads.

concurrent queries. While we do not issue queries, we still create the corresponding views for Umbra and PipelineDB and track all required aggregates for Flink. For both Flink and Umbra, we report the throughput under full load. For PipelineDB, we report the highest measured throughput. The results are shown in Figure 4.9, again for an increasing thread count. PipelineDB's throughput peaks at around 50K events per second consistently, which seems to be the limit of the non-scaling background workers. Both Flink and Umbra scale well with an increasing thread count for up to 5 threads. Flink does not scale beyond 5 threads. This can be expected as Flink is optimized for scale-out, not for multiple concurrent queries on a single node. However, Flink still offers the highest insert rates. Umbra scales better for higher thread counts and achieves insert rates of more than 1.5M events per second.

Query Throughput. For the final AIM experiment, we look at the isolated read throughput for individual queries. To report actual query results, we initialize the aggregates with random values for Flink and 10K random events for both PipelineDB and Umbra. We report the average query throughput for all individual queries, exemplifying for two threads, in Table 4.4. As the reported query throughput is for full query round-trips for both PipelineDB and Umbra, we only see a slight difference for most queries. The majority of the time is spent in parsing and sending the query, not in execution. Still, Umbra is able to outperform PipelineDB consistently throughout all queries. As Flink does not materialize full query results, it needs to calculate parts of the query on the fly, thus staying behind for all queries. When increasing the number of

Table 4.4: Average query throughput in queries per second without concurrent writes

	SF 1			SF10		
	Umbra	**PDB**	**Flink**	**Umbra**	**PDB**	**Flink**
Q1	**10748**	6725	40	**11161**	208	4
Q2	**10761**	7274	46	**11171**	991	4
Q3	**10677**	6123	19	**11061**	100	2
Q4	**10710**	125	17	**11074**	124	2
Q5	**10501**	6987	13	**10835**	10619	1
Q7	**10709**	7957	26	**11169**	9582	2

customers to 100M (SF=10), we can see the advantages of result caching for Umbra and PipelineDB. While Flink degrades linearly, PipelineDB's throughput stays constant for queries not grouping by customer, and Umbra's throughput stays constant for all queries.

4.3.2 TPC-H Benchmark

After evaluating our strategy on a concurrent streaming benchmark, we compare it to modern view maintenance using DBToaster [87], and the scale-up SPE Trill [31], on a more analytical workload based on TPC-H. We configure all systems to treat *lineitem* as a stream and all other relations to be static. For our experiments, we choose queries 1, 3, and 6, where most of the work of our approach happens at insert time. For these queries, all systems should act similarly. Further, we choose query 15 as a grouped stream-stream join and query 20 to represent queries where the majority of analytical evaluation is performed on a grouped stream. As neither Trill nor DBToaster support order by or limit in their streaming API, we remove all order by and limit clauses from the queries. We rewrite queries for our competitors to fit their syntax and feature set where necessary.

Configuration

We configure and implement our competitors as follows:
DBToaster. We use DBToaster v2.3[2] and allow it to handle streams as append-only using the `IGNORE-DELETES` flag. Further, we distinguish two versions for our experiments: The query-optimized DBT^{Q} performing full refreshes, and

[2] `https://dbtoaster.github.io/download.html`

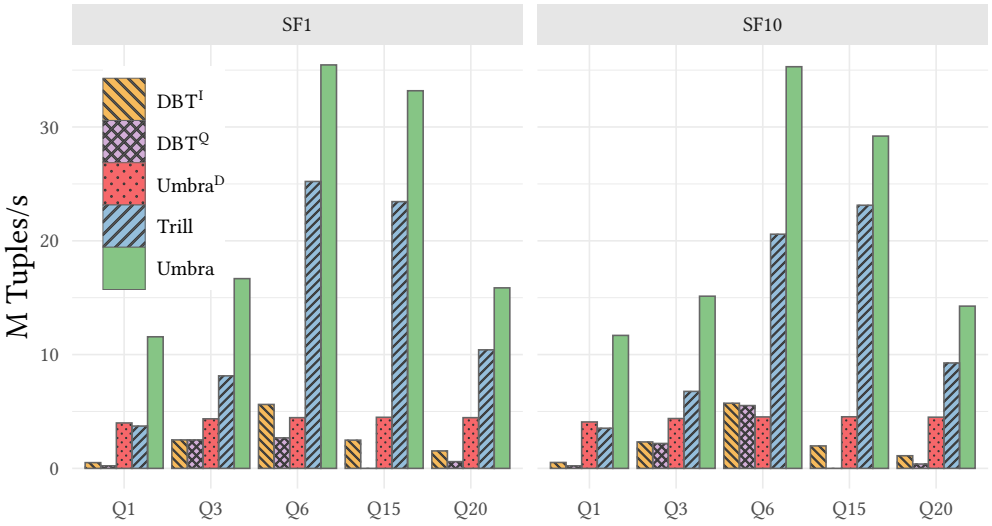


Figure 4.10: Insert throughput against TPC-H baselines for scale factors 1 and 10.

the insert-optimized DBT^{I}. DBT^{I} is allowed to perform only partial refreshes at insert time (EXPRESSIVE-TLQS flag).

Trill. We use Trill v2019.9.25.1[3] and implement all queries using the LINQ-style interface. We only operate on cached inputs for both streams and tables and pre-calculate all joins and subtrees not involving lineitem. Both tables and streams are configured to have an infinite lifetime to keep them in the same join-window. We optimized join order and execution hints to the best of our knowledge.

Flink. We implement the queries in Flink 1.9.2 using the BatchTables API and the SQL interface.

Umbra^{D}. We implement deferred maintenance in Umbra based on full query evaluation at refresh.

As the semantics of concurrent queries is quite different for all approaches, we focus on insert throughput of streams and view refresh times. For all experiments, we report the average of 10 runs after performing 3 warm-up runs.

Experiments

Insert Throughput. We compare the insert throughput of all systems when inserting the *lineitem* table once. As the DBToaster release currently only offers a single-threaded execution, we measure all systems using one thread. To obtain

[3] `https://www.nuget.org/packages/Trill/`

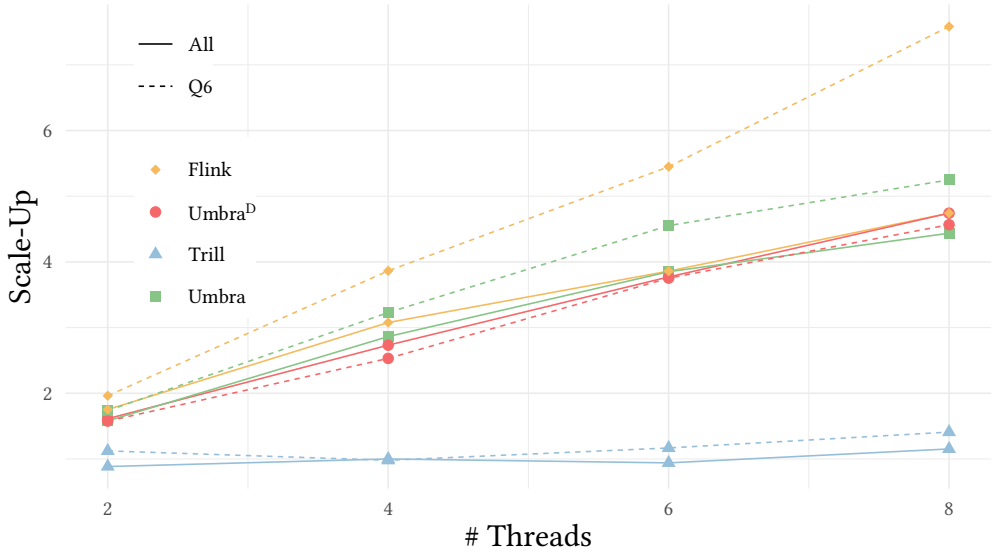


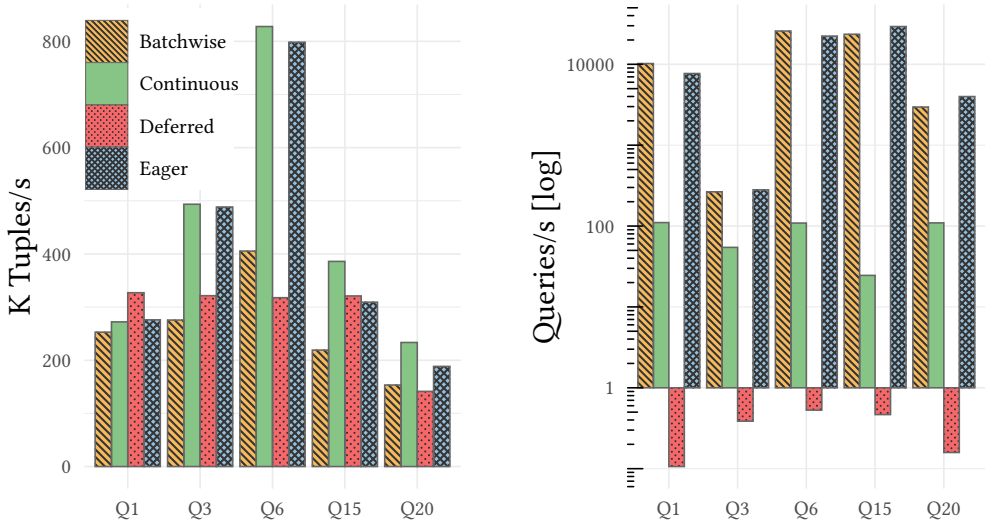
Figure 4.11: Scale-up of Trill, Flink, and Umbra views relative to single-threaded performance.

comparable results, we hold data in a cached stream buffer for Trill. For both DBToaster and Umbra, we only measure processing time without input parsing. We cannot easily recreate this processing-time-only setup in Flink, therefore we exclude it from this experiment. To even the scores, we add the cost of a single view refresh to all systems that do not provide the result immediately. As the runtime for query 15 in DBT^{Q} exceeded reasonable limits for the full table, we report throughput based on a 1000-element sample.

The throughput, displayed in Figure 4.10, is largely independent of the scale factor with minor fluctuation, e.g., at Q6. Overall, Umbra and Trill offer the highest throughput, with a slight advantage for Umbra. Both stay ahead of both DBT variants independent of query and scale factor. Umbra^{D} stays ahead of DBToaster for most queries and can even outperform Trill for Q1. For the simple queries Q1 to Q6, both DBT approaches perform similarly. However, partial refreshes pay off for DBT^{I} for Q15 and Q20.

Scalability. To investigate the scalability of our approach, we repeat the insert experiment above for scale factor 10 with multiple threads. Figure 4.11 reports the average scaling relative to single-threaded performance for each system based on full query execution times. Flink and both Umbra variants scale nicely, Flink even shows near-perfect scaling for Q6.

However, we see little improvement for Trill. We found that, while simple queries like Q6 scale in Trill, for complex queries, the majority of work still happens single-threadedly, and we see negative scale-up. We attribute this to



(a) Insert (b) Query

Figure 4.12: Comparison of continuous views with traditional views in PostgreSQL on TPC-H. Queries consist of refreshing and fetching the result.

the complex nature of the queries and the introduced scheduling overhead in multi-threaded execution.

Traditional View Maintenance. Finally, we want to look at the trade-off space between traditional view maintenance and continuous views on a common runtime. We implement continuous views in PostgreSQL, as described in Section 4.2.9. Further, we manually implement eager and batchwise incremental views for the TPC-H queries and include PostgreSQL's deferred views. Incremental views buffer 10K tuples before propagating changes. Neither eager nor continuous views store any input tuples. We again insert all tuples of lineitem for scale factor 1. As we measure query round-trip times, we insert chunks of 1K tuples to reduce overhead.

Figure 4.12 reports the insert and query throughput of the approaches. For inserts, continuous views are fastest overall, except for Q1, due to the high contention on few groups. The lower throughput for deferred maintenance for Q20 is caused by an index, which we need to maintain to keep the refresh from timing out. For simple queries, like Q1 to Q3, continuous and eager views behave the same conceptually, as all processing occurs at insert time. The measured difference in query throughput is only caused by the current PostgreSQL implementation of continuous views, wherein we needlessly copy the result from the stream pipeline view to the result view for simple queries.

As expected, the eager and incremental approaches offer the highest query throughput. Nevertheless, even on PostgreSQL, continuous views can offer up to a hundred refreshes per second without the need of hand optimizing views, as is the case for eager views, while still offering higher insert throughput on average.

4.3.3 Microbenchmarks

Finally, we want to study our approach's internal behavior, which is not directly visible in end-to-end query benchmarks. For this, we conduct three microbenchmarks, focusing on the memory footprint, the load balancing technique outlined in Section 4.2.5, and the overhead of maintaining multiple views in parallel.

Memory Usage

For the first experiment, we will focus on the memory requirements of a continuous view during its lifetime. In contrast to regular queries, continuous views have an unlimited lifetime. Therefore, we must ensure they are memory efficient and only perform required memory allocations.

Configuration. We create four views, each grouping by a key of the input and calculating one, two, four, or six aggregates over the remaining columns, respectively. Inserting one million tuples into the stream, we record the current memory usage per view every 100 inserts. To only measure the overhead of tuples, not our queuing mechanism, we run the experiment in a single-threaded configuration.

Experiment. The results are displayed in Figure 4.13. At the top of the figure, we show the overall memory consumption throughout the insertion of all 1M tuples. One can see a sharp growth in memory usage on the left-hand side that flattens and does not further change for more than 50K elements. This memory footprint is what we aimed for, an initial growth as aggregates are created and no further changes once an aggregate for the key already exists in the grouping hash table. Further, we see the expected higher memory usage for queries with more aggregates, scaling with the number of aggregates.

When focusing only on the first 50K tuples at the bottom of the figure, we see that memory usage grows similarly for all views. The slight differences stem from our optimized memory allocation, allocating space for whole morsels instead of single tuples. As inserts are created randomly, the growth continues after 10K elements but slows as new keys are seen less frequently. We see no further growth at 35K inserted keys, indicating that each key was encountered once.

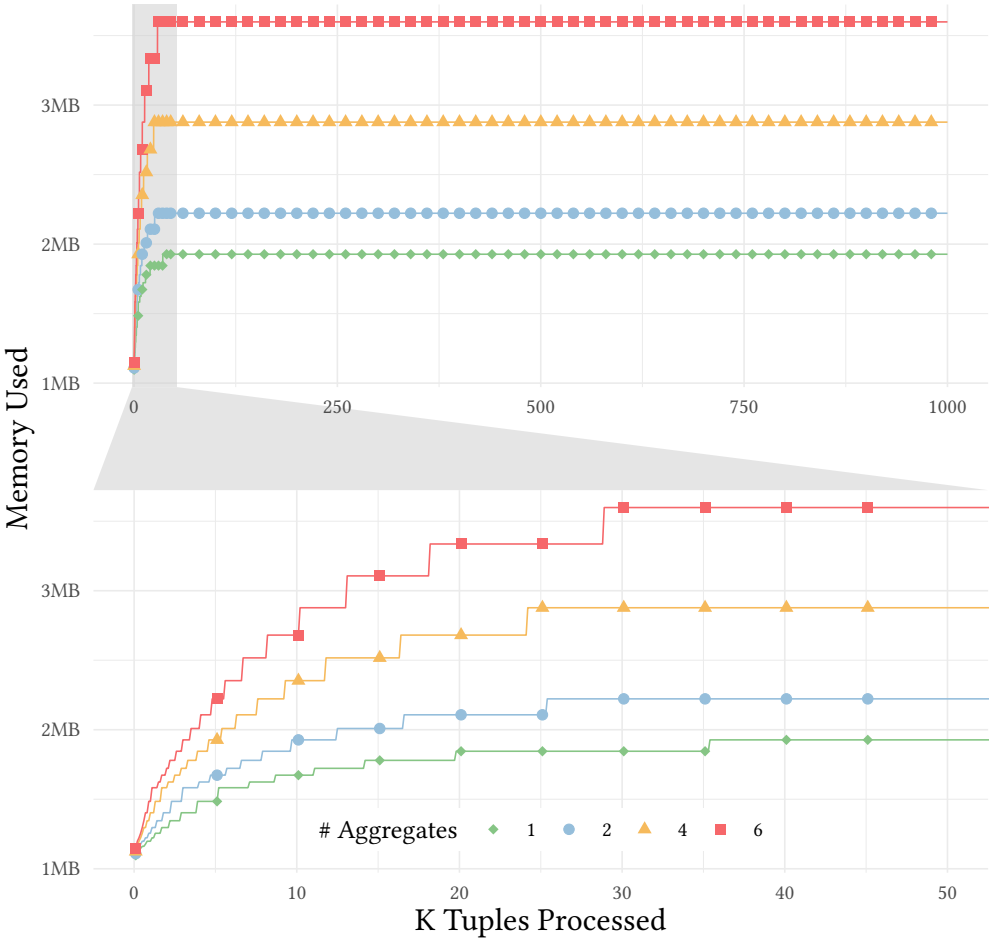


Figure 4.13: Memory consumption of continuous views over time for a varying number of aggregates. The top displays the consumption over 1M inserts. The bottom focuses on the first 50K elements.

Load Balancing Queue

Having outlined the memory consumption of views in a single-threaded setup, we want to focus on the feasibility of our load balancing for parallel view maintenance. To alleviate load spikes, our queue should not grow unlimitedly, as high memory consumption for the queue could decrease the overall performance. Inserts into this queue happen whenever (a) a query locks the view for materialization or (b) there is a concurrent insert blocking the view. As queries empty the queue in case (a), it can never grow endlessly when concurrent queries are triggered. Therefore, we consider case (b) and an insert-only workload, which could trigger an unbounded queue growth.

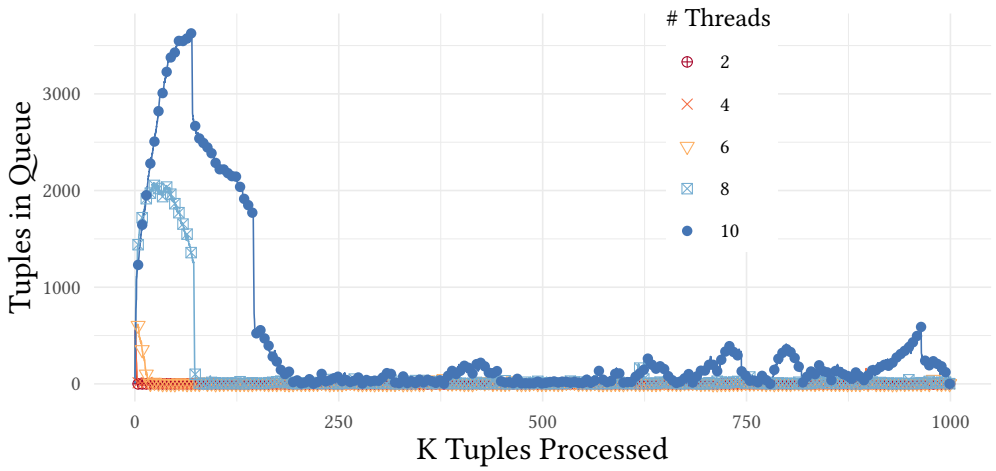


Figure 4.14: Queue length of a single continuous view with varying number of threads.

Configuration. For this experiment, we create a workload of one stream. Tuples of this stream comprise six integer values between 1 and 10,000. Views group by a single key and calculate the sum of one column. We use a single view bundling all inserts to achieve the maximum load on the queue.
Experiment. We display the results of this experiment in Figure 4.14. Queue length is reported after every 100 inserts. The average queue length for ten threads grows slightly after the initial spike. This growth could indicate the need for active maintenance for even higher thread counts. However, one can see that it drops to near zero consistently throughout the experiment, even though the experiment represents the worst case for queue growth. For fewer threads, we only see a small initial spike at around 10K to 50K tuples, with queue length scaling with thread count. Overall the queue works as intended, accepting tuples in the beginning when the load is high due to hash table groups being created. Once all groups have been created, the buffered tuples are processed, and processing threads empty the queue.

