DISS. ETH NO. 22063

Work Sharing Data Processing Systems

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences of ETH Zurich

presented by
GEORGIOS GIANNIKIS
Master of Science ETH, ETH Zürich, Switzerland
born on 19 August 1982
citizen of Greece

accepted on the recommendation of
Prof. Dr. Gustavo Alonso (ETH Zurich), examiner
Prof. Dr. Donald Kossmann (ETH Zurich), co-examiner
Prof. Dr. Anastasia Ailamaki (EPFL), co-examiner
Prof. Dr. Peter Boncz (CWI Amsterdam), co-examiner

2014

Abstract

Since the birth of the Internet, the way people consume information on-line has been in a state of constant evolution. The introduction of dynamic web content, allowed every web user to create, read, search and delete information on remote data services, enabling a wide variety of industries, such as travel reservation, social networking and e-banking. Furthermore, the fast flow of information created the need for real time planning and on-line analytics in order to support business intelligence.

As both the number of users and the amount of data increase, the data services and systems that support these industries are put under a constant strain. The high throughput of user requests, combined with the increased complexity of analytical queries has created the need for a more efficient data processing. Ideally, the data processing tier should handle high throughputs of user requests and evaluate long-running decision support queries, while exhibiting robust, scalable performance.

Traditional relational database solutions fail to meet these new requirements. What a big set of these systems have in common, is that they execute each query individually. Execution can be optimized by taking advantage of hardware (i.e., cache conscious algorithms, SIMD instructions) or software (i.e., better algorithmic design, efficient locking and latching), in order to provide higher performance. Nevertheless, when many of these optimized queries are executed concurrently, system resources become sparse. As a result contention arises, resulting in unpredictable performance. At the end, the response time of an individual query may be multiple times higher than the expected one.

This dissertation presents an alternative way of implementing data processing in relational database systems: SharedDB/TX. SharedDB/TX implements a push-based, data flow network of always-on operators that process queries and updates in batches. Deep, aggressive sharing in terms of both computation and resources is the key component of the design. SharedDB/TX is able to support a high throughput of queries while ensuring system robustness and proper query isolation.

SharedDB/TX consists of three components, each of which has been designed and implemented from scratch: the SharedDB core, the Work Sharing Optimizer and the Work Sharing Transaction Manager.

SharedDB defines the model of a work sharing operator and implements the most common relational operators. SharedDB sidesteps the traditional query-at-a-time approach and processes all pending queries in a single batch. This radical design makes SharedDB significantly more robust to high loads. Furthermore, it introduces a novel query-data model that is used to share the common work across each batch of queries. This dissipates the impact of heavy analytical queries that typically process a vast amount of data, across all concurrently executed queries.

The Work Sharing Optimizer (WSO) implements query optimization for work sharing systems, like SharedDB. Instead of trying to optimize each query individually, the goal of WSO is to generate a globally optimal query plan. WSO takes into account the sharing possibilities of the whole workload, which allows SharedDB to handle higher loads.

Finally, the Work Sharing Transaction Manager (WSTM) implements ACID transaction processing in SharedDB. Compared to traditional approaches, WSTM is designed for batch processing on a data flow network. The overhead of transactions is optimized, by removing most synchronization, while still maintaining strong consistency.

SharedDB/TX combines these three components into a single system. The experimental results show that work sharing is an efficient approach to data processing that results in higher and more robust performance. In describing the design and implementation of SharedDB/TX, this dissertation contributes to our scientific understanding of how to efficiently share work in data processing systems, how to apply work sharing in all SQL supported relational operators, how to optimize query access paths to allow higher degree of sharing, and how to support transactional properties in systems that share work.

Zusammenfassung

Seit der Geburtsstunde des Internets hat sich die Art, wie Menschen Informationen online aufnehmen, fortlaufend geändert. Dynamische Webinhalte ermöglichen heute jedem Internetbenutzer das Erzeugen, Durchsuchen und Löschen von Informationen auf Datenverwaltungssystemen von einem beliebigen Ort aus, was zu einer Vielzahl von neuen Industriezweigen, wie zum Beispiel Flugreservations-Plattformen, sozialen Netzwerken oder e-Banking geführt hat. Des Weiteren entstand durch den schnellen Informationsfluss eine Anforderung für Echtzeit-Planung und Online-Analyse zur Unterstützung von firmeninternen Informationsdiensten.

Da nicht nur die Anzahl der Nutzer sondern auch die zu verarbeitende Datenmenge stetig wächst, kommen Datenverwaltungssystemen und anderen Anwendungen, die solche Industriezweige unterstützen, eine zunehmende Bedeutung zu. Der hohe Durchsatz von Benutzeranfragen, in Verbindung mit der gestiegenen Komplexität von analytischen Anfragen führt zu einem steigenden Bedürfnis nach effizienteren Datenverarbeitungsmethoden. Mit Hilfe eines idealen Datenverarbeitungssystem wäre es möglich, einen hohen Durchsatz von Benutzeranfragen zu beantworten und dabei gleichzeitig lang laufende Entscheidungshilfe-Anfragen (englisch "decision support queries") zu verarbeiten. Zudem sollte ein solches System robust und skalierbar sein.

Traditionelle Datenbanklösungen können diesen Anforderungen nicht gerecht werden, da der Grossteil dieser Systeme jede Anfrage separat bearbeitet. Die Verarbeitung kann allerdings optimiert werden, indem man die folgenden zwei Faktoren berücksichtigt: moderne Hardware (z.B. Cache-gewahre Algorithmen und spezialisierte Mehrweg Daten Instruktionen, englisch "Single-Instruction-Multiple-Data", SIMD) und bessere Software (z.B. bessere Algorithmen und effiziente Locks und Latches). Einerseits führen diese Faktoren zu einer verbesserten Leistung, andererseits werden die Ressourcen durch die Vielzahl gleichzeitiger Anfragen schnell erschöpft. Dies macht es wiederum schwer, die Leistung genau vorherzusagen. Daher kann es vorkommen, dass die Bearbeitungszeit

einer individuellen Anfrage deutlich höher ausfällt als erwartet.

Diese Dissertation zeigt eine alternative Möglichkeit auf, wie Datenverarbeitung umgesetzt werden kann: SharedDB/TX. SharedDB/TX ist ein push-basiertes Datenfluss-Netzwerk von ständig verfügbaren Operatoren, die Anfragen und Datenaktualisierungen chargenweise durchführen. Die Hauptidee besteht darin, Ressourcen wenn immer möglich zu teilen und Berechnungen, die mehrfach vorkommen, nur einmal durchzuführen. Durch diese Teilung der Arbeit erreicht SharedDB/TX einen hohen Durchsatz von Anfragen und gleichzeitig Robustheit und garantierte Isolation der Anfragen.

SharedDB/TX besteht aus drei Hauptkomponenten, die alle von Grund auf neu entwickelt wurden: Dem SharedDB System, dem Arbeitsteilungs-Optimierer und dem Transaktionsmanager zum Implementierung der Arbeitsverteilung.

SharedDB definiert zuerst das Modell eines kooperativen Operators (englisch "work sharing operator") und implementiert die meist gebrauchten relationalen Operatoren in Form eines solchen kooperativen Operators. Dabei verzichtet SharedDB auf den traditionellen Ansatz, wonach nur eine Anfrage gleichzeitig verarbeitet wird und bearbeitet stattdessen alle offenen Anfragen gemeinsam. Dieser Ansatz trgt entscheidend zu SharedDBs Robustheit bei hoher Last bei. Des Weiteren wird ein neuartiges Modell der Anfragedaten eingeführt, das verwendet werden kann, um gemeinsame Berechnungen in einer Charge zu teilen. Dadurch verringert sich auch der negative Einfluss der berechnungsintensiven analytischen Anfragen, da ein grosser Teil der Berechnungskosten für diese Anfragen auf andere Anfragen der gleichen Charge, welche die gleichen Berechnungen machen müssen, "verteilt" werden kann.

Der Arbeitsteilungs-Optimierer (englisch "Work Sharing Optimizer", WSO) ist für die optimale Arbeitsteilung von Anfragen zuständig und kann für jegliche Art von Arbeitsteilungssystemen verwendet werden, insbesondere auch für SharedDB. Statt jede Anfrage separat zu optimieren, ist das Ziel des WSO, einen Ausführungsplan zu berechnen, der aus globaler Sicht optimal ist. WSO betrachtet dabei die möglichen Einsparungen aufgrund gemeinsamer Berechnungen in Bezug auf die gesamte Menge von Anfragen, die gestellt werden können. Dadurch erreicht SharedDB den benötigten hohen Durchsatz.

Der Arbeitsteilungs-Transaktionsmanager (englisch "Work Sharing Transaction Manager", WSTM) ist für die Verwaltung von ACID Transaktionen in SharedDB zuständig. Im Unterschied zu anderen Transaktionsmanagern, konzentriert sich WSTM auf das Verarbeiten von ganzen Chargen in einem Datenfluss-Netzwerk. Der Mehraufwand für Transaktionen wird minimiert, indem der Grossteil der (oft unnötigen) Synchronisation weggelassen wird und zwar in einer Art und Weise, in der strikte Konsistenz gewährleistet bleibt.

SharedDB/TX kombiniert die drei genannten Komponenten in einem einzigen System.

Unsere Experimente zeigen, dass Arbeitsteilung eine effiziente Methode für Datenverarbeitung ist und zu verbesserter und vorhersagbarer Leistung führt. Durch das Design und die Umsetzung von SharedDB/TX trägt diese Dissertation wesentlich zum wissenschaftlichen Verständnis bei, wie Arbeit in Datenverarbeitungssystemen effizient geteilt werden kann, wie Arbeitsteilung in SQL-unterstützten Operatoren umgesetzt werden kann, wie Zugriffspfade von Anfragen optimiert werden können um noch weitergehende Arbeitsteilung zu ermöglichen und wie Transaktionen in solchen Systemen unterstützt werden können.

Credits

The work presented in this dissertation was funded by the the Enterprise Computing Center of ETH Zurich (`www.ecc.ethz.ch`) and supported by the Swiss National Science Foundation under grant number `PDFMP2-122963`.

Acknowledgments

So this is what five years of work can be compressed into. Unfortunately, only the "good" results are included in this dissertation. The road was a long one, yet full of knowledge and adventure. All the bad turns I took while riding the road, all the bad designs, wrong decisions and crazy ideas have been left out, in order to make room for the scientifically important findings. Nevertheless, this work would not have been possible without the wrong turns, the bad designs and the reckless mistakes and, most importantly, without all the people who helped me, advised me and supported me in this effort.

First of all, I have to thank my advisors Prof. Gustavo Alonso and Prof. Donald Kossmann for their personal engagement in this work and for the countless expert advises and discussions on it. I really appreciate the numberless times they guided and promoted this work. I feel all the more grateful and honored for their trust, support and encouragement throughout my PhD studies.

I would also like to thank Natassa Ailamaki and Peter Boncz for being in my committee, providing all their important feedback to polish this dissertation and coming to Zurich in person for my examination.

SharedDB/TX would not have been possible without Philipp Unterbrunner who helped me during my Master studies and my first years as a PhD student. I learned a lot next to you my friend and I will always appreciate it.

I am also indebted to my colleagues at the Systems Group. Especially to Darko Makreshanski, who was a great contributor to my work. I am sincerely grateful to Lucas Braun, Tudor-Ioan Salomie, Ionut Subasu, Simon Loesing, Jana Gičeva, Markus Pilman, Dimitris Karampinas, Kyumars Sheykh Esmaili and anyone I may have forgotten, for their time and for taking my mind off whenever necessary.

I would also like to avail the opportunity and thank my friends who made the years of my PhD studies a fun experience and unforgettable time. Giorgo, Evdokia, Nikoleta, Evi, Panagioti, Dimo, Vasso, Sofia, Marie, Ektora, Gerasime, Anna, Kosta, Agni, Kosta,

Maria, Dimitri, Niko, Kosti, Pavlo, Pano, Dimitri, Pano, Filippe, Ntino, Gianni, Taso, Mano: Thank you.

Last but not least, I thank my parents and my brother for all their love and support. I am grateful to my family for giving me the chance to follow my dreams even though that meant that I would be 2,000 km away.

I wanted to separate
data from programs,
because data and instructions
are very different.

– KEN THOMPSON

Contents

1

Introduction

In the last decade, there has been a radical increase in the use of data services. With the number of internet users increasing, these systems are faced with loads that often involve hundreds or thousands of queries submitted at the same time. These loads are proliferated by the widespread usage of web services that map user requests to multiple systems queries. The problem is further aggravated by the increasing number of strategic decision making systems that have to execute complex analytical queries on the same dataset. Travel reservation, financial, insurance and social networking are just some examples of such data intensive systems. To make matters worse, end users of such systems expect a constant response time, independently of the load of the system or the complexity of the concurrently executed queries.

The changes in modern workloads have made a profound impact on the design of relational database management systems (RDBMS). Researchers have focused on improving the performance of these systems by inventing faster algorithms, making more efficient use of hardware and employing better query optimizers. Such techniques focus on achieving the best performance for each query, making them effective for conventional database engines that evaluate each query individually.

Even with these optimizations, traditional RDBMS have been struggling to meet the requirements of modern workloads. With very high loads, the independent query plans

compete with each other for resources, as they are unaware of each other, causing a load interaction problem.

This has led to the introduction of alternative data processing architectures, like NOSQL [Cat11], and the notion that one size does not fit all [SC05]. A number of systems designed around these ideas manage to support higher loads. To achieve this, they make radical simplifications of the programming model in one or more dimensions (query language, data model, consistency model). While they are able to support the high loads of modern workloads, the simplified model of these systems reduces their applicability, as they have limited support for complex analytical queries.

In this dissertation, we present another alternative approach to data processing that is able to support high loads of queries, while also evaluating analytical queries. The key component of our approach is deep sharing of system resources and processing power. To enable deep sharing, we step aside from the conventional query-at-a-time data processing paradigm and instead, process multiple queries concurrently in a single execution context. This allows sharing of common work across queries making the design extremely robust to high loads and hotspots of data.

Our concrete implementation, SharedDB/TX is a push-based, work-sharing relational database engine that supports all SQL operations. SharedDB/TX is robust to mixed (read-write) workloads that consist of diverse (transactional-analytical) queries. It guarantees consistency of data, while achieving multiple times the performance of traditional data processing. Contrary to conventional RDBMS, SharedDB/TX does not provide the best performance per individual query, but instead it focuses on maximizing the performance of all concurrently executed queries. Rather than assigning system resources to queries, SharedDB/TX assigns resources to a network of always-on storage and data processing operators. This data flow network forms a shared global query plan that is used to evaluate *all* queries. In fact, the same global query plan can be used for long periods of time, possibly the whole life cycle of the system.

SharedDB/TX consists of three key components: SharedDB, Work Sharing Optimizer, and Shared Transaction Manager. SharedDB is the core data processing engine that implements the data flow network and powers robustness to high loads of modern workloads. The Work Sharing Optimizer implements query optimization that takes advantage of SharedDB's properties. That is, it generates a global query plan that increases the degree of sharing. Finally, the Shared Transaction manager allows SharedDB to support SQL transactions, allowing for any isolation level from read uncommitted to serializable.

In this dissertation, we present this novel work sharing approach to data processing, as well as, the system design and the implementation details of the concrete implementation

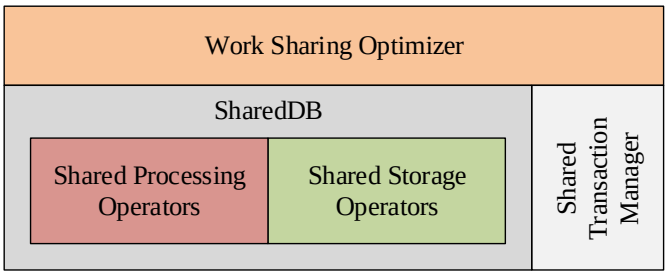


Figure 1.1: Solution Overview

of SharedDB/TX. Additional, this dissertation contributes to, among other things, our scientific understand of how to efficiently share resources and common work across all queries, without sacrificing the applicability or robustness of the system.

1.1 Solution Overview

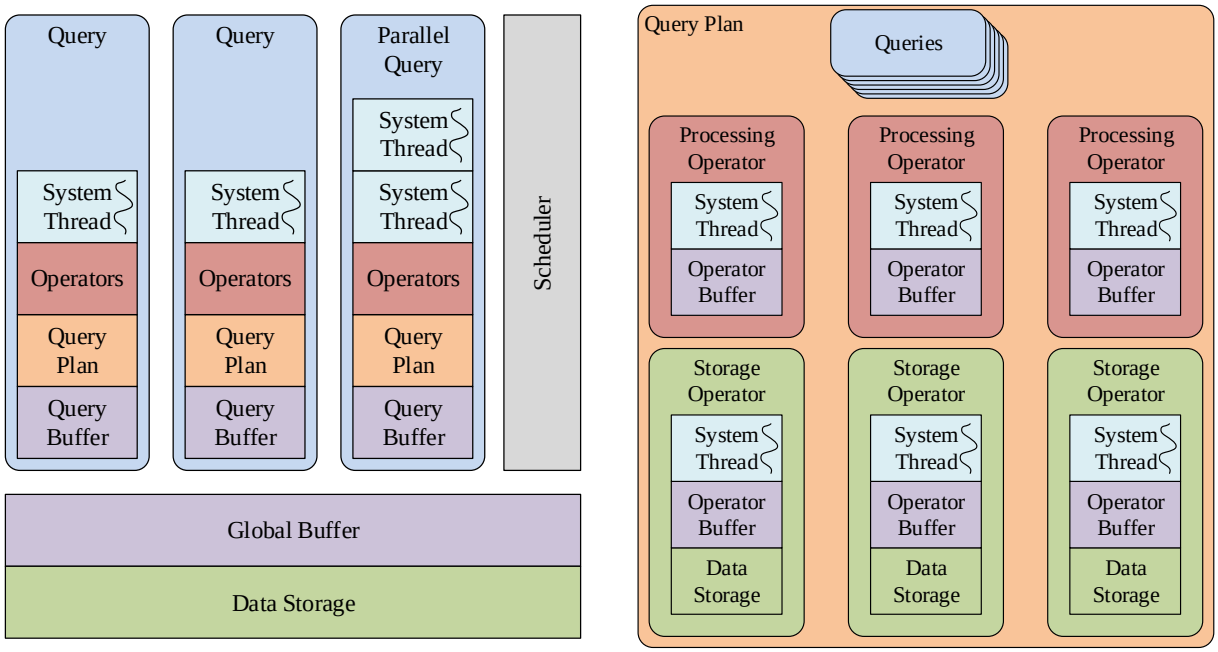
In this section we provide a high level overview of the solution to answering high loads of modern, diverse workloads. Figure 1.1 shows a high-level visualization of the architecture.

1.1.1 SharedDB

One of the reasons that conventional data processing architectures fall short in efficiently answering modern workloads, is load interaction. The response time of a request depends not only on the request itself, but on various other parameters. Amongst these parameters is the number of concurrently executed queries, as well as their complexity. Needless to say, these parameters are unpredictable and most importantly transparent to the end-users, who expect quasi-constant response times.

The load interaction problem is triggered by the way queries are executed: each query is evaluated in a execution context (usually a thread). Various optimizations can be applied at this point, like cache-aware algorithms. However, when multiple queries are executed at the same time, contention arises, and as a result, performance drops.

In order to reduce load interaction, SharedDB uses a different approach to data processing that revolves around operators instead of queries. SharedDB uses a small set of operators that run on a dedicated context (usually a CPU core), and perform a very specific task, for instance a join or a sort. Each operator handles multiple (possibly thousands) queries concurrently, in order to support high loads. This operator centric approach reduces



(a) Query-at-a-time Execution Model

(b) Multiple-queries Execution Model

Figure 1.2: Data Processing Models

contention for resources, and as a result, is a able to sustain higher loads. Figure 1.2 illustrates the key differences of the two processing models.

In short, we can say that the differences of the two processing models resemble the differences of commuting with a car or public transportation. Query-at-a-time approaches are similar to using your car to travel in a city. This guarantees smaller commute time under normal conditions, as you can travel exactly to your destination and chose the shorter route. However, the travel time is not constant. If you happen to commute during rush hours, where there is high contention on the streets, the commute time becomes unpredictable. Traffic lights (synchronization points) get crowded, and any car collisions (transaction rollbacks) further increase travel time. And most importantly, these conditions do not depend on how fast your car may be, or the optimality of the chosen route, rather than on the accumulated traffic status of the city.

On the other hand, the multiple-query execution model that is used in SharedDB resembles commuting with public transportation. In this case, you cannot travel directly from the origin to the destination, as some walking may be required. The route is not always optimal and it depends on how good the traffic plan of the city is. In most cases, you have to switch transportation medium, which implies that some waiting in the bus stop is necessary. In certain cases, you arrive in a intermediary stop right before the next

bus leaves. In the worst case though, you might arrive right after the bus has left, thus having to wait for the next one. While the travel time per individual person is higher with public transportation, it is more efficient when lots of people want to commute. There is less contention on the streets, as each transportation medium transfers multiple people simultaneously, and if the traffic plan is implemented correctly, like for example in Switzerland, less synchronization points (i.e. traffic lights) are required. Last but not least, the cost of commuting with public transportation is smaller.

SharedDB may be inferior to query-at-a-time engines for small query loads. But when a critical mass of queries is accumulated, it guarantees good and, most importantly, predictable good performance. In short, SharedDB is based on the following principles:

Batch Processing Instead of processing one query at a time, SharedDB batches queries and updates and evaluates each batch as a whole.

Work Sharing SharedDB introduces a new type of work sharing operators that process a whole batch of queries concurrently. Any common work across the batch is not repeated, allowing SharedDB to be robust to high loads.

Staged Operators Instead of allocating resources to individual queries, SharedDB allocates resources to always-on operators, reducing scheduling overhead and resource contention.

Global Query Plan SharedDB's operators form a data flow network where result tuples flow through processing operators. The performance of a query depends on the length of its path in this global query plan, allowing for higher load isolation as heavy analytical queries have minimal influence on smaller transactional ones.

Chapter 2 presents SharedDB. Comprehensive experimental results are presented in Section 2.4 of this dissertation. To give a rough idea, SharedDB is able to outperform a high-end commercial query-at-a-time engine by a factor of 2 when tested under a standardized online bookstore workload.

1.1.2 Work Sharing Optimizer

As with every database engine, SharedDB/TX includes a specialized data access path optimizer. In traditional data engines, the optimizer attempts to determine the most efficient (faster) way to execute a given query by considering possible query plans. A tremendous amount of research has focused in making the optimizer better and faster, as choosing a inefficient plan adds a big strain on the database system.

In Work Sharing Data Processing, a new optimizer is required. Since multiple queries are executed by the same operators, at the same time, a Work Sharing Optimizer should consider the query plans of all concurrently executed queries. Furthermore, the goal of the optimizer needs to be adapted. Instead of trying to find the faster way to execute a query, a work sharing optimizer should find a plan that answers *all* queries with the smaller cost.

In the public transportation analogy, the optimizer plays the role of the route planner. When commuting by car, which resembles query-at-a-time systems, the optimizer's job is to find the shortest route from the destination to the target. There are different tricks that such an optimizer can play, like avoiding traffic lights. The goal is to reduce either time or distance, which is similar to query response time and memory footprint.