Parallel Views

As the final experiment, we look at the overhead introduced when attaching multiple views to a stream.
Configuration. For this experiment, we measure the average throughput when inserting 10M tuples in a stream with 1, 2, 4, or 8 views attached without concurrent queries. Each view groups by one key and calculates the sum of another column. We repeat this experiment for 1 to 10 threads to further investigate the scale-out.

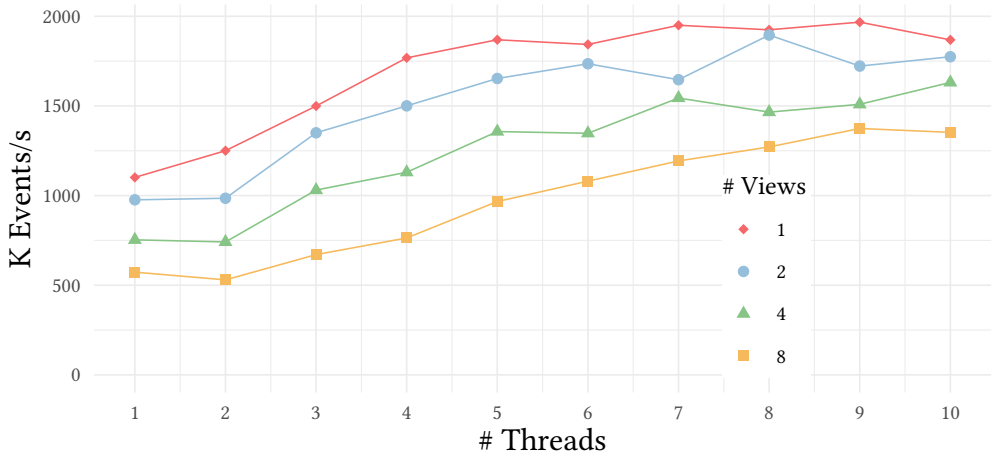


Figure 4.15: Isolated insert throughput for a varying number of attached views. Scale out is shown for an increasing number of threads on the x-axis.

Experiment. The results of this experiment are shown in Figure 4.15. Overall we see the two expected effects: It is cheaper to attach fewer views, and a higher thread count results in higher throughput. Besides, we see other interesting influences. Inserts to a single view do not scale to more than seven workers, while we see a continuous increase for eight parallel views. We attribute this to the high contention on the single view, where all threads compete to access the insert callback. When comparing the scale-out of all numbers of views, we see that the more views are attached, the better the scale-out. Further, the overhead of attaching an additional view is not strictly linear. This non-linear scaling, again, can be attributed to lower contention of inserts for multiple views as fewer threads will compete for inserts on the same view.

4.4 Related Work

To the best of our knowledge, we are the first system to implement analytical views for high-velocity data with a specialized maintenance strategy. There is, however, extensive previous work on materialized view maintenance, stream processing, and continuous query evaluation. We will use this section to summarize those most relevant to our work.

4.4.1 View Maintenance

Managing materialized query results in the form of materialized views is a well-studied problem [69, 141]. Our continuous views can be thought of as

materialized views specialized for streaming data, where full refreshes are only performed at query time. Materializing parts of queries has been previously suggested by Mistry et al. [112] in the context of multi-view optimizations. They share common parts of queries between multiple views for more efficient maintenance. The concept of reusing partial results has been extended to regular query processing [93, 186].

Delaying the maintenance task has been previously described in the form of both deferred maintenance [35] and lazy evaluation [185]. In contrast to our approach, these systems need to have access to the changes to the base table, either in the form of deltas or of auxiliary tables. Storing only changes required for maintenance is known as incremental view maintenance [22, 66, 176]. We apply similar techniques for stream pipelines where incremental deltas are held in local states and materialization in the global state. Incremental view maintenance has been further optimized using hierarchical deltas by Koch et al. [87]. In contrast to our insert-optimized approach, they optimize for query throughput, mostly triggering full refreshes for every insert.

4.4.2 Stream Processing

Stream processing is a broad area of data processing, spanning both dedicated systems and traditional databases. We will focus on recent work and analytical systems.

Modern Stream Processing. There is a wide range of recent work in dedicated stream processing engines, recently surveyed by Zeuch et al. [180], but most systems are optimized for stateless or simple stateful queries. To increase performance, Grulich et al. [67] also utilize query compilation for stream processing. Incremental updates, which we use for data streams, are utilized by SPEs as well [3, 31, 115, 179]. To support aggregating queries in stream processing more efficiently, some work proposes sharing state between long-running queries [63, 97]. As many SPEs do not support complex analytical workloads, dedicated solutions for more stateful queries on data streams have been developed [27, 31, 53, 70, 138].

While we know of no other work implementing stream views with specialized maintenance for database systems, using views to speed up simple analytical queries over streams [60, 176], as well as materializing continuous query results [12] has been previously suggested. For analytical streaming systems, custom query languages are common, as shown in recent surveys [74, 94]. Our approach allows for stream processing using regular SQL statements. Finally, accessing historic stream data with new queries, as is possible by querying our continuous views, has been described by Chandrasekaran et al. [32].

Stream Processing in Databases. Apart from dedicated and stand-alone solutions for stream processing, recent work has focused on integrating data streams in relational database systems and data warehouses [116, 167]. Jain et al. propose a streaming SQL standard [77], an idea that has been further refined by Begoli et al. [18] to allow a single standard and simplify the usage of both tables and streams in SQL statements. Meehan et al. [110] describe how stream processing and OLTP transactions with isolation guarantees can be combined in a single system. Others have made a case for a greater need for in-database processing of signal streams [121].

There have also been some open-source extensions to databases that allow for stream processing within the database [124]. Most work for high-velocity data processing focuses on the previously described maintenance of materialized views. Nikolic et al. [122] further extend higher-order incremental view maintenance to batches and distributed environments, which are common in streaming systems, e.g., in Spark [179]. They further outline the advantages of pre-filtering and aggregation in these batches for incremental maintenance, which we extend to join processing. We apply this to the whole stream instead of batches in our stream pipelines. Our work continues all these efforts by describing an insert-optimized stream view completely expressible with regular SQL statements.

Continuous Query Evaluation. Finally, our work touches upon continuous queries. While continuous views are not full continuous queries, the underlying concept of updating the query result for arriving tuples and reporting intermediate results is similar. Continuous query evaluation [16, 106, 148] focuses on keeping track of changes to an underlying query like we do within the views.

In contrast to our approach, the goal is to alert users whenever the query matches defined triggers. There has been some work on whole systems dedicated to such monitoring and change detection [4, 30, 84]. While we currently do not support triggers, those could be realized by periodic queries to the continuous views.

4.5 Summary

In this chapter, we introduce continuous views, a new type of materialized views optimized for high-velocity streaming data. To maintain these views, we introduce a novel split maintenance strategy, performing parts of the query at insert time and finalizing query processing when results are required. We demonstrate the feasibility of our approach by integrating these views into our state-of-the-art database system Umbra, using the compiling query execution engine as well as the query optimizer for fast and low-overhead maintenance. We

explain how this integration allows users to access stream results in analytical workloads and durable state for queries on streams efficiently.

To demonstrate the capability of our views' split maintenance strategy, we compare it to modern stream processing engines, as well as view maintenance strategies in both Umbra and PostgreSQL. Our approach outperforms the former on analytical workloads and the latter on insert throughput, often by order of magnitude, creating an ideal fusion of analytical query processing and high-velocity data.

CHAPTER 5

Communication-Optimal Parallel Reservoir Sampling

Excerpts of this chapter have been published in [171].
With contributions from Moritz Sichert and Altan Birler.

So far, this thesis has presented two strategies for full end-to-end stream analytics within a relational database system. However, end-to-end analytics on live data is not the only streaming workload that relational database systems face. With the widespread deployment of cheap connected sensors and the increased use of off-premise systems, there is an ever-growing need to analyze the log and data streams generated by these sensors and systems remotely. Due to this data's high volume and high velocity, analyzing all generated entries and values is often infeasible.

Therefore, many analyses are performed on a reduced version of the data. Some applications rely on aggregates over windows or subsets of the data, e.g., monitoring average temperature readings in a given area in continuous queries. Others, e.g., for analyzing log streams, apply highly selective filters to reduce the input cardinality, showing only relevant error messages and surrounding entries. However, for some applications, representative and unfiltered data points are desirable for later analytics. For such workloads, sampling is employed to reduce the data stream to a manageable size for analytics systems.

By drawing a uniform sample from a stream of data points, each point of the stream has an equal probability of being part of the resulting sample, and thus the result is representative of the entire stream. Sampling is utilized in a wide range of applications, e.g., for workload statistics [21, 105], machine learning [133, 135, 136], and big data [107]. While some sampling algorithms take a variable-sized subset of the input, e.g., selecting $x\%$ of the arriving tuples at random, their resulting sample size can still be infeasible for analyzing high-velocity unbounded data streams. Therefore, we will focus on those algorithms

that produce a fixed-sized sample independent of the input size, i.e., reservoir sampling.

While reservoir sampling produces a uniform random sample in a single pass over the input in $\mathcal{O}(n(1 + \log(N/n)))$ [103], the sheer volume of data can lead to bottlenecks during the sampling phase. In the past, the growth in data could be compensated by the performance improvements of new hardware. However, with Moore's law coming to its end, a single core can often no longer keep up. Therefore, many solutions for stream processing and analysis, such as dedicated stream processing engines [29, 179, 181] and in-database stream-processing approaches [170], have focused on processing incoming streams in a parallel and distributed manner.

In this chapter, we describe a mechanism allowing fully parallel reservoir sampling without communication between sampling threads. By keeping thread-local samples over chunks of the input, we can construct a complete sample of size n in parallel from p worker samples using only $2p + n$ messages in an efficient k-way merge. The low message overhead is especially beneficial in distributed environments, where communication takes place using comparably slow and expensive network connections.

Further, we describe an $\mathcal{O}(p + n\log(p))$ merge strategy optimized for small sample sizes and show that it can guarantee uniform random samples, independent of how input elements are assigned to workers. By constraining all communication to our merge stage, our approach can scale almost linearly in many-core machines. Implementing our approach within the code-generating Umbra database system [118], we demonstrate that our approach is applicable in real-world systems. Using this implementation, we evaluate our novel merge strategy against a merge strategy based on a hypergeometric distribution in many-core applications for different sample sizes, showing that our proposed merge is beneficial for small sample sizes. Overall, our communication optimal reservoir sampling can scale linearly along the number of available workers and sustain a sampling throughput of more than 300 million tuples per second per thread, independent of the distribution used in the merge. In summary, this chapter addresses the following key points:

- We describe a communication-optimal parallel merge process.
- We device a novel merge strategy optimized for small reservoir sizes.
- We evaluate the performance of our technqiues within the Umbra database system [118].

The remainder of the chapter is structured as follows: We introduce relevant concepts and algorithms to our approach in Section 5.1 and discuss related

work. In Section 5.2, we present the design and implementation of both of our techniques and prove the correctness of our novel merge strategy. To demonstrate our approach's applicability and performance, we evaluate its scalability and throughput in Section 5.3 before summarizing in Section 5.4.

5.1 Background and Related Work

Approximate results are often deemed acceptable to gain instant query response times for analytics over large volumes of data. Simple random samples of fixed size are a reliable tool to reduce the costs of computing approximate statistics [131]. We will argue why this tool is particularly interesting and discuss related work in constructing such samples.

When using a random sample, a small random subset of the data is picked and processed instead of the entire data, significantly improving query response times. Other statistical tools, such as histograms [37] and distinct count sketches [57], are useful for approximate statistics. However, they are inflexible as the filters that they can evaluate are limited. A histogram can only evaluate simple predicates, such as one (or few) dimensional ranges. Samples, on the other hand, can evaluate arbitrary predicates, as they are smaller representatives of the entire data set. In the absence of complex filters, histograms can provide useful upper bounds, while samples can only provide probabilistic estimates. However, with large enough samples, the variance of the estimates a sample provides is relatively low and can be relied upon. Additionally, with complex filters, a histogram's upper bounds can be too high to be useful, as it can only consider simple range predicates.

There are many ways to pick a random sample. For computing unbiased statistics, simple random samples without replacement are a good fit as every subset of the data is selected with equal probability. Tillé [150] describes the theoretical background of simple random sampling and computation of statistics from a simple random sample. Ting [151] provides efficient implementations for a range of algorithms for sampling without replacement. We focus on samples of fixed size. In contrast to Bernoulli sampling, where every tuple is picked with independent probability θ, fixed-size samples do not grow alongside the input data size. One might assume a larger sample would be a better fit for a larger data set. This is true for predicates whose selectivity decrease with increasing data set size, such as filters for fixed timespan on a data set that grows over time. However, for predicates of constant selectivity, the size of the data set has little to no effect on the quality of the sample. We will try to build a simplified intuition as to why and refer the reader to the detailed theoretical analysis by Moerkotte and Hertzschuch [113] for further details.

Given a sample of size n, a data set of size $N = \rho n$, and a predicate of constant selectivity σ, we want to estimate the number of matches $K = \sigma N = \sigma n \rho$ where ρ is the ratio of the size of the data set to the size of the sample. This task is common in cardinality estimation within database systems [73]. As a strategy, we evaluate the predicate on the sample and count the number of matches k, which is a random variable from the hypergeometric distribution $HG(\rho n, \sigma \rho n, n)$. We use the simple estimator $\hat{K} = k\rho$ as our estimate of $K = \sigma n \rho$. As a simple cost metric, we define the expected relative mean squared error of our estimate as:

$$\text{rMSE} = \text{E}\left[\left(\frac{\hat{K} - K}{K}\right)^2\right] = \text{E}\left[\left(\frac{k\rho - \sigma n \rho}{\sigma n \rho}\right)^2\right] = \frac{1}{\sigma n}\,\text{Var}\left[k\right] = \frac{1-\sigma}{\sigma}\frac{\rho - 1}{\rho n - 1} \approx \frac{1-\sigma}{\sigma}\frac{1}{n} \tag{5.1}$$

Assuming ρ is relatively large, it disappears from our error estimate, implying that the relative size of the sample has little to no effect on the accuracy of this estimate. On the contrary, the predicate's selectivity and the sample's absolute size directly influence the error. An intuitive explanation of this result is that we can assume that the data set from which we are sampling is itself a random sample drawn from an infinite distribution. Resampling from this *intermediate sample* simply means that we construct a smaller sample of the original data set. The size of the intermediate sample has only a small effect on the final sample's contents. So, given requirements on the error and information on the selectivity of predicates, it makes sense to pick a *fixed* sample size rather than having sample sizes adapt to the data set size as adapting the size of a sample is a costly operation requiring that the sample be rebuilt.

The optimal way to compute a simple random sample depends on various factors. Our proposed approach focuses on concurrently sampling from many parallel data streams in environments where communication costs are high (we are optimal in the number of communications), and memory usage is not a limiting factor. Due to these assumptions, our approach is flexible and an excellent fit for distributed environments.

There are simpler algorithms for when the size of the data set is known beforehand: The draw-by-draw procedure iterates over the sample and picks tuples one by one with potentially high costs from random accesses into the data [150]. The sequential selection-rejection method described by Fan et al. [49] instead iterates over the data to produce the sample. Vitter [159, 160] further improves the selection rejection method by computing skip lengths; rather than iterating over the data one tuple at a time, one can probabilistically generate the number of tuples to skip before the next tuple is selected, significantly reducing the number of random number generations. This approach is parallelized by Sanders et al. [132] by distributing the task of random sampling among different

workers. Chickering et al. [33] describe merging parallel reservoir samples using a hypergeometric distribution, which we utilize for larger sample sizes, and offer proof that the resulting sample is still uniformly random. However, they send all local samples to a centralized coordinator for the merge, leading to communication overhead.

To maintain a sample of size n in a single pass over a data stream, a set of more than n values, the so-called reservoir, needs to be processed to guarantee a simple random sample at any point during processing. All procedures maintaining a simple random sample in a single pass over a data stream are variants of the reservoir sampling algorithm defined by Vitter [161]. Reservoir sampling iterates over input tuples and selects them for the sample with a probability proportional to the number of tuples seen so far. Vitter [161] improves on this by probabilistically generating skips, a contiguous amount of tuples not contained in the sample, thus reducing the costs for generating random numbers from $\mathcal{O}(N)$ to $\mathcal{O}(n(1 + \log(N/n)))$. Li [103] improves on the approach by Vitter by proposing a simpler distribution to generate skips.

These sequential approaches only support sampling data from a single stream. Hübschle-Schneider and Sanders [75] also parallelize reservoir sampling by independently collecting multiple reservoirs and merging them afterward. However, their approach needs to maintain a distributed priority queue, which incurs additional communication costs but potentially reduces the sizes of their independent samples. For situations where memory is scarce, Tirthapura and Woodruff [152] maintain a single sample at a central coordinator. Their approach has optimal communication complexity for the centralized sample setting. Birler et al. [21] reduce the communication costs of Tirthapura and Woodruff's approach by accepting temporary imperfections in the central sample that are eventually corrected.

Our approach, in contrast, is communication optimal among all possible distributed reservoir sampling algorithms. For this property, we accept a slight increase in per-stream memory consumption, which is acceptable in analytical workloads as our local samples are fixed-sized and small compared to all the other data the streams need to maintain.

The performance characteristics of the various approaches can be analyzed by looking at communication costs, total processing costs, and total memory use. These three metrics are influenced by the utilized sampling approach, the sample size, the data size, and the number of workers. Shared-sample-based approaches [21, 152] benefit from low memory use and total processing costs. Thus, they are well applicable to low communication cost environments, such as a single machine with a single CPU socket. However, in settings with high communication costs, such as manycore machines with multiple sockets or distributed networks, distributed sampling approaches [75] are beneficial as

they sacrifice memory and some computation to cut down on inter-worker communication.

These trade-offs are necessary as Tirthapura and Woodruff [152] prove that communication costs may not be optimal when only one shared sample exists. In our approach, where we focus on minimizing communication costs, we must maintain a full sample per worker. Otherwise, we must incur additional communication or provide weaker guarantees, such as a probability of failure [132].

5.2 Approach

Having outlined the background in sampling, we can describe our merge-based parallel reservoir sampling technique and the novel post-sampling merge strategy optimized for small sample sizes. Our approach aims to draw a sample of n tuples uniformly random in parallel from an input of N tuples using p workers. Each worker i draws a thread-local reservoir sample of size n out of the N_i tuples assigned to it, denoted as $s_{i,1} \dots s_{i,n}$, using algorithm L [103]. This sample is merged into a global sample on demand.

We assume no prior knowledge about the workload, only the reservoir size n has to be known, and we materialize only tuples selected for a local reservoir. Throughout this section, we will assume a work-stealing, morsel-based [99] distribution of input chunks to workers, as this is the parallelization strategy of our system Umbra. However, our approach is applicable to any input distribution strategy. This chapter describes two techniques. First, we detail the communication-optimal merge process, which improves upon prior work independent of the reservoir size and merge strategy used. Subsequently, we discuss our novel merge strategy optimized for small reservoir sizes.

5.2.1 Communication-Optimal Process

Our approach consists of two phases, the sampling and the merge phase, as shown in Figure 5.1. In the sampling phase ①, all threads create local reservoirs of size n over chunks of the input. While sampling, workers fetch arriving chunks independently from one another, ensuring that each chunk is assigned to exactly one worker. Reservoir sampling guarantees that all local samples are uniformly random at any point, with each tuple having a probability of $\frac{n}{N_i}$ to be contained in the corresponding thread-local sample. To generate the whole sample, each worker first reports the cardinality of all chunks it processed, N_i, to a coordinator ②.

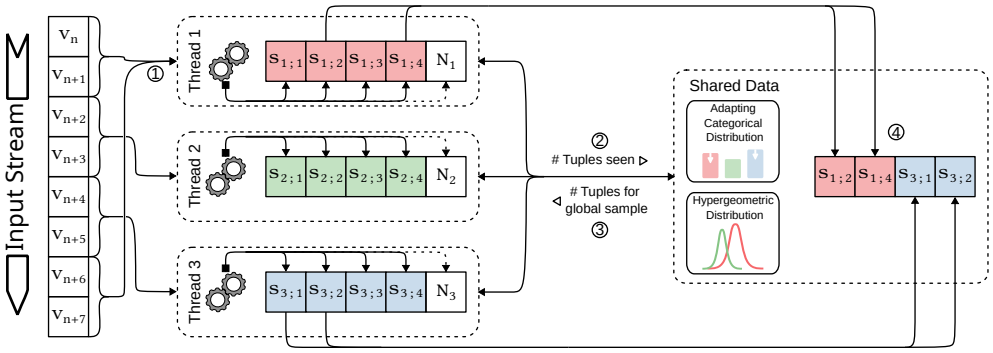


Figure 5.1: Overview of the sampling process with three workers consisting of ① thread-local sampling, ② the transfer of local cardinalities, ③ determining thread shares in the global sample, and ④ the transfer of the resulting sample tuples.

This coordinator can be an external worker or one of the sampling workers. First, the coordinator determines the share in tuples that each thread-local sample has in the global sample using either the hypergeometric distribution, or the proposed merge strategy outlined in detail below. Then, the coordinator notifies each thread of the number of tuples it has to choose for the global sample ③, which in turn selects the desired amount uniformly at random from their sample and reports it to the global sample ④. In contrast to prior work [33] sharing all local samples with the coordinator in pn messages, our approach needs at most $2p+n$ messages: 2 per worker to communicate the local cardinality and the number of tuples to contribute to the global sample, and n to send the selected tuples.

5.2.2 Merge Strategy for Small Reservoirs

Conceptually, our approach for small reservoirs relies on iteratively evaluating and updating a categorical distribution over all threads for each position of the final sample to determine which thread to select for this reservoir slot. In contrast to strategies based on the hypergeometric distribution, the categorical distribution has to be evaluated per sample slot and not per thread. While this is too costly for large sample sizes, it avoids the computationally expensive hypergeometric distribution. Each thread i is selected with a probability of $\frac{N_i}{N}$ for the first slot. As we sample without replacement, we decrease the cardinality of the selected thread N_i and the global cardinality N by 1 for the next draw. All threads j with $j \neq i$ have the probability $\frac{N_j}{N-1}$ of being selected for the next

Algorithm 5.4 Calculating per-thread share of global sample

```
1:  function calculateThreadShare(localCardinalities[])
2:      globalCardinality ← sum(localCardinalities)
3:      fenwickTree ← FenwickTree::build(localCardinalities)
4:      samplesPerThread ← []
5:      slot ← 0
6:      while slot < sampleSize and globalCardinality > 0 do
7:          selTuple ← pickRandom(0, globalCardinality − 1))
8:          selThread ← fenwickTree.rank(selTuple)
9:          samplesPerThread[selThread] ← samplesPerThread[selThread] + 1
10:         fenwickTree.add(selTuple, −1)
11:         globalCardinality ← globalCardinality − 1
12:         slot ← slot + 1
13:     return samplesPerThread
```

reservoir slot, the selected thread has a probability of $\frac{N_i-1}{N-1}$. We repeat this until we have selected the source for all n spaces in the sample, decreasing N and N_i for the selected i at every draw. Note that while conceptually drawing slot by slot, we do not care about the order of the sample or the actual slot to select from. This allows us to only track the number of tuples per thread.

Algorithm 5.4 shows our implementation. In the first step, we calculate the global cardinality from the thread-local information and build a Fenwick tree over the local cardinalities (Line 3). Fenwick trees, as described in [52], allow efficient operations over prefix sums while requiring only linear space. Building a Fenwick tree is possible in linear time, whereas rank and update operations have logarithmic runtime. Following the setup, we can pick the shares for each thread. For this, we first generate a random integer $r \in [0, N)$ (Line 7). From this value, we pick the corresponding thread by using the prefix-sums stored in the Fenwick tree. For $r \in [0, N_1)$, we pick thread 1, for $r \in [N_1, N_2)$, thread 2, and so forth. Mapping r to a thread this way is possible in $\mathcal{O}(\log(p))$ using the rank operation of the Fenwick tree (Line 8). Then, we update the cardinality of the selected thread and the global cardinality (Lines 10 and 11) for the next draw from the updated categorical distribution. We repeat the draw and distribution update until we have either selected every tuple or filled every slot in the final sample. Our algorithm does not require floating point arithmetic, allowing a fast and exact evaluation. The Fenwick tree construction dominates the setup step with a runtime of $\mathcal{O}(p)$. The loop of Line 6 is evaluated n times, requiring $\mathcal{O}(\log(p))$ to update the Fenwick tree, resulting in an overall runtime complexity of $\mathcal{O}(p + n\log(p))$.

5.2.3 Proof

To show that the merge strategy outlined above does not change the resulting samples' probability, we show that it selects tuples from local reservoirs equal to the hypergeometric distribution. For this, we will use the probability mass function $P(X = k)$ where X denotes the number of tuples selected from thread i. For our proof, we use the fact that the positions for which i is selected are irrelevant. Consider first the case where all k selections of i happen in the first k draws, followed by $n - k$ draws of $j \neq i$. This results in the probability

$$P(\text{first } k \text{ from } i) = \frac{N_i}{N} \times \frac{N_i - 1}{N - 1} \times \cdots \times \frac{N_i - k + 1}{N - k + 1} \times \frac{N - N_i}{N - k} \times \frac{N - N_i - 1}{N - k - 1} \times \cdots \times \frac{N - N_i - n + k + 1}{N - n + 1} \tag{5.2}$$

$$= \frac{N_i \times (N_i - 1) \times \cdots \times (N_i - k + 1) \times (N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1)}{N \times (N - 1) \times \cdots \times (N - n + 1)}. \tag{5.3}$$

It is clear that, through the commutative property of the product, the draws for i, N_i to $N_i - k + 1$, can be moved to any of the draws N to $N - n + 1$ without changing the resulting probability. For the full $P(X = k)$, we additionally need to select the k positions for our i draws, resulting in the probability

$$P(X = k) = \frac{N_i \times (N_i - 1) \times \cdots \times (N_i - k + 1) \times (N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1)}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k}. \tag{5.4}$$

Using $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ and $x \times (x-1) \times \cdots \times (x-m+1) = \frac{x!}{(x-m)!}$ we get

$$P(X = k) = \frac{(N_i \times (N_i - 1) \times \cdots \times (N_i - k + 1)) \times ((N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1))}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k} \tag{5.5}$$

$$= \frac{N_i!}{(N_i - k)!} \times \frac{(N - N_i) \times (N - N_i - 1) \times \cdots \times (N - N_i - n + k + 1)}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k} \tag{5.6}$$

$$= \frac{N_i!}{(N_i - k)!} \times \frac{(N - N_i)!}{(N - N_i - n + k)!} \times \frac{1}{N \times (N - 1) \times \cdots \times (N - n + 1)} \times \binom{n}{k} \tag{5.7}$$

$$= \frac{N_i!}{(N_i - k)!} \times \frac{(N - N_i)!}{(N - N_i - n + k)!} \times \frac{(N - n)!}{N!} \times \binom{n}{k} \tag{5.8}$$

$$= \frac{N_i!}{(N_i - k)!} \times \frac{(N - N_i)!}{(N - N_i - n + k)!} \times \frac{(N - n)!}{N!} \times \frac{n!}{k!(n - k)!} \tag{5.9}$$

$$= \frac{N_i!}{k!(N_i - k)!} \times \frac{(N - N_i)!}{(N - N_i - (n - k))! \times (n - k)!} \times \frac{(N - n)!n!}{N!} \tag{5.10}$$

$$= \binom{N_i}{k} \times \binom{N - N_i}{n - k} \times \frac{1}{\binom{N}{n}} \tag{5.11}$$

$$= \frac{\binom{N_i}{k} \times \binom{N - N_i}{n - k}}{\binom{N}{n}} \tag{5.12}$$

which is the probability mass function of the hypergeometric distribution.

5.3 Evaluation

Having outlined the implementation of our fully parallel communication-optimal reservoir sampling approach, we demonstrate its performance, focussing on

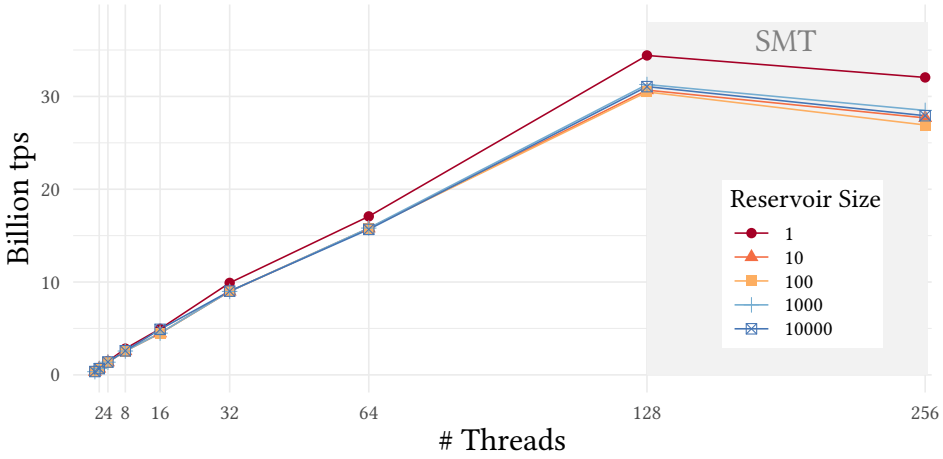


Figure 5.2: Total sampling throughput with an increasing number of threads: Our communication-optimal sampling approach scales nearly linearly to more than 30 billion tuples per second.

scalability and the impact of our proposed merge strategy for small sample sizes. The experiments are conducted using an implementation within Umbra on a server equipped with 2 AMD EPYC™ 7713 CPUs with 64 cores each and 1 TiB of main memory. To reduce the impact of IO bottlenecks, we generate all data using the PostgreSQL-derived `generate_series`[1] command. All results reported are based on averages over 9 runs, each sampling $N = 100$ billion records.

5.3.1 Scalability and Performance

In the first experiment, we investigate the scalability of our approach. As we require no communication between threads during the sampling phase, we expect the sampling phase to scale perfectly along the thread count. Additionally, the runtime of the merge phase is independent of the input size, so we expect it to amortize for large data sets. We measure the total throughput of our implementation with different reservoir sizes and an increasing number of threads and report the results in Figure 5.2. As expected, the throughput of processed tuples scales nearly linearly with the number of threads. For a reservoir size of 1, our implementation can process up to 35 billion tuples per second when using all 128 physical cores. The experiment samples 8 B integers, so in total, our system processes up to 280 GB of data per second.

[1] `https://www.postgresql.org/docs/current/functions-srf.html`

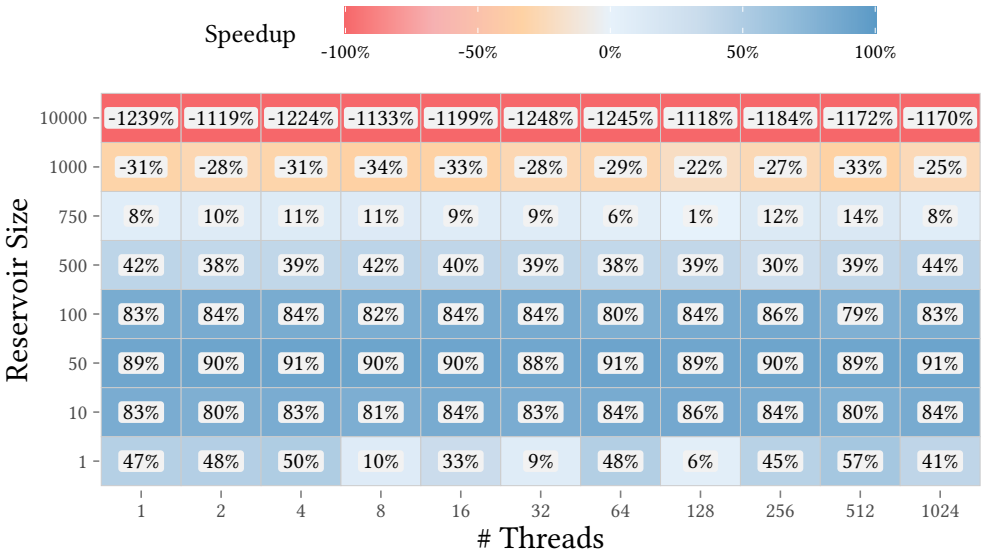


Figure 5.3: Speedup of our merge strategy based on a categorical distribution over using a hypergeometric distribution: For sample sizes below 1000, our approach achieves up to 91% speedup independent of the number of threads.

Our approach requires each thread to collect a full-size local sample. For this reason, the memory usage of sampling increases linearly with the number of threads. Therefore, we expect higher sampling overhead and lower throughput for increasing sample sizes. However, the results show that the overhead is manageable even for larger sample sizes. For example, even when collecting a sample of size 10000, our implementation can still process over 30 billion tuples per second.

5.3.2 Merge Strategy Comparison

Our approach requires communication between threads only in the merge phase. We want to show that our merge strategy, which employs a categorical distribution, can be more efficient than a hypergeometric distribution while producing equivalent results. To evaluate our strategy, we compare the runtime of the merge phase for both distributions. Figure 5.3 shows the relative speedup of our merge strategy with varying numbers of threads and sample sizes. Note that the merge phase itself always runs single-threaded on a coordinator node. The number of threads in the figure refers to the number of locally collected samples to be merged.

For sample sizes up to 750, our approach consistently outperforms using the hypergeometric-distribution-based merge. The main reason for the efficiency

of our merge strategy is that it uses no floating-point operations. However, we need to perform two operations on the Fenwick tree for every element in the sample, while the merge phase using a hypergeometric distribution only generates one value from the distribution for every thread, independent of the sample size. Therefore, our merge strategy is inefficient for larger sample sizes. The experiment further shows that our approach's speedup is generally independent of the number of threads: Our strategy consistently achieves similar speedup across all sample sizes, even for several hundred threads.

5.4 Summary

In this chapter, we introduce a new communication-optimal parallel reservoir sampling technique, requiring only $2p + n$ messages for environments with p workers and a sample size of n. Our technique relies on an efficient merge of thread-local reservoir samples, each taken over arbitrarily distributed chunks of the input. In addition, we describe a novel merge strategy optimized for small sample sizes and provide proof that this strategy is statistically equal to the hypergeometric distribution. Finally, with our implementation of communication-optimal parallel reservoir sampling in the Umbra database system, we evaluate its overall performance, and both merge strategies. Achieving more than 370 million samples per thread, we show a near-linear scale-up to 128 workers and a clear advantage of our optimized merge for samples smaller than 1000 tuples.

CHAPTER 6

On-Demand State Separation

Excerpts of this chapter have been published in [168].

The high flexibility and cost-efficiency of cloud databases, such as Snowflake [39] and Amazon Redshift [68], are attracting an increasing range of customers. While these systems offer solutions for petabytes of data and optimize for scalability, they also attract smaller customers and workloads. These workloads do not require the full elasticity offered and can often be handled by one or a few machines. However, finding the optimal instance to provide cost optimality is still not trivial [102]. To understand the challenges of cloud data warehousing for smaller workloads, one needs to look at the dominant warehouse architectures and their characteristics outlined in Figure 6.1.

First, there is the classic shared-nothing architecture prominent in on-premise deployments and used in the initial version of Amazon Redshift [68]. In this architecture, both storage and compute are co-located on a worker. While this offers the best performance, it cannot scale resources independently. Second, there are storage separated architectures, such as Snowflake [39]. These allow compute and storage to be scaled separately. Keeping the working state of a query at the compute node still achieves excellent performance. However, this does not permit elasticity and fault tolerance for individual queries. Third, state-separated architectures, like Microsoft POLARIS [7], fully decouple state and compute. The high flexibility and elasticity of state separation come at the cost of network overhead when syncing the state between tasks. This overhead is acceptable when data has to be shuffled between workers after each task. Finally, modular systems, like Apache Spark [179], do not follow a specific architecture fully but can be configured similarly to one or more architectures. For Spark, e.g., state separation can be achieved by strategically placing checkpoints in the query plan [139, 173]. We argue that, due to network transfer costs, stateless architectures are not profitable for smaller workloads. Nevertheless, there is a

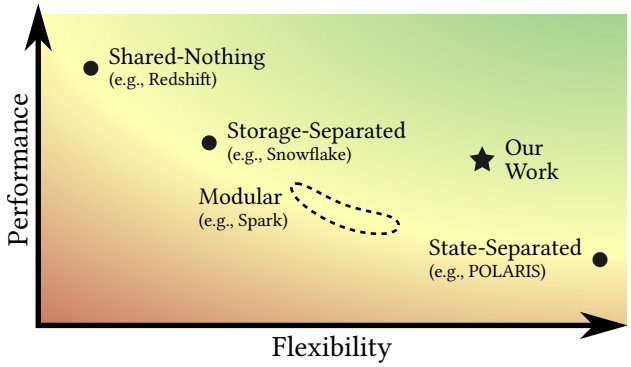


Figure 6.1: Classification of cloud data warehouse architectures by performance and flexibility. Flexibility here is the ability to adapt to changes in the execution environment and provide scalability for processing. Performance is defined by query throughput and latency.

growing need for higher flexibility for such workloads: Ambati et al. [10] propose speculatively executing queries to find the best query-to-worker matching, losing all progress achieved when the worker has to be changed. In addition, Garefalakis et al. [59] have described the need for suspendable tasks to provide low latency for time-sensitive tasks when resources are limited.

In this chapter, we propose on-demand state separation to provide the desired flexibility of stateless architectures without incurring the performance cost. This way, we can utilize the full performance of storage-separated architectures for local queries, while still allowing for query migration and elasticity with minimal overhead when necessary. To achieve this, we run the query as if it were only storage-separated. When the need for state separation arises, we cache all relevant intermediate results over the network and continue the query on these on the target worker. This query migration can be beneficial in a number of settings. It allows the utilization of more powerful or cheaper servers that become available in a cluster for already running queries. Furthermore, it enables load balancing between servers in multi-tenancy settings, as well as the utilization of spot instances for query processing. Queries can be started on such instances and migrated when the spot instance expires, which has been proposed for VM instances [140, 145, 175] and using predefined checkpoints [139, 173] in Apache Spark. Contrary to those solutions, we only need to migrate the current working state of a query and do not require a priori knowledge about the workload. This chapter covers the following key points:

- We provide an analysis of the query states occurring in mid-sized cloud workloads on the exemplifying workload of all queries of the TPC-DS benchmark [1] at a scale factor of 100. This dataset of roughly 100GB

represents a medium-sized workload, which can be reasonably executed on a single server. To the best of our knowledge, we are the first to describe query states in a state-separated architecture.

- We describe the constraints for the deployment environment necessary for (on-demand) state-separated architectures on such workloads.
- We show the design and implementation of on-demand state separation in an OLAP database system using the code-generating DBMS Umbra [118].
- We evaluate the performance and overhead of on-demand state separation for various use cases.

The remainder of the chapter is structured as follows: Section 6.1 defines the goal of on-demand state separation and the relevant concepts. Then, we analyze the state of OLAP workloads based on TPC-DS in Section 6.2. Section 6.3 describes the design and implementation of on-demand state separation, which we evaluate in Section 6.4. We discuss related work in Section 6.5 before summarizing in Section 6.6.

6.1 Problem Definition

Before describing our novel approach to on-demand state separation, we first need to formalize the problem statement, as well as the requirements that queries and systems have to fulfill to support our approach. The goal of on-demand state separation is to provide flexibility for traditional relational databases in cloud settings. To achieve this flexibility, we utilize state separation [7].

Definition 1. ***State Separation:*** *State separation is the process of decoupling the working state and progress of a query from the machine executing it.*

A state-separated query can, thus, be resumed on any machine, even if the new machine's configuration differs from the old one. Changing the executing server at runtime has been employed in the past, e.g., in multi-engine environments [6, 143]. These systems focus on migration between different engines at pre-planned points in the query plan for performance. In contrast, we aim for flexibility by migrating on demand without prior planning.