When commuting by public transportation, the analogous of work-sharing systems, the optimizer has to decide on an optimal layout of all public transportation routes. That is, it has to consider the destination and target of all people who commute, and strategically place the bus stops in locations that are useful. Moreover, the goal is not to reduce the travel time of each individual person, rather than reduce the accumulative travel time.

Another important difference of a work sharing optimizer compared to conventional optimizers, is the life cycle of the generated plans. In query-at-a-time systems, these plans have to be generated quite often, thus a fast optimizer is important. Even when using prepared statements, small variations on the query parameters or the dataset cardinalities may require a re-optimization of all query plans. In work sharing systems, the global query plan can be used for much longer. Since they execute multiple (possibly hundreds or thousands) queries at the same time, the *average* variation of query parameters is expected to be small. As a result, a fast optimizer is not a necessity.

Figure 1.3 the process of optimizing and building a global query plan. In certain cases, this plan can be used for the entire life cycle of the system. Alternatively, SharedDB/TX allows a database administrator to dynamically re-optimize the global query plan.

Obviously, a work sharing optimizer has a enormous solution space to explore. All queries that will be executed have to be taken into account. Then, all operators of these queries have to be reordered and strategically placed in the global query plan. In Chapter 3 we provide the problem definition of work sharing optimization, and analyze the size of the solution space. In order to reduce the solution space, two heuristics are used. The implementation of SharedDB/TX's optimizer uses an algorithm based on branch and bound to methodically explore the possible solutions.

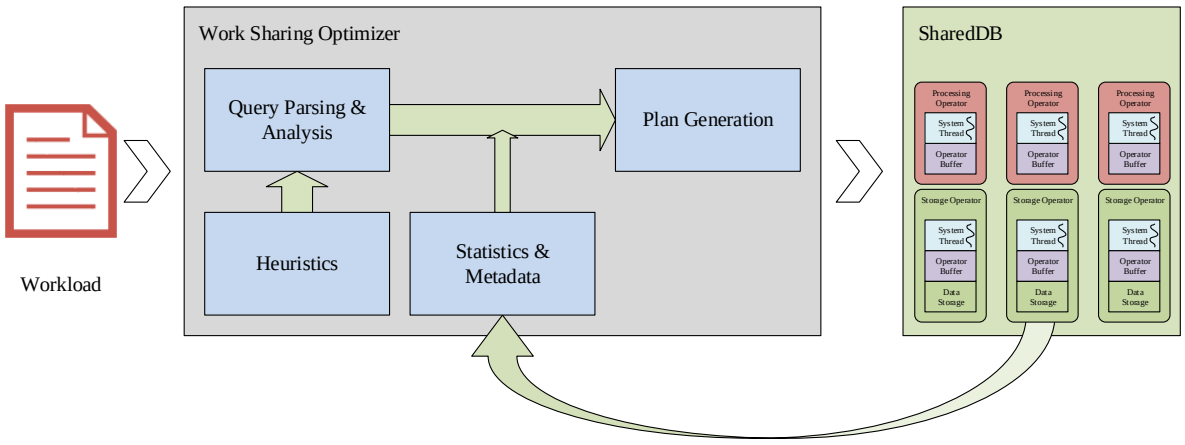


Figure 1.3: Workload Optimization and Building of a Global Query Plan

1.1.3 Shared Transaction Manager

Finally, the last part of SharedDB/TX is a set of extensions to SharedDB with the goal of supporting SQL transactions. While some modern workloads are designed to require only session consistency, the Shared Transaction Manager enables support for atomic read-write operations in SharedDB/TX.

The consistency guarantees provided by SharedDB is write monotonicity and read your own writes. These two are enough to provide session consistency. The Work Sharing Transaction Manager (WSTM) builds on top of these and provides serializability guarantees. In fact, WSTM allows each transaction to define a different isolation level, supporting all four isolation levels, as defined in the ANSI/ISO SQL standard: read uncommitted, read committed, repeatable read and serializable.

WSTM shows the flexibility and extensibility of SharedDB's framework. It requires no modifications in the data processing operators, the Global Query Plan, or the Work Sharing Optimizer. In fact, WSTM is implemented as another SharedDB operator. Obviously, to enable the new isolation levels, WSTM requires certain support from the storage engine operators. SharedDB's design allows mixed data consistency models, where some relations use strong consistency, while others provide only write monotonicity.

Work Sharing Data Processing favors optimistic multi-version concurrency control. Locking is not a good candidate, as each lock can potentially stall the execution of multiple queries. To make matters worse, deadlock detection is extremely complicated in a data flow network and requires coordination across storage engines. Chapter 4 presents an analysis of how different concurrency control methods fit into work sharing, their advantages and disadvantages when implemented in SharedDB, the design of Shared Transaction Manager

and its consistency model, as well as, the modified storage engine operators.

1.2 Contributions

In describing the design and implementation of SharedDB, this dissertation contributes to our scientific understanding of how to efficiently share work in data processing systems, how to apply work sharing in all SQL supported relational operators, how to optimize query access paths to allow higher degree of sharing, and how to support transactional properties in systems that share work.

The concrete, technical contributions of this dissertation include:

SharedDB A work-sharing relational database engine which batches queries and evaluates these queries concurrently [GAK12, GMAK13]. SharedDB's design take full advantage of modern multi-core hardware. It guarantees an upper limit on the execution time of each query that depends only on the complexity of the query rather than the queries that are concurrently executed, thus providing a higher degree of workload isolation, and lower load interaction. Notable components of SharedDB include a novel data-query model that allows queries to be handled as data, a push-based staged database design that processes data similar to a data flow network, a novel query-execution model where resources are allocated to operators instead of queries, a number of algorithms that implement work sharing in all SQL supported relational operators, as well as two implementations of storage engines that use cooperative scans and cooperative B-Tree lookups.

Work Sharing Optimizer A novel multiple query optimizer with the goal of reducing the total work to execute a given workload [GMAK14]. The WSO algorithm takes as an input a set of prepared statements and generates a globally optimized query plan. The generated plan can be used possibly for the whole life cycle of the workload, or alternatively, the system can be progressively re-optimized to adapt to evolutions of workload and dataset. In this dissertation, we formulate the problem of work sharing query optimization, analyze its complexity, discuss alternatives on the implementation of work sharing, introduce two heuristics to reduce the complexity of the optimization problem, present its algorithm solution, and describe its efficient implementation.

SharedDB/TX A set of extensions to the design of SharedDB that allow efficient handling of SQL transactions. SharedDB/TX extends SharedDB's global query plan

by introducing new transaction manager operators, as well as, a storage engine that supports SQL transactions. The number of changes to the SharedDB core is minimal, which shows the extensibility of the operator model of SharedDB. We present an extensive analysis of different approaches to implement transactions in work sharing systems, discuss on their effectiveness and scalability of these approaches in the context of SharedDB, present the atomicity, isolation and consistency guarantees that SharedDB/TX provides, present the transaction manager operator and its implementation, present comprehensive experimental results on SharedDB/TX and analyze the overhead of transactions in SharedDB/TX.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents SharedDB and its implementation. Chapter 3 introduces the problem of work sharing and a novel algorithm that solves it. Chapter 4 presents a set of modifications that allow SharedDB to handle transactional operations, and Chapter 5 concludes the dissertation. The Appendix contains details about the workloads that were used in the experiments of this dissertation.

2

SharedDB

Over the last decades, tremendous efforts have been invested into query evaluation with the goal of achieving the best possible performance for each individual query in a query-at-a-time processing model. To this end, sophisticated query compile-time techniques (e.g., cost-based optimization [SAC+79]) and many kinds of database operator implementations (e.g., [Sha86]) have been developed. While highly effective, unfortunately, all these approaches are not sufficient to meet common requirements of modern database applications if used in a query-at-a-time model. Often, modern applications need to meet service-level agreements (SLAs) that involve a maximum response time for, say, 99 percent of the queries. Furthermore, such SLAs may specify isolation levels and/or data freshness guarantees for workloads that involve queries and updates. The query-at-a-time processing model is not good to meet such SLAs because it may result in resource contention and interference in high load situations.

Unfortunately, performance guarantees are difficult to make with traditional database management systems. These systems are designed to achieve best performance for each individual query, in isolation. To this end, they require constant monitoring, maintenance, and tuning by highly trained administrators. But what users really want are response time and throughput guarantees for a concurrent, diverse, evolving workload as a whole, without the need for an army of administrators.

As a step toward a solution to this problem, in this chapter we present a novel relational database, SharedDB [GAK12], specifically designed to meet SLAs in high load situations for complex and highly dynamic workloads. SharedDB is based on a new processing model that batches queries and updates in order to share computation across them.

The key idea of SharedDB can be best described using an example involving two queries and the well known schema of `Customers` and `Orders`, as depicted in Figure 2.1. The first query asks for all `Orders` of the Swiss `Customers`. The second query asks for all `Orders` of German `Customers` in the Year 2013.

Executing these two queries in a query-at-a-time fashion is straightforward. For Query 1, a selection is made over all `Customers` to fetch the rows with `Country` equal to `'CH'`. The results are joined with all the `Orders` by i.e. hashing the `Customer_Id` and probing the hash table with all the `Orders` rows. The second query is executed in the same fashion but with different selection predicate on the `Orders` and an additional selection predicate on the `Customers` relation.

SharedDB executes these two queries in a single `Customer-Order` join operation joining *all* `Orders` with the union of German and Swiss `Customers` and routing the results of the big join to the corresponding queries. This procedure is shown in Figure 2.1c and a typical result set is shown in Figure 2.1d.

At a first glance, SharedDB looks like a bad idea. In the example, SharedDB performs extra work when compared to a traditional query-at-a-time approach. Specifically, SharedDB compares Swiss `Customers` and `Orders` of the `Year` 2012 as part of its join. When processing the two queries individually and pushing down selection predicates below the join, these combinations are never considered. Because of this, SharedDB is likely to perform poorly in low-throughput situations. SharedDB, however, was designed to handle high throughput with response time guarantees. If hundreds of concurrent queries involving a `Customer-Order` join need to be processed, then there will be a significant overlap between the sets of `Customers` and `Orders` relevant for these queries. The more queries, the higher the overlap, even if the queries involve totally different kinds of predicates on the `Customer` and `Order` tables. Furthermore, SharedDB defines an upper bound for the amount of work that needs to be carried out for a set of concurrent `Customer-Order` join queries. In the worst case, the whole `Customer` table needs to be joined with the whole `Order` table, independently of the number of concurrent queries processed. In contrast, the effort of a traditional, query-at-a-time system grows linearly with the number of queries. Moreover, SharedDB handles hotspots ideally. Popular tuples are processed exactly once for all concurrently executed queries. This way, SharedDB is able to maintain response time guarantees in high-throughput situations.

```
Q1  SELECT * FROM Customers JOIN Orders WHERE Customers.country = 'CH'
Q2  SELECT * FROM Customers JOIN Orders WHERE Orders.year = 2013
```

(a) Sample Queries

Customers

Id	Name	Country
1	...	CH
2	...	DE
3	...	NL
4	...	GR
5	...	CH
6	...	DE
...	...	...

⋈

Orders

Customer	Year	Amount
4	2014	420
5	2012	1,100
1	2013	600
4	2013	700
3	2011	2,000
6	2013	800
5	2013	900
...	...	...

(b) Sample `Customers ⋈ Orders`

σ(Customers)

Id	Name	Country	Query
1	...	CH	1
2	...	DE	2
5	...	CH	1
6	...	DE	2
...	...	...	...

Orders

Customer	Year	Amount	Query
4	2014	420	1
5	2012	1,100	1
1	2013	600	1, 2
4	2013	700	1, 2
3	2011	2,000	1
6	2013	800	1, 2
5	2013	900	1, 2
...	...	...	...

(c) Shared Execution of Selections

Id	Name	Country	Customer	Year	Amount	Query
5	...	CH	5	2012	1,100	1
1	...	CH	1	2013	600	1
6	...	DE	6	2013	800	2
...	...	...	...	...	...	...

(d) Result Set of Shared Execution

Figure 2.1: Example of Query Execution in SharedDB

SharedDB is designed for diverse workloads that contain short-running queries as well as long-running analytical queries. Performance may be inferior to traditional solutions for

their sweet spot, but it exhibits good and, more importantly, predictably good performance for all workloads. This is achieved by combining and extending a number of database techniques in the context of multi-query optimization and data stream processing. In particular, SharedDB adopts some of the ideas developed as part of QPipe [HSA05], CJoin [CPV09, CPV11] and DataPath [ADJ+10]. These systems, however, are only effective for certain kinds of queries and, thus, their application is limited to OLAP workloads with complex queries. In contrast, we will show that the design principles of SharedDB are general and can be applied to any kind of query and update. As a result, SharedDB is able to process OLTP workloads in addition to OLAP and mixed workloads. It is this generality and its ability to meet SLAs in high load situations that distinguishes SharedDB from its closest competitors. Technically, it is the batch-oriented query processing model with a new way to share computation that makes SharedDB unique.

In summary, this chapter makes the following contributions to the state of the art of query processing:

- a novel data-query model which allows queries to be treated as data and be part of data processing;
- a novel query-execution model where resources are allocated to operators instead of queries;
- a new relational operator model that is always on and processes batches of queries, executing the whole batch concurrently;
- a new push-based execution model that is based on a Global Query Plan, a data access plan that is able to answer the queries of all a whole workload, instead of one query at a time;
- a comprehensive experimental evaluation of SharedDB, including a performance comparison to traditional relational database systems; and, finally,
- the (perhaps surprising) insight that predictable performance need not be the result of careful isolation of concurrent activities, but can instead be achieved by rigorous, deep sharing of resources.

The remainder of this chapter is organized as follows. Section 2.1 discusses the state of the art and related work. Section 2.2 gives an overview of the high level system design and explains the data and operator model, as well as the query execution model of SharedDB. Section 2.3 explains the implementation details of the system, a selection of data processing

algorithms designed for shared data processing and a number of optimizations that were applied to make the most of work sharing. Section 2.4 shows the results of an extensive experimental evaluation, and finally, Section 2.5 makes concluding remarks.

2.1 State of the Art

SharedDB is designed to robustly execute possibly thousands of complex queries concurrently, while providing predictable performance guarantees. This research topic is not a new one, as robustness is becoming a hot topic in query execution.

In this section, we discuss the traditional approach toward work sharing, multi query optimization and data flow execution, followed by a selection of recent, related work from the robust execution, predictable performance, work sharing and stream-processing domains.

2.1.1 Work Sharing

Sharing work across concurrently executed queries is a field that gained great momentum in research during the last decade. Shared scans have been implemented in the context of disk-based database systems such as Red Brick DW [Fer94], DB2 UDB [LBMW07], MonetDB/X100 and Vectorwise [ZHNB07, ZB12], with the goal of sharing disk bandwidth and maximizing buffer-pool utilization across queries. Driven by the advent of multi-core machines and scans' inherent parallelizability, Raman et al. [RSQ+08] and their system called Blink have demonstrated that shared main-memory scans are a scalable access path with predictable, low ("constant") latency.

A follow-up paper by Qiao et al. [QRR+08] investigates the optimization problem of sharing a main-memory scan cursor between a subset of pending queries with `GROUP BY` clauses. In Blink, `GROUP BY`s are implemented via hashing, which implies a certain working set associated with each query, turning this into a bin-packing problem.

Crescando [UGA+09, GUM+10] is a relational table where data is accessed explicitly via full table scans rather than indexes. Crescando has been shown to sustain high loads by letting concurrent queries share scan cursors, as well as creating short-lived indexes of query predicates; an idea that originates from the domain of publish-subscribe systems, such as Le Subscribe [FJL+01, PFL+00].

Sharing work across scans allows more robust performance compared to executing them one at-a-time, yet all the aforementioned work is limited to a single relation. CJoin [CPV09] extends the ideas of work sharing to join operations. CJoin is a multi-query star-join

algorithm for analytic read-only workloads (data warehousing). It achieves high scalability, handling up to 256 concurrent queries, by using a single always-on plan of join operators. The approach is, however, limited to star schemata. Additionally, CJoin updates the common hash tables in a single thread for each individual query, which is problematic for more general schemata where hash tables are substantially larger. Nevertheless, CJoin showcases that sharing work in relational operators is indeed beneficial and results in better overall performance.

SharedDB is based on existing ideas in terms of work sharing. Crescando has been modified and used as a storage engine of SharedDB. Hash join operators based on CJoin have been implemented and extended to support more complex schemata. Finally, SharedDB supports a generic `GROUP BY` operator based on the ideas implemented in Blink.

2.1.2 Multi Query Optimization

Another approach of sharing work across queries is multi-query optimization (MQO) [Sel88, Fin82]. MQO is a different model of query execution that involves detection of common subexpressions in concurrently executed queries and evaluation of these subexpressions only once. This idea of shared computation was first devised in the Eighties and has been the basis of several studies to optimize OLAP workloads.

Applying MQO in OLAP workloads is very efficient and indeed eliminates duplicating work. Yet, MQO is not efficient under more generic workloads, in particular transactional workloads. Such workloads involve short running point queries, thus there are not many opportunities to factor out common subexpressions. Additionally, MQO introduces a high cost of detecting common subexpression, which grows as more queries have to be taken into account. A recent paper by Zhou et. al. [ZLFL07] analyzes this issue and describes several optimizations that allow faster common subexpression detection. Nevertheless, even with this optimization, experimental results on MQO are always limited to less than 16 concurrent queries at a time. Furthermore, MQO combines queries by generating broader predicates. This is extremely efficient during query processing (especially during data access and retrieval), yet it requires certain post-filtering to distinguish the results of each individual query. As the number of combined queries grows, post-filtering becomes more complicated.

For these reasons, MQO has been mostly applied to complex OLAP queries that process large amounts of data and involve larger common subexpressions (e.g., the scan of a fact table in a data warehouse). In such workloads the investment of detecting common subexpressions is offset by the benefits of shared computation. As explained in the example

of the introduction, SharedDB is able to carry out shared computation without common subexpression detection and is, thus, applicable to OLTP and mixed workloads.

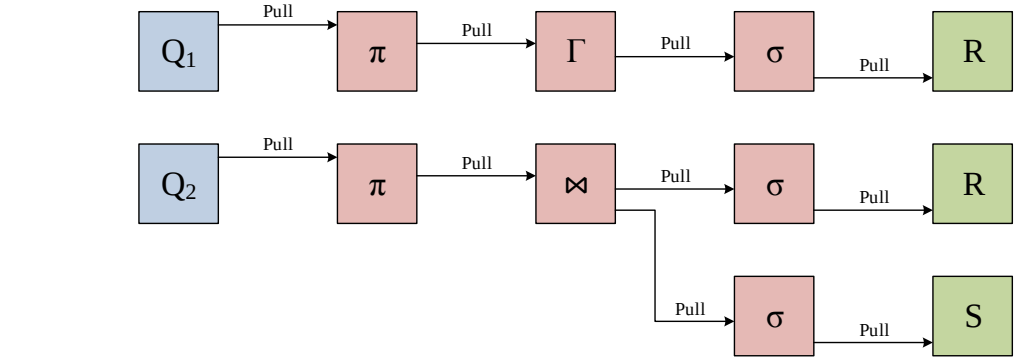
Another problem of MQO is the *synchronization* of the execution of queries with common subexpressions when queries are submitted at different moments in time. To this end, the QPipe system developed a new query processing model that allows to exploit MQO if the queries were executed within a certain time frame [HSA05].

The big advantage of this approach is that sharing does not slow down queries if the queries arrive at different points in time. Another contribution of the QPipe project is to differentiate between *data sharing* and *work sharing* and to provide a comprehensive taxonomy on the different forms of sharing for different kinds of database operators. Opportunities for data sharing arise in scan operations through base data; i.e., shared scans. Opportunities for work sharing in QPipe, however, only arise in the presence of common sub-expressions. In order to unfold its full potential, therefore, QPipe also relies on common subexpression detection. Thus it has only been studied for OLAP workloads, just as classic MQO techniques, and does not address the issue of sharing work across short running queries.

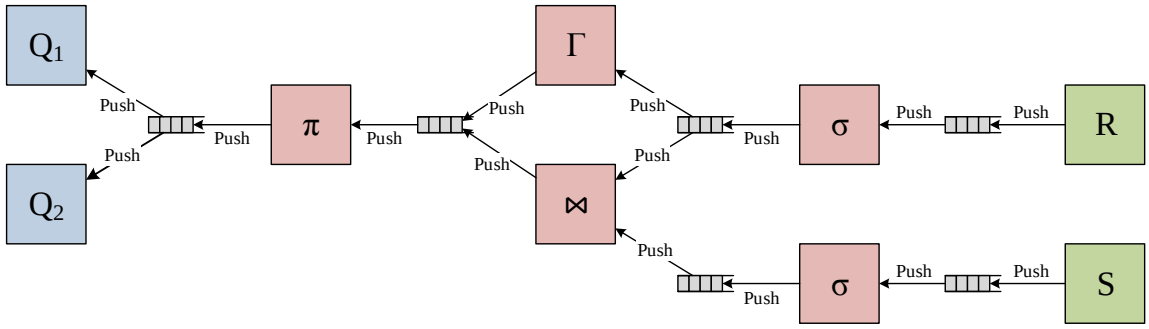
2.1.3 Push-based Execution

Both QPipe and SharedDB fall into the larger class of push-based database engines. Consider Figure 2.2 for an illustration of the difference between a traditional, pull-based execution model, and a push-based execution model. In a pull based model, operators (selection, join, projection, etc.) and resources (threads, memory, I/O) are allocated to query plans. In this computation driven model, each query plan independently pulls data from the lower level operators (typically storage engines) to the higher level operators by requesting the "next tuple" (or next batch of tuples). Consequently, pull-based engines are easy to modularize and extend, but suffer from bad code and data access locality, since any thread of execution constantly jumps from one operator to another [HA05].

In contrast, push-based engines allocate resources over a network of processing stages and operators. StagedDB [HA05] was one of the first database system to implement such a push-based data flow network. Queries and intermediate results are streamed (pushed) through this network of stages and operators, connected by buffered queues. Each operator runs in its own context and can be shared by concurrent queries. Because individual operators execute in tight loops and can be bound to specific CPU cores and memory regions, code and data access locality is generally much better than in pull-based engines. QPipe [HSA05] is also based on a similar design.



(a) Pull-based Execution Model



(b) Push-based Execution Model

Figure 2.2: Pull vs Push Query Execution Model

A push-based, data-flow model for query processing was used by the Eddies project [AH00]. While Eddies have a number of technical similarities to SharedDB, they were designed for a very different purpose. They were build to provide runtime-adaptivity of query execution where static query plan generation is not sufficient, however they can not provide either high throughput for concurrent workloads, or robust performance. Additionally, Eddies cannot provide any response time guarantees. In contrast, SharedDB is static and does not adapt the query plan at runtime. As a result, SharedDB will not always achieve the best possible response time for any given query, but SharedDB is able to provide response time guarantees which is the primary goal of this system.