Goal. *On-demand state separation achieves state separation retroactively with minimized progress loss and minimized query state without hampering the performance of local execution.*

```
with customer_revenue as (
  select     customer.id, sum(orders.price)
  from       customers, orders
  where      customers.id = orders.customer_id
  group by   customer.id
)
select       c.name, c.adress, c.birthday
from         customer c, customer_revenue r
where        c.id = r.id and r.revenue >= 0.9 *
               (select max(revenue) from customer_revenue)
```

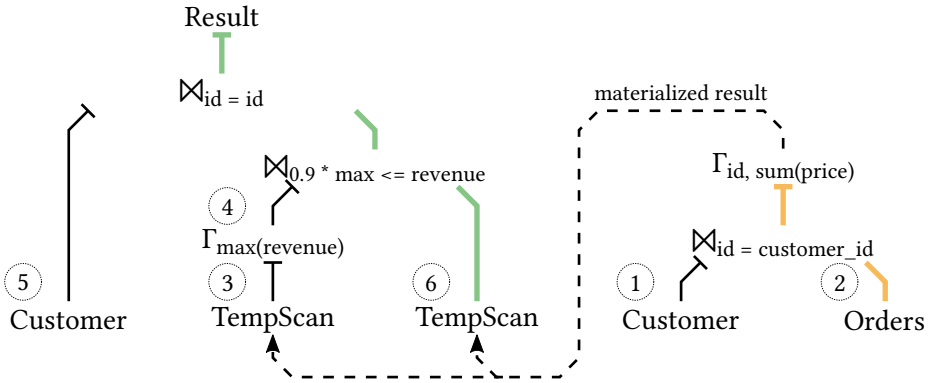


Figure 6.2: Exemplifying SQL OLAP query (top) and corresponding query plan (bottom) with pipelines. Pipelines continuing through an n-ary operator are marked bold and color-coded. Circled pipeline IDs also denote execution order.

6.1.1 Background

Having defined the goal of on-demand state separation, we need to discuss the requirements a database system has to fulfill to support it, as well as the properties of relational queries we use for our approach. For this, we will use the exemplifying query in Figure 6.2 and its query plan, which serves as a running example throughout this chapter. First, we find high-grossing customers using the common table expression *customer_revenue*. Then, we reconnect these customers to the customer table to extract all information required to send them birthday cards.

Query Properties

Our approach is based on relational queries following a query plan such as the one displayed in Figure 6.2. Plans consist of operators, such as joins, aggrega-

tions, and filters. These operators can be grouped into two categories, blocking and filtering operators. Blocking operators, such as the *id, sum* aggregation, materialize all tuples before reporting the result. Filtering operators, on the other hand, do not materialize tuples. Operators can exist as a filtering and a blocking operator simultaneously. The *id = customer_id* join, e.g., will materialize all tuples from the customer relation but only filter tuples from the orders relation without materializing them. We call all paths in the query plan in which a tuple is not materialized, i.e., between two blocking operators, pipelines.

Each pipeline is executed exactly once for all tuples of its input. Nevertheless, the result of a pipeline may be used multiple times. The result of pipeline (2) in Figure 6.2, for example, is scanned twice, namely in pipelines (3) and (6). Furthermore, a pipeline fully depends on all of its inputs. Before pipeline (6) can start, pipelines (2), (4), and (5) must finish their execution. These dependencies result in the execution order denoted by the pipeline IDs in Figure 6.2.

System Requirements

A system has to fulfill three main requirements to support on-demand state separation, which we will discuss below. As we have implemented our approach within the Umbra database system [118], we give a brief overview of how Umbra adheres to these requirements.

Plan-Based Execution. In our approach, we process queries using relational operators and pipelines. Therefore, we also require the system to process queries based on relational operators. Query plans are the default execution model for relational databases. Therefore, most existing database systems adhere to this requirement. Furthermore, the system must support the serialization and deserialization of plans for execution, either through dedicated formats or by emitting SQL. Umbra, e.g., uses a pipeline-based execution model, which we describe in Section 2.2. In this model, the query is split into pipelines, which in turn are translated into code and compiled for execution. Umbra further supports the export and import of query plans to and from JSON format, which we use to share queries between instances.

Query Progress Information. As one goal is to preserve already achieved query progress, systems have to offer insights into the progress of running queries. This progress information is already part of query execution in interpreting systems, such as MonetDB [23]. Compiling systems, such as Umbra, which convert queries to machine code, require active progress-keeping. In Umbra, it is not the entire query but its individual pipelines that are converted to machine code. This pipeline-based conversion allows us to keep track of the query progress at pipeline granularity.

Accessible Intermediate Results. Finally, our approach requires access to the intermediate results held for a query as they materialize the progress achieved. While interpreting engines access these results directly for query processing, compiling engines typically only access them through generated code. For state separation, however, compiling systems need to maintain information on these intermediate results outside of generated code as well. Umbra, for example, manages the state for queries in two different regions, thread-local and global state, which we describe in Section 2.3. The former is used for intra-pipeline processing and is thus not relevant to our approach. The global state, on the other hand, holds all data shared between pipelines, such as materialized results, and allows us to access them from the database. We will discuss the access in detail in Section 6.3.4.

6.1.2 State Model

After describing the goal and system model for state separation, we can define what *state* is relevant for extraction. As we discussed in Section 6.1.1, the intermediate results of a query materialize its progress. Thus, we only have to focus on these results. This allows us to remove all database-wide information, like indices, from our consideration, as this information is available to all nodes in a cluster in a uniform fashion.

Intermediate results are commonly materialized in blocking operators, i.e., at the end of pipelines. Vectorized systems such as MonetDB [23] materialize results in every operator. In our example in Figure 6.2, the results are, e.g., materialized in the join hash table after pipeline (1) and the aggregates after pipeline (3). Further, we abstract from the physical representation of the result, such as hash tables. This physical representation can vary greatly between systems and even between different instances of the same system. For example, even switching from a hash join to a blockwise-nested-loop-based join implementation for the same query and system will change the materialization of the results, even though the results will be identical. Therefore, we only consider tuples in the materialized results, not their surrounding index structures. Finally, we only consider results that are still required for query processing. After pipeline (2) has finished, the join hash table produced by pipeline (1) is no longer required and thus not considered part of the state. The results of pipeline (2), in turn, will be used by both pipelines (3) and (6). It is, therefore, part of the query state until both have finished. To summarize, we define query state as follows:

Definition 2. Query State: *The query state comprises all tuples materialized within the blocking operators of finished pipelines connecting to not yet finished pipelines.*

We will assume this definition when speaking of *state* in the remainder of this chapter. After pipelines (1) to (4) are finished in Figure 6.2, for example, the query state would contain all tuples in the *id, sum(price)* aggregation, as well as all tuples in the $max \leq revenue$ join hash table.

6.2 State Analysis

Having defined what constitutes the state of a query, we can now analyze the state of typical OLAP workloads. We base this analysis on the well-known TPC-DS OLAP benchmark [1], which models a warehouse for a decision support system. To represent medium-sized workloads, we choose scale factor 100 (SF100), which roughly equals 100GB of data. We analyze the state of each of the 103 TPC-DS queries after every pipeline in query plans generated by the Umbra database system [118]. TPC-DS distributes queries uniformly. Therefore, we include all queries and variants once in our analysis. As the state is comprised of only required tuples and columns in SQL-defined data types, the state size only depends on the query plan and join order, and not on the system used.

6.2.1 State Size Distribution

As the first analysis, we look at the distribution of state sizes occurring throughout all queries. For this, we measure the size of tuples stored after each pipeline and the number of blocking operators materializing these tuples. Figure 6.3 displays the results. One can see that the vast majority of states comprise few operators with less than ten megabytes of data. While most states are small, several states are larger than five gigabytes. Forty-six states exceed one gigabyte in size, while 626 are smaller than 1MB. Overall, 30% of states contain a single blocking operator with less than 1MB of data, and 86% of states do not exceed 100MB. Even though the median state size is only 133KB, the mean state size is 265MB. While 90% of states comprise fewer than five operators, up to 15 operators are involved for some queries.

All intermediate results of a query are relevant for state-separating architectures as the state has to be synchronized after every task. Therefore, we also look at the sum of intermediate result sizes occurring for each query. Figure 6.4 depicts the distribution of average and total state size per query. One can see that the total state size far outweighs the average. Compared to the mean size of a single state, the mean of all states is 2.6GB, and thus, 10× larger. This sum of state sizes is an upper bound for the state of a query, as it can contain the same pipeline result multiple times. For our example in Figure 6.2, the result of pipeline (2) is part of all states starting from pipeline (3). However, entirely

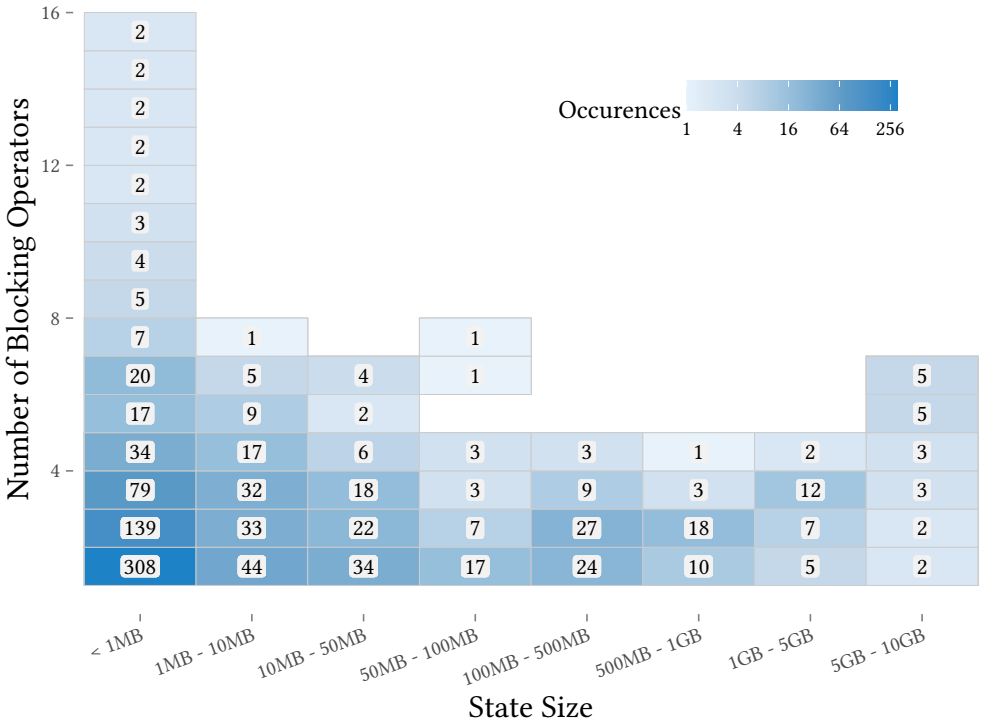


Figure 6.3: Distribution of state sizes occurring within TPC-DS SF100 by the number of blocking operators involved.

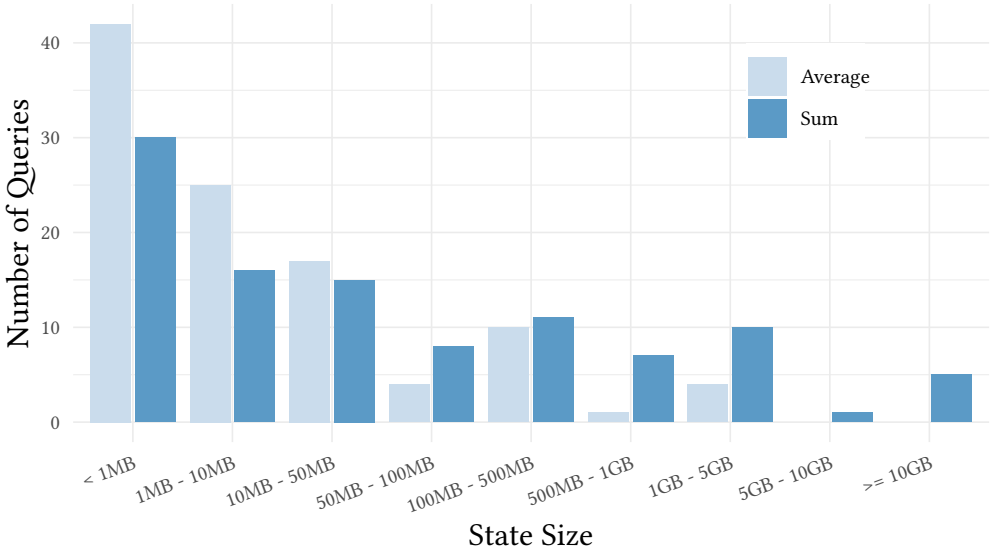


Figure 6.4: Comparison of the distribution of average and total state sizes per query for TPC-DS SF100.

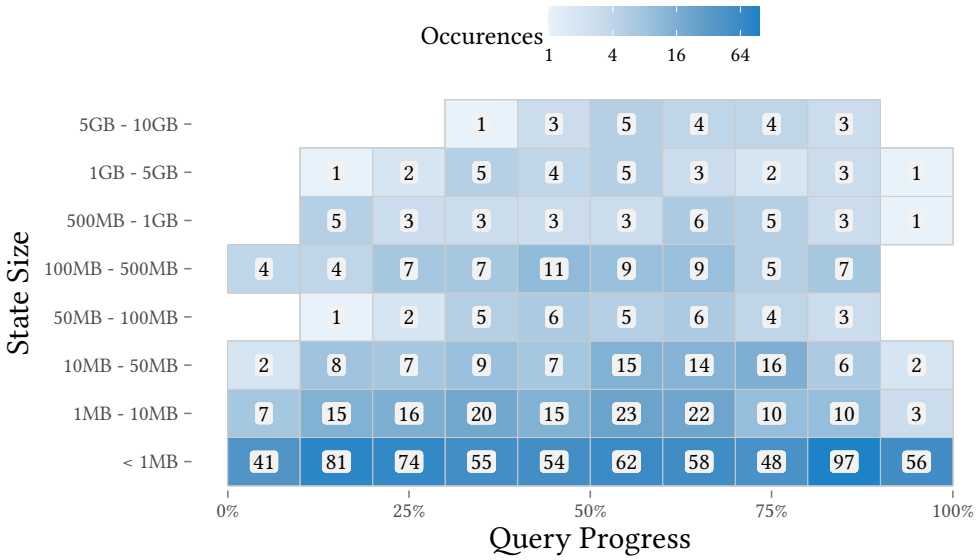


Figure 6.5: Distribution of state sizes occurring within TPC-DS SF100 by query progress.

excluding these duplicates would be inaccurate as well, as they have to be transferred to workers multiple times. E.g., the result of pipeline (2) is required by workers for pipelines (3) and (6).

6.2.2 Influence of Query Progress

Given the overall state distribution, we want to analyze further if and how the state distribution relates to query progress. Therefore, we must first define query progress for our model. As we are only interested in the states occurring after pipelines, we define the progress metric based on pipelines only:

Definition 3. Query progress $= \frac{\#\ of\ finished\ pipelines}{\#\ of\ total\ pipelines}$

While this does not account for the runtime of individual pipelines, it considers the task-based scheduling of cloud jobs. The distribution of state sizes along this query progress for all TPC-DS queries is shown in Figure 6.5. While large states seem to occur less frequently close to the start and end of queries, the overall distribution shows no significant trends. Both large and small states can occur during every phase of query execution. However, this distribution could be skewed by a few queries with a large state. Figure 6.6 displays the distribution of state size normalized to the maximum state size for each query to account for this skew. One can see that the trend partly revealed in Figure 6.5

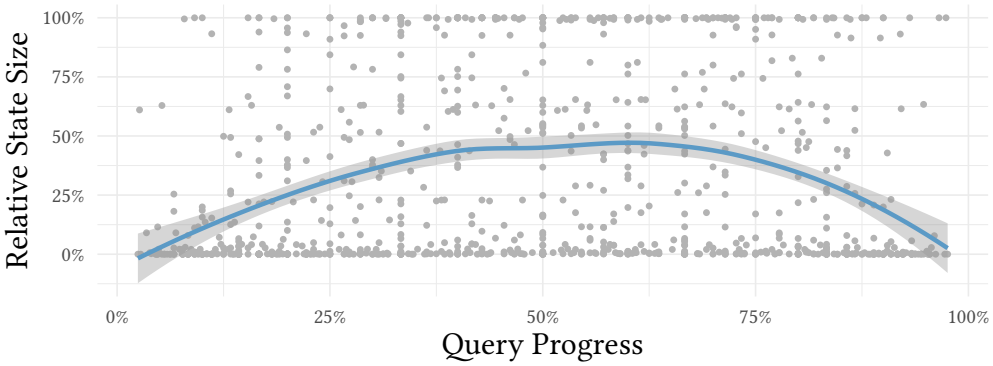


Figure 6.6: Evolution of relative state size during queries. State size is relative to the maximum state size reached for a query.

is more apparent here. On average, the state size grows until around 40% of pipelines are completed and plateaus until around 70%. From there on, the state continuously shrinks. The structure of query plans can explain this. In the beginning, queries collect a lot of data, e.g., in join build sides. In Figure 6.2, e.g., the first two states include only a single pipeline result. From there on, at least two results are part of the state: at least one join build side and the state of the probe pipeline. After pipeline (5), the state is maximal with three materialized pipelines results ((2), (4), and (5)).

6.2.3 Discussion

In this section, we have shown the overall distribution and trends in the query state of all TPC-DS queries. However, we have not discussed the implications of state separation arising from this data. Overall, the state sizes in Figure 6.3 are promising for state separation. Assuming a 10Gbit/s network connection between servers, a round trip for the mean state size takes only 424ms. Still, a complete round trip after every state can add up quickly. In the worst case, up to 9.1GB must be transmitted for a single state, resulting in a 14.6 second round trip. For these large states, transfer time alone can already exceed the execution time of local queries, making re-execution in case of failure more profitable than state separation.

When considering all states occurring for queries (cf. Figure 6.4), the potential network overhead increases further. Examining the sum of state sizes, the mean of 2.6GB and a maximum of 162.8GB lead to a 4.2 and 261 second round trip, respectively. Nevertheless, given that 86% of single states can be transferred to other workers in less than 160ms, state separation of single states can be profitable for the vast majority of queries.

Evaluating the distribution of state sizes during individual queries in Figure 6.6 shows that state separation is best early on or close to the end of a query. However, as the relative cost of a restart increases with query runtime, migrating the larger states occurring between 25% and 75% progress might still pay off.

6.3 On-Demand State Separation

The advantages of state separation are well known for cloud environments. Being able to add and remove workers and handle worker crashes offers the flexibility desired by customers. However, synchronizing the state over the network can be expensive, especially for single-worker queries. It is not necessary for those to shuffle state to workers between tasks, and thus, every network transfer is overhead in query execution. As shown in Section 6.2, sending every state over the network instead of only one can make a 10× difference on average. Furthermore, our approach can optimize for local execution, generating no execution overhead when no state separation is required. While on-demand state separation and migration solutions exist based on virtual machines (VMs), these treat the VM as a blackbox. Therefore, they either have to restart tasks [175] or migrate the entire VMs memory state [140, 145], which is bound to be much larger than just the query state. Approaches that use extensive knowledge about the inner state of queries [139, 173] rely on pre-defined checkpoints. These must be defined before a task starts and cannot be created retroactively on demand.