Another example of a push-based engine that supports full SQL processing is DataPath [ADJ+10]. DataPath introduces a push-based architecture, where query computation is driven by data that is pushed by the storage engine. Chunks of data traverse through virtual waypoints (selection, join, aggregation waypoints) before they are pushed as output to the clients. If there is not enough computation power, chunks are dropped and re-pushed by the storage engine afterwards. While chunk dropping might be a good load balancer for disk databases, a different technique is required for in-memory databases. Dat-

aPath uses a single hash table for all join operations. One reason is to allow asynchronous query admission, similarly to CJoin. While DataPath demonstrates that combining a work-sharing system with a push-based query execution model is feasible and efficient, the system design is hindering performance, as it does not implement deep state sharing across queries. Also, the approach of trying to push data as much as possible from the storage engine reduces sharing opportunities and creates significant redundant work. As a result DataPath's experimentation is limited to up to 7 concurrent queries.

A specific problem of shared computation is that it may result in *deadlocks* in a pull-oriented query processor [DSRS01]. This problem can be alleviated by a *push-oriented* query processing approach which is the approach adopted by SharedDB. This push-based processing model has its roots in data stream processing technology; e.g., YFilter [DAF+03]. Even though it is applied to a totally different query processing paradigm, SharedDB has a number of additional commonalities with the YFilter approach: Just as YFilter, SharedDB compiles a set of queries into a single query plan and continuously uses the same plan for different generations of queries. In most applications, this approach becomes possible because the *kinds* of queries are known in advance and running an application over time involves executing the same queries with different parameter settings. The implementation of the TPC-W benchmark, for instance, involves about thirty different JDBC PreparedStatements that are executed with different parameter settings. The idea of such an *always-on-query plan* for short-lived queries is also used in CJoin, and is one of the commonalities between SharedDB, CJoin, and DataPath.

The idea of shared execution in the context of push-based systems has gained some momentum recently. Psaroudakis et. al. [PAA13] have conducted an extensive review of sharing methodologies and performed a comparison of two different approaches to implement sharing in push-based systems. The work compares existing research prototypes and conducts a sensitivity analysis of how sharing affects the overall performance.

There are a number of similarities in the design of SharedDB and all the aforementioned systems, but there are also fundamental differences. At the core of CJoin and DataPath are special, dedicated join algorithms in order to facilitate shared computation in a *pipelined* query execution model à la QPipe. In contrast, SharedDB is based on a *batched* query execution model. Furthermore, SharedDB uses standard query processing techniques such as index nested-loops, hashing and sorting for any kind of operator of the relational algebra (e.g., joins, grouping, ranking, and sorting) whereas CJoin and DataPath are limited to shared computation of joins, and to cases in which the particular CJoin and DataPath join methods show good performance.

2.1.4 Predictable Performance Optimization

In traditional database management systems, a query optimizer is responsible for automatically translating a declarative (SQL) query into a physical execution plan. The introduction of System R [CAB+94] triggered a lot of research on improving the query optimizer. Markl et al. [MRS+04] propose "progressive query optimization", which is to validate cardinality estimates against actual numbers during query execution, in order to abort query processing and re-optimize if strong discrepancies are detected. Yet, the goal of all this research focuses on delivering better performance for each individual query. Under high throughput the generated plans fail to deliver predictable performance, as resources become scarce and queries compete for them.

Of course, there is a great deal of other related work on optimizing databases to meet SLAs in high-throughput situations. Examples include indexing, materialized views, and the design of advisors to get the best physical database design for a given workload [CN97]. Other examples include caching (e.g., [DFJ+96]), reuse of query results (e.g., [IKNG09]), or optimization techniques that control the data placement in a distributed system (e.g., [KFDA00]). Various aspects of shared scans, one particular building block exploited in SharedDB (and QPipe, CJoin, and DataPath), have been studied in [Fer94, ZHNB07, QRR+08, UGA+09].

2.2 System Overview

This section presents the key ideas that combined provide the unique performance and predictability characteristics of SharedDB for a large range of different workloads: the data-query model, the global query plan, batched query execution, and shared operators. Furthermore, this section contains a detailed comparison of SharedDB versus its closest competitors. The implementation specific details of SharedDB are described in Section 2.3.

2.2.1 Data-Query Model

SharedDB features a data-query model to represent (shared) intermediary query results. This data-query model extends the relational data model of all intermediate results by adding an additional column which keeps track of the identifiers of queries that are potentially interested in a tuple. Specifically, the schema of an (intermediary) Relation R is represented as follows in this data-query model:

$$\{R_a, R_b, ..., R_n, \texttt{query\_id}\}$$

Here, R_i are the (normal) attributes of R. The `query_id` attribute uniquely identifies a query that is currently active in SharedDB. Figure 2.3a shows an example of how SharedDB uses this model to keep track of which query is interested in a tuple. As will be shown in the following sub-sections, the `query_id` can be used in relational operators just as any other attribute. For instance, it could be part of the join predicate between two relations.

In a traditional, "first normal form" relational database system, the implementation of the data-query model would result in a great deal of redundancy. If a tuple is relevant for multiple queries, multiple copies of this tuple would have to be generated and processed; one for each relevant query. As a result, the complexity of query processing and the main memory footprint would grow linearly with the number of queries.

The goal of SharedDB is to do better. Therefore, SharedDB implements this column internally as a *set-valued attribute* (i.e., using the NF2 model). As a result, an operator (e.g., a predicate) must be applied to a tuple only once independent of the number of concurrent queries and updates that may have subscribed to that tuple. Furthermore, the main memory footprint is reduced significantly. For illustration, Figure 2.3b shows the significant compression of intermediate results sets when using this set-valued attribute.

There is a question of how to implement set-valued attributes most efficiently. In the literature, two data structures have been proposed: (a) bitmaps and (b) lists. For instance, both CJoin [CPV09] and DataPath [ADJ+10] represent tuple membership in queries using bitmaps. CJoin's approach is to have a global mapping from queries to bit locations in the bitmap, which will be problematic when the size of the system in terms of operators and relations increases. DataPath does not explain the way bits are mapped to queries in their system.

Handling query_ids in lists outperforms bitmaps in cases of short-running point queries (OLTP), while using bitmaps is more efficient in cases of long-running range queries. This is because bitmaps have a fixed memory footprint. If the overlap of queries is small, then storing query_ids in lists requires a smaller memory region. On the other hand, query membership (testing whether a tuple contains a given query id) is slower when using lists, as in the worst case, the whole list has to be tested. In the case of bitmaps, query membership has a complexity of $O(1)$: a tuple belongs to the qth query, if bit q is set.

The first prototype of SharedDB uses lists, due to their simplicity in implementation and reduced memory requirement. Further optimizations in SharedDB (Section 2.3.6) required switching to bitmap representation, due to the fast query membership test, as well as the inherent advantages of bitmaps when conjuncting and disjuncting sets.

Sample Dataset:

RowID	Name	City	Country	Balance	Other Attributes
1	John Smith	Zurich	CH	150.00	...
2	Kate Johnson	Lausanne	CH	100.00	...
3	George Dallas	Munich	DE	200.00	...
4	Jim Simpson	Bern	CH	450.00	...
5	Karl Smith	Koln	DE	130.00	...

Sample Queries:
Query 1: `SELECT * FROM CUSTOMERS WHERE COUNTRY = 'CH'`
Query 2: `SELECT * FROM CUSTOMERS WHERE BALANCE` $\geq$ `150`

Sample Resultset:

RowID	Name	City	Country	Balance	Other	QueryId
1	John Smith	Zurich	CH	150.00	...	1
1	John Smith	Zurich	CH	150.00	...	2
2	Kate Johnson	Lausanne	CH	100.00	...	1
3	George Dallas	Munich	DE	200.00	...	2
4	Jim Simpson	Bern	CH	450.00	...	1
4	Jim Simpson	Bern	CH	450.00	...	2

(a) Example of SharedDB's Data-Query Model

Sample Resultset (NF2):

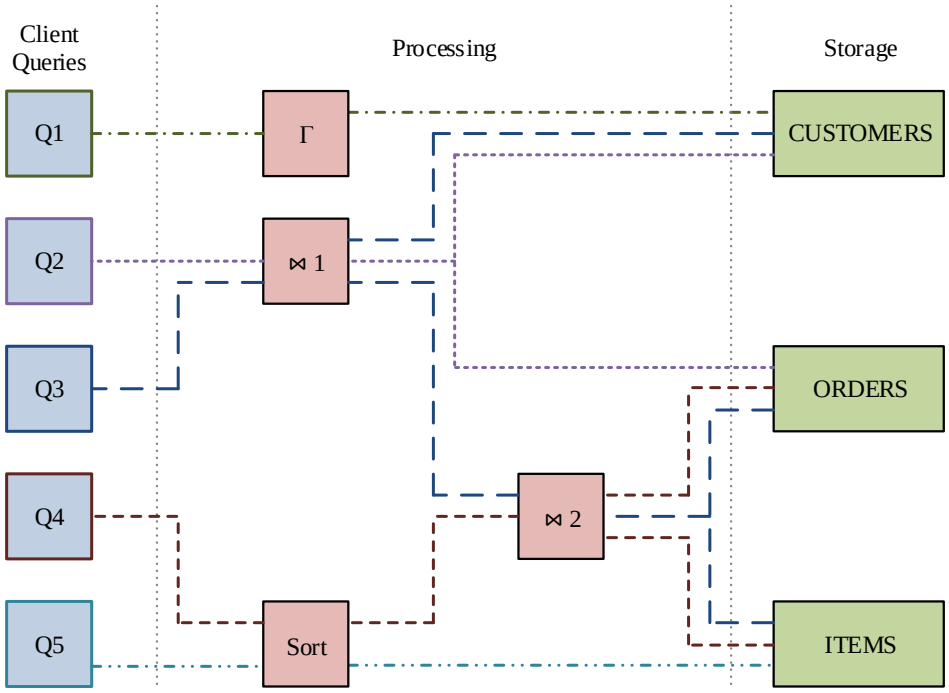
RowID	Name	City	Country	Balance	Other	QueryIds
1	John Smith	Zurich	CH	150.00	...	1, 2
2	Kate Johnson	Lausanne	CH	100.00	...	1
3	George Dallas	Munich	DE	200.00	...	2
4	Jim Simpson	Bern	CH	450.00	...	1, 2

(b) Set-Value Representation of Resultset

Figure 2.3: Data-Query Model

2.2.2 Global Query Plan

Instead of compiling every query into a separate query plan, SharedDB compiles the whole workload of the system into a single global query plan. The global query plan may serve hundreds or thousands of concurrent SQL queries and updates, and may be reused over a long period of time, possibly for the entire lifetime of the system. As stated in Section 2.1, this approach was pioneered in the context of continuous query processing in data stream



Query	SQL
Q1	`SELECT COUNTRY, SUM(CUSTOMER_ID) FROM CUSTOMERS GROUP BY COUNTRY`
Q2	`SELECT * FROM CUSTOMERS C JOIN ORDERS O ON C.CUSTOMER_ID = O.CUSTOMER_ID WHERE C.USERNAME = ? AND O.STATUS = 'OK'`
Q3	`SELECT * FROM CUSTOMERS C JOIN ORDERS O ON C.CUSTOMER_ID = O.CUSTOMER_ID JOIN ITEMS I ON O.ITEM_ID = I.ITEM_ID WHERE I.AVAILABLE < ?`
Q4	`SELECT * FROM ORDERS O JOIN ITEMS I ON O.ITEM_ID = I.ITEM_ID WHERE O.DATE > ? ORDER BY I.PRICE`
Q5	`SELECT * FROM ITEMS I WHERE I.CATEGORY = ? ORDER BY I.PRICE`

Figure 2.4: Example of a Global Query Plan

processing systems (e.g., YFilter [DAF+03]) and first applied to traditional, short-lived queries as part of the CJoin system [CPV09].

Figure 2.4 shows an example of such a global query plan. In this example, four database operators are executed on three tables to evaluate five different query types. Figure 2.4 shows how different kinds of queries can share operators in different ways. For instance, both Q_2 and Q_3 involve a join between the *Users* and *Orders* table; consequently, this

join is shared between queries of these two types (denoted as $\bowtie 1$ in Figure 2.4). Likewise, queries of types Q_3 and Q_4 share the join between the *Orders* and *Lineitems* table. The sort on *price* can be shared between queries of types Q_4 and Q_5.

Just as important as sharing across different types of queries is sharing within the *same* type of query. For instance, the plan shown in Figure 2.4 could be used to execute hundreds of concurrent queries of type Q_4 (in addition to hundreds of concurrent queries of the other types), all with different parameter settings for the `ORDERS.DATE` predicate. All these concurrent Q_4 queries would share the same join and sort operators so that only a single big join and sort would be carried out for all these Q_4 queries.

To understand the execution model of SharedDB, it is important to define what *concurrent* means. As mentioned in Section 2.1, SharedDB facilitates the sharing of operators in a very different way than QPipe, CJoin, DataPath, and other related systems that support operator-level sharing of computation. These systems start executing each query as soon as it arrives. Unfortunately, this approach limits the opportunities to share computation for many database operators as observed in [HSA05]. In order to overcome these limitations, CJoin and DataPath devise specific join methods. In contrast, SharedDB *batches* queries and updates; it is this batching that enables SharedDB to exploit shared computation in a scalable and generic way, thereby making use of traditional, best-of-breed algorithms to implement joins, sorting, and grouping.

SharedDB batches queries and updates in the following way: While one batch of queries and updates is processed, newly arriving queries and updates are queued. When the current batch of queries and updates has been processed, then the queues are emptied in order to form the next batch of queries and updates. Metaphorically, SharedDB works like the blood circulation: With every "heartbeat", tuples are pushed through the global query plan in order to process the next generation of queries and updates. For OLTP workloads, these heartbeats can be frequent in the order of hundreds of milliseconds or even less.

The queueing of queries is fine-grained per input relation and query operator: In the example of Figure 2.4, queries of Type Q_1, Q_2, and Q_3 would queue for reading the `CUSTOMERS` relation, queries of Type Q_2, Q_3, and Q_4 would queue for reading the `ORDERS` relation and so on. Then `CUSTOMERS` tuples (generated for all three query types) would queue for the Γ and $\bowtie 1$ operators which in turn would process these tuples in batches so that all `CUSTOMERS` tuples belonging to a specific query are processed within a single batch. Hard-wired pipelines are only created for certain operators; for instance, an operator can stream its output into the *build* phase of a hash join and even then, tuples are processed in batches following a *vector model* of execution for better instruction cache locality [HA05, HNZB07]

(Section 2.3).

SharedDB works particularly well if most query types are known in advance; e.g., as part of JDBC PreparedStatements. In this case, SharedDB can generate a global query plan for all the query types known in advance and exploit sharing in the best possible way for these queries. Ad-hoc queries need to be processed individually. Nevertheless, even ad-hoc queries can take advantage of sharing. For instance, an ad-hoc query that asks for the ten users that have placed the most orders could share $\bowtie 1$ with all queries of type Q_2 and Q_3 in the global plan of Figure 2.4. The Top 10 operation of that ad-hoc query, however, would have to be compiled and executed separately just as in any other traditional database system. In some sense, all operators of the global plan can be regarded by the query compiler as materialized views which are available to speed-up the processing of ad-hoc queries. This observation has also been made in the QPipe project [HSA05].

2.2.3 Shared Execution

One of the key innovations of SharedDB is the way it implements relational operators that share execution. These *Shared Operators* take advantage of the Data-Query Model that was described in Section 2.2.1, to process the (intermediary) tuples of all concurrently executed queries.

The Shared Operators process a multiplexed stream of tuples that are fetched by different queries in a push based fashion. To best describe the concept and compare it to query-at-a-time execution, as well as multi-query optimization, we will use an example of three prepared statements that include a join of the well known `CUSTOMERS` and `ORDERS` relations.

Consider the prepared statements of Figure 2.5a. All three statements ask for a join of the same relations, yet they have very different predicates. Figure 2.5b shows how these three queries would be implemented in a query-at-a-time system. Each query is parsed and compiled individually, thereby pushing down predicates. This step is typically referred to as *logical query optimization* [HFLP89]. To evaluate the queries, both relations have to be accessed three times, one for each statement. In case that no index is suitable for answering the queries, this translates into six full table scans to evaluate the three queries.

An improved plan to execute these statements is shown in Figure 2.5c. This plan uses multi-query optimization (MQO) [Sel88] in order to minimize storage accesses. The query predicates are combined into a complex disjunction of predicates that selects all the tuples required by all three queries. On the second step, the selected tuples are joined as if the tuples are fetched by a single query. Finally, post-filtering is performed in order to map the result tuples to the appropriate query.

MQO is able to reduce the storage accesses as well as share some computation across queries. It is extremely efficient as long as the number of queries is relatively small. For instance, executing the same plan with thousands of different parameters for each prepared statement would require an extremely complex disjunction expression. Most importantly, the post-filtering phase always repeats work, as the selection predicates have to be re-evaluated for all generated tuples.

To avoid repeating unnecessary work, SharedDB uses the approach shown in Figure 2.5d. After applying the logical query optimization, the three individual query plans are merged into a single global plan. That is, rather than processing three *small* joins (one for each query), one *big* join is executed that meets the requirements of all three queries. Technically, the *union* of all `CUSTOMERS` and `ORDERS` tuples that the three queries are interested in are considered as part of the join. Furthermore, the join predicate is amended, thereby considering the `query_id`. This way, an `CUSTOMERS` tuple that is only relevant for Query Q_1 does not match an `ORDERS` tuple that is only relevant for Query Q_2. Finally, the routing of the join results to the relevant queries is carried out using a *grouping operator* (Γ) by `query_id`.

Even though Shared Processing seems similar to Multi Query Optimization, the two techniques are very different. As already explained MQO is extremely efficient when the number of concurrent queries is low. Shared Processing is not limited, as there is no need for either a complex conjunctive expression, or post-filtering. Additionally, MQO requires detection of common subexressions. If thousands of queries are executed, the analysis of all subexpressions and their commonalities will curtail the performance of the system. SharedDB does not require such commonalities to exist, as discussed in Section 2.1.

As stated in the introduction, this way to process joins sounds like a bad idea at first glance: It is usually better to process a few small joins than to process one big join. This approach only becomes advantageous if there are many (possibly hundreds) of concurrent queries and there is overlap in the tuples that need to be processed for a set of queries. One particularly nice way in which the global join plan of Figure 2.5d supports scalability with the number of concurrent queries is by making the `query_id` part of the join predicate. This way, the `query_id` can be indexed; for instance, a hash join could be used that builds a hash table on the `query_id` of `CUSTOMERS`. This observation shows nicely how SharedDB achieves scalability by turning queries into data that can be processed using traditional query processing techniques. This is another key idea that SharedDB has adopted from data stream processing systems.

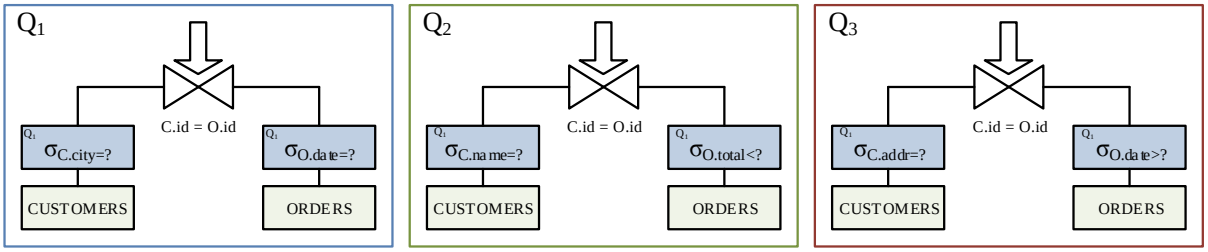
Another crucial advantage of the global join plan of Figure 2.5d is that any join method can be used; e.g., hashing, sorting, index-based, and nested-loops. In particular, any

```
SELECT *
FROM CUSTOMERS C
     JOIN ORDERS O ON C.id = O.id
WHERE
  AND C.city = ?
  AND O.date = ?
```

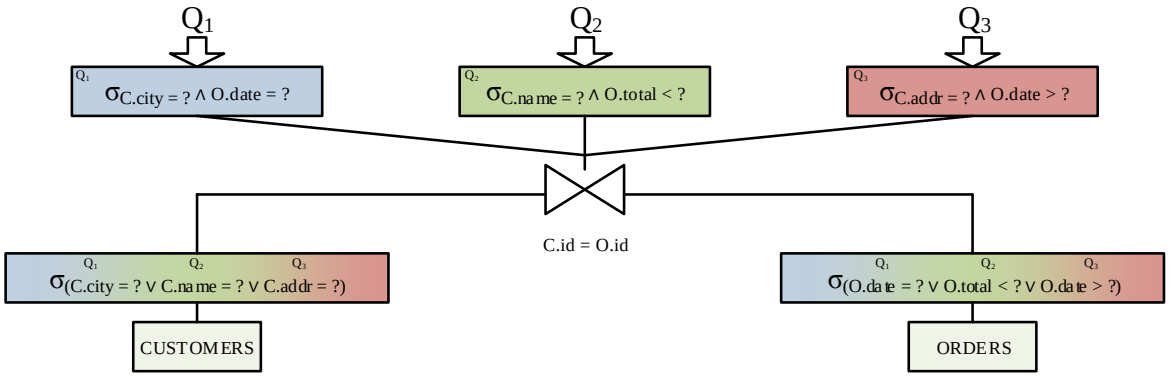
```
SELECT *
FROM CUSTOMERS C
     JOIN ORDERS O ON C.id = O.id
WHERE
  AND C.name = ?
  AND O.total < ?
```

```
SELECT *
FROM CUSTOMERS C
     JOIN ORDERS O ON C.id = O.id
WHERE
  AND C.addr = ?
  AND O.date > ?
```

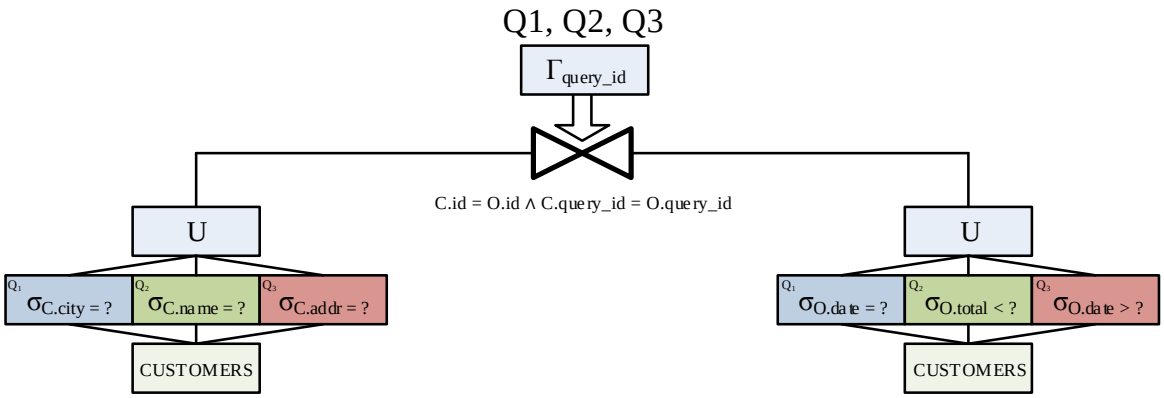
(a) Sample Prepared Statements



(b) Traditional Query Processing



(c) Multi Query Optimization



(d) Shared Query Processing

Figure 2.5: Traditional Processing, MQO and Shared Processing

parallel or cache-aware join methods can be used (e.g., [KKL+09]). Furthermore, as observed in the previous paragraph, either `CUSTOMER.id = ORDERS.id` or `CUSTOMER.query_id = ORDERS.query_id` can be used as primary join predicates. If the latter, a set-based join is carried out as studied in [HM97]. In our implementation, we use a simple hash table that maps a `query_id` to a set of pointers that reference the corresponding tuples.