To achieve a minimal migration state without the need for less flexible checkpoints, we propose scanning and extracting the currently materialized query state, as defined by Definition 2. As this state is part of the execution process, it is accessible at any time without prior preparation. In the remainder of this section, we will describe the high-level design of our approach and the prototypical implementation within the Umbra system using the exemplifying use case of query migration:

Definition 4. ***Query Migration.*** *Query migration is the process of moving the processing of a query q from an executing server A to a server B without losing the progress achieved for q on A.*

6.3.1 Deployment Environment

The deployment environment is of great importance to enable on-demand state separation, especially in the presence of transient compute resources. We show a possible server configuration in Figure 6.7. In order to keep the state held in finished pipelines when a worker fails or is taken away, it has to be kept in a durable, external location. Therefore, when state separation is required, all

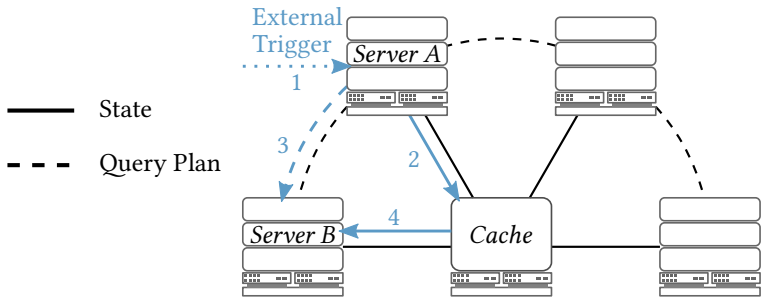


Figure 6.7: Server cluster for on-demand state separation and query migration. State is only shared via a network cache, query plans can be migrated peer to peer. Exemplifying data flow for a migration from server A to server B is highlighted in blue.

tuples that are part of the state must be cached externally. As state sizes can reach gigabytes (cf. Section 6.2), all workers (e.g., servers A and B) must access the cache through high-bandwidth connections.

Furthermore, the workers have to communicate to transfer the accompanying query plan. In our current setup, this is realized by peer-to-peer connections between workers. A peer-to-peer setup is the most lightweight option for coordination, requiring few messages and no additional servers. It is, thus, ideal for small and stable deployments. However, it is also possible to handle these transfers using a dedicated coordinator in larger deployments. While this adds communication overhead and requires an additional server, it offers greater flexibility. For example, a coordinator can monitor the running instances to detect migration needs and deal with servers joining and leaving the cluster. While Figure 6.7 only focuses on those servers and components relevant for state separation and query migration, real deployments will also include additional servers for the cache and storage servers required for storage separation.

There are additional constraints for caches. For one, the workload is different from traditional key-value store workloads. In contrast to those, our data is ephemeral. For query migration, the state is written and read exactly once, often directly after each other. Once the data is read, it is no longer required and, thus, discarded. Furthermore, the state sizes can pose a problem. Many cloud key-value stores limit the maximum value size. The popular key-value database Redis, e.g., has a limit of 512MB [127] for individual values. However, the state analysis shows that the state size can reach gigabytes easily. For our workload, we require high throughput and low latency under high write and read loads for large value sizes. We found the system closest to our requirements to be Apache Crail [144], as it is optimized for ephemeral data and has no limit on value size.

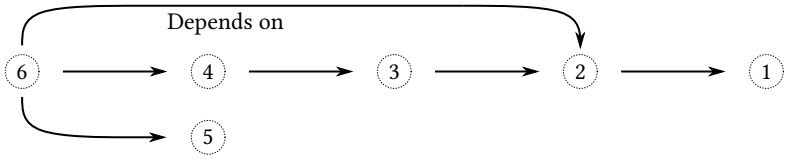


Figure 6.8: Dependency graph between all pipelines of the query plan of Figure 6.2. Transitive dependencies are irrelevant for state selection.

6.3.2 Process Overview

Having defined the deployment environment, we can describe the outline of our on-demand state separation process. We will use the query migration use case of Definition 4, as it is the most involved. Other possible use cases for state separation, such as deferring execution to prioritize other queries or snapshotting, can be realized with the functionality utilized for query migration. For example, consider the migration of the query in Figure 6.2 from server A to server B in Figure 6.7.

First, the need for migration is detected and reported to the server (1). Migration can be triggered by several events, e.g., the indication by the cloud provider that a transient compute resource is being taken away soon or the availability of a faster or cheaper spot instance. Then, the current state of a query according to Definition 2 has to be identified at server A and extracted from structures such as hash tables. Server A then transfers the extracted state to the external cache (2). On server A, the query plan is then adapted to continue from the current state and transferred to the receiving server B (3). Server B then compiles the received query plan and continues the execution. Whenever a partial result from the state has to be scanned for the first time, it is fetched from the cache and kept locally (4). Snapshots can be realized by periodically sending the current state and the adapted query plan to the cache. Deferring queries is a special case of snapshotting, as the worker does not need to change. Therefore, the state and modified query plan can also be kept locally, thus saving the cost of network transfer. We will use the remainder of this section to describe the steps above in detail.

6.3.3 State Selection

Once a server has been notified of a desired state separation, the execution of the current query is halted. Then, we identify all operators that are part of the state. For this, we track the current query progress in the form of finished pipelines throughout query execution. Further, to prune all pipeline results no longer required for execution, we calculate all direct dependencies between pipelines.

Algorithm 6.5 Selecting blocking operators contained in a state

```
1:  function SelectStateOperators(finishedPipelines, dependencies)
2:      dependents ← invert(dependencies)
3:      stateOperators ← ∅
4:      for p ∈ finishedPipelines do
5:          anyUnfinished ← false
6:          for dep ∈ dependents[p] do
7:              if dep ∉ finishedPipelines then
8:                  anyUnfinished ← true
9:          if anyUnfinished then
10:             append(stateOperators, p.blockingOperator)
11:     return stateOperators
```

Definition 5. Direct pipeline dependency: *A pipeline A directly depends on a pipeline B if pipeline A directly requires the result of pipeline B for execution.*

In Figure 6.2, e.g., pipeline (3) depends on pipeline (2), but not on pipeline (1). Figure 6.8 depicts dependencies of all pipelines of our example. One can see that the dependencies form a DAG. However, it is still possible that a pipeline is another pipeline's direct and transitive dependency. Given these dependencies, we can now select those finished pipelines still part of the state. The algorithm for this is shown in Algorithm 6.5. First, we invert the dependency mapping to get all dependents for a pipeline (line 2). Then, we iterate over all finished pipelines, identifying those that are part of the state (line 4). If all dependents of a pipeline are finished, the pipeline's materialized result is no longer required and, therefore, no longer considered part of the state (lines 6 - 8). Finally, we collect all blocking operators of finished pipelines with at least one dependent for state extraction (line 10).

Directly after pipeline (4) finished executing in our running example, the finished pipelines contain pipelines (1), (2), (3), and (4). Of those, state selection discards (1) and (3) as all their dependents ((2) and (4), respectively) are already finished. We remember the blocking operators for state extraction for the two remaining pipelines, namely, the grouped aggregation *id, sum(price)* and the $max \leq revenue$ join.

6.3.4 State Extraction

Having found all operators containing state, we have to extract the individual tuples that comprise this state. While a closer mapping to the current state would be to migrate tuples within index structures, such as hash tables, we

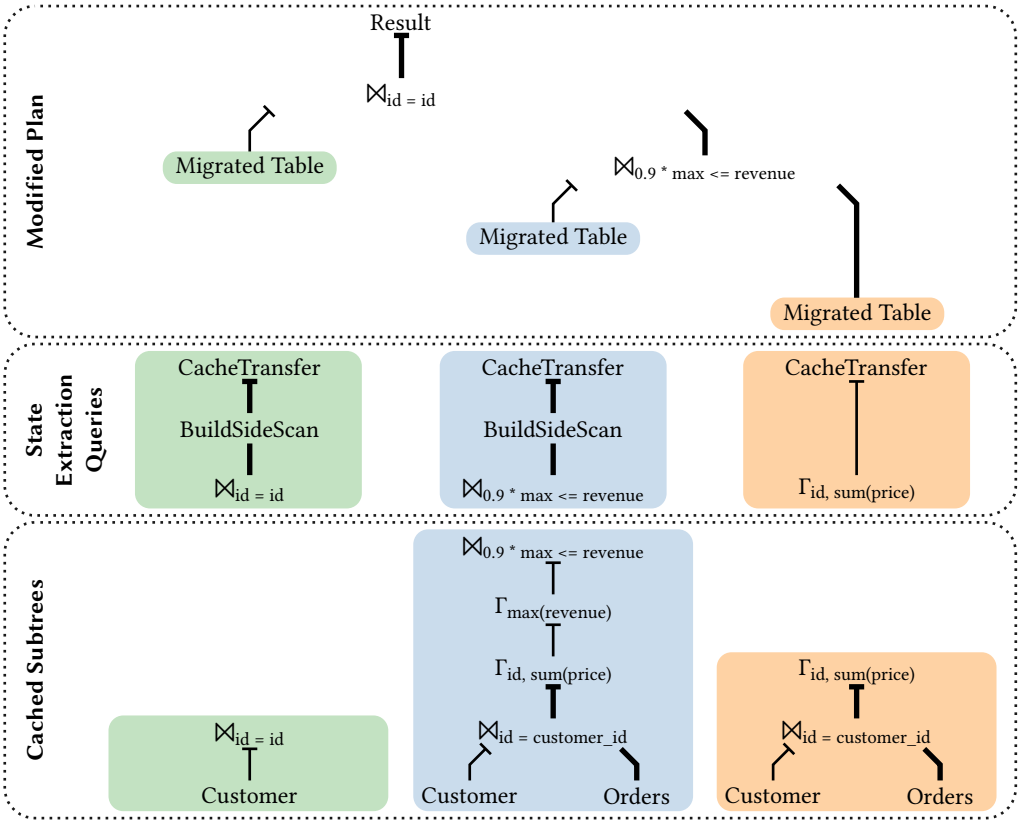


Figure 6.9: Query migration artifacts of the query in Figure 6.2 when migrating after pipeline 5. Bottom: Subtree to be extracted as state and held in the cache. Mid: State extraction queries to be run on the migration source. Top: Modified plan to continue execution on the target server.

extract state as defined in Definition 2. State held in operators is optimized for efficient local processing, e.g., by keeping it in pointer-referenced storage and hash tables. This configuration differs between operators and is hard to serialize for network transfer. Furthermore, optimal state structures might differ between servers. Migrating only tuples allows the target system to re-create this per-operator state in a configuration optimized for the local deployment as if it resided in a table. We first want to highlight the high-level process of this tuple-based state extraction before giving a detailed description of the implementation within our system. In general, we have to distinguish between two different kinds of blocking operators.

The first type is operators that only appear as blocking operators within a query, which we call scan-optimized operators. This category comprises unary blocking operators, such as aggregations, sorting, set operations, and

potential specialized operators such as K-Means, window functions, or sampling. These operators are the easiest to extract tuples from, as they already offer functionality to scan all tuples. Such operators are always sources of pipelines using their results. Therefore, they produce all tuples when scanned, allowing us to re-use this scan functionality. In many systems, operators can be scanned repeatably to optimize queries, like the *id, sum(price)* aggregation in Figure 6.2. Functionality for repeated scans further enables state extraction for snapshots without interfering with query execution.

The second type of operator is those with more complex access patterns, such as joins, which appear as a blocking and as a filtering operator. Hash joins, e.g., are optimized for point accesses on the join predicate and often do not offer functionality for full scans. Fortunately, there are only a few operator types in this category. This category only contains different join implementations with non-linear access optimizations in our system. Nevertheless, joins frequently occur in queries and should be considered for query migration. While these operators are not optimized for full scans, their internal structures often still support such scans. Blockwise-nested-loop joins, e.g., materialize their build-side fully without additional indices, making scans easy. Most index structures, such as hash tables and trees, can be scanned efficiently, allowing us to support the migration of all operators currently implemented within our system.

In the following, we will describe the implementation of query migration for both scan-optimized and index-optimized operators. As the implementation of operators varies heavily between systems, we will limit the implementation to the Umbra [118] system. We outline the requirements and possibility for state extraction in other systems in Section 6.1.1.

Implementation

To best utilize existing infrastructure within the database, we implement the extraction process as a regular query. All tuples are materialized within operators during execution, often nested in a complex operator state. Extracting this state with a query allows us to re-use existing logic and access paths. Furthermore, this allows us to access all optimizations and features of in-database query execution, such as specialized code generation [117] and morsel-driven parallelism [99]. Especially for scan-optimized operators, the state extraction can be realized almost entirely with existing code and logic, allowing easy integration into an existing system.

The extraction query plans differ for scan- and index-optimized operators, which we will describe below. Once we have generated this plan, the remaining steps are identical: The query plan is compiled and given access to the state of the query to be migrated or snapshotted. In contrast to regular queries, our state

extraction queries do not report the result to the user. Instead, the resulting tuples, i.e., the query state, are collected in a compact format and sent to the cache. Our approach schedules all extraction queries for immediate execution and exclusively to prevent modifications to the state before the extraction is complete.

Scan-Optimized Operators

State from scan-optimized operators can be extracted using only the extracted operator's logic. For this, we duplicate the existing operator into a new query plan and link the copy to the state of the operator selected for extraction. An example of such an extraction plan for a scan-optimized operator can be seen in Figure 6.9 for the *id, sum(price)* aggregation. We can again use the fact that scans of an operator's state are non-destructive and reference the state of the existing operator, avoiding a costly copy of the whole operator state.

Index-Optimized Operators

Index-optimized operators require a more in-depth analysis of the state to extract tuples. While it would be possible to generate extraction plans using only existing query logic, e.g., by modifying joins to run against a single tuple that joins with all build-side tuples, we opted to implement dedicated extraction operators instead. Using dedicated operators, we can often bypass the operator's access paths and directly access the data for a scan. This more efficient access strategy comes at the cost of implementing extraction logic for all index-optimized operators. However, as stated above, there are only a few operators in this category that often occur. As all these extraction operators follow a similar pattern, we will not detail the implementation for every operator. Instead, we will describe the high-level implementation based on a hash join.

Consider, e.g., the *id=id* join extraction in the top left corner of Figure 6.9. In it, we need to extract all tuples stored in the build-side hash table of the hash join. All operator states are well-defined within Umbra, as outlined in Section 2.3. Therefore, we can locate the hash table from the operator state and make it accessible to our build-side-scan operator. This scan operator then loops over all buckets, extracting all key-value pairs stored within to recreate the tuples. All other specialized extraction operators in our system follow this pattern of accessing the structure holding tuples in the operator's state to be extracted. Again, this scan is non-destructive, and therefore, we do not have to copy the hash table to extract tuples.

Algorithm 6.6 Modifying the query plan to use the extracted state

```
1:  function ModifyPlan(stateOperators)
2:      opsToExtract ← ∅
3:      for op ∈ sortPreOrder(stateOperators) do
4:          if isIndexOptimized(op) then
5:              toReplace ← op.buildSide
6:              replaceIn ← op
7:          else
8:              toReplace ← op
9:              replaceIn ← op.parents
10:         migratedTable ← buildTable(toReplace.types)
11:         for location ∈ replaceIn do
12:             if location.isValid() then
13:                 location.replace(toReplace, migratedTable)
14:                 append(opsToExtract, op)
15:     return opsToExtract
```

6.3.5 Plan Modification

In the final step on the source server, we need to adapt the query plan to incorporate the cached state instead of the subtrees it represents. To achieve this, we modify a copy of the existing query plan. Algorithm 6.6 displays the pseudo-code for this query plan modification. For every operator in the state, we again have to differentiate whether it is scan- or index-optimized in the current query plan. Index-optimized operators are not the blocking operator of the final pipeline passing through them. Therefore, we cannot replace them entirely with the tuples contained in their state. Instead, we mark the build-side child for migration in the operator itself (line 5). For simplicity, we only consider binary operators with one build side in Algorithm 6.6. The procedure for n-ary operators is orthogonal, replacing all finished pipelines ending at the state operator with the corresponding state. In Figure 6.9, this can be seen for the *id = id* and *max ≤ revenue* joins. Our approach replaces only the build side subtrees and not the entire joins in the modified plan.

Scan-optimized operators can only be part of the state operators if all their inputs are finished. Therefore, we can replace the entire operator with the state held within (line 8) without losing progress. However, in contrast to index-optimized operators, it is possible that we need to replace the operator in multiple places as scan-optimized operators can be scanned multiple times within the same query. The *id, sum* aggregation of Figure 6.2, e.g., is scanned twice. Therefore, a migration after pipeline (2) needs to replace it in both parent

pipelines (3) and (6). One can see that this can lead to conflicting replacements: For example, in the migration displayed in Figure 6.9, the *id, sum* aggregation is part of the state, and thus, replaced in both parents. However, one parent is further replaced in the *max* ≤ *revenue* join's build side. For such cases, we always want to ensure only the topmost replacement takes place, as it preserves the most progress. To achieve this, we perform replacements top-down by sorting the state operators (line 3) and always check whether the replacement location is still valid, i.e., contained in the query plan (line 12). This way, replacements will always be optimal, as the topmost operator is considered first. Replacements in the lower part of the tree will either be performed later on or will not occur if the location is no longer valid. To prevent needless network transfers, we only extract state from operators that are part of the final query plan, i.e., all operators part of a valid replacement (line 14).

6.3.6 Query Migration and Continuation

Once we extracted all tuples that are part of the state and have generated a query plan utilizing this state, the query can be sent to the desired target. In the scenario outlined above, migrating a query from server *A* to server *B*, neither the state nor the query are initially available at the destination server. In the first step, server *A* sends the modified query plan to server *B* and then aborts the local execution. In turn, the query plan is compiled and executed on server *B*. Whenever the execution reaches the first scan of a migrated table, the table is fetched in parallel from the network cache and held locally for potential subsequent scans. In the case of a migration, the cache can discard each stored value after the first read.

Pausing a query works orthogonally without the need for network transfers. Instead of caching the query plan and state externally, our approach would materialize them in the memory or persistent storage of the worker. Once both are materialized, we abort the query locally to free all working memory for the prioritized query. When the prioritized query finishes, we load the plan from disk and continue its execution. Finally, snapshots register both the state and query plan with the cache. Once all data is cached externally, execution continues on the local server.

6.3.7 Applications

So far, we have focused on migrating queries between servers. Migration alone already offers several benefits. It can save cost by utilizing cheap spot instances without risk and improve performance by changing to better-suited instances at runtime. However, we understand on-demand state separation as a toolkit that

can also be applied in other scenarios. First, as already discussed in the previous subsection, our approach allows users to suspend queries cheaply to prioritize latency-sensitive tasks when compute resources are limited. The snapshotting mechanism of Section 6.3.6 can be used to deal with worker failures, which we have not discussed so far. In the after-the-fact query migration use case, we rely on prior notice to migrate, which is unavailable in the case of crashes. In order to avoid restarts, this mechanism can be used to create periodic snapshots of a query. In case of a failure, we assign the latest snapshot of the query plan to a new worker, which again fetches migrated tables on demand and continues execution. Furthermore, applications of our approach are not restricted to single-worker queries alone. The extracted state caches independent subtrees of a query, as can be seen in Figure 6.9. Thus, these subtrees could be executed in parallel on different workers and combined using the steps outlined in this section, effectively enabling scale-out for existing systems.

While they are the focus of our work, possible applications of on-demand state separation are not limited to distributed settings. Materializing tuples with information about their corresponding subtree (cf. Figure 6.9) can be used to share and re-use intermediate results with other queries [76, 137]. Further, our approach can be used to re-plan queries in the event of network delays [11, 156] or cardinality misestimation in the optimizer [15, 108].

6.4 Evaluation

Our evaluation is twofold. In the first part, we provide an in-depth analysis of the amount and sources of the overhead of on-demand state separation on query processing in a series of microbenchmarks. In the second part, we demonstrate the feasibility of our approach for typical cloud use cases. We conduct all experiments in this section using our approach within the Umbra database system [118].

6.4.1 Setup

To emulate a cloud environment, we run all experiments in this section in a cluster of 4 nodes. Each node is equipped with an Intel Xeon CPU E5-2660 v2 (2.20GHz) and 256GB of DDR3 RAM. The nodes connect to the cluster through a Mellanox ConnectX–3 VPI network interface card (up to 56Gbit/s FDR Infiniband) via a Mellanox SX6005 switch. While an RDMA Infiniband configuration would be most performant, many cloud providers rely on Ethernet connections between servers. For this reason, our implementation uses the TCP network

stack as well, and we configure our cluster to run on IP-over-Infiniband (IPoIB) instead of full-fledged Infiniband to emulate a more typical cloud setup.