It is even possible to use multi-dimensional join methods which are commonly used in spatial and temporal database systems; e.g., [DSTW02]. In summary, SharedDB can always make use of the best-of-breed algorithms. In contrast, the processing model of CJoin and DataPath constrains the use of join methods used: In some cases, using the dedicated join method of DataPath might show good performance; in other cases, however, its performance may be terrible. SharedDB makes use of multiple join methods for the same reasons as traditional database systems.

In addition to supporting multiple join methods, SharedDB does not constrain join ordering thanks to its flexible data-query model. In Figure 2.4, for instance, the `ORDERS` / `ITEMS` join ($\bowtie 2$) of Q_3 could be carried out before or after the join with *Users* ($\bowtie 1$). As a result, the following join ordering for Q_3 would also be possible:

$$(\mathit{Users} \bowtie \mathit{Orders}) \bowtie \mathit{Items}$$

This join order for Q_3 would also enable sharing $\bowtie 1$ with Q_2 and $\bowtie 2$ with Q_4, in the same way as the original join order for Q_3 depicted in Figure 2.4. To share a join across queries, SharedDB only fixes the join method and the inner and outer relation of the join for all queries that are share the join.

In Chapter 3 we discuss the details of a query compiler as well as an optimizer that is designed for such global query plans. For the purposes of this chapter, we considered plans that are based on the two-step logical approach (i.e., determine join orders for each query individually and then merge the plans for the individual queries into a single global plan), and hand tuned based on microbenchmarks.

The idea shown in Figure 2.5 to evaluate shared joins can also be applied to evaluate any other operator of the relational algebra. Figure 2.6 illustrates the principle for a shared sort using two example queries and a few tuples of a *Users* table. Again, in theory, it is better to have a few small sorts than one big sort, but sharing may more than offset this effect. In this example, it is more efficient to do *one* sort with four tuples than to do *two* sorts with three tuples each. Obviously, the overlap increases with the number of queries.

The Top-N operator is an extension of the sort operator and can, thus, benefit from sharing in a similar way as the sort operator. In the SharedDB implementation, the shared Top-N

Shared Sort Queries

Query A:
```
SELECT * FROM USERS
WHERE BIRTHDATE > 1980.01.01
ORDER BY NAME
```

Query B:
```
SELECT * FROM USERS
WHERE ACCOUNT > 1000
ORDER BY NAME
```

Relation: USERS

Name	Account	Birthdate
John Smith	3,000	1980.03.05
Kate Johnson	800	1976.04.11
Bill Harisson	1,230	1978.03.02
Nick Lee	540	1982.02.09
James Meyer	2,300	1981.03.09

Selection

Query_Ids
A, B
B
A
A, B

Sorted Output

Name	Account	Birthdate	*Query_Ids*
Bill Harisson	1,230	1978.03.02	*B*
John Smith	3,000	1980.03.05	*A, B*
James Meyer	2,300	1981.03.09	*A, B*
Nick Lee	540	1982.02.09	*A*

Figure 2.6: Shared Sort

operator first sorts all the tuples that are relevant for all the active queries; thus, the sorting is shared. Then, it filters the Top N results for each query individually.

Like Top N, the Group-By operator is carried out in two phases. In the first phase, the input tuples are grouped. Again, this phase can be shared so that all the tuples that are relevant for all active queries are grouped in one big batch. Also, any kind of grouping algorithm (e.g., hashing or sorting) can be used for this purpose. In the second phase, `HAVING` predicates and aggregation functions are applied to the tuples of each group. Just as for Top N, this second phase must be carried out for each query individually. Fortunately, grouping is the most expensive part so that sharing can be applied to reduce the cost of the most performance critical operation.

The storage manager of SharedDB provides two operators for accessing tables: *shared table scans* and *shared index probes*. We do not claim any novelty here. Shared table scans have been studied extensively in previous work ([Fer94, ZHNB07, QRR+08, UGA+09]) and SharedDB uses Crescando and the ClockScan algorithm proposed in [UGA+09]. Shared index probes have also been studied in the past (e.g., [FK05]) and SharedDB simply adopts the well established techniques in that domain, too. Both operators generate tuples in the data-query model such as those shown in Figure 2.3a.

2.2.4 Discussion

This section summarizes the main advantages and disadvantages of SharedDB in comparison to traditional, query-at-a-time systems and other systems that exploit shared computation.

2.2.4.1 SharedDB vs. "query-at-a-time"

The main *dis*advantage of SharedDB is that it adds latency to each query due to its batch-based execution model. In contrast, traditional database system start processing queries as soon as they are submitted by an application. In the worst case, batching increases latency by a factor of 2: one cycle of queuing and one cycle of actual query processing.

The biggest *advantage* of SharedDB is that it is able to bound computation and scales with the number of concurrent queries and updates. In the worst case if there are many concurrent queries that involve all the tuples of the whole database, SharedDB joins and sorts the *whole* relations as part of the global query plan, independent of the number of concurrent queries. This way, SharedDB can give response time guarantees which is critical for many modern applications to meet SLAs. For instance, if SLAs specify that all queries must be processed within 3 seconds, then SharedDB would provision enough CPU cores such that a batch of queries can be processed in at most 1.5 seconds in the worst case. All this is predictable and can be planned upfront: There is no interference and resource contention between concurrent queries because SharedDB schedules the data flow and the utilization of cores at compile-time as part of its global plan. In contrast, the work carried out by traditional database systems grows linearly with the number of concurrent queries. Furthermore, traditional database systems allocate a separate thread for each query and these threads might compete for shared resources (e.g., the main memory bus or processor caches) in an unpredictable and uncontrollable way.

SharedDB is not ideal for transactional workloads that contain only short running point-queries. These queries are executed so fast that it is very hard to gather a critical mass of

queries, such as the benefits of batching are visible. A query-at-a-time system would be a better fit for such workloads, as the amount of queries fighting for resources is always small. Additionally, such queries have very small overlap. They usually are very selective, fetching and processing only a couple of tuples. The probability of two queries that belong in the same batch, ask for the same tuples is very small, nullifying the benefit of work sharing.

On the other hand, SharedDB is better fit for workloads that contain long running analytical queries, as sharing possibilities are more common and the higher query execution time allows more queries to be batched together. The amount of analytical queries in the workload can be as low as 2% to 3%, as our experimental study of Section 2.4 shows. Such workloads are very common nowadays, due to the importance of real-time analytics and online business intelligence.

As mentioned in Section 2.2.3, SharedDB might result in extra work if the load on the system is light and there is little or no overlap in the data processed by the queries that share a common operator. With an increasing load, however, the overlap increases. In theory, if each query Q_i needs to process n_i tuples, there are k concurrent queries, $n = \Sigma_{i=1}^{k} n_i$, and o is the number of tuples that at least one query needs to process ($o \leq n$), then SharedDB will save work for an operator with complexity $\mathcal{O}(f(n))$ if:

$$f(o) < \Sigma_{i=1} k f(n_i)$$

For operators with *linear* complexity this equation is always fulfilled, unless $o = n$ which is the worst case for SharedDB. This is the case for table scans and joins under certain circumstances. For operators with a complexity of $f(n) = n * logn$ (e.g., sorts and certain joins), the advantage of SharedDB depends on o and n: In such cases, SharedDB will result in extra work in the worst case ($o = n$). But, even in these bad cases, SharedDB maintains its property of bounded computation and predictable performance.

2.2.4.2 SharedDB vs. "pipelined sharing"

The most significant *dis*advantage of SharedDB as compared to other, recent approaches to effect shared computation (e.g., QPipe, CJoin, and DataPath) is again that SharedDB adds latency due its batched processing model. In contrast, QPipe, CJoin, and DataPath have a continuous query processing model. The big advantage of this batched processing model in concert with the special way in which joins and other operators are processed, is its generality to any kind of operator of the relational algebra, any kind of algorithm (e.g., join method), and to the processing of updates. This generality enables SharedDB

to process any kind of workload (OLTP, OLAP, and mixed) without any special tuning, whereas QPipe, CJoin, and DataPath have so far only been shown to work well for OLAP workloads with queries that each involve processing a large portion of the entire database. As observed in the QPipe work [HSA05], the kinds of sharing are limited depending on the query operator in a continuous query processing model. As a result, CJoin and DataPath rely on specific, dedicated join methods in order to carry out shared join computation; other operators (e.g., sorting and grouping) need to be carried out for each query individually in these systems. Furthermore, these dedicated join methods only show good performance for specific kinds of workloads. Just as a traditional database system, SharedDB supports multiple join methods in order to adapt to different kinds of workloads.

A specific advantage of SharedDB as compared to QPipe and DataPath is its ability to meet SLAs and bound the response time of queries. Predictable performance is also supported in CJoin, but only for star-join queries. Another advantage of the batched processing model of SharedDB is that it supports concurrent updates and strong consistency (e.g., Snapshot Isolation). Section 2.3 gives more details on how updates are processed in SharedDB. Again, updates and transaction processing have not been studied in the context of QPipe and DataPath; CJoin does support concurrent updates, but the model is more complicated than in SharedDB.

2.3 Implementation Details

This section describes a number of implementation details of SharedDB that will help the reader get a better understanding of the system.

2.3.1 Operator Model

The building block of a SharedDB Global Query Plan is an operator, a common characteristic of all staged database systems. Figure 2.7 presents the high level operator model in SharedDB.

Operators communicate by exchanging data structures through two queues. This message passing interface is very common in modern software implementation that are designed for scalability and robustness (i.e. [BPS+09]). The message-passing primitive can make more efficient use of the interconnect and reduce latency over sharing data structures between cores. The first queue is the “Query Queue”, where a consumer operator places a query for this operator. For instance, in the case of the global query plan that implements the

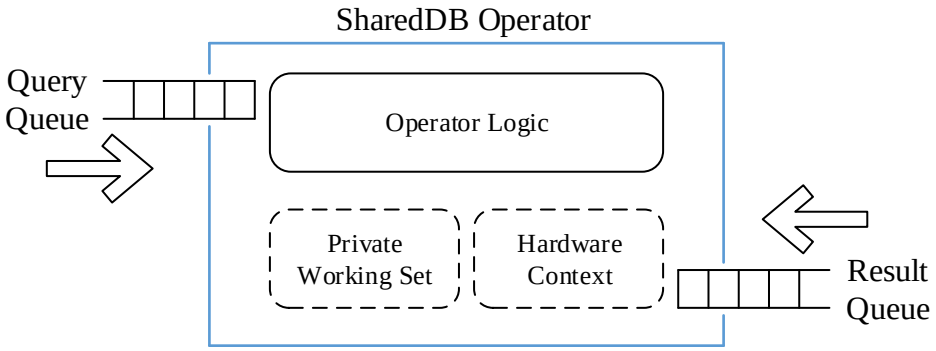


Figure 2.7: Overview of a Generic SharedDB Operator

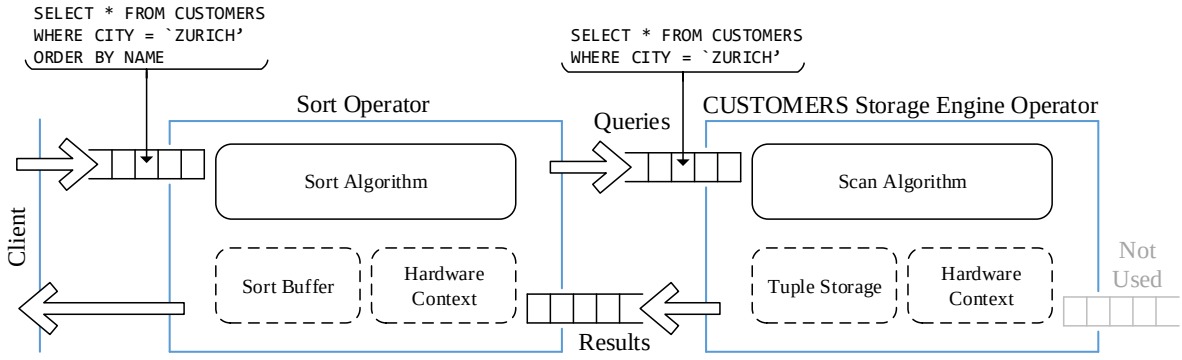


Figure 2.8: Query Evaluation and Chaining of Operators in SharedDB

query:

```
SELECT * FROM CUSTOMERS WHERE CITY = 'ZURICH' ORDER BY NAME
```

the `ORDER BY` operator will enqueue the respective *sub* query to the operator that implements the `CUSTOMERS` relation. In this case, this subquery will be the selection of customers that comply with the predicate `CITY = 'ZURICH'`. The whole setup of this example is shown in Figure 2.8.

The second queue is the "Result Queue". This is used by the producing operators to pass result tuples to the consumers. In the previous example, the `CUSTOMERS` operators will place all matched tuples in the result queue of the `ORDER BY` operator. In order to result the number of messages exchanged between operators, result tuples are inserted into batches in the result queue.

SharedDB operators implement the data processing logic by using the messages in these two queues. Most operators use a dedicated hardware context (CPU core) and run exclusively on this context, using hard processor affinity. This ensures data cache locality, as well as instruction cache locality, which effectively improves performance. Specialized

operators may have multiple hardware contexts. For instance, a shared parallel join uses multiple threads, each one running on a different core, to process the tuples of the result queue. Other operators might not require a dedicated hardware context. This is the case for lightweight filtering operators, for instance a `LIMIT` operator. In this case the hardware context is shared with another operator, most likely the producer that feeds the result queue of the `LIMIT` operator.

Operators that require a big working set, for instance the sort buffer of a `ORDER BY` operator, use a private working set. This memory region is exclusive to this operator, which reduces the need for thread consistency primitives. This allows for higher performance, as a i.e. `HASH JOIN` operator does not require any hash bucket locks to insert items into the hash table, even if result tuples originate from different producing operators. Finally, in order to reduce memory access time, the private working set of all operators, including the thread stack are all allocated in the local NUMA (non-uniform memory access) region. This is critical on modern hardware NUMA architectures and allows for maximum memory bandwidth and minimum access latency.

Another innovation of SharedDB is the way queries are handled by operators. Instead of processing one query at a time, SharedDB operators process multiple of them *concurrently*. Once an operator is idle, it dequeues all currently queued queries from the query queue. This batch of queries is called the active set of queries, and the operator must finish evaluating it before processing anything else. The operator specific algorithm runs for this batch, until all work has been completed. Finally, a special *end-of-stream* result tuple is send to all consumers. In this case the consumers are all the different operators that enqueued a query in the first place. Then the operator is idle again and checks for any enqueued queries that are waiting to be handled, and the process repeats. The time it takes to evaluate a batch of queries, is called a *cycle*. Any additional queries that arrive after the cycle has started, are queued and will be evaluated during the next cycle. This leads to a concept of *always-on* operators that wait for incoming work, which is either queries that have to be evaluated, or tuples that have to be processed.

Algorithm 1 describes an abstract SharedDB operator that processes queries in batches. At the beginning of the cycle, the operator dequeues the pending queries and activates them by issuing their subqueries to the respective operators. Then it receives the result tuples they generated, processes them and forwards the processed output to the issuers of the queries. For example, in the case of a filter operator, like SQL `LIKE`, the `ProcessTuple` function tests the tuple against the `LIKE` expression. If it matches, the result tuple is pushed to the next operator in the pipeline.

Blocking operators, such as the `SORT` operator, can use the function `ProcessTuple` to

Algorithm 1: Skeleton of a SharedDB Operator

```
Data: SyncedQueue iqq;                          /* incoming queries queue */
Data: SyncedQueue irq;                          /* incoming result tuples queue */
while true do
    Array aq ← ∅ ;                              /* Array of active queries */
    /* Operators that will receive the produced results                    */
    Array consumers ← ∅;
    /* Activate all queries in the incoming queue                          */
    while ¬IsEmpty(iqq) do Put(aq, Get(iqq)) ;
    /* Enqueue subqueries to underlying operators                          */
    foreach Query q ∈ ag do
        Query subQuery ← GetSubQuery(q);
        Operator op ← GetOperator(subQuery);
        EnqueueQuery(op, subQuery);
        Operator consumer ← GetConsumer(q);
        Put(consumers, consumer);
    end foreach
    /* Loop until all active queries have finished                         */
    Number queriesLeft ← Size(aq);
    while queriesLeft do
        /* Receive a tuple from the underlying operators                   */
        Tuple t ← Get(irq);
        /* Process the incoming tuple. Depending on the operator, this function
           might generate results                                          */
        Tuple resultTuple ← ProcessTuple(t);
        if ¬IsNull(resultTuple) then
            SendResult(consumers, resultTuple);
        end if
        if IsEndOfStream(t) then
            queriesLeft ← queriesLeft − 1;
        end if
    end while
    /* Notify the consumers that processing has finished                   */
    SendEndOfStream(consumers);
end while
```

append the tuple to a buffer structure (i.e., a vector). The same buffer structure is used for all the queries that belong to the same batch. In this case, no results are produced from `ProcessTuple`. Once all the result tuples have been received, the buffer structure is sorted and it is pushed to the consumers as part of the `SendEndOfStream` function.

2.3.2 Query Model

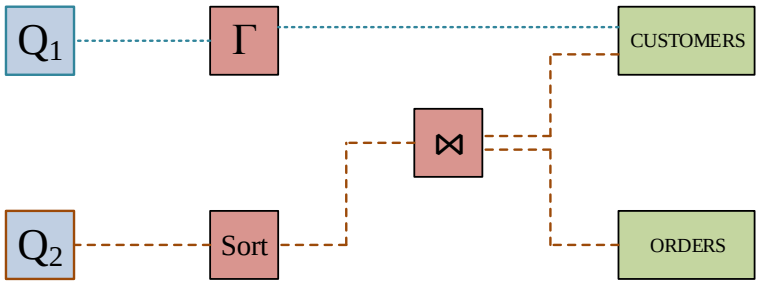
As described in Section 2.2, SharedDB evaluates queries using a data flow network of always-on database operators. There are no individual query plans and instead a global plan is always active.

Every SharedDB query describes a path of operators in the data flow network. This means that operators are not hard-wired in the global query plan, but instead they are able to receive messages (queries) from any other operator. The path, or the single-query plan is represented as a chain (or tree) of subqueries. Each subquery has three properties that depend on what operator executes it. These are the `query_id`, the chained query or queries, and the configuration. As explained in Section 2.2.1, the `query_id` uniquely identifies each currently active query in the whole system. The chained query is the query that has to be issued to the underlying operator, which in that case is the producing operator. Special queries have zero chained queries. This is the case for, i.e., table access operators that require no producer. Queries are also allowed to have more than one chained query. For instance a join query requires two chained queries: one to fetch the tuples of the inner relation and one for the outer relation. Finally, the configuration is operator specific and describes what the operator has to execute in order to evaluate this query.

An example of how queries are logically represented inside SharedDB is shown in Figure 2.9. The two queries of the example form the global query plan of Figure 2.9a. Figure 2.9b shows how the subqueries are chained as well as their sample configuration. In this example, the `GROUPBY` operator receives a query from the Output operator of Query 1. In order to execute it, another query is issued to the `TableScan:CUSTOMERS` operator. Result tuples are generated by the TableScan, passed to the GroupBy operator and finally sent to the clients.

2.3.3 Result Management

We described how intermediary results look like inside SharedDB in Section 2.2.1. They use an extended data-query model, in which tuples are augmented with information about the query that generated them. The query information is represented with `query_id`s, that



(a) Sample Global Query Plan

Query 1

	Operator	Configuration
1.	TableScan:Customers	CUSTOMERS WHERE LAST_LOGIN < 2011.01.01
2.	GroupBy	GROUP(CUSTOMERS.COUNTRY) SUM(CUSTOMERS.BALANCE)
3.	Output	Network, TCP Port 5843

Query 2

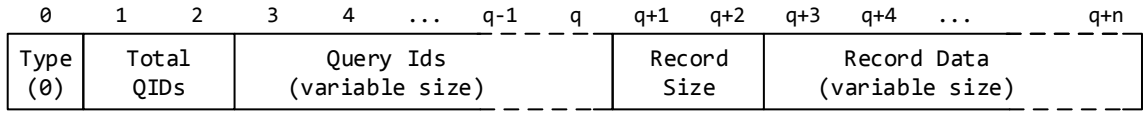
	Operator	Configuration
1.	TableScan:Customers	CUSTOMERS WHERE CITY = 'ZURICH'
–	TableScan:Orders	ORDERS WHERE AMOUNT > 1,000
2.	HashJoin	CUSTOMERS.C_ID = ORDERS.C_ID
3.	Sort	ORDERS.AMOUNT
4.	Output	Network, TCP Port 5844

(b) Queries Logical Representation

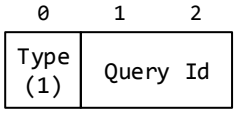
Figure 2.9: An Example of Query Representation in SharedDB

can be implemented either with a variable sized arrays, or a bitset. Besides the normal results that contain tuples, SharedDB uses a special type of results, the *end-of-stream* results. End-of-stream results are used to notify the consumer operator that evaluation of a query is complete.

Figure 2.10 shows the physical layout of results in SharedDB. The result tuples, which are the results that carry data around, have two variable sized fields. The `query_ids` and the record data (tuple). Independent of whether an array or a bitset is used to represent the `query_ids` field, the physical layout of the record is the same. The bitset implementation



(a) Physical Layout of Result Tuples



(b) Physical Layout of End-of-Stream Results

Figure 2.10: Traditional Processing, MQO and Shared Processing

of course provides constant, $\mathcal{O}(1)$ check for the a specific `query_id`, while an array requires linear, $\mathcal{O}(q)$ time. Nevertheless, a bitset requires a size that is equal to $\left\lceil \frac{max(\texttt{query\_id})}{8} \right\rceil$, while an array has a memory footprint of $\mathcal{O}(q)$, as it contains only as many `query_id`s as necessary. An extended analysis on the representations of `query_id`s will be presented in Section 2.3.6. The special end-of-stream results contain only one attribute, the `query_id`.

To reduce the number of messages exchanged between operators, results are batched. The size of the batch depends on the operator itself. For instance, a key-value store should use smaller batches, because it mostly answers point queries that as a rule of thumb, generate only a few tuples. On the other hand, an `Order By` operator uses huge batches which contain the whole sorted intermediary relation. In this case, using smaller batch sizes makes no sense, as `Order By` operators generate a burst of results only when their buffer is fully sorted.

Since SharedDB uses a data-flow network of loosely coupled operators, all operators must be able to distinguish if an incoming tuple should be handled, that is if the operator should be the consumer of this tuple. Result membership is well studied problem of publish subscribe systems that use similar data-flow networks [FJL+01]. In SharedDB we considered different implementations of result membership.

2.3.3.1 Rewrite `Query_ids`

One way to handle result membership is by rewriting the ids of queries in all operators. This requires that all operators have a unique range of `query_id`s and any query or result tuple with this `query_id` has to be processed by this operator. A centralized component, the Operator Manager is responsible for keeping track of all operators and assign ranges to each of them. The Operator Manager can either be an operator itself, that is use the

same interface of two queues that an operator has, or a common component of the system with no separate runtime.

In this approach, when an operator is initiated, it requests from the Operator Manager a range of `query_id`s. When a query is enqueued to an operator, the operator modifies the `query_id` of the query and all chained queries to a different one that is in the range of his own range of `query_id`s. Additionally, it maintains a mapping of the newly assigned `query_id` to the original one. Since the chained queries have the new `query_id`, all incoming result tuples that are intended for this operator will have this `query_id` as well. Result membership can be determined by doing a simple range check on the `query_id`s of all incoming result tuples.

However, there are a few disadvantages in this approach. First of all, any tuple produced by an operator needs to have the `query_id`s rewritten. This is why a mapping of new `query_id`s to old ones is required. Most importantly, this means that result tuples are not immutable and have to be modified by possibly multiple consumers. In order to support this, the producing operator has to send a physical copy of the result tuple to each consumer that is interested in tuples. For instance, the `Tablescan:CUSTOMERS` operator in Figure 2.9a has to generate two copies of each result that is produced. One for the `HashJoin` operator and one for the `GroupBy` operator. The algorithm of result membership by writing `query_id`s is shown in Algorithm 2.

2.3.3.2 `Query_id`s in Operator State

A second implementation that was considered involves storing the set of `query_id`s in the state of the operator. This means that each operator remembers which queries are active at every point in time and before processing a tuple, it checks whether this tuple is intended to be executed by this operator.

Since the incoming queries may have very different `query_id`s, a hashmap is used to store them. The hashmap maintains a mapping of `query_id` $\rightarrow$ *Query* and can be used to either just check if this query should be handled, or to retrieve the query itself. The latter is useful for operators that want to retrieve the configuration of each query. For instance, when a `LIMIT` operator is asked to process two queries with different limits, the operator has to fetch this meta-information for every processed tuple. The algorithm of this approach is shown in Algorithm 3.

As with `query_id` rewriting, this approach has also drawbacks. The hashmap has to be probed for every incoming tuple multiple times, once for each `query_id` that is present in the tuple. Additionally, the hashmap itself has a memory footprint, that may affect cache

Algorithm 2: Result Membership by `query_id` Rewriting

```
Data: OperatorManager op_man;          /* The centralized operator manager. */
Data: Query_Id low_qid, high_qid;      /* the range of query_ids for this operator. */
Data: Query_Id next_qid;               /* The next available query_id in the range. */
Data: Map[Query_Id → Query_Id] qid_map; /* query_id mapping. */

Function InitializeOperator():
    {low_qid, high_qid} ⟵ op_man.RegisterOperator();
    next_qid ⟵ low_qid;
    qid_map.Reset();
end

Function HandleIncomingQuery(q): /* Called for each incoming query. */
    Input: Query q
    qid_map.Map (q.query_id → next_qid);   /* Required in PostProcessTuple. */
    q.query_id ⟵ next_qid;                 /* Modify query_id before pushing it down. */
    next_qid ⟵ next_qid + 1;
end

Function IsMyQueryId(qid): /* Checks if a query_id should be handled. */
    Input: Query_Id qid
    return (qid ⩾ low_qid) ∧ (qid ⩽ high_qid);
end

/* Called after processing a tuple and before pushing it to any consumer operator.
   Remaps query_ids back to the original ones. */
Function PostProcessTuple(t):
    Input: Tuple t
    for i = 1 to t.no_query_ids do
        if IsMyQueryId(t.query_ids[i]) then
            t.query_ids[i] ⟵ qid_map.Get (t.query_ids[i]);
        end if
    end for
end

Function EndOfCycle(): /* Called at the end of each cycle. */
    next_qid ⟵ low_qid;
    qid_map.Reset();
end
```

Algorithm 3: Result Membership in Operator State

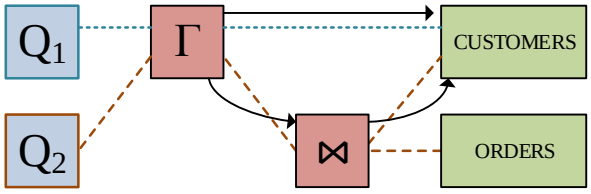
```
Data: Map[Query_Id → Query] qid_map;        /* query_id to Query mapping. */

/* Called for each incoming query.                                        */
Function HandleIncomingQuery(q):
    Input: Query q
    qid_map.Map (q.query_id → q);                    /* Remember qid. */
end

/* Checks if a query_id should be handled. This function is called for every
   incoming tuple in order to decide if the tuple should be processed by this
   operator.                                                              */
Function IsMyQueryId(qid):
    Input: Query_Id qid
    return qid_map.contains (qid);
end

/* Called at the end of each cycle.                                       */
Function EndOfCycle():
    qid_map.Reset();
end
```

conscious algorithms. Yet, there is no need to modify the result tuples and, as a result, they are immutable and there is no need to generate multiple copies as was the case with `query_id` rewriting.

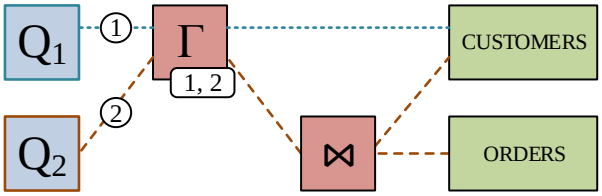
After conducting a set of microbenchmarks, using a hashmap to store the `query_id`s proved to be more efficient, especially in the cases of high loads. Furthermore, by using a carefully tweaked, compact hashmap implementation, we managed to minimize the cache misses caused by probing the hashmap.

2.3.3.3 Cyclic Paths

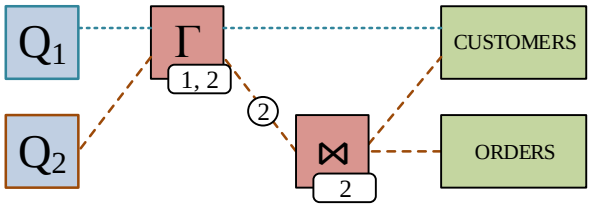
Algorithm 3 solves the result membership problem for any acyclic path in the data flow network. Queries may combine results from any operator as long as no combination of queries form a cycle. Once any number of queries form a cycle in the path, result membership by maps becomes problematic. To better explain the problem, consider an example of two queries that form the global query plan of Figure 2.11a. This global query plan forms a cycle consisting of the `GroupBy`, `Join` and `CUSTOMERS` operators. The cycle is



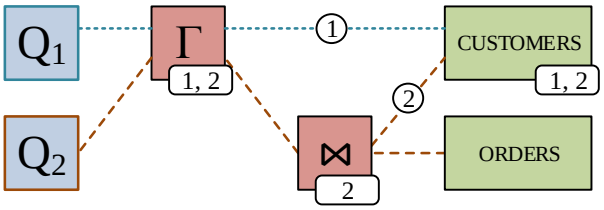
(a) Cycle in a Global Query Plan.



(b) Execution of Queries 1 and 2. `GroupBy` should process any results with `query_id` $\in \{1, 2\}$



(c) Query 2 pushed to the `Join` operator.



(d) Queries 1 and 2 are pushed to the `CUSTOMERS` operator.

TID	Name	City	...	Query_Ids
1	John Smith	Zurich	...	{1}
2	Kate Johnson	Lausanne	...	{1, 2}
3	George Dallas	Munich	...	{2}
4	Jim Simpson	Bern	...	{1, 2}
Result Done				{1}
Result Done				{2}

(e) Set of Results generated by `CUSTOMERS` that flow to `GroupBy` and `Join`

Figure 2.11: Example of a Problematic Cycle in a Global Query Plan

not created from a single query, but requires a combination of both Q_1 and Q_2. If these queries are never executed concurrently, there is no issue.

However, if they are executed concurrently, synchronization mismatch renders wrong results. Figure 2.11b shows the first part of evaluation, where both queries are enqueued in the `GroupBy` operator. The `GroupBy` operator inserts into the query map the two `query_ids`, 1 and 2, and then begins forwarding queries to producers. First query 2 is forwarded to the `Join` operator as shown in Figure 2.11c. Then query 1 is forwarded to the `CUSTOMERS`

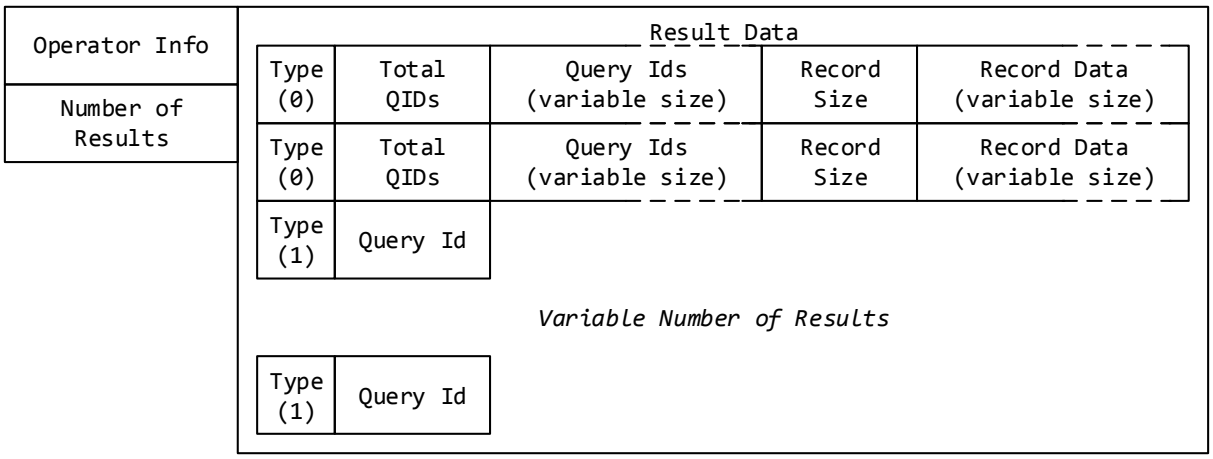


Figure 2.12: Result Messages in SharedDB

operator. If the `Join` operator was idle, then at the same time query 2 will be forwarded from the `Join` operator to the `CUSTOMERS` operator (Figure 2.11d). After this sequence of events, the `CUSTOMERS` operator begins a cycle where it has to evaluate queries 1 and 2 concurrently. As far as the `CUSTOMERS` operator is concerned, the consumers of the generated results are both the `Join` and the `GroupBy` operators. Figure 2.11e shows a sample result set of the `CUSTOMERS` operator. This result set is pushed to both consumers. The outcome is that the final result of query 2 will be incorrect, as i.e., tuple with `TID` 3 will be grouped by twice: once through the `Join` operator, and once directly from the `CUSTOMERS` operator.

In order to allow Algorithm 3 to support cycles in the global query plan, we have to extend the logic to use origin information. That means that each operator should be aware that it should process a set of `query_id`s, if they come from the same operator where the respective query was enqueued. In the example of Figure 2.11, this means that the `GroupBy` operator should process results with `query_id` = 1, if they originate from the `CUSTOMERS` operator only, and results with `query_id` = 2, if they originate from the `Join` operator.

The first modification involves the result queue. Figure 2.12 shows the structure of results that are communicated between operators. The operator information (an identifier that uniquely identifies an operator) has been added. The data structure is communicated via the result queues across all operators. Any operator receiving a result is now able to distinguish the producer of a set of result data.

Additionally, the algorithm needs to be modified to support this information. Algorithm 4 shows how origin information can be taken into account to avoid erroneous results. In fact, distinguishing the producer of each result tuple further improves the performance of result membership via state. The reason is that we do not have to query the top level map

Algorithm 4: Result Membership with Cycles in Global Query Plan

```
/* A map that maintains operator_id to Query mappings for all operators. */
Data: Map[OperatorId → Map[Query_Id → Query] qid_map;
/* A temporary pointer to a map that maintains query_id to Query mappings for
   the operator that send the most recent result. */
Data: Map*[Query_Id → Query] result_map;

/* Called for each incoming query. */
Function HandleIncomingQuery(q):
    Input: Query q
    /* Remember that the operator with id = q.operator_id should be the producer
       for all results with query_id = q.query_id. */
    qid_map.Map (q.operator_id → (q.query_id → q));
end

/* Called for each incoming result data structure (Figure 2.12). */
Function HandleIncomingResult(r):
    Input: ResultData r
    /* Modify the result_map to point to the map of this producer */
    result_map ⟵ qid_map.Get (result.operator_id);
end

/* Checks if a query_id should be handled. This function is called for every
   incoming tuple in order to decide if the tuple should be processed by this
   operator. Uses the helper result_map to adjust to the specific producer. */
Function IsMyQueryId(qid):
    Input: Query_Id qid
    return result_map.contains (qid);
end

/* Called at the end of each cycle. */
Function EndOfCycle():
    qid_map.Reset();
    result_map ⟵ ∅;
end
```

(*qid_map*) for every result tuple. This map is queries once for every set of results. Instead, a smaller map (*result_map*) is queries for each tuple, thus reducing the cache usage in the innermost loop.

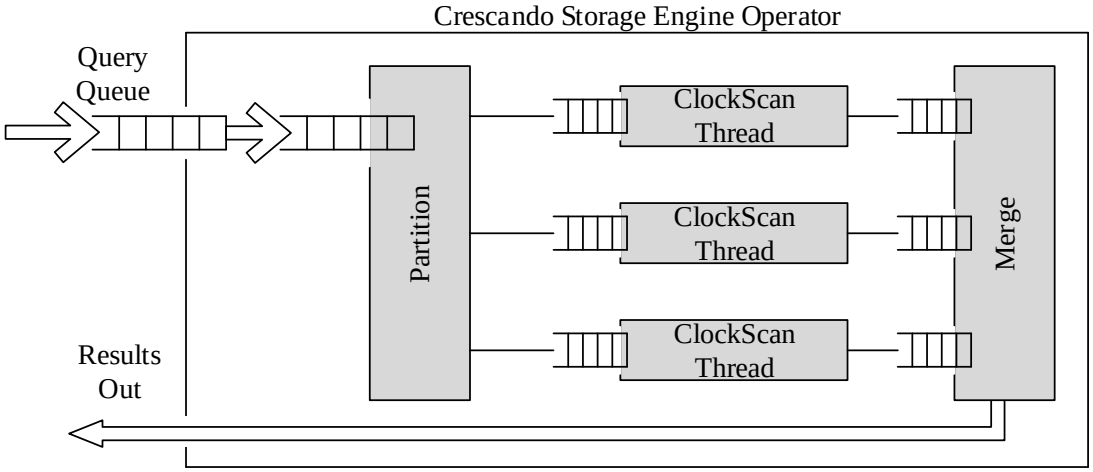


Figure 2.13: Overview of the Crescando Storage Engine Operator
Grey Components belong to Crescando's Implementation

2.3.4 Storage Engine

SharedDB implements two different storage engine operators. A built-from-scratch multi key value store, and Crescando storage manager. The two storage engines have very different characteristics in terms of performance, but they both use the same abstract operator model that was presented in Section 2.3.1. This shows the flexibility and extensibility of SharedDB: as long as the two-queue interface is used, the implemented logic can be very different. As far as the other operators are concerned, an operator is a black box that consumes queries and produces results. Finally, each storage engine operator implements a single relation. To support multiple relations, SharedDB uses different instances of these operators, one for each relation.

2.3.4.1 Crescando Storage Engine Operator

Crescando [UGA+09] implements a push-based relational table that is robust to mixed (read-write), unpredictable, evolving workloads. It scales linearly on modern multi-core hardware, guarantees snapshot isolation and write monotonicity, and most importantly is able to provide predictable performance. In this dissertation we only describe the properties of Crescando and how SharedDB uses it. An extended analysis of Crescando as well as implementation details can be found in [Unt12]. The Crescando storage manager is not a purely academic exercise. It has been successfully deployed to serve demanding update-intensive operational business intelligence workloads of the travel industry. It is currently in production at Amadeus, the market leader for airline reservation.

Algorithm 5: Wrapping Crescando in a SharedDB Operator

```
Data: Crescando csd;                                   /* instance of Crescando */
Data: SyncedQueue iqq;                                 /* incoming queries queue */
while true do
    Array aq ← ∅ ;                                     /* Array of active queries */
    /* Operators that will receive the produced results                          */
    Array consumers ← ∅ ;
    /* Activate all queries in the incoming queue                                */
    while ¬IsEmpty(iqq) do Put(aq, Get(iqq)) ;
    /* Beginning of cycle. Push queries to Crescando.                            */
    foreach Query q ∈ ag do
        CrescandoEnqueueQuery(csd, q);
        Put(consumers, GetConsumer(q));
    end foreach
    /* Pull results from Crescando and push them to consumers.                   */
    Number queriesLeft ← Size(aq);
    while queriesLeft do
        /* Get results from Crescando and forward them to the consumers.         */
        CrescandoResult csd_result ← CrescandoGetResult(csd);
        SendResult(consumers, ConvertResult(csd_result));
        if IsEndOfStream(csd_result) then
            queriesLeft ← queriesLeft − 1;
        end if
    end while
    /* Notify the consumers that processing has finished                         */
    SendEndOfStream(consumers); /* End of cycle.                                 */
end while
```

Figure 2.13 shows the Crescando Storage Engine operator of SharedDB. The algorithm of this operator is sketched in Algorithm 5. In Crescando data is accessed only via carefully optimized full table scans. The lack of indexes allows Crescando to handle write-intensive workloads gracefully by applying any updates in place, as there is no need to maintain any secondary structures. Table scans are carried out using a novel algorithm, namely the *ClockScan* algorithm, described in detail in [UGA+09]. ClockScan batches queries and updates in the same way as SharedDB and processes a whole batch of queries and updates. As a result, the ClockScan algorithm fits nicely into the overall SharedDB design.

Crescando uses main memory to store data in a row-wise fashion. Since a full table scan of row wise data is trivial to parallelize, Crescando is able to use multiple instances of ClockScan to increase the throughput and decrease the latency of the system. Each partition, called *Segment* in Crescando's terminology, is executed on a different hardware context and handles a horizontal partition of the data. Different partitioning schemes are supported, like round-robin, hashing on an attribute partitioning and range partitioning. Regarding durability, Crescando supports full recovery by lazy checkpointing and physical logging writes to disk. The overhead of recovery is minimal.

To further increase performance, ClockScan indexes the query predicates instead of the data and performs query-data joins, a technique widely used in publish-subscribe systems [FJL+01, LJ03] and data-stream processing [CF02, WCY04]. Updates are executed in arrival order as part of the same scan that executes the queries.

With regard to isolation, Crescando supports natively Snapshot Isolation if only one ClockScan thread is used. With multiple ClockScan threads, Crescando guarantees session consistency (read your own writes) and write monotonicity. An extension of Crescando that is supported in SharedDB allows reads to be treated as writes that do not modify the dataset, but return all matched tuples. This ensures Snapshot Isolation even with more than one thread, however this comes at a much higher cost due to the specifics of the ClockScan algorithm. Snapshot Isolation complements nicely the batch-oriented shared query processing model of SharedDB. The design of the system favors optimistic and multi-version concurrency control because any kind of locking would result in unpredictable response times due to lock contention and blocking. Further details about transactional properties of SharedDB are discussed in Chapter 4.

The Crescando storage engine operator falls into the category of specialized operators in SharedDB. It uses more than one hardware contexts and as a result has multiple NUMA allocated memory regions. These memory regions are used to store the whole relation. Also there is no incoming result queue in this operator, as this is a producing operator.

2.3.4.2 Key-Value Storage Engine Operator

While Crescando is a scalable and predictable storage engine, it is not designed to serve transactional workloads that require sub millisecond response times. In order to support such fast queries, SharedDB implements a second storage engine operator that supports indexes. The key-value storage engine operator is not as predictable as Crescando is. Certain update sequences might force a reorganization of the indexes, and as a result cause hiccups to the response time of the system. Yet, it is able to answer batches of point or range queries while providing good response time most of the time.

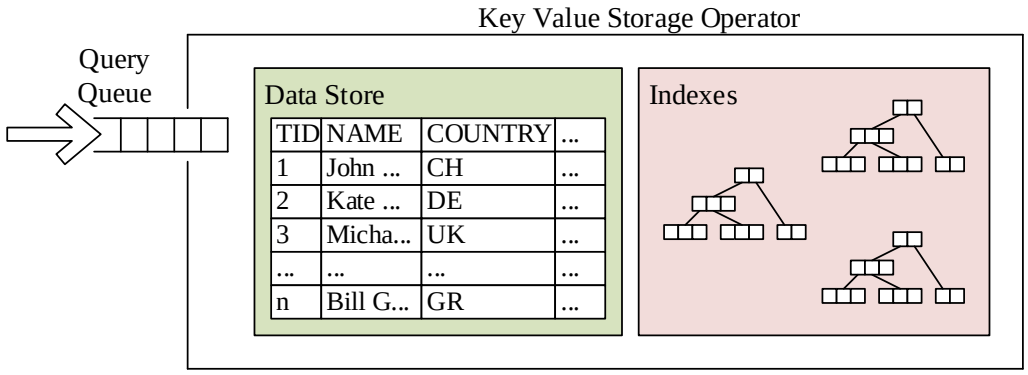


Figure 2.14: Overview of the Key-Value Storage Engine Operator

Figure 2.14 shows a high level overview of the key-value storage operator. It consists of a datastore and a number of $B+$ Tree structures. The datastore is implemented as a dynamic array of tuples that are accessible via a tuple identifier.

The operator uses a single thread that handles all incoming queries and updates, and processes them in batches. Having a single thread removes any necessity for synchronization mechanisms, like locks and latches, on the data. The logic of the key-value storage probe operator is similar to Algorithm 2.3.1. Queries are enqueued in the pending query queue which is emptied at the beginning of each cycle. During the cycle, the updates are executed in the arrival order and multiple B-Tree look-ups are used to evaluate all the select queries. Executing multiple look-ups in one cycle allows for better instruction and data cache locality [FK05]. Furthermore it allows for result sharing, in case there is any overlap across the queries of the current batch. Just as Crescando, the key-value operator guarantees that all select queries will read a consistent snapshot.

2.3.4.3 Replication

The design of our system allows replication in SharedDB. Replicating a storage operator does not hurt consistency, because updates are always executed in the same order as they were received by the system. However, replication has an impact on Snapshot Isolation, as reads are interleaved with updates and their execution order is not guaranteed.

In Chapter 4 shared transactions are introduced and an improved version of these operators is presented that supports multiple replicas of storage engines without sacrificing Snapshot Isolation.

Moreover, data processing operators can be also replicated in order to reduce the effects of bottlenecks and hotspots in the data flow network. For instance, if a specific operator

becomes a bottleneck, SharedDB can partition the load across two replicas of the same physical operators.

2.3.5 Data Processing Operators

In this Section we discuss the specifics of the most common data processing operators in SharedDB. All operators follow the abstract operator model that was presented in Section 2.3.1 and process queries and results in batches.