Two nodes act as source and target servers for query migration, which is the main focus of this evaluation. Each server runs an Umbra instance on a local copy of the TPC-DS SF100 database held in an in-memory file system. This way, we guarantee equal access to the base data, simulating storage separation. The two remaining nodes form the Apache Crail-based network cache [144], with one acting as a namenode and one acting as a datanode. While Crail offers an optimized RDMA-based mode, we again opt for a TCP-based infrastructure to better simulate a typical cloud setup.

6.4.2 Microbenchmarks

On-demand state separation comprises many individual steps, as outlined in Section 6.3. Before demonstrating the feasibility through end-to-end benchmarks, we first want to analyze the sources of the introduced overhead in these steps. The two main categories in our analysis are network and execution overhead. The first arises from the topology of the cluster setup and external components, such as network caches, which we cannot directly influence. Overhead stemming from our approach is mainly execution-based, that is, analyzing and extracting the state locally and continuing execution on the remote server.

Configuration. For all experiments in this section, we report overheads based on an average of 5 runs. To provide a detailed analysis, we measure the individual runtime of all sub-steps of migrating after every pipeline occurring in the 103 TPC-DS queries. Further, we perform full migrations and configure both the source and target server to run an identical configuration of Umbra, thus minimizing any configuration influence on runtime. However, we still detected runtime variance in preliminary experiments for local-only and migrating runs, even with identical configurations. Thus, we report overheads as a percentage of the runtime of an entire migration.

Execution Overhead. In the first microbenchmark, we want to highlight the overhead caused by our approach. Multiple factors comprise this overhead: On the source server, this includes state selection (Section 6.3.3), plan modification (Section 6.3.5), and compiling state extraction queries, as well as running the extraction up to, but excluding, network transfer (Section 6.3.4). Further, the execution overhead includes parsing and compiling the received query plan on the target server. We compare the overhead generated solely by our approach for two server configurations. Once Umbra is allowed to use up to four worker threads, once up to eight. Figure 6.10 shows the resulting overheads.

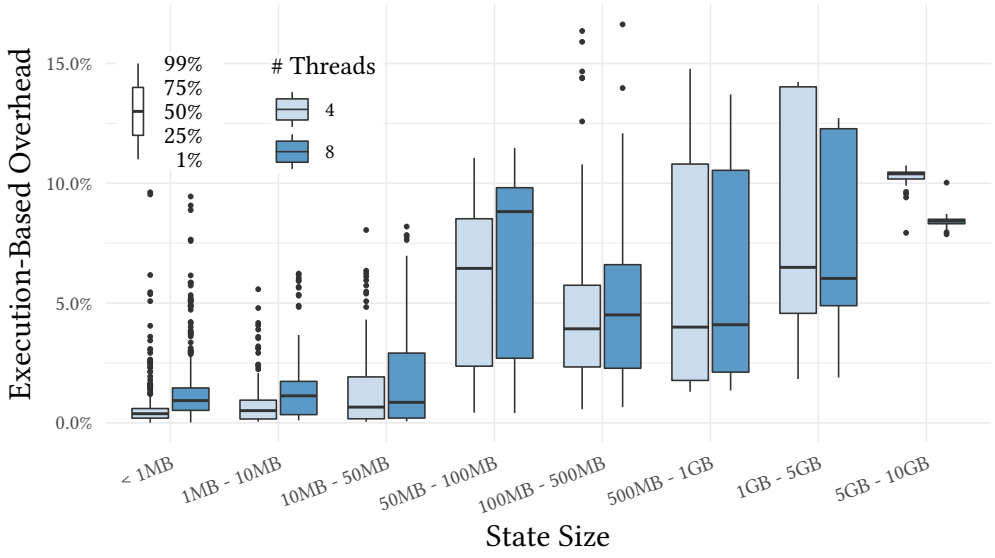


Figure 6.10: Execution overhead by state size when migrating TPC-DS queries.

One can see that there is a trend along with the state size for both configurations. When migrating larger states, the overhead grows as well. We expect this increase, as all tuples must be scanned at least once for extraction when materializing them for network transfer. Furthermore, there is no apparent difference between four and eight threads in terms of overhead, indicating that our parallel extraction scales as well as Umbra's query execution framework. This scaling further shows the benefits of utilizing extraction queries in our approach, through which we gain access to parallelism and scheduling optimizations already present in the database.

For both configurations, one can see several outliers for small state sizes. These are primarily from small and fast queries with execution times in milliseconds, where execution does not fully amortize the cost of compiling extraction queries. Nevertheless, on average less than 11% of overall query runtime is spent processing state extraction and migration, independent of server configuration and state size. For states smaller than 50MB, which make up 83% of all states, the mean overhead does not exceed 1.9% independent of the server configuration.

Network Overhead. Having analyzed the processing overhead caused directly by our approach, we want to analyze the overhead caused by the necessary network transfers. While this overhead does not stem from our approach directly, and we thus cannot influence it within our system, it is crucial to understand the overall cost of on-demand state separation. The network overhead measured here is the time required to send and receive the extracted state to and from the

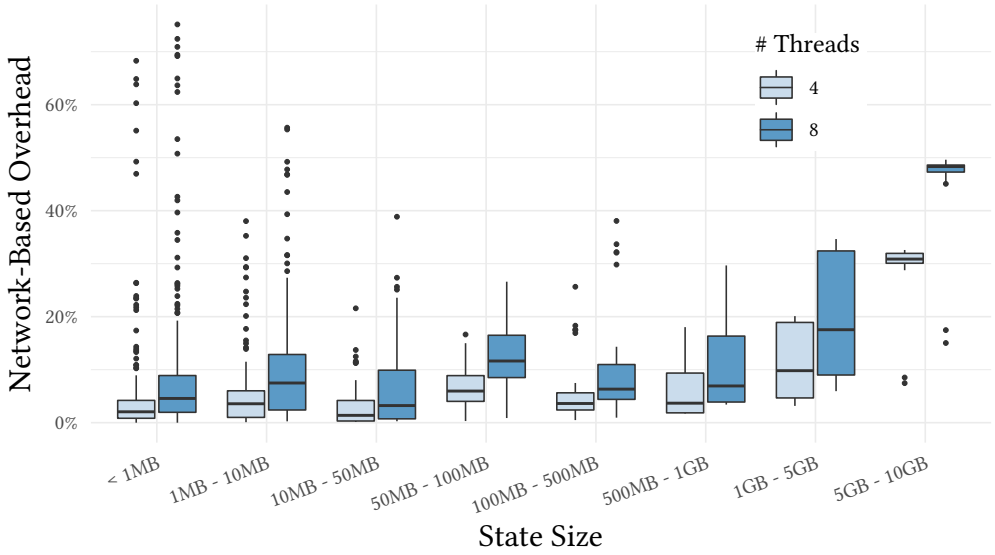


Figure 6.11: Network overhead by state size when migrating TPC-DS queries.

cache. Again, we compare the overhead for migrations between two instances with an equal number of worker threads and display the resulting overhead for all migrations in Figure 6.11.

Overall, the network overhead again clearly grows with the state size migrated. We expect this growth, as network bandwidth is limited and slower than local processing of tuples within a query. However, this overhead is less linear than we have seen for local processing, reaching an average of 45% for states between five and ten gigabytes when using eight worker threads. Furthermore, in contrast to the execution-based overhead, one can see that there is a noticeable difference between the configurations. The network transfer is not limited by the compute resources available but by the network bandwidth and latency. Even though network overhead exceeds processing overhead for most state sizes in both configurations, the mean overhead for states smaller than 50MB does not exceed 11% of query runtime.

6.4.3 Query Migration

Given the individual overheads from the microbenchmarks, we investigate how this translates into the cost of end-to-end query migrations compared to local execution. In addition to migrating between identically configured servers, we further investigate the advantage of fixing an adverse query-to-worker matching by migrating to a more powerful server. Furthermore, we highlight the advantage of our on-demand separation by comparing it to full-fledged state

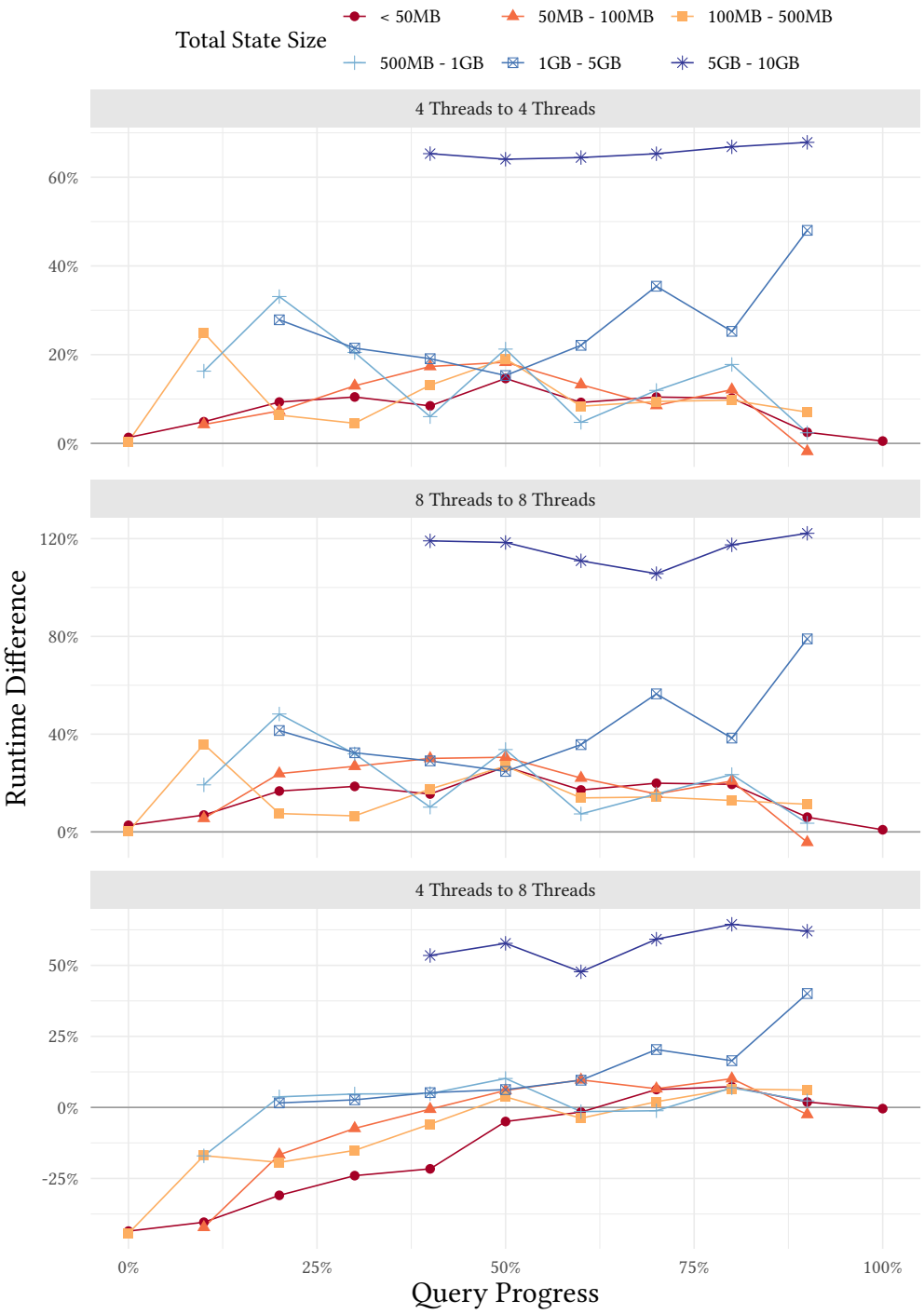


Figure 6.12: Execution time difference of query migration compared to local execution for 3 server configurations. First thread count denotes source worker threads, execution times without migration are measured on source server.

separation, where the state separation of Section 6.3 is performed after every pipeline.

Configuration. To capture the total cost of migration, we base all experiments in this section on end-to-end query runtime. Query runtime includes every step of query processing, from receiving the SQL query to fully reporting the result, either locally or, in the case of migration, on the remote server. We again migrate all 103 TPC-DS queries and report the average of five runs while re-using the server configurations from Section 6.4.2. Unless stated otherwise, all experiments in this subsection report the relative runtime difference between migrating a query and local-only execution on the source server. We show all states $<$ 50MB as a single group for better visibility as they behaved almost identically in all experiments. To better highlight trends, we round query progress to the nearest 10% and report averages within this interval throughout this subsection.

Symmetric Migration. In the first two experiments, we focus on the overall cost of migration. For this, we migrate between identical instances of Umbra for configurations with four and eight worker threads. This experiment simulates a transient worker being taken away and replaced by another at any part of the query. The results are displayed left and center in Figure 6.12. We again compare the results for different state sizes. One can see that states smaller than 1GB behave similarly, independent of server configurations. However, there is a significant difference between the two configurations, even for smaller states. We attribute this to the differences in network overhead, which we already identified in Figure 6.11. In addition to the state size, the migration point also influences the overhead. One can see that migrating between 30% and 80% of query progress is slightly more expensive than at the beginning and end of a query, even when state sizes are similar. We found that states in the middle of a query comprise more operators on average, leading to an increased overhead for compiling and managing state extraction even when the resulting state is of a similar size.

Most query migrations cause less than 25% overhead, making migrating queries more profitable than restarts right from early on. On average, migrating states smaller than 1GB causes 9.3% overhead when using four worker threads and 16.4% when using eight. However, it seems that migrating states larger than 5GB is seldom profitable, especially for the eight-thread configuration. The explosion in overhead for states between one and five gigabytes is caused by only two queries that are no longer compensated for by other states for progress $>$50%.

It is evident that the migration of large states is rarely profitable and will be outperformed by restarts. However, this does not mean that on-demand state

separation cannot be profitable for queries with large states. Because state scans are non-destructive, it is possible to extract older, potentially smaller states at the cost of some progress loss. This way, restarts are only required if a query does not have any small states.

Migrating to Better Instance. On-demand state separation can not only be utilized to migrate between identical instances. It further allows using more powerful instances when they become available, e.g., through spot instances or servers finishing their current query. Utilizing such instances promises the potential to speed up query processing. To investigate the benefits of migrating running queries to faster servers, we migrate from an Umbra instance with four worker threads to one with eight workers. The results, displayed on the right in Figure 6.12, show that utilizing a faster instance can speed up query processing in many cases, especially for smaller state sizes. As expected, migrating early will lead to the biggest speedup in processing because the faster compute can be used the longest. However, there are instances where migration pays off in every execution phase. Even when 90% of the query is completed, some small states' migrations are still beneficial. Migration is not the only possibility to leverage faster workers. In addition, one needs to consider restarting queries on the new worker.

Ideally, restarting on a worker with twice the compute power will speed up query processing by a factor of two as well. On-demand state separation can outperform such query restarts for many queries in our experiments. On average, migration outperforms restarts once a query reaches 30% progress. When a query has progressed more than 50% on the source server, migration is 67.5% faster when compared to a restart on the destination. Again, larger states are not profitable for on-demand state separation, and query restarts would outperform them throughout the experiment.

On-Demand vs Full State Separation. Having analyzed the cost of on-demand state separation for migrations, we want to compare it to full state separation. Full separation extracts state and synchronizes it with a cache after every pipeline. We compare the average cost of on-demand state separation with the cumulative cost that state separation after every pipeline will have incurred so far. For both approaches, state separation is performed in Umbra as outlined in Section 6.3. While this leads to a larger state than necessary (cf. Section 6.2.1), and approaches with full state separation could optimize for smaller states, the overall trends will prevail. The results of each blocking operator still have to be transferred at least once. In contrast to the previous experiments, the runtime overhead no longer includes a full migration, which would disproportionately affect full state separation. Instead, the overhead comprises the work required to extract the state at the migration source only.

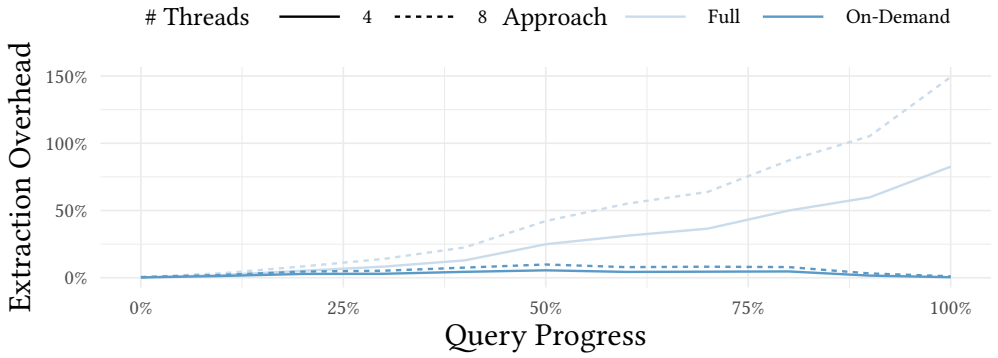


Figure 6.13: Extraction-caused runtime overhead for full and on-demand state separation for TPC-DS.

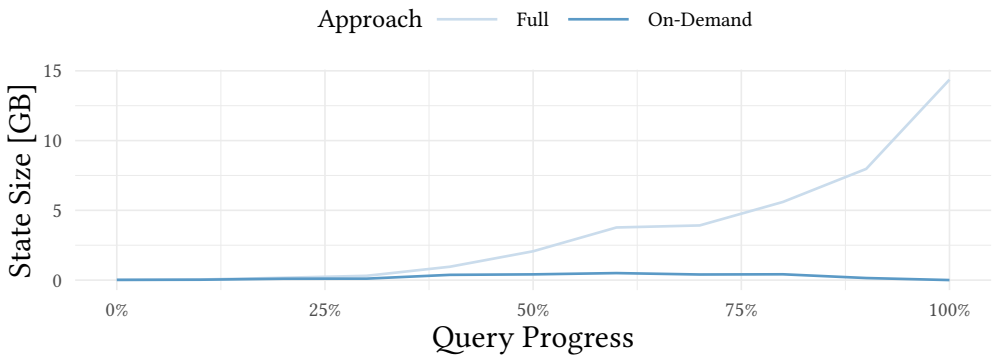


Figure 6.14: Size of state transferred to the cache for full and on-demand state separation for TPC-DS.

Figure 6.13 shows the resulting overheads in runtime. The execution time overhead shows the advantage of on-demand separation. While the overhead grows for full migration with query progress, the overhead of a single migration is almost constant throughout. A full migration causes more than 100% overhead for eight threads, whereas the overhead of on-demand separation never exceeds 10%. The influence of the number of worker threads identified in Figure 6.12 prevails for both on-demand and full state separation, even when only considering the overhead at the source. The more powerful the servers, the higher the overhead of state separation.

Following, we analyze the space required to cache the state of a query, displayed in Figure 6.14. As state size is independent of the number of worker threads involved, we do not distinguish worker configurations. The advantages of on-demand separation are again amply clear. While the average on-demand

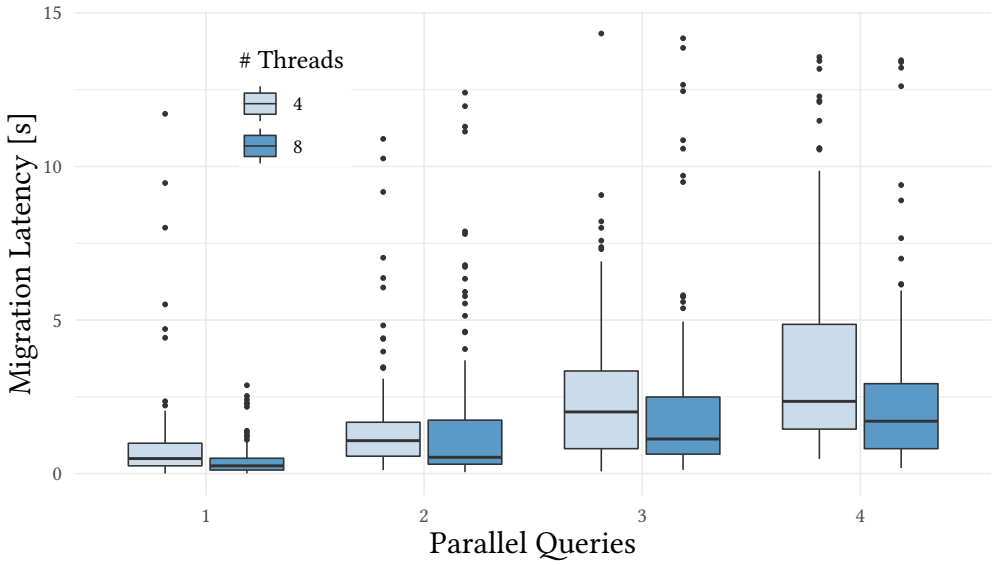


Figure 6.15: Migration latency when executing multiple queries in parallel.

migration never transfers more than 503MB of data, the average cumulative sum of states transferred to the cache in the last progress interval (95-100%) exceeds 14GB. The discrepancy to the mean sum of 2.6GB identified in Section 6.2.1 is caused by the per-interval mean. Queries comprised of more pipelines create more data for the mean, thus leading to an over-representation of longer queries. These queries further accumulate more state, leading to a 5× higher mean. Nevertheless, even when considering the mean of 2.6GB, on-demand state separation requires at least 5× less space in the cache throughout a query on average.

In environments where sudden worker failures are common, the increased cost of full state separation may pay off. However, we argue that the benefits of after-the-fact on-demand separation will outweigh the risk of losing progress in most deployments.

Migration Latency. Finally, we want to demonstrate the capabilities of our approach in real-world applications. We investigate an exemplifying use case of vacating a spot instance running multiple queries, e.g., when the cloud provider indicates that they will take it away soon. It is critical to react quickly to a migration request in this scenario. Therefore, we will investigate the migration latency, the time from the notification until the current server is fully vacated, and all progress is held externally. While we here migrate all running queries, it is, of course, also possible to extract only a few queries for load balancing in multi-tenant scenarios [98]. We investigate the latency by running randomly-

selected queries in parallel in a loop and triggering migration after a randomly selected duration between 10 and 30 seconds. The migration latency reported is the time from triggering the migration until every query's state and plan are sent to the cache and target server, respectively. Furthermore, to not lose progress, all in-flight tasks, i.e., pipelines, are finished before migrating, which is also included in the latency.

Figure 6.15 shows the migration latency for 4 and 8 worker threads for 100 runs per thread and query count combination. We removed 19 outliers for better visibility. Of course, one can see that there is linear growth with an increasing number of parallel queries. However, the gap between 4 and 8 threads shrinks for more queries. This shrinkage is again attributable to the network limitation already identified in Figure 6.11. Furthermore, as we optimize for keeping all progress, the latency includes finishing the current task. It could be further reduced if faster migration is valued over progress kept. On average, even with this additional delay, it takes less than 2.6 seconds to vacate a server, allowing us to react quickly to changes in dynamic environments.

6.5 Related Work

As more data moves to the cloud, ample research has focused on optimizing data processing for this distributed and flexible environment. This section will provide an overview of the research most essential and relevant to our on-demand state separation approach.

Cloud-Optimized Database Architectures. Cloud-optimized databases, such as Snowflake [39], Google BigQuery [8, 111], and Amazon Redshift [68] optimize for massive parallelism for queries on ever-growing data. Some surveys [146, 183] investigate the challenges and opportunities for databases in cloud environments. Analogous to cloud-optimized databases, we strive to increase flexibility for analytical queries in the cloud. In some systems, this flexibility is enabled through disaggregated storage [28, 39, 44, 126, 157, 165, 182]. Dremel [111], e.g., employs storage disaggregation, as well as memory disaggregation through a shuffle layer, for flexiblity and scalability. We see storage disaggregation as one pillar of flexibility in our approach. To further improve the performance of storage-separated systems, Yang et al. propose a combination of caching and pushing compute to storage to reduce network cost [174]. We also minimize network overhead by migrating a minimal query state. In addition, modern big data systems [92, 149, 179] offer flexibility by directly accessing tables stored in remote storage [9, 54].

Building on the ideas of storage-separated architectures, Aguilar-Saborit et al. [7] describe the state-separating POLARIS system. In addition to storage, they

further keep the query state externally, thereby enabling intra-query worker changes. We build upon this idea for our approach. However, we keep state externally only when necessary, thus reducing network overhead. Keeping state in the form of intermediate results externally for transient compute resources has also been proposed for Apache Spark [173, 175]. To materialize intermediate results, Stuedi et al. [86, 144] propose a data store optimized for temporary data in distributed settings.

Adaptive Query Processing. Ample research has been conducted on modifying query execution during runtime in the context of adaptive query processing [14, 41, 65]. While this area focuses on adapting the query plan at runtime, and we currently do not modify execution order when migrating, we still share similar ideas. For example, Xing et al. [172] discuss migrating processing on the fly for load balancing in streaming engines. Orthogonal to our work in case of migration, some works have focused on keeping progress in the case of plan changes in ETL MapReduce [80] and traditional database systems. Keeping progress has been described using artificially introduced operators [15] and unary blocking operators [108] for database systems. These works re-use intermediate results on the same system and do not consider network transfer. To mitigate network dealys, Urhan et al. [11, 156] propose query scrambling.

Instance Migration in Cloud Environments. Many cloud providers offer transient compute resources to customers to increase resource utilization within their datacenters. Often, such transient workers, e.g., spot instances in Amazon AWS, are cheaper than reserved instances. Therefore, utilizing such transient resources has been the focus of recent research. Kraska et al. [95] analyze deployment strategies for fault tolerance mechanisms, such as query restarts and checkpoints. These checkpoints often comprise the entire VM and application state [81, 140, 164, 177, 178]. In contrast, we optimize for a small, system-specific query state that can be extracted at any time. Other systems also optimize for a minimized state [19, 139, 173] but rely on pre-defined checkpoints. Yan et al. [173] propose adaptive fine-grained checkpointing for Apache Spark based on recomputation cost and failure probability. Kaulakiene et al. [81] propose migrating tasks to cheaper or more powerful instances using VM snapshots to optimize the cost and runtime of jobs in a cloud setting. We have identified such migrations as a primary use case for our on-demand state separation approach and optimized specifically for the migration of database workloads. While our work focuses on analytical workloads, migrating between servers is also interesting for transactional workloads [40, 48].

Migration of Intermediate Results. The presented idea to migrate partial query results is inspired by past work in multi-engine environments. These so-called polystores span a combination of stream processing, big data, and

database systems [43, 58, 61, 78, 104], some surveyed by Tan et al. [147]. While we only consider migrating to other instances of the same engine and optimize for flexibility in our work, works on multi-engine environments focus on a range of optimization criteria. For example, Agrawal et al. [6] optimize performance by selecting a combination of execution engines for a single task and migrating intermediate results between these engines. Simitsis et al. [143] describe an optimizer for data workflows comprising multiple engines, taking both the execution and data shipping cost into account. Focussing specifically on data migration, Dziedzic et al. [47] discuss challenges and solutions for sharing results between engines. While we discuss our approach in the context of flexibility, it can also be used to optimize for different metrics.

6.6 Summary

In this chapter, we present a novel on-demand state separation approach for data processing in the cloud. In contrast to existing state-separating architectures, our approach can establish state separation after-the-fact, e.g., when migrating between workers. This way, our approach only incurs the overhead of syncing state externally when necessary. To motivate our approach, we provided an extensive analysis of query state occurring within the TPC-DS benchmark, showing that on-demand extraction can reduce the transferred state by an order of magnitude. Our approach exploits existing access paths to extract state with specialized extraction queries, allowing state extraction with minimal code and runtime overhead while utilizing all features of modern query engines.

We demonstrate the feasibility of our approach using an implementation of on-demand state separation in the Umbra database system. The experimental analysis shows that our approach can outperform full state separation and query restarts in many scenarios. Our in-depth cost analysis demonstrates that the majority of overhead stems from network overhead. With the roll-out of more powerful network infrastructure in the future, we expect our approach to be beneficial in even more use cases.

CHAPTER 7

Conclusions and Future Work

Recent changes in the workload and landscape of data analytics have brought forward unprecedented challenges and opportunities for relational database systems. In this thesis, we devised four strategies allowing relational database systems to conquer the challenges of ephemeral data and dynamic workloads.

First, we engineered a ring-buffered streaming relation that enables stream enrichment queries, which can be easily integrated into existing systems. Further, we demonstrated its performance against two dedicated stream processing engines, consistently outperforming them on analytical workloads.

Following this, we devised a new type of materialized view called continuous view to support complex queries over streamed and durable inputs in relational database systems. Our novel split strategy to maintain continuous views divides the maintenance work between inserts and queries. Integrating these views into the Umbra database system, we highlighted their performance for several analytical workloads. Our experiments showed that our views are not only competitive against dedicated stream processing engines but often outperform them by order of magnitude.

In addition, we outlined a communication-optimal process to sample large volumes of data, such as streams, in parallel for later analysis. Our process necessitates no communication between workers in the sampling phase and minimizes communication when merging local samples. Further, we described a new merge strategy optimized for small sample sizes. We highlighted the ability of our process to keep up with even the highest-velocity inputs in a range of experiments, showing a near-linear scale-up to more than one hundred worker threads.

Finally, to fully leverage the opportunities of flexible cloud architectures, we devised a novel cloud data warehousing architecture based on on-demand state separation. Our approach relies on a storage-separated architecture to leverage the flexibility of independently scalable relations for all workloads.

We demonstrated how state can be extracted from running queries on demand when additional flexibility is required, incurring no performance penalty when not. We extensively studied the state occurring in TPC-DS queries to highlight the applicability of on-demand state separation and showed that such state separation can be performed quickly with little overhead, outperforming query restarts in many scenarios.

While the four techniques introduced in this thesis solve many challenges of dynamic workloads, we are convinced that future work can extend their applicability to even more use cases. One compelling concept for our in-database stream processing approaches is transactionality. So far, all our strategies consider streams non-transactional and decouple stream processing from the transactional semantics of database systems. While this is the desired behavior for finite queries such as those supported by our relation-based stream processing, it can lead to outdated results for infinite queries in continuous views. Streams are not the only source of change in relational database systems. Inserts, updates, and deletes to relations can influence the outcome of queries combining both streams and durable relations as well. We already use the state management of Umbra to extract the state of running queries. Using even finer-grained access, exchanging the state of a running query would also be possible, enabling us to propagate changes to base tables into continuous views. Such state change propagations could be performed periodically or on-demand, ensuring a frequently fresh state for all tables involved in continuous queries.

We see the largest potential for our techniques in future research on novel cloud data warehousing architectures. The concepts and methods of modular query processing we used for on-demand state separation and split maintenance in this thesis could enable a range of flexible execution models. The state extraction for independent subtrees can, e.g., also be used to distribute these subtrees to multiple workers to execute them in parallel. An extracted state can not only be used to continue processing the same query on another server. It could also be used to share the results of subtrees with other queries on the same machine, avoiding unnecessary recalculation of intermediate results. Deploying networks with bandwidth above 400 Gbit/s, as was recently announced for AWS, would allow sharing of intermediate results not only with queries on the same machine but in the entire data center efficiently. We are excited to see how the techniques presented in this work can help address the current and future challenges of processing dynamic workloads.

Bibliography

[1] Transaction Processing Performance Council (TPC). *TPC benchmark DS: Standard specification*. 2021. URL: http://www.tpc.org/.

[2] Transaction Processing Performance Council (TPC). *TPC benchmark H: Standard specification*. 2021. URL: http://www.tpc.org/.

[3] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. "The Design of the Borealis Stream Processing Engine". In: *CIDR*. www.cidrdb.org, 2005, pp. 277–289.

[4] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. "Aurora: a new model and architecture for data stream management". In: *VLDB J.* 12.2 (2003), pp. 120–139.

[5] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. "TSpoon: Transactions on a stream processor". In: *J. Parallel Distributed Comput.* 140 (2020), pp. 65–79.

[6] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. "RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -". In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1414–1427.

[7] Josep Aguilar-Saborit and Raghu Ramakrishnan. "POLARIS: The Distributed SQL Engine in Azure Synapse". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3204–3216.

[8] H Ahmadi. "In-memory Query Execution in Google BigQuery". In: *Google Cloud Blog* (2016).

[9] Amazon. *Cloud Object Storage - Amazon S3*. 2022. URL: https://aws.amazon.com/s3/.

[10] Pradeep Ambati, Noman Bashir, David E. Irwin, and Prashant J. Shenoy. "Good Things Come to Those Who Wait: Optimizing Job Waiting in the Cloud". In: *SoCC*. ACM, 2021, pp. 229–242.

[11] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. "Scrambling Query Plans to Cope With Unexpected Delays". In: *PDIS*. IEEE Computer Society, 1996, pp. 208–219.

[12] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. "STREAM: The Stanford Stream Data Manager". In: *SIGMOD Conference*. ACM, 2003, p. 665.

[13] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. "Amazon Redshift Re-invented". In: *SIGMOD Conference*. ACM, 2022, pp. 2205–2217.

[14] Shivnath Babu and Pedro Bizarro. "Adaptive Query Processing in the Looking Glass". In: *CIDR*. www.cidrdb.org, 2005, pp. 238–249.

[15] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. "Proactive Re-optimization". In: *SIGMOD Conference*. ACM, 2005, pp. 107–118.

[16] Shivnath Babu and Jennifer Widom. "Continuous Queries over Data Streams". In: *SIGMOD Rec.* 30.3 (2001), pp. 109–120.

[17] Maximilian Bandle, Jana Giceva, and Thomas Neumann. "To Partition, or Not to Partition, That is the Join Question in a Real System". In: *SIGMOD Conference*. ACM, 2021, pp. 168–180.

[18] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth L. Knowles. "One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables". In: *SIGMOD Conference*. ACM, 2019, pp. 1757–1772.

[19] Carsten Binnig, Abdallah Salama, Erfan Zamanian, Muhammad El-Hindi, Sebastian Feil, and Tobias Ziegler. "Spotgres - parallel data analytics on Spot Instances". In: *ICDE Workshops*. IEEE Computer Society, 2015, pp. 14–21.

[20] Altan Birler. "Scalable Reservoir Sampling on Many-Core CPUs". In: *SIGMOD Conference*. ACM, 2019, pp. 1817–1819.

[21] Altan Birler, Bernhard Radke, and Thomas Neumann. "Concurrent online sampling for all, for free". In: *DaMoN*. ACM, 2020, 5:1–5:8.

[22] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. "Efficiently Updating Materialized Views". In: *SIGMOD Conference*. ACM Press, 1986, pp. 61–71.

[23] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. "MonetDB/XQuery: a fast XQuery processor powered by a relational engine". In: *SIGMOD Conference*. ACM, 2006, pp. 479–490.

[24] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *CIDR*. www.cidrdb.org, 2005, pp. 225–237.

[25] Irina Botan, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. "Transactional stream processing". In: *EDBT*. ACM, 2012, pp. 204–215.

[26] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. "Scalable and robust latches for database systems". In: *DaMoN*. ACM, 2020, 2:1–2:8.

[27] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. "Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database". In: *SIGMOD Conference*. ACM, 2015, pp. 251–264.

[28] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. "PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers". In: *SIGMOD Conference*. ACM, 2021, pp. 2477–2489.

[29] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[30] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. "Monitoring Streams - A New Class of Data Management Applications". In: *VLDB*. Morgan Kaufmann, 2002, pp. 215–226.

[31] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. “Trill: A High-Performance Incremental Query Processor for Diverse Analytics”. In: *Proc. VLDB Endow.* 8.4 (2014), pp. 401–412.

[32] Sirish Chandrasekaran and Michael J. Franklin. “Streaming Queries over Streaming Data”. In: *VLDB*. Morgan Kaufmann, 2002, pp. 203–214.

[33] David M Chickering, Ashis K Roy, and Christopher A Meek. “Distributed reservoir sampling for web applications”. US Patent 7,308,447. 2007.

[34] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[35] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. “Algorithms for Deferred View Maintenance”. In: *SIGMOD Conference*. ACM Press, 1996, pp. 469–480.

[36] Andrew A. Cook, Goksel Misirli, and Zhong Fan. “Anomaly Detection for IoT Time-Series Data: A Survey”. In: *IEEE Internet Things J.* 7.7 (2020), pp. 6481–6494.

[37] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. “Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches”. In: *Found. Trends Databases* 4.1-3 (2012), pp. 1–294.

[38] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490.

[39] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. “The Snowflake Elastic Data Warehouse”. In: *SIGMOD Conference*. ACM, 2016, pp. 215–226.

[40] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. “Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration”. In: *Proc. VLDB Endow.* 4.8 (2011), pp. 494–505.

[41] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. “Adaptive Query Processing”. In: *Found. Trends Databases* 1.1 (2007), pp. 1–140.

[42] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. "Hekaton: SQL server's memory-optimized OLTP engine". In: *SIGMOD Conference*. ACM, 2013, pp. 1243–1254.

[43] Katerina Doka, Nikolaos Papailiou, Victor Giannakouris, Dimitrios Tsoumakos, and Nectarios Koziris. "Mix 'n' match multi-engine analytics". In: *IEEE BigData*. IEEE Computer Society, 2016, pp. 194–203.

[44] Dominik Durner, Badrish Chandramouli, and Yinan Li. "Crystal: A Unified Cache Storage System for Analytical Databases". In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2432–2444.

[45] Dominik Durner, Viktor Leis, and Thomas Neumann. "JSON Tiles: Fast Analytics on Semi-Structured Data". In: *SIGMOD Conference*. ACM, 2021, pp. 445–458.

[46] Dominik Durner, Viktor Leis, and Thomas Neumann. "On the Impact of Memory Allocation on High-Performance Query Processing". In: *DaMoN*. ACM, 2019, 21:1–21:3.

[47] Adam Dziedzic, Aaron J. Elmore, and Michael Stonebraker. "Data transformation and migration in polystores". In: *HPEC*. IEEE, 2016, pp. 1–6.

[48] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. "Zephyr: live migration in shared nothing databases for elastic cloud platforms". In: *SIGMOD Conference*. ACM, 2011, pp. 301–312.

[49] C. T. Fan, Mervin E. Muller, and Ivan Rezucha. "Development of Sampling Plans by Using Sequential (Item by Item) Selection Techniques and Digital Computers". In: *Journal of the American Statistical Association* 57.298 (1962), pp. 387–402. ISSN: 01621459. URL: http://www.jstor.org/stable/2281647 (visited on Oct. 6, 2022).

[50] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database – An Architecture Overview". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33.

[51] Philipp Fent and Thomas Neumann. "A Practical Approach to Groupjoin and Nested Aggregates". In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2383–2396.

[52] Peter M. Fenwick. "A New Data Structure for Cumulative Frequency Tables". In: *Softw. Pract. Exp.* 24.3 (1994), pp. 327–336.

[53] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. “Integrating scale out and fault tolerance in stream processing using operator state management”. In: *SIGMOD Conference*. ACM, 2013, pp. 725–736.

[54] Apache Software Foundation. *Apache Hadoop*. 2021. URL: https://hadoop.apache.org/.

[55] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting Worst-Case Optimal Joins in Relational Database Systems”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904.

[56] Michael J. Freitag, Alfons Kemper, and Thomas Neumann. “Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2797–2810.

[57] Michael J. Freitag and Thomas Neumann. “Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates”. In: *CIDR*. www.cidrdb.org, 2019.

[58] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron J. Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. “The BigDAWG polystore system and architecture”. In: *HPEC*. IEEE, 2016, pp. 1–6.

[59] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter R. Pietzuch. “Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications”. In: *SoCC*. ACM, 2019, pp. 233–245.

[60] Thanaa M. Ghanem, Ahmed K. Elmagarmid, Per-Åke Larson, and Walid G. Aref. “Supporting views in data stream management systems”. In: *ACM Trans. Database Syst.* 35.1 (2010), 1:1–1:47.

[61] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. “MuSQLE: Distributed SQL query execution over multiple engine environments”. In: *IEEE BigData*. IEEE Computer Society, 2016, pp. 452–461.