Operators can be classified in two categories: blocking operators and non-blocking operators. Blocking operators require that the full set of input result tuples has been processed before they start producing tuples. The typical example of blocking operators is `OrderBy` and `Aggregate` operators. Non blocking operators are able to produce results while still incoming results are arriving. This is the case for filtering operators.

2.3.5.1 Filters

Filtering operators belong to the non-blocking class of operators. Their purpose is to select a set of tuples based on a set of query specific predicates. In general in Shared selection is pushed as low as possible and is implemented inside the storage engines. However there are certain cases where filtering operators are useful. For instance, post filtering the results of a complex operator requires a filtering operator. This is the case for queries like this:

```
SELECT CUSTOMERS.COUNTRY, SUM(ORDERS.AMOUNT) AS TOTAL_AMOUNT
FROM CUSTOMERS JOIN ORDERS ON CUSTOMERS.CUSTOMER_ID = ORDERS.CUSTOMER_ID
WHERE TOTAL_AMOUNT > ?
GROUP BY CUSTOMERS.COUNTRY;
```

Another example of filtering operators is `LIMIT` operators that output only a subset of incoming result set.

Implementing filter operators in SharedDB is straightforward. The main algorithm is based on the abstract operator model (Algorithm 1) where `ProcessTuple` implements the query specific selection. Algorithm 6 shows an example of a `LIMIT` operator.

Furthermore, filter operators can be used to evaluate complex selection predicates that should not be part of storage engines, as this would introduce a big overhead for all queries. Regular expression queries, sql `LIKE` predicates, as well as even prefix and postfix queries are a few examples. For this king of selection criteria, SharedDB can use special operators that take advantage of query batching in order to reduce evaluation time. For instance, a SharedDB filtering operator for prefix predicates is able to create a trie of all

Algorithm 6: Implementation of the `LIMIT` operator's logic in SharedDB

```
Data: Result_Map map;                    /* Result's query_id map (Algorithm 4). */
/* Called for every incoming tuple from Algorithm 1.                            */
Function ProcessTuple(t):
    Input: Tuple t
    Query_Ids output_qids;               /* The query_ids of the output tuple */
    /* Iterate over all query_ids in this tuple.                                */
    for i = 1 to t.no_query_ids do
        /* Check if this query_id should be handled by this operator.           */
        if IsMyQueryId(t.query_ids[i]) then
            Query q ⟵ map.Get(qid);
            if ¬q.LimitReached() then
                output_qids.Add(t.query_ids[i]);
                q.LimitSubstract(1);
            end if
        end if
    end for
    if output_qids.Size()> 0 then
        /* At least one query accepts this tuple                                */
        t.query_ids ⟵ output_qids;
        return t
    else
        return ∅
    end if
end
```

the predicates. The trie is created when the cycle starts and contains the lookup prefixes of all active queries. Upon receiving a tuple, the operator can use a technique called query-data join to lookup the tuple on the trie[CF02]. This means that the complexity of evaluating n queries is no longer $O(n)$, but $O(log(n))$.

2.3.5.2 Projection

In general, projection operators are not useful in the context of shared execution. Projecting certain attributes out of a tuple reduces the possibilities of sharing. For instance, consider the following two queries:

```
Q1  SELECT CUSTOMERS.NAME, ORDERS.AMOUNT
    FROM CUSTOMERS JOIN ORDERS
         ON CUSTOMERS.CUSTOMER_ID = ORDERS.CUSTOMER_ID
    WHERE CUSTOMERS.COUNTRY = ?;
Q2  SELECT CUSTOMERS.COUNTRY, ORDERS.DATE
    FROM CUSTOMERS JOIN ORDERS
         ON CUSTOMERS.CUSTOMER_ID = ORDERS.CUSTOMER_ID
    WHERE ORDERS.SHIPPED = ?;
```

A traditional query-at-a-time system would apply the projection as early as possible. In most systems, this is before executing the join of the two relations. However by doing so, the two queries can no longer share the join.

One may argue that the join may still be shared, as long as we project the superset of all projected attributes. For this example, this means the projection of `(CUSTOMER_ID, NAME, COUNTRY)` from `CUSTOMERS` and `(CUSTOMER_ID, AMOUNT, DATE)` from `ORDERS`. Even though this is correct, it heavily increases the complexity of the system. The problem of detecting the superset of projected attributes in queries is similar to the problem detecting common subexpressions across queries. Last but not least, SharedDB is designed to handle a big amount of queries. Based on the law of big numbers, when SharedDB handles thousands of queries concurrently, the possibility that some attribute is not projected is tiny. As a result, SharedDB does not project attributes during query evaluation.

Nevertheless, SharedDB implements a projection operator that can be used on the top level of query evaluation. That is before delivering the result tuples to the clients. Implementing such an operator is trivial and evolves extending the abstract Operator Model algorithm with an implementation of `ProcessTuple` that outputs a projected tuple.

2.3.5.3 Sort

SharedDB implements a blocking sort operator to handle `ORDER BY` clauses. The sort operator buffers all incoming tuples in the private working set memory and at the end sorts the buffer, as shown in Algorithm 7.

Sorting in a shared data processing environment can be problematic, as performance is not always better compared to non-shared data processing. For instance, consider the example of Figure 2.15. In this example three queries require to sort a subset of `CUSTOMERS`. If Q_1 is executed concurrently with Q_3, then there is no overlap of the two intermediary result sets. If on average each queries fetch t tuples, the sort buffer will contain $2 * t$ tuples and the complexity of sorting the buffer is $O(2t\texttt{log}(2t))$. If these two queries were evaluated

Algorithm 7: Implementation of the `ORDER BY` operator's logic in SharedDB

```
Data: Buffer sort_buffer;                          /* The sort buffer */
Data: Array consumers;                  /* The consumers of this operator */

/* Called for every incoming tuple from Algorithm 1.                     */
Function ProcessTuple(t):
    Input: Tuple t
    if IsMyTuple(t) then
        sort_buffer.Append(t);
    end if
end
/* Called before the end of the cycle in Algorithm 1.                    */
Function SendEndOfStream():
    sort_buffer.Sort();
    PushBuffer(consumers, sort_buffer);
end
```

in a query-at-a-time system, the complexity of the sort operator is $2 * O(t\log(t))$ which is faster. However, if there is some overlap across queries then a shared sort is faster. For instance, if Q_1 is executed concurrently with Q_2, then the sort buffer will have t tuples. The complexity of sorting the buffer will be only $O(t\log(t))$.

However, if the number of concurrent queries is high, the probability of any overlap is also high. As a result sorting in SharedDB is quite efficient. Moreover, as with any other operator, the nature of SharedDB imposes an upper limit to the $O(2t\log(2t)$ complexity, as $2t$ cannot be bigger than the size of the relation itself. Thus, this means that once the superset of all the results sets is equivalent to the original relation, adding more queries to the system has minimal impact on the performance.

2.3.5.4 Joins

Join is notoriously expensive, which is why it has been extensively studied in literature [BTAO13, BLP11, CAGM07, KKL+09]. A join operator combines result sets from two different relations into a new intermediary relation. There are different implementations of joins in literature, but all of them fall in one of the four main categories: nested loop joins, index joins, sort merge joins and hash joins. Each of this class has a number of advantages and disadvantages, rendering it better for specific workloads.

Not all classes of joins can be implemented in SharedDB. For instance, index joins can

Q1: `SELECT * FROM CUSTOMERS WHERE COUNTRY = 'CH' ORDER BY NAME;`

Q2: `SELECT * FROM CUSTOMERS WHERE CITY = 'ZURICH' ORDER BY NAME;`

Q3: `SELECT * FROM CUSTOMERS WHERE COUNTRY = 'US' ORDER BY NAME;`

(a) Sample Queries

Id	Name	Country	City
1	Jens	CH	Zurich
2	Mathias	CH	Bern
3	John	US	New York
4	Fabian	CH	Biel
5	Bill	US	Boston
6	Michael	US	Seattle
7	Lukas	CH	Zurich
8	Jack	US	Chicago
9	Kate	US	Indianapolis
...	...	...	...
n	Dominik	CH	Zurich

(b) Sample `CUSTOMERS` Dataset

Query_Id
1, 2
1
3
1
3
3
1, 2
3
3
...
1, 2

(c) `Query_id` Lists

Figure 2.15: Example of Sorting in a Shared Data Processing Environment

not be implemented, as this would require exposing the index structure of an index-based storage engine. That would first of all break the operator model of SharedDB, but most importantly, it requires synchronization mechanisms as the index needs to be accessed by two threads concurrently: the join operator thread and the storage engine thread.

Sort-merge joins can be used in SharedDB, however as explained in the discussion regarding sort operators, they may have a behavior that does not favor SharedDB. Additionally, they require full sorting of the inner and outer relations. In query-at-a-time systems this may be efficient as usually both inner and outer relations are a selection on the original relations. In SharedDB, as multiple queries are executed, the superset of all inner and outer relations might be even the whole relations. Currently there is no implementation of a sort-merge join, yet this is not because the system design makes it impossible, rather than because a sort-merge join is not favored in any of the microbenchmarks conducted.

Hash join is the preferred join class in SharedDB as it shows a constant $O(1)$ behavior as the size of the result set increases. For SharedDB, a hash join operator is a special case of operator. The operator has two phases. During the first phase, results from the

Inner Tuple:

Id	Name	Country	City	Query_Ids
14	John	CH	Zurich	10, 54, 1, 5, 58, 69, 39

Outer Tuple:

CId	Date	Amount	Query_Ids
14	2014.06.20	3,403.2	83, 54, 12, 5, 39

Result Tuple:

Id	Name	Country	City	Date	Amount	Query_Ids
14	John	CH	Zurich	2014.06.20	3,403.2	54, 5, 39

Figure 2.16: Joining `query_id`s During the Materialization in a Hash Join

inner relation are hashed and stored in a table. This phase is a blocking phase, as no results can be produced. The second phase involves parsing result tuples from the outer relation and probing the hashtable. Any successful probe generates result tuples which are immediately pushed to the consumers. A naïve implementation of hash join uses a single dynamic hash table. The main algorithm of the operator is show in Algorithm 8. The `Materialize` function, shown in Algorithm 9 is responsible for executing the `query_id` join and in case a `query_id` is common on both the inner and the outer relation tuples, forward a joined tuple to the consumers.

Obviously the performance of the hash table is critical, as it is probed more frequently as the sizes of the relations get bigger. Also the `Materialize` function is critical, and most specifically the implementation of the `JoinQueryIds` function. This function takes the sets of `query_id`s of two tuples and outputs the common elements, as shown in Figure 2.16.

Obviously, not all join algorithms are appropriate for `JoinQueryIds`. A first approach was to use a hashset. The hashset is build from the inner tuples's `query_id`s and then probed with the `query_id`s from the outer tuple. This approach scales gracefully to the number of `query_id`s as it has a constant behavior, however it increases the amount of both cache and data misses. `JoinQueryIds` is called at the inner most loop, between probes of the hash table. Building a hashset at this point pollutes the cache, and makes both hash table probing and execution of `IsMyTuple` slow.

A more suitable join algorithm for the `JoinQueryIds` function is a sort merge join. In order to avoid cache misses, we avoid sorting `query_id`s inside `JoinQueryIds`. Instead we make sure that all tuples have their `query_id`s sorted a priori. This is extremely easy in SharedDB, as all queries are executed in the order they are activated. As long as the storage engines sort the active queries on the `query_id` attribute, then all tuples will be generated with the set of `query_id`s sorted. Finally, what `JoinQueryIds` has to do is a linear search for common elements in the `query_id` sets of both tuples. This technique is the fastest when lists of `query_id`s are used. In Section 2.3.6, a number of optimizations are presented, including a more efficient `JoinQueryIds` implementation that relies on bitsets

Algorithm 8: Implementation of the hash join operator's logic in SharedDB

```
Data: SyncedQueue iqq;                          /* incoming queries queue */
Data: SyncedQueue irq;                          /* incoming result tuples queue */
Data: HashTable htable;          /* A common hash table for all active queries */
while true do
    Array aq ← ∅ ;                                  /* Array of active queries */
    /* Activate all queries in the incoming queue                            */
    while ¬IsEmpty(iqq) do Put(aq, Get(iqq)) ;
    foreach Query q ∈ ag      /* Enqueue subqueries to underlying operators */ do
        Query subQuery ← GetInnerRelationSubQuery(q);
        EnqueueQuery(op ← GetOperator(subQuery), subQuery);
    end foreach
    /* Build phase: Loop until all active queries have finished.             */
    Number queriesLeft ← Size(aq);
    while queriesLeft do
        Tuple t ← Get(irq) /* Receive a tuple from the underlying operators */
        if IsMyTuple(t) then htable.Insert(t); ;
        if IsEndOfStream(t) then queriesLeft ← queriesLeft − 1; ;
    end while
    /* Probe phase.                                                          */
    foreach Query q ∈ ag      /* Enqueue subqueries to underlying operators */ do
        Query subQuery ← GetOuterRelationSubQuery(q);
        EnqueueQuery(op ← GetOperator(subQuery), subQuery);
    end foreach
    queriesLeft ← Size(aq);
    while queriesLeft do
        Tuple t ← Get(irq) /* Receive a tuple from the underlying operators */
        if IsMyTuple(t) ∧ ¬htable.Probe(t).IsNull() then
            Materialize(htable.Probe(t), t);
        end if
        if IsEndOfStream(t) then queriesLeft ← queriesLeft − 1; ;
    end while
    SendEndOfStream();
end while
```

Algorithm 9: Implementation of the `Materialize` tuples on a hash join

```
Function Materialize(innerTuple, outerTuple):
    Input: Tuple innerTuple ;                    /* The inner relation tuple. */
    Input: Tuple outerTuple ;                    /* The outer relation tuple. */
    QueryIds outqids ⟵ JoinQueryIds(innerTuple.query_ids,
    outerTuple.query_ids);
    if ¬outqids.IsEmpty then
        Tuple output;
        output.query_ids ⟵ outqids;
        output.record ⟵ MergeRecords(innerTuple.record, outerTuple.record);
        PushTuple(output);
    end if
end
```

of `query_id`s.

2.3.5.5 Group By, Aggregate

Grouping and aggregation are the only operations that do not benefit from execution in a shared environment. The reason is that each query has probably different predicates and as a result aggregation results will always be different.

Yet, it is possible to share the grouping phase of a `GROUP BY` operator, as long as all queries group on the same attribute. For instance consider the following two queries:

Q1:
```
SELECT COUNTRY, COUNT(NAME) FROM CUSTOMERS
WHERE BALANCE > ?  GROUP BY COUNTRY;
```

Q2:
```
SELECT COUNTRY, COUNT(NAME) FROM CUSTOMERS
WHERE LAST_LOGIN > ?  GROUP BY COUNTRY;
```

If we execute these two queries in SharedDB, there is no possibility to share the `COUNT` evaluation. The `COUNT` is the output of different results sets. The SharedDB Group By operator is still able to share grouping. The result tuples are partitioned on the given `GROUP BY` attribute. Once all tuples have been consumed, the algorithm evaluates the aggregation function on each of the partitions for each query. Even though SharedDB is not very efficient in grouping and aggregation, it cannot be worse than executing these queries separately. The aggregation function has to be applied on each query's result set in any case.

2.3.6 Optimizations

In this section, we discuss a number of optimization techniques that can be applied in the presented SharedDB design. The goal of these optimizations is to further increase the performance of SharedDB by employing smarter algorithms, higher cooperation with modern hardware, as well as techniques that have been proposed in research and can be applied in the context of SharedDB as well.

2.3.6.1 Operator Deployment

One of the design goals of SharedDB is to take advantage of modern multi core hardware architectures. In these systems, CPU cores that are located on the same chip, share components of the memory architecture, like the L3 cache and the NUMA main-memory region. SharedDB implements a data flow network, where each operator (node) is assigned to a number of hardware contexts (CPU cores). To increase memory bandwidth and reduce memory latency, SharedDB takes advantage of modern NUMA hardware architectures by allocating the working set of each operator in the local NUMA main-memory region.

Moreover, careful deployment of database operators across CPU cores can further exploit modern multi core hardware architectures. Database operators that access similar sets of records can be assigned to "adjacent" CPU cores in order to benefit from the sharing of these memory components. The problem of optimal operator deployment can be mapped to a max-flow graph problem with linear constraints. Similar to the deployment of database operators to CPU cores, replicating a specific operator is a task that needs to be performed by a deployment optimizer, based on the global query plan.

Our existing implementation supports custom placement of operators. Currently the assignment of operators to CPU cores has been performed manually by examining the data access paths of every operator. A possible extension of SharedDB can support a placement optimizer to automatically deploy operators to the proper CPU cores.

2.3.6.2 `Query_Id` Bitsets

The system design of SharedDB supports either lists of `query_id`s or bitsets. The described implementation of Section 2.3 utilizes arrays, as they are more compact when queries have higher selectivities. The alternative approach of bitsets has been extensively analyzed in [Mak12].

Using bitsets simplifies the result membership problem that was explained in Section 2.3.3. When using bitsets, checking whether a tuple should be processed by an operator requires

Algorithm 10: Using `Query_Id` Bitsets to Answer the Result Membership Problem. Extension of Algorithms 3 and 4.

```
/* A map that maintains operator_id to Bitset Masks for all operators.   */
Data: Map[OperatorId → Bitset] masks;
/* A pointer to the mask associated with the operator that send the latest result. */
Data: Bitset mask;

Function HandleIncomingQuery(q): /* Called for each incoming query.      */
    Input: Query q
    /* Remember that the operator with id = q.operator_id should be the producer
       for all results with query_id = q.query_id.                       */
    qid_map.Get (q.operator_id).SetBit(q.query_id);
end

Function HandleIncomingResult(r): /* Called for each incoming result.    */
    Input: ResultData r
    /* Modify the current mask to point to the mask of this producer     */
    mask ⟵ masks.Get (result.operator_id);
end

/* Checks if a tuple should be handled                                   */
Function IsMyTuple(t):
    Input: ResultTuple t
    return mask ∧ t.query_ids ≠ ∅;
end

/* Called at the end of each cycle.                                      */
Function EndOfCycle():
    masks.Reset();
end
```

less processing power and has smaller memory (and cache) impact. Algorithm 10 presents the implementation. Each operator maintains masks that, when applied to the bitsets carried by tuples, can answer the result membership problem.

Furthermore, the use of bitsets simplifies any other processing that deals with `query_ids`. For instance, the `JoinQueryIds` function of a hash join can be simplified. In fact there is no need to join `query_ids`, and instead a binary `AND` operation can be used on the tuples (Algorithm 11). This simplifies the inner most loop of the hash join. A careful analysis of the performance of set intersection using bitsets and lists has been conducted

Algorithm 11: Implementation of the `Materialize` tuples on a Hash Join with Bitsets. Extension of Algorithm 9.

```
Function Materialize(innerTuple, outerTuple):
    Input: Tuple innerTuple ;                    /* The inner relation tuple. */
    Input: Tuple outerTuple ;                    /* The outer relation tuple. */
    if innerTuple.query_ids ∧ outerTuple.query_ids then
        Tuple output;
        output.query_ids ⟵ (innerTuple.query_ids ∧ outerTuple.query_ids);
        output.record ⟵MergeRecords(innerTuple.record, outerTuple.record);
        PushTuple(output);
    end if
end
```

(Section 2.4.1).

One challenge when using bitsets is that their size can get extremely large as `query_id`s increase. For this reason, SharedDB has to re-use any `query_id`s that have been finished. This can be implemented very easily in the centralized Operator Manager that was introduced in Section 2.3. Section 2.4.1 presents an analysis of the memory footprint of bitsets versus arrays.

2.3.6.3 Parallel Operators

The whole design of SharedDB resembles a pipeline, where data flows through. As with every pipeline system, the slowest component dominates the performance of the system. In order to handle bottleneck operators, a design of multi-core operator has been introduced in Makreshanski's Master Thesis [Mak12]. This Master Thesis focuses mostly on exploiting multi-core architectures to evaluate joins in the context of SharedDB. Still the technique can be applied to other data processing operators as well.

Literature provides a lot of parallel operator implementations. The work of Pasetto et al. [PA11] presents a collection of various multi-core parallel sort implementations. Albutiu et al. [AKN12] introduces a very efficient way to execute massively parallel joins in main memory. All these algorithms can be implemented in SharedDB, under the model of a parallel operator. Crescando falls under the same class, being a parallel scan operator.

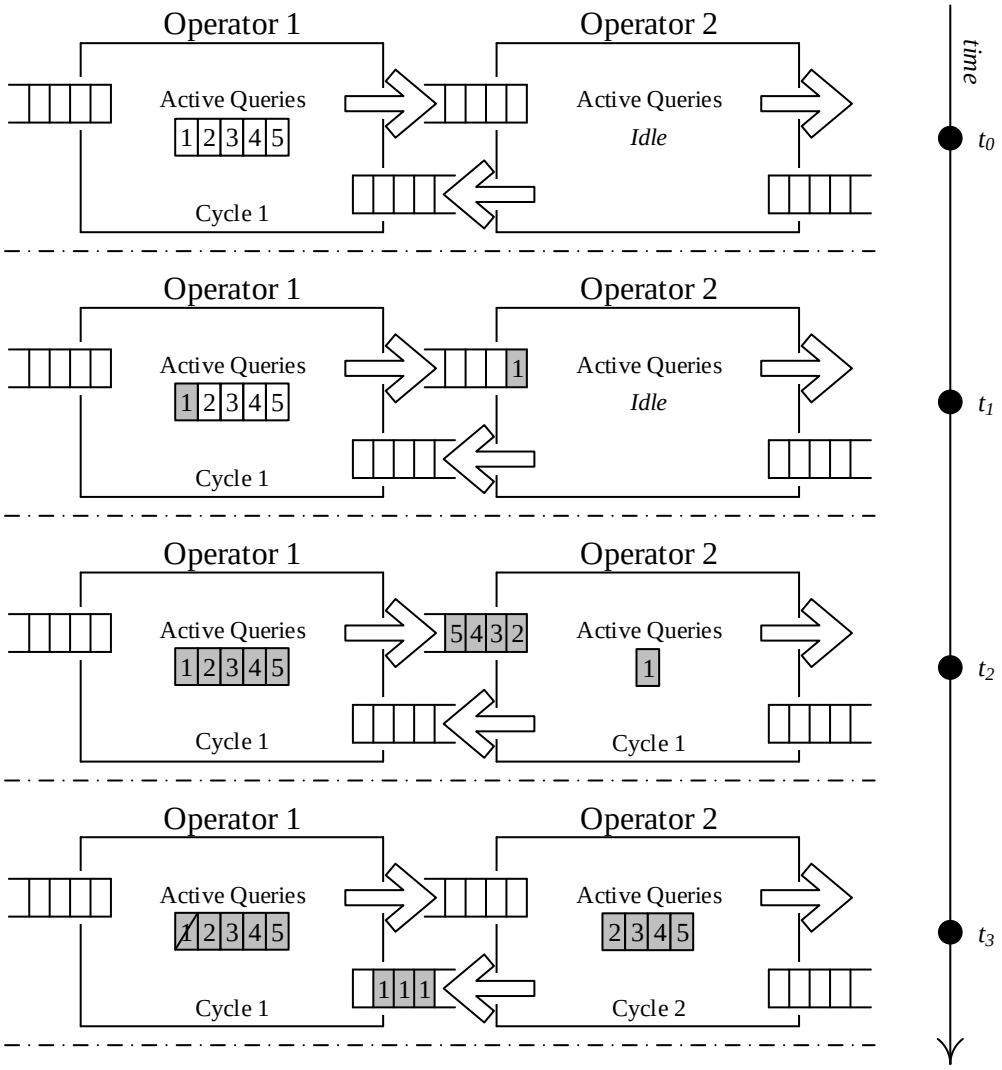
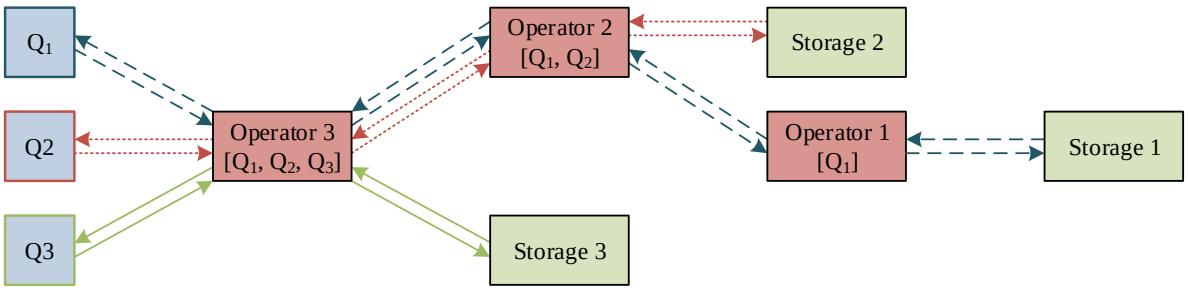


Figure 2.17: Example of Desynchronized Operators

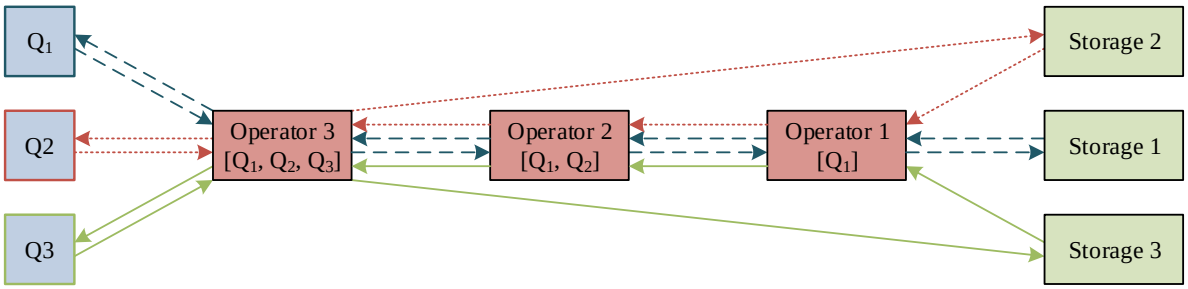
2.3.6.4 Fused Pipelined Operators

Another optimization that was considered deals with pairs of operators that are strongly connected in the data flow network. This means that the output of the first operator is always consumed by the second one. An example of such a use is a `TOP-N` queries that sort a relation and then filter (limit) the output. As long as no other operator is interested in the output of the sort operator, the limit and the sort operator are strongly connected.

Strongly connected operators can be *fused* together. That is, they can implement the logic of both operators in one context. Fusing operators can be beneficial in cases where the two



(a) Example of a Global Query Plan



(b) Example of a Global Query Plan with Shadow Queries

Figure 2.18: Example of using *Shadow* Queries in SharedDB

operators never evaluate tuples in parallel. In the previous example, the limit operator is idle while the sort operator accumulates tuples and sorts them. Then the limit operator starts processing the sorted output, while the sort operator remains idle.

Besides better CPU utilization, another advantage of fusing operators is synchronization. If operators are fused, the batches of queries are always in sync across operators. Consider for instance the example shown in Figure 2.17, which involves two chained operators. At t_0 the first operator has to push five queries to the second one, that happens to be idle (i.e. the cycle has finished and no queries were enqueued in the incoming query queue). At t_1 `Operator1` starts enqueueing the queries. Once the first query is enqueued, `Operator2` picks it up immediately and starts evaluating it. As a result, at t_2 `Query1` is active, while the remaining four queries have remained in the queue and will be executed in the next cycle of `Operator2`.

In Makreshanski's Master Thesis [Mak12], this idea has been extended to allow operators to be fused, even if they are not strongly connected. The concept involves introducing no-op queries, called *shadow* queries. The key idea is to change the request-response model of the message passing interface to a more sophisticated, where the response can follow a different path compared to the path followed by the request.

Figure 2.18a shows an example of a SharedDB global query plan without shadow queries.

In this case, results flow in the reverse direction of the requests. No operators are strongly connected, as all operators have two different producers. Introducing shadow queries modifies the global query plan to the one of Figure 2.18b. Shadow queries allow results to be routed through another path of operators. Any operator that receives shadow results forwards them to the consumers without processing them. This allows operators 1, 2 and 3 to be fused into a single big operator.

The approach of shadowing queries can be used only under specific global query plans. In Section 2.4, we present experimental evaluation of global query plans that use shadow queries and demonstrate the performance gains of this approach. In Chapter 4 introduces transactional operators that make heavy usage of shadow queries and results.

2.4 Experimental Evaluation

In order to evaluate the performance of SharedDB, we carried out a series of performance experiments. We present a series of microbenchmarks that were conducted in order to analyze the time critical parts of SharedDB, followed by a set of experiments on two well know benchmarks, namely the TPC-W and the TPC-H.

2.4.1 Microbenchmarks

As explained in Section 2.3, the `query_id` set implementation plays an important role in the performance of SharedDB. There are two different ways to implement `query_id` sets. The first one uses an array of `query_id`s, while the second one represents it using a bitset. The first SharedDB implementation used arrays as the required algorithms to maintain them were simpler.

Each implementation has different advantages and disadvantages. A bitset has fixed memory footprint that only depends on the number of queries. Each query has a unique index in the bitset. In SharedDB, we need as many bits as the number of queries in the whole system. Arrays on the other hand, have variable memory footprint that depends on how many queries are set. Figure 2.19 shows a comparison of memory footprint for the two implementations. The σ parameter of arrays is the selectivity. A σ value of $\frac{1}{1}$ means that the array contains all different `query_id`s that exist in the system. An important observation in this figure is that bitsets are more predictable. The memory footprint depends only on the overall workload when using bitsets. With arrays, the memory footprint depend also on the popularity of a tuple. A "hot" tuple will have a bigger impact compared to a "cold" tuple.

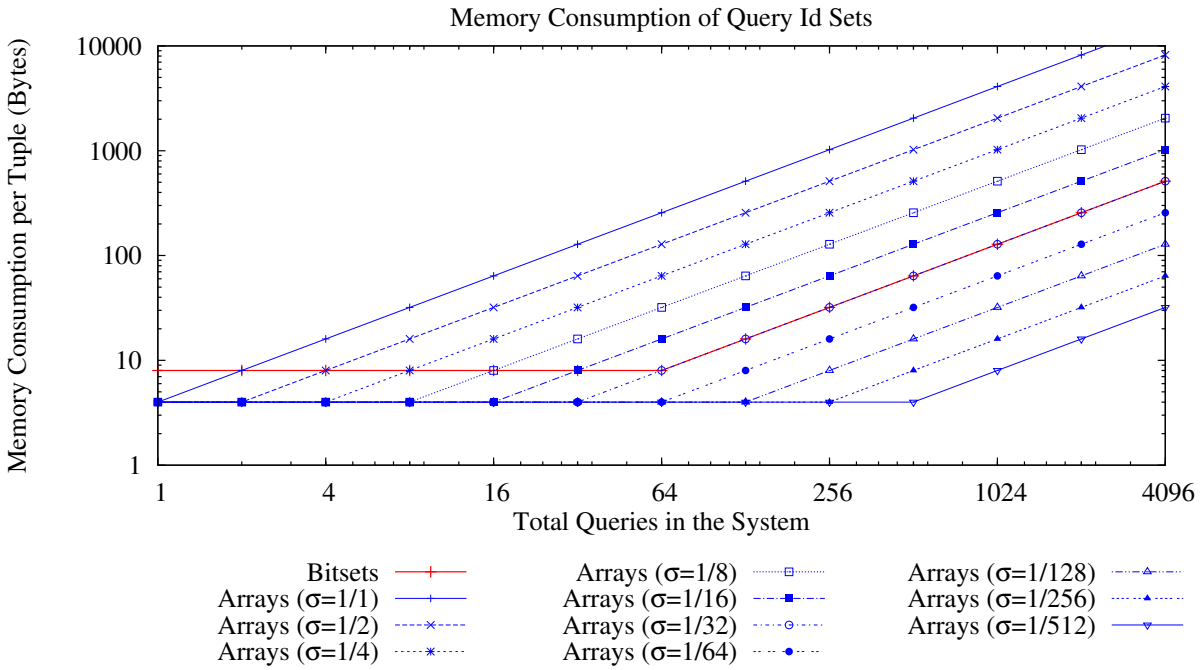


Figure 2.19: Memory Footprint of Different Implementations of `query_id` Sets

In order to evaluate the performance of the two implementations, we conducted a set of experiments that intersect randomized `query_id` sets. Figure 2.20 shows the performance in terms of throughput and latency.

The results show that the performance of bitsets is comparable to the array implementation. Bitsets are more robust, as their performance is independent of selectivity and most of the times, it is higher compared to arrays. A negative aspect is that `query_id`s can no longer be arbitrary numbers. `Query_id` recycling, as presented in Section 2.3.6.2 is necessary. Yet, recycling of `query_id`s adds only a small overhead per query.

2.4.2 TPC-W Evaluation

In this section, we present results of comprehensive evaluation using the TPC-W benchmark and the results of micro benchmarks with individual queries of the same workload, in order to demonstrate specific effects.

The TPC-W benchmark models an online bookstore. It assesses the performance of multi-tier information systems which contain a database layer, a web server layer and a client layer. Every client of the system is an emulated browser that issues http requests (web interactions in TPC-W terminology) to the web server layer. Between two web interac-

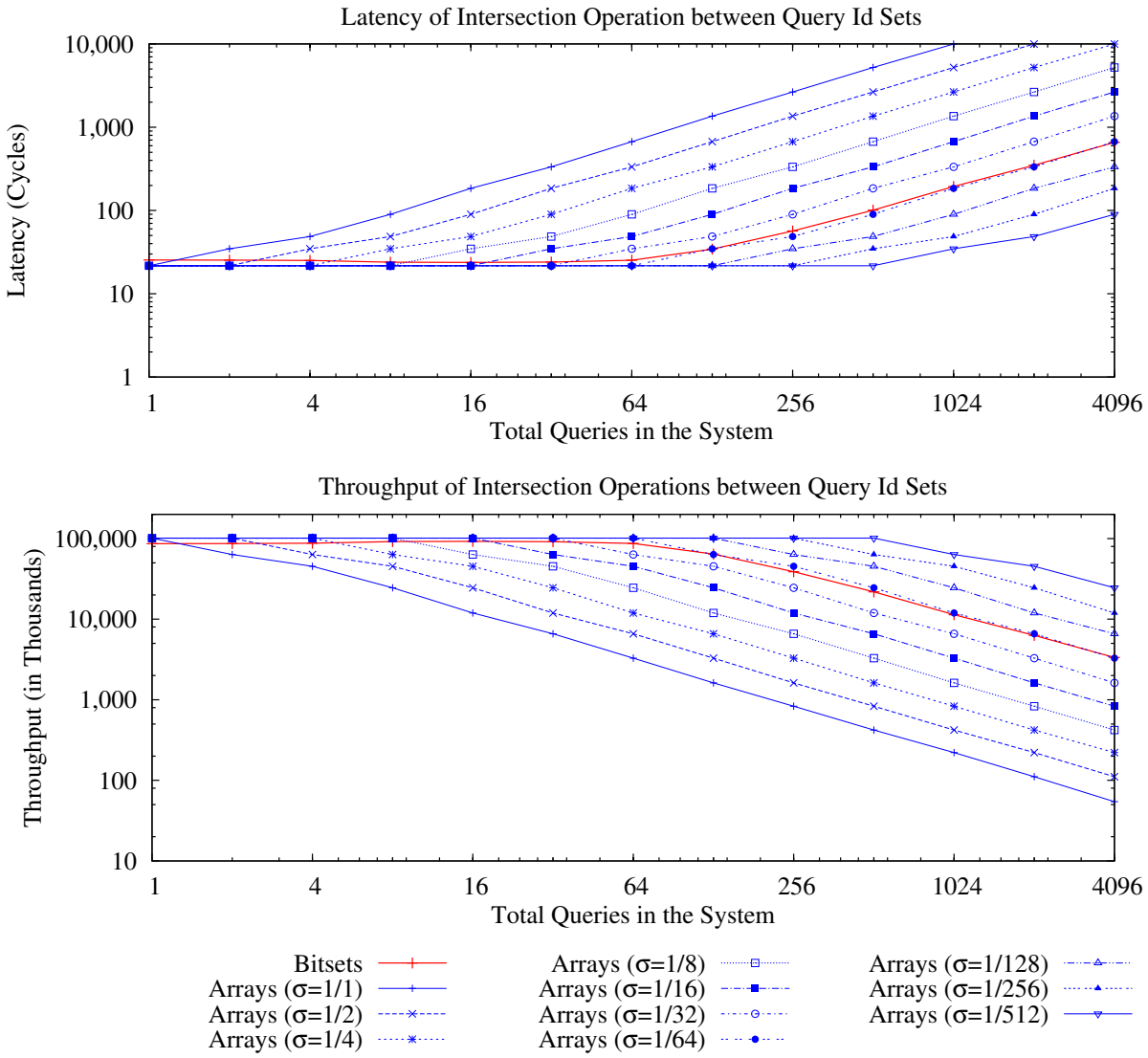


Figure 2.20: Performance of Intersecting `query_id` Sets for Bitsets and Arrays

tions, there is a "thinktime" which is defined by a negative exponential distribution and has an average of 7 seconds. The web servers accept the clients' requests and issue queries to the database layer in order to retrieve the requested data. Each client interaction is translated to a number of database queries, depending on the type of the interaction. For instance, to execute the "Home" web interaction (i.e., a user visits the application's home page), two queries have to be evaluated: The first query fetches a set of promotion items, and the second query retrieves the profile of the user.

In order to be compliant with the TPC-W specification, every web interaction needs to be answered in a predefined amount of time. The timeout depends on the type of the

web interaction, ranging from 2 seconds for small, point queries, up to 20 seconds for long running analytical queries. Any web interaction that exceeds this timeout, is not valid.

TPC-W contains a total of 14 different web interactions. Every web interaction has a different probability of appearing in the workload. The probabilities of all the web interactions are given by the "workload mix". TPC-W defines three different workload mixes: Browsing, Shopping, and Ordering. The Browsing mix is a read-mostly, search intensive workload with few updates and many analytical queries. The Ordering mix is a write-intensive workload with only a few analytical queries. The Shopping mix is somewhere in between with some updates and some analytical queries.

In all experiments reported in this section, we used a 48 core machine as a database server. This machine features four twelve-core AMD Opteron 6174 ("Magny-Cours") sockets and is equipped with 128 GB of DDR3 1333 RAM. Each core has a 2.2 GHz clock frequency, 128 KB L1 cache, 512 KB L2 cache, and is connected to a shared 12 MB L3 cache. The operating system used in all experiments was a 64-bit SMP Linux. The emulated browsers were executed on up to eight client machines, each having 16 CPU cores and 24 GB of DDR3 1066 RAM. The clients also ran the application logic; that is, the clients issued queries directly to the database server. This is a slight simplification of the TPC-W set-up and justified because we were interested in the performance of the database system under high load. The client machines were connected to the database server machine using a 1 Gbps ethernet.

We implemented the full TPC-W workload in SharedDB. Figure 2.21 shows the global query plan that was used. It consists of 26 database operators in addition to shared scans and index probe operators to access the nine base tables of the TPC-W benchmark. The current version of SharedDB does not feature any parallel join or sort algorithms and does not support partitioning or replication of base data. As a result, we used at most 32 cores for SharedDB, one core for each operator and nine cores for the shared table scans. We have *fused* all filter operators as explained in Section 2.3.6.4, and as a result they do not require a separate CPU core. The same applies to the `Distinct*` operator which is evaluated as part of the underlying `Hash`$\bowtie$ operator.) SharedDB is able to make use of the additional CPU cores by replicating operators, as explained in Section 2.3.4.3. However, we avoided using replication of data and operators in all our experiments as the goal is to study the benefits of sharing work rather than the benefits of replication. In order to demonstrate the scalability of SharedDB, we varied the number of cores between 1 and 32 in some experiments. We used the kernel parameter `maxcpus` to limit the number of available CPU cores. If not stated otherwise, 24 cores were used.

In all experiments reported, SharedDB held all the data in main memory. Disk I/O was

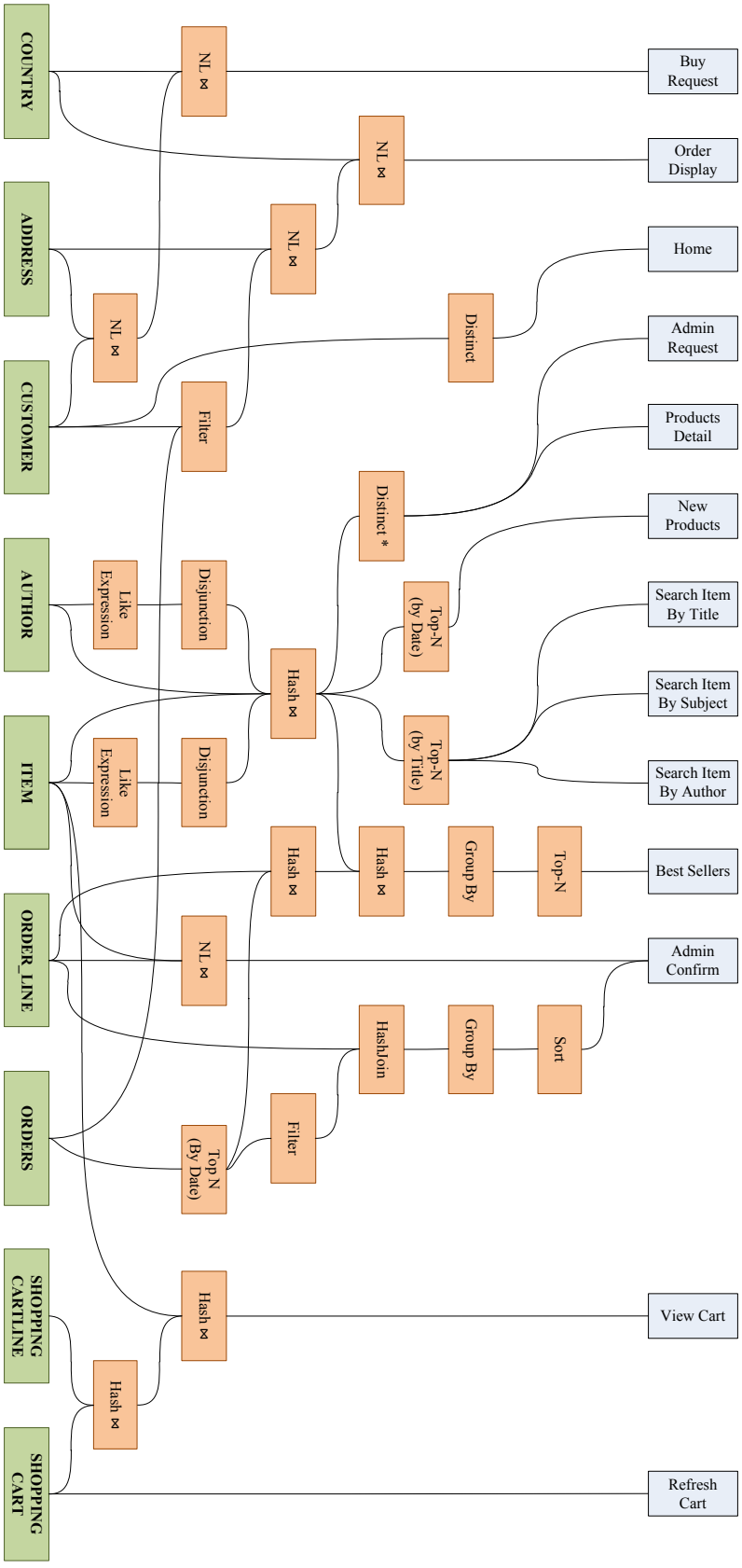


Figure 2.21: Global Plan for the TPC-W Benchmark. Updates have been omitted for simplicity.

only required to log updates as part of the Crescando storage manager (Section 2.3.4.1). As a result, running the TPC-W benchmark on SharedDB was CPU-bound. It was also CPU-bound for the two baseline systems that we describe in the next section.

2.4.2.1 Baselines

To put the performance of SharedDB into perspective, we compared it against two existing database systems. The first one is a popular commercial system that will be referred to as SystemX. The second one is a widely used database system, MySQL 5.1 using the InnoDB storage engine. Just like SharedDB, MySQL and SystemX are general-purpose database systems, designed to perform well for any kind of workload. Other existing solutions may reach better performance for specific workloads; e.g., column stores for OLAP workloads and lock-free, single-threaded systems for OLTP workloads. We do not compare SharedDB to these systems because we wanted to understand specifically the performance tradeoffs of SharedDB's shared execution model as opposed to the traditional, query-at-a-time model.

Clearly, the comparison of SharedDB with MySQL and SystemX is not an apples-to-apples comparison, but we tried to make the comparison as fair as possible by fine-tuning the two baseline systems. We built all the necessary indexes on the two systems and we used an in-main-memory filesystem for the data files of both systems. Furthermore, we provided a big memory buffer pool for MySQL and SystemX, big enough to hold the database and all indexes. Finally, we filled this buffer pool by carrying out full table scans and a warm-up phase of the benchmark. As a result, neither MySQL nor SystemX performed any disk I/O to carry out queries. Disk I/O was only performed by SystemX and SharedDB as part of logging in order to persist updates. For MySQL, all recovery options were turned off so that MySQL did not even perform disk I/O for logging. Overall, however, the workload was CPU-bound for all three systems. The isolation level used was "read committed", as TPC-W requires only session consistency.

We disabled query and result caching in both MySQL and SystemX. Query caching allows a database system to skip execution of a query as long as it has been executed before and no change in the dataset has occurred. Result caching performs the same optimization on each individual subquery. By disabling any caching, we guarantee that we measure the performance of evaluating every query, rather than fetching pre-calculated results from a cache.

In summary, we did everything to make a fair comparison between the three systems. Overall, however, our goal is not to show that SharedDB is better than MySQL, SystemX, or any other existing database system. Our goal was to show that batch-oriented sharing can result in predictable performance under high load. To this end, we also present the

results of micro-benchmarks that study the behavior of all systems under growing load and isolates the effects of "query-at-a-time" vs. "shared" query processing.