[62] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. “Deployment of Query Plans on Multicores”. In: *Proc. VLDB Endow.* 8.3 (2014), pp. 233–244.

[63] Lukasz Golab, Kumar Gaurav Bijay, and M. Tamer Özsu. “Multi-query optimization of sliding window aggregates by schedule synchronization”. In: *CIKM*. ACM, 2006, pp. 844–845.

[64] Philipp Götze and Kai-Uwe Sattler. “Snapshot Isolation for Transactional Stream Processing”. In: *EDBT*. OpenProceedings.org, 2019, pp. 650–653.

[65] Anastasios Gounaris, Efthymia Tsamoura, and Yannis Manolopoulos. "Adaptive Query Processing in Distributed Settings". In: *Advanced Query Processing (1)*. Vol. 36. Intelligent Systems Reference Library. Springer, 2013, pp. 211–236.

[66] Timothy Griffin and Leonid Libkin. "Incremental Maintenance of Views with Duplicates". In: *SIGMOD Conference*. ACM Press, 1995, pp. 328–339.

[67] Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. "Grizzly: Efficient Stream Processing Through Adaptive Query Compilation". In: *SIGMOD Conference*. ACM, 2020, pp. 2487–2503.

[68] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. "Amazon Redshift and the Case for Simpler Data Warehouses". In: *SIGMOD Conference*. ACM, 2015, pp. 1917–1923.

[69] Himanshu Gupta. "Selection of Views to Materialize in a Data Warehouse". In: *ICDT*. Vol. 1186. Lecture Notes in Computer Science. Springer, 1997, pp. 98–112.

[70] Daniel Gyllstrom, Eugene Wu, Hee-Jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. "SASE: Complex Event Processing over Streams (Demo)". In: *CIDR*. www.cidrdb.org, 2007, pp. 407–411.

[71] Gabriel Haas, Michael Haubenschild, and Viktor Leis. "Exploiting Directly-Attached NVMe Arrays in DBMS". In: *CIDR*. www.cidrdb.org, 2020.

[72] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. "Database Servers on Chip Multiprocessors: Limitations and Opportunities". In: *CIDR*. www.cidrdb.org, 2007, pp. 79–87.

[73] Axel Hertzschuch, Guido Moerkotte, Wolfgang Lehner, Norman May, Florian Wolf, and Lars Fricke. "Small Selectivities Matter: Lifting the Burden of Empty Samples". In: *SIGMOD Conference*. ACM, 2021, pp. 697–709.

[74] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. "Stream Processing Languages in the Big Data Era". In: *SIGMOD Rec.* 47.2 (2018), pp. 29–40.

[75] Lorenz Hübschle-Schneider and Peter Sanders. "Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams". In: *SPAA*. ACM, 2020, pp. 543–545.

[76] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. "An architecture for recycling intermediates in a column-store". In: *ACM Trans. Database Syst.* 35.4 (2010), 24:1–24:43.

[77] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. "Towards a streaming SQL standard". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1379–1390.

[78] Abdulrahman Kaitoua, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. "Muses: Distributed Data Migration System for Polystores". In: *ICDE*. IEEE, 2019, pp. 1602–1605.

[79] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. "H-store: a high-performance, distributed main memory transaction processing system". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1496–1499.

[80] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. "Dynamically optimizing queries over large scale data platforms". In: *SIGMOD Conference*. ACM, 2014, pp. 943–954.

[81] Dalia Kaulakiene, Christian Thomsen, Torben Bach Pedersen, Ugur Çetintemel, and Tim Kraska. "SpotADAPT: Spot-Aware (re-)Deployment of Analytical Processing Tasks on Amazon EC2". In: *DOLAP*. ACM, 2015, pp. 59–68.

[82] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *ICDE*. IEEE Computer Society, 2011, pp. 195–206.

[83] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra". In: *VLDB J.* 30.5 (2021), pp. 883–905.

[84] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. "Detecting Change in Data Streams". In: *VLDB*. Morgan Kaufmann, 2004, pp. 180–191.

[85] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Scalable Analytics on Fast Data". In: *ACM Trans. Database Syst.* 44.1 (2019), 1:1–1:35.

[86] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *OSDI*. USENIX Association, 2018, pp. 427–444.

[87] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. "DBToaster: higher-order delta processing for dynamic, frequently fresh views". In: *VLDB J.* 23.2 (2014), pp. 253–278.

[88] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries". In: *ICDE*. IEEE Computer Society, 2018, pp. 197–208.

[89] André Kohn, Viktor Leis, and Thomas Neumann. "Building Advanced SQL Analytics From Low-Level Plan Operators". In: *SIGMOD Conference*. ACM, 2021, pp. 1001–1013.

[90] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. "SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures". In: *SIGMOD Conference*. ACM, 2016, pp. 555–569.

[91] Arnd Christian König, Yi Shan, Tobias Ziegler, Aarati Kakaraparthy, Willis Lang, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. "Tenant Placement in Over-subscribed Database-as-a-Service Clusters". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2559–2571.

[92] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. "Impala: A Modern, Open-Source SQL Engine for Hadoop". In: *CIDR*. www.cidrdb.org, 2015.

[93] Yannis Kotidis and Nick Roussopoulos. "A case for dynamic view management". In: *ACM Trans. Database Syst.* 26.4 (2001), pp. 388–423.

[94] Nick Koudas and Divesh Srivastava. "Data Stream Query Processing". In: *ICDE*. IEEE Computer Society, 2005, p. 1145.

[95] Tim Kraska, Elkhan Dadashov, and Carsten Binnig. "Spotlytics: How to Use Cloud Market Places for Analytics?" In: *BTW*. Vol. P-265. LNI. GI, 2017, pp. 361–380.

[96] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing". In: *Proc. NetDB*. Vol. 11. 2011, pp. 1–7.

[97] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. "On-the-fly sharing for streamed aggregation". In: *SIGMOD Conference*. ACM, 2006, pp. 623–634.

[98] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. "Towards Multi-Tenant Performance SLOs". In: *IEEE Trans. Knowl. Data Eng.* 26.6 (2014), pp. 1447–1463.

[99] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age". In: *SIGMOD Conference*. ACM, 2014, pp. 743–754.

[100] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. "LeanStore: In-Memory Data Management beyond Main Memory". In: *ICDE*. IEEE Computer Society, 2018, pp. 185–196.

[101] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. "Efficient Processing of Window Functions in Analytical SQL Queries". In: *Proc. VLDB Endow.* 8.10 (2015), pp. 1058–1069.

[102] Viktor Leis and Maximilian Kuschewski. "Towards Cost-Optimal Query Processing in the Cloud". In: *Proc. VLDB Endow.* 14.9 (2021), pp. 1606–1612.

[103] Kim-Hung Li. "Reservoir-Sampling Algorithms of Time Complexity O(n(1 + log(N/n)))". In: *ACM Trans. Math. Softw.* 20.4 (1994), pp. 481–493.

[104] Harold Lim, Yuzhang Han, and Shivnath Babu. "How to Fit when No One Size Fits". In: *CIDR*. www.cidrdb.org, 2013.

[105] Richard J. Lipton and Jeffrey F. Naughton. "Query Size Estimation by Adaptive Sampling". In: *PODS*. ACM Press, 1990, pp. 40–46.

[106] Ling Liu, Calton Pu, Roger S. Barga, and Tong Zhou. "Differential Evaluation of Continual Queries". In: *ICDCS*. IEEE Computer Society, 1996, pp. 458–465.

[107] Mohammad Sultan Mahmud, Joshua Zhexue Huang, Salman Salloum, Tamer Z. Emara, and Kuanishbay Sadatdiynov. "A survey of data partitioning and sampling methods to support big data analysis". In: *Big Data Min. Anal.* 3.2 (2020), pp. 85–101.

[108] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. "Robust Query Processing through Progressive Optimization". In: *SIGMOD Conference*. ACM, 2004, pp. 659–670.

[109] John Meehan, Cansu Aslantas, Stan Zdonik, Nesime Tatbul, and Jiang Du. "Data Ingestion for the Connected World". In: *CIDR*. www.cidrdb.org, 2017.

[110] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. "S-Store: Streaming Meets Transaction Processing". In: *Proc. VLDB Endow.* 8.13 (2015), pp. 2134–2145.

[111] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. "Dremel: A Decade of Interactive SQL Analysis at Web Scale". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3461–3472.

[112] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. "Materialized View Selection and Maintenance Using Multi-Query Optimization". In: *SIGMOD Conference*. ACM, 2001, pp. 307–318.

[113] Guido Moerkotte and Axel Hertzschuch. "alpha to omega: the G(r)eek Alphabet of Sampling". In: *CIDR*. www.cidrdb.org, 2020.

[114] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. "Rethinking Stateful Stream Processing with RDMA". In: *SIGMOD Conference*. ACM, 2022, pp. 1078–1092.

[115] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: a timely dataflow system". In: *SOSP*. ACM, 2013, pp. 439–455.

[116] Kousuke Nakabasami, Toshiyuki Amagasa, Salman Ahmed Shaikh, Franck Gass, and Hiroyuki Kitagawa. "An architecture for stream OLAP exploiting SPE and OLAP engine". In: *IEEE BigData*. IEEE Computer Society, 2015, pp. 319–326.

[117] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550.

[118] Thomas Neumann and Michael J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance". In: *CIDR*. www.cidrdb.org, 2020.

[119] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *SIGMOD Conference*. ACM, 2015, pp. 677–689.

[120] Thomas Neumann and Bernhard Radke. "Adaptive Optimization of Very Large Join Queries". In: *SIGMOD Conference*. ACM, 2018, pp. 677–692.

[121] Milos Nikolic, Badrish Chandramouli, and Jonathan Goldstein. "Enabling Signal Processing over Data Streams". In: *SIGMOD Conference*. ACM, 2017, pp. 95–108.

[122] Milos Nikolic, Mohammad Dashti, and Christoph Koch. "How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates". In: *SIGMOD Conference*. ACM, 2016, pp. 511–526.

[123] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. "Stateful Scalable Stream Processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1634–1645.

[124] *PipelineDB - High-performance time-series aggregation for PostgreSQL*. `https://github.com/pipelinedb/pipelinedb`.

[125] Hasso Plattner. "A common database approach for OLTP and OLAP using an in-memory column database". In: *SIGMOD Conference*. ACM, 2009, pp. 1–2.

[126] Presto. *Distributed SQL Query Engine for Big Data*. 2022. URL: `https://prestodb.io/`.

[127] Redis. *Redis - Data types*. 2022. URL: `https://redis.io/topics/data-types`.

[128] Maximilian Reif and Thomas Neumann. "A Scalable and Generic Approach to Range Joins". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 3018–3030.

[129] Alice Rey, Michael Freitag, and Thomas Neumann. "Seamless Integration of Parquet Files into Data Processing". In: *BTW*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 235–258.

[130] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Global Value Numbers and Redundant Computations". In: *POPL*. ACM Press, 1988, pp. 12–27.

[131] Viktor Sanca and Anastasia Ailamaki. "Sampling-Based AQP in Modern Analytical Engines". In: *DaMoN*. ACM, 2022, 4:1–4:8.

[132] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. "Efficient Random Sampling - Parallel, Vectorized, Cache-Efficient, and Online". In: *CoRR* abs/1610.05141 (2016).

[133] Sebastian Schelter, Juan Soto, Volker Markl, Douglas Burdick, Berthold Reinwald, and Alexandre V. Evfimievski. "Efficient sample generation for scalable meta learning". In: *ICDE*. IEEE Computer Society, 2015, pp. 1191–1202.

[134] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. "ArrayQL for Linear Algebra within Umbra". In: *SSDBM*. ACM, 2021, pp. 193–196.

[135] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. "In-Database Machine Learning with SQL on GPUs". In: *SSDBM*. ACM, 2021, pp. 25–36.

[136] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. "Recursive SQL and GPU-support for in-database machine learning". In: *Distributed Parallel Databases* 40.2-3 (2022), pp. 205–259.

[137] Timos K. Sellis. "Multiple-Query Optimization". In: *ACM Trans. Database Syst.* 13.1 (1988), pp. 23–52.

[138] Salman Ahmed Shaikh and Hiroyuki Kitagawa. "StreamingCube: A Unified Framework for Stream Processing and OLAP Analysis". In: *CIKM*. ACM, 2017, pp. 2527–2530.

[139] Prateek Sharma, Tian Guo, Xin He, David E. Irwin, and Prashant J. Shenoy. "Flint: batch-interactive data-intensive processing on transient servers". In: *EuroSys*. ACM, 2016, 6:1–6:15.

[140] Supreeth Shastri and David E. Irwin. "HotSpot: automated server hopping in cloud spot markets". In: *SoCC*. ACM, 2017, pp. 493–505.

[141] Oded Shmueli and Alon Itai. "Maintenance of Views". In: *SIGMOD Conference*. ACM Press, 1984, pp. 240–255.

[142] Moritz Sichert and Thomas Neumann. "User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases". In: *Proc. VLDB Endow.* 15.5 (2022), pp. 1119–1131.

[143] Alkis Simitsis, Kevin Wilkinson, Malú Castellanos, and Umeshwar Dayal. "Optimizing analytic data flows for multiple execution engines". In: *SIGMOD Conference*. ACM, 2012, pp. 829–840.

[144] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. "Crail: A High-Performance I/O Architecture for Distributed Data Processing". In: *IEEE Data Eng. Bull.* 40.1 (2017), pp. 38–49.

[145] Supreeth Subramanya, Tian Guo, Prateek Sharma, David E. Irwin, and Prashant J. Shenoy. "SpotOn: a batch computing service for the spot market". In: *SoCC*. ACM, 2015, pp. 329–341.

[146] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. "Choosing A Cloud DBMS: Architectures and Tradeoffs". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2170–2182.

[147] Ran Tan, Rada Chirkova, Vijay Gadepally, and Timothy G. Mattson. "Enabling query processing across heterogeneous data models: A survey". In: *IEEE BigData*. IEEE Computer Society, 2017, pp. 3211–3220.

[148] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. "Continuous Queries over Append-Only Databases". In: *SIGMOD Conference*. ACM Press, 1992, pp. 321–330.

[149] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. "Hive - a petabyte scale data warehouse using Hadoop". In: *ICDE*. IEEE Computer Society, 2010, pp. 996–1005.

[150] Yves Tillé. "Sampling Algorithms". In: *International Encyclopedia of Statistical Science*. Springer, 2011, pp. 1273–1274.

[151] Daniel Ting. "Simple, Optimal Algorithms for Random Sampling Without Replacement". In: *CoRR* abs/2104.05091 (2021).

[152] Srikanta Tirthapura and David P. Woodruff. "Optimal Random Sampling from Distributed Streams Revisited". In: *DISC*. Vol. 6950. Lecture Notes in Computer Science. Springer, 2011, pp. 283–297.

[153] Quoc-Cuong To, Juan Soto, and Volker Markl. "A survey of state management in big data processing systems". In: *VLDB J.* 27.6 (2018), pp. 847–872.

[154] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. "Storm@twitter". In: *SIGMOD Conference*. ACM, 2014, pp. 147–156.

[155] Tolga Urhan and Michael J. Franklin. "XJoin: A Reactively-Scheduled Pipelined Join Operator". In: *IEEE Data Eng. Bull.* 23.2 (2000), pp. 27–33.

[156] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. "Cost Based Query Scrambling for Initial Delays". In: *SIGMOD Conference*. ACM Press, 1998, pp. 130–141.

[157] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. "Eon Mode: Bringing the Vertica Columnar Database to the Cloud". In: *SIGMOD Conference*. ACM, 2018, pp. 797–809.

[158] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases". In: *SIGMOD Conference*. ACM, 2017, pp. 1041–1052.

[159] Jeffrey Scott Vitter. "An efficient algorithm for sequential random sampling". In: *ACM Trans. Math. Softw.* 13.1 (1987), pp. 58–67.

[160] Jeffrey Scott Vitter. "Faster Methods for Random Sampling". In: *Commun. ACM* 27.7 (1984), pp. 703–718.

[161] Jeffrey Scott Vitter. "Random Sampling with a Reservoir". In: *ACM Trans. Math. Softw.* 11.1 (1985), pp. 37–57.

[162] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2662–2675.

[163] Adrian Vogelsgesang, Tobias Mühlbauer, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Domain Query Optimization: Adapting the General-Purpose Database System Hyper for Tableau Workloads". In: *BTW*. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 313–333.

[164] William Voorsluys and Rajkumar Buyya. "Reliable Provisioning of Spot Instances for Compute-intensive Applications". In: *AINA*. IEEE Computer Society, 2012, pp. 542–549.

[165] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. "Building An Elastic Query Engine on Disaggregated Storage". In: *NSDI*. USENIX Association, 2020, pp. 449–462.

[166] Benjamin Wagner, André Kohn, and Thomas Neumann. "Self-Tuning Query Scheduling for Analytical Workloads". In: *SIGMOD Conference*. ACM, 2021, pp. 1879–1891.

[167] Yousuke Watanabe, Shinichi Yamada, Hiroyuki Kitagawa, and Toshiyuki Amagasa. "Integrating a Stream Processing Engine and Databases for Persistent Streaming Data Management". In: *DEXA*. Vol. 4653. Lecture Notes in Computer Science. Springer, 2007, pp. 414–423.

[168] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. "On-Demand State Separation for Cloud Data Warehousing". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2966–2979.

[169] Christian Winter, Andreas Kipf, Christoph Anneser, Eleni Tzirita Zacharatou, Thomas Neumann, and Alfons Kemper. "GeoBlocks: A Query-Cache Accelerated Data Structure for Spatial Aggregation over Polygons". In: *EDBT*. OpenProceedings.org, 2021, pp. 169–180.

[170] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. "Meet Me Halfway: Split Maintenance of Continuous Views". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2620–2633.

[171] Christian Winter, Moritz Sichert, Altan Birler, Thomas Neumann, and Alfons Kemper. "Communication-Optimal Parallel Reservoir Sampling". In: *BTW*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 567–578.

[172] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. "Dynamic Load Distribution in the Borealis Stream Processor". In: *ICDE*. IEEE Computer Society, 2005, pp. 791–802.

[173] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. "TR-Spark: Transient Computing for Big Data Analytics". In: *SoCC*. ACM, 2016, pp. 484–496.

[174] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. "FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS". In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2101–2113.

[175] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. "Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters". In: *EuroSys*. ACM, 2017, pp. 575–588.

[176] Yuke Yang, Lukasz Golab, and M. Tamer Özsu. "ViewDF: Declarative incremental view maintenance for streaming data". In: *Inf. Syst.* 71 (2017), pp. 55–67.

[177] Sangho Yi, Artur Andrzejak, and Derrick Kondo. "Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances". In: *IEEE Trans. Serv. Comput.* 5.4 (2012), pp. 512–524.

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

[179] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. "Apache Spark: a unified engine for big data processing". In: *Commun. ACM* 59.11 (2016), pp. 56–65.

[180] Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. "Analyzing Efficient Stream Processing on Modern Hardware". In: *Proc. VLDB Endow.* 12.5 (2019), pp. 516–530.

[181] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. "The NebulaStream Platform for Data and Application Management in the Internet of Things". In: *CIDR*. www.cidrdb.org, 2020.

[182] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. "CompuCache: Remote Computable Caching using Spot VMs". In: *CIDR*. www.cidrdb.org, 2022.

[183] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. "Rethinking Data Management Systems for Disaggregated Data Centers". In: *CIDR*. www.cidrdb.org, 2020.

[184] Shuhao Zhang, Juan Soto, and Volker Markl. "A Survey on Transactional Stream Processing". In: *CoRR* abs/2208.09827 (2022).

[185] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. "Lazy Maintenance of Materialized Views". In: *VLDB*. ACM, 2007, pp. 231–242.

[186] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. "Efficient exploitation of similar subexpressions for query processing". In: *SIGMOD Conference*. ACM, 2007, pp. 533–544.

[187] Tobias Ziegler, Carsten Binnig, and Viktor Leis. "ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA". In: *SIGMOD Conference*. ACM, 2022, pp. 685–699.