Unfortunately, we could not carry out performance experiments on QPipe, CJoin, or DataPath, the closest competitors of SharedDB in terms of shared computation. These systems are research prototypes and were not available for experimentation. Furthermore, we believe that significant research is necessary before the special, continuous (as opposed to batch-oriented) sharing featured by these systems can be applied to comprehensive workloads such as the TPC-W benchmark. So far, these systems have only been applied to OLAP queries of the TPC-H and SSB benchmarks.

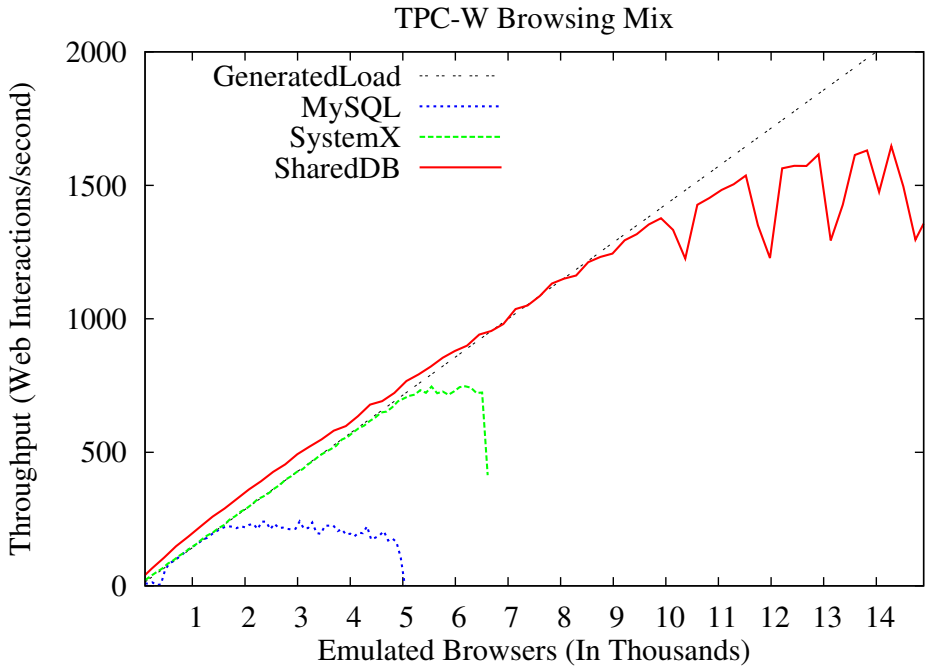
2.4.2.2 Performance under Varying Load

In the first set of experiments, we compared the performance of SharedDB, MySQL and SystemX on the three workload mixes of TPC-W. We varied the load of the system by increasing the number of emulated browsers and measured the web interactions that were successfully answered by the system in the response time limit that is defined by the TPC-W specification. All web interactions that exceeded this limit were not accounted as successful. For this experiment, we configured the database system to use 24 CPU cores.

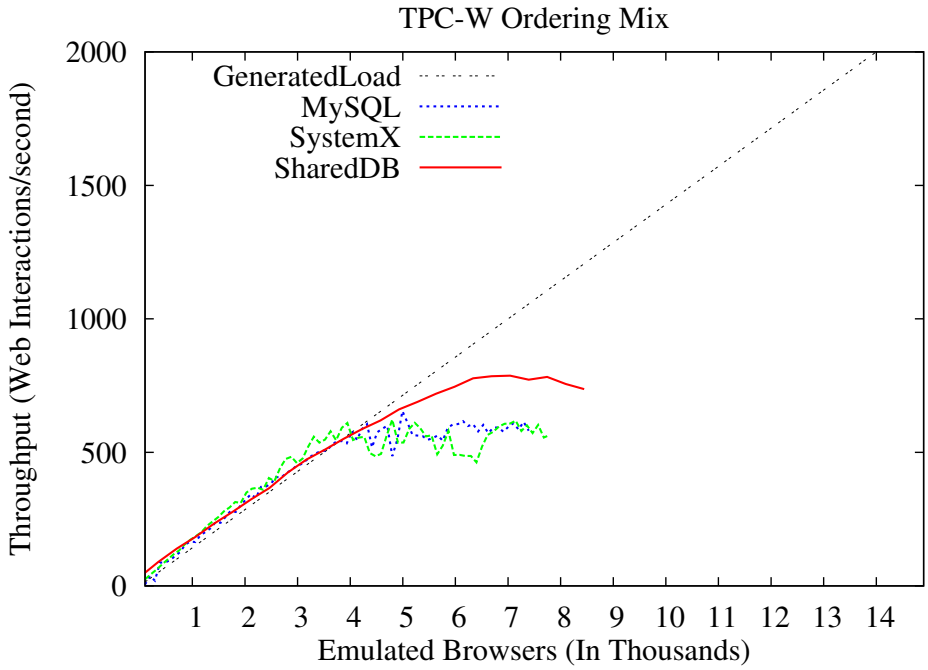
Figure 2.22 shows the results. SharedDB is able to achieve higher throughput in all three TPC-W workload mixes. For instance, in the Browsing mix, we see that SharedDB is able to sustain twice the throughput of SystemX and eight times the throughput of MySQL. The Browsing mix (Figure 2.22a) involves use cases of customers searching for items. Most of the search queries are heavy queries that involve a number of joins and sorts. This result confirms previous results with MQO on TPC-H queries (e.g., [HSA05]) that shared computation is beneficial if heavy queries need to be executed.

The Ordering mix (Figure 2.22b) involves few searches for items, and instead places many new orders. For this mix, MySQL is able to execute as many web interactions as SystemX. Again, MySQL had a bit of an unfair advantage as compared to SystemX and SharedDB here because all recovery options were turned off for MySQL. SharedDB still wins in this experiment, but the margins are lower. In the Ordering mix, most queries are point queries that can be executed highly efficiently with an index look-up in a traditional, query-at-a-time fashion. Furthermore, there is a little benefit for sharing for such point queries.

Finally, Figure 2.22c shows the throughput results with varying load of the three systems for the Shopping mix. Again, SharedDB is the clear winner and again the reason is that SharedDB takes advantage of shared computation for heavy and medium-sized queries.

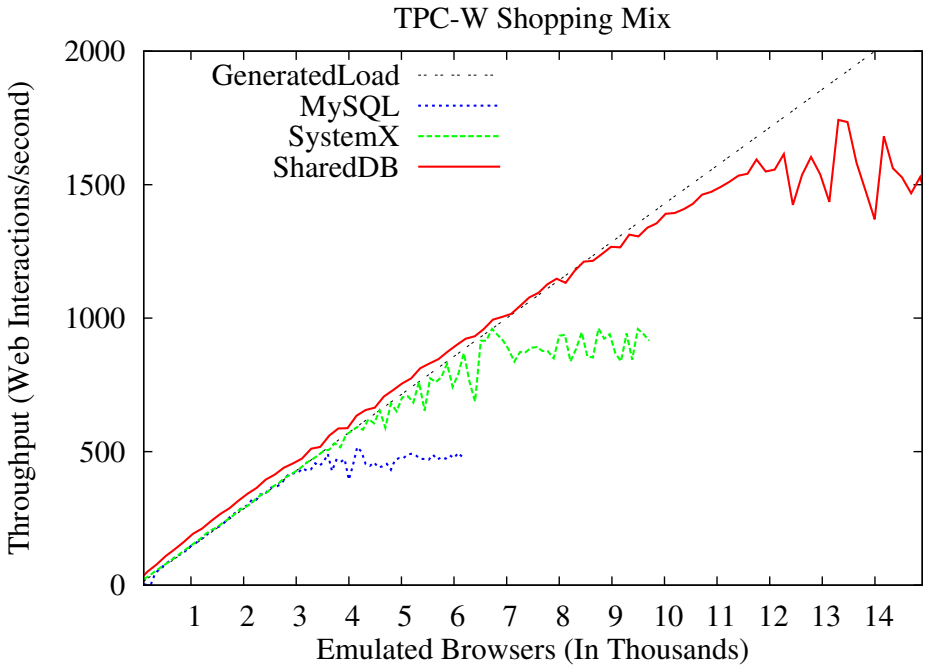


(a) TPC-W Browsing Mix



(b) TPC-W Ordering Mix

Figure 2.22: TPC-W Throughput: Varying Load, All Mixes



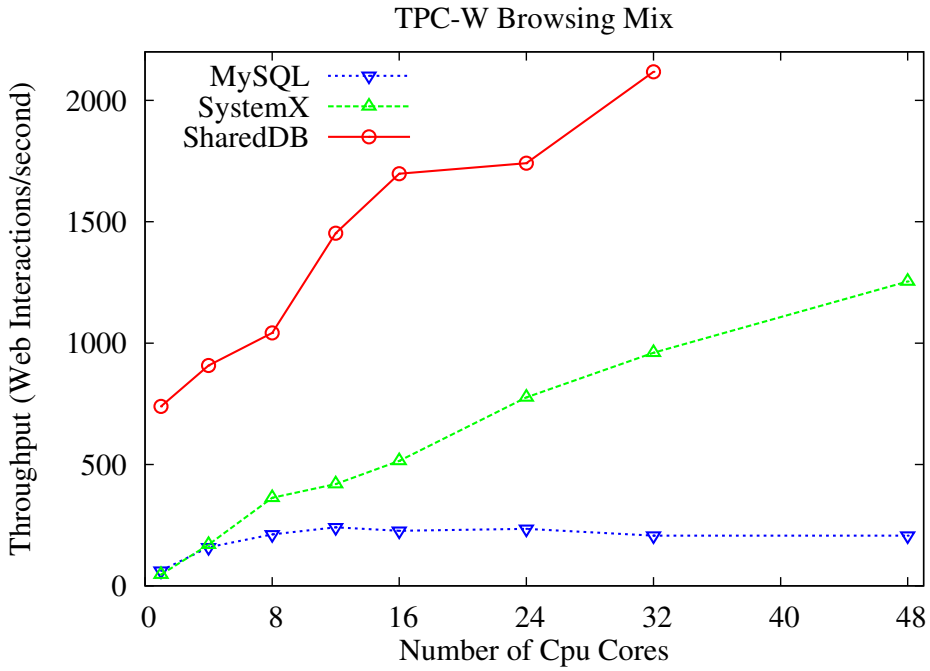
(c) TPC-W Shopping Mix

Figure 2.22: TPC-W Throughput: Varying Load, All Mixes

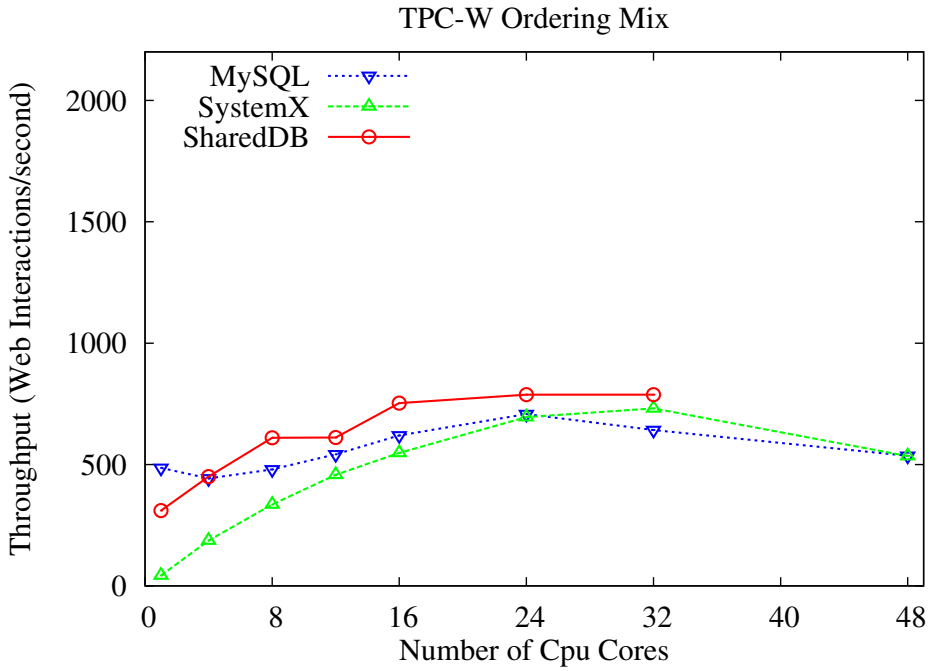
We would like to reiterate that SharedDB used the *same* global plan for all three mixes. SharedDB was not adapted or tuned in any way to meet the specific requirements of these workloads. Obviously, a "query-at-a-time" system can adapt much better to the workload and, for instance, optimize a query based on the currently available resources. We do not know how, for example, SystemX does so. It is clear, however, that shared computation is more critical to sustain high throughputs with response time guarantees than any adaptation technique implemented in MySQL and SystemX.

2.4.2.3 Scaling with the Number of Cores

In the second set of experiments, we explored the impact of having additional CPU cores on the database server. For this reason, we varied the number of available CPU cores of the database server machine from 1 to 48 and repeated the experiments of Section 2.4.2.2 for each configuration. As mentioned in Section 2.4.2, we varied the number of cores for SharedDB only between 1 to 32 because SharedDB cannot take advantage of additional cores for the TPC-W benchmark in the configuration used for these experiments. We measured the maximum number of successful web interactions per second each system could achieve.

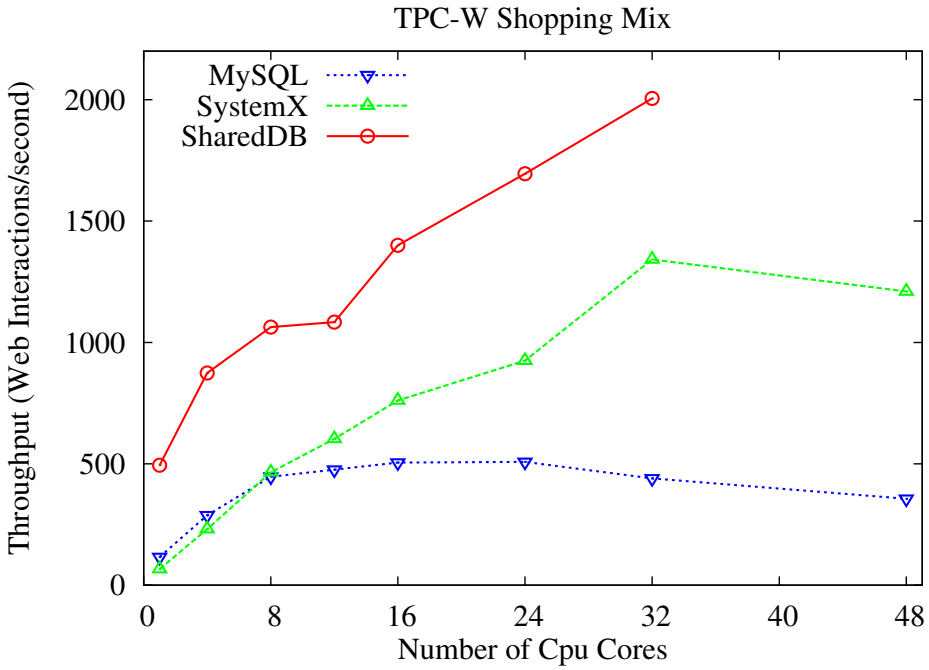


(a) TPC-W Browsing Mix



(b) TPC-W Ordering Mix

Figure 2.23: Max. Throughput: Vary # Cores, All Mixes



(c) TPC-W Shopping Mix

Figure 2.23: Max. Throughput: Vary # Cores, All Mixes

Figure 2.23 shows that SharedDB again is the clear winner for all three workload mixes and almost independent of the number of cores. SharedDB is only outperformed by MySQL for the Ordering mix if the database server is constrained to using only a single core. Again, the magic ingredient is sharing and the special architecture of SharedDB. Both SharedDB and SystemX scale nicely with the number of cores for the Browsing and Shopping mixes. Scalability is limited for the update-intensive Ordering mix because at some point concurrency control and transaction management limit the throughput of the system. MySQL does not scale beyond twelve cores, independent of the workload. This observation was also made in a recent study by Salomie et al. [SSGA11].

2.4.2.4 Analysis of Individual Web Interactions

The TPC-W benchmark involves a variety of different web interactions, each involving a different set of queries. For instance, the home web interaction involves two simple point queries (fetching promotion articles and a user's profile). Other web interactions involve point queries and several updates. Finally, there are also web interactions that involve heavy, analytical queries with multiple joins, grouping, and sorting. Figure 2.24 shows the maximum throughput that each of the three systems can achieve if the clients

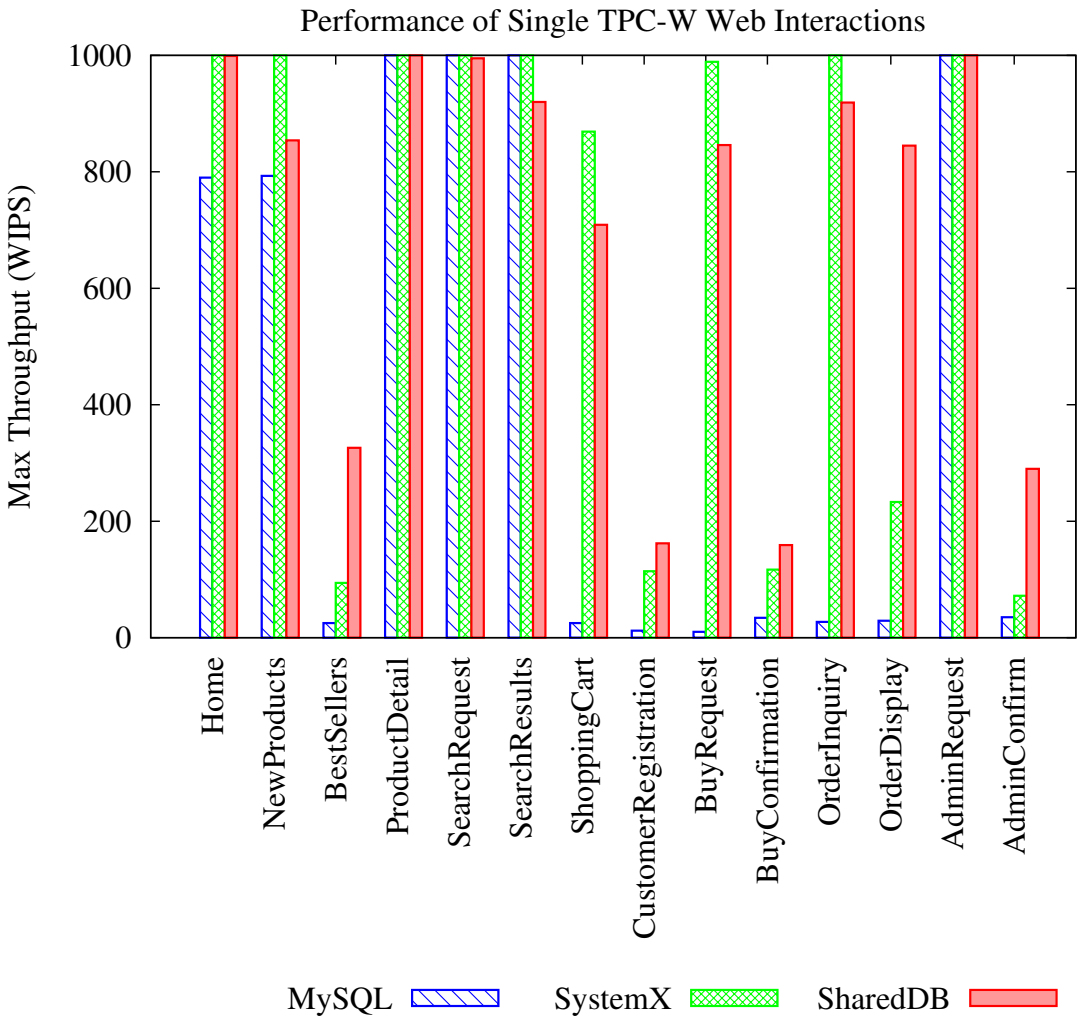


Figure 2.24: Analysis of Individual Web Interactions

are configured to issue only queries that correspond to a single web interaction. These experiments were carried out in a configuration with 24 cores for the database server.

SharedDB wins in this experiment for many kinds of web interactions (e.g., BestSellers and CustomerRegistration). Again, sharing is the main reason for SharedDB's success. Keep in mind that SharedDB does not only support sharing across queries of different types but also sharing between concurrent queries of the same type, executed with different parameter settings.

Figure 2.24, however, also shows that SharedDB loses for several web interactions as compared to SystemX (e.g., NewProducts and ShoppingCart). These web interactions involve mostly point queries and/or updates for which sharing does not help much. SystemX wins

because it is the more mature system and carries out the same work more efficiently than SharedDB. In other words, SharedDB can only beat SystemX if it carries out *less work* as a result of shared computation.

2.4.2.5 Heavy Queries vs. Light Queries

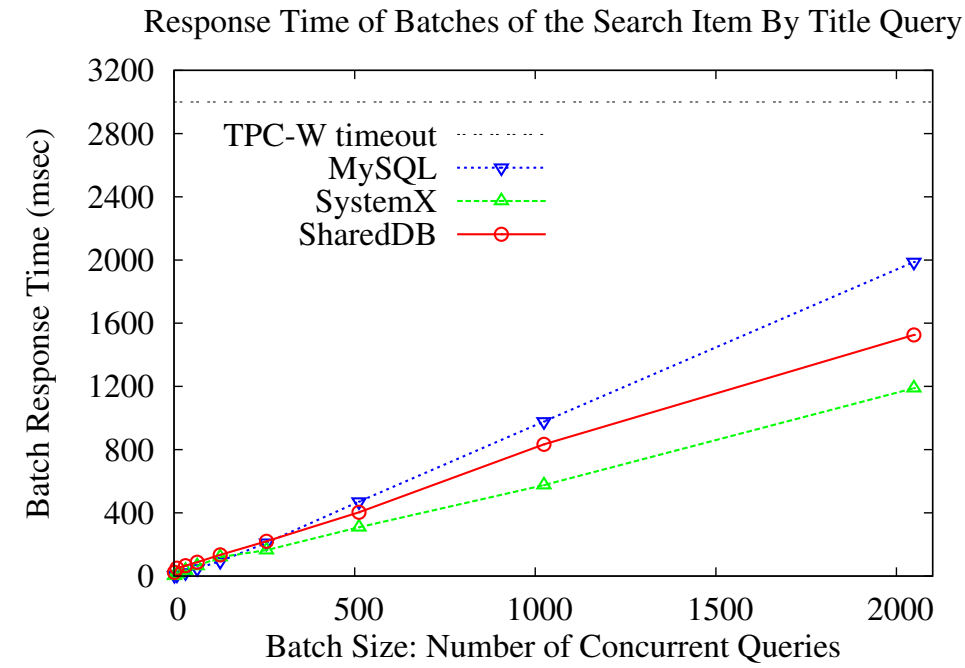
Next, we analyzed the performance of the three test systems under two very different queries of the TPC-W benchmark. The first one uses a key identifier to select an item and its author. This query is part of the `ProductDetail` web interaction. It is a lightweight query that performs a join of two relations and fetches one record from each of them:

```
SELECT * FROM item, author
WHERE item.author_id = author.author_id AND item.item_id = ?
```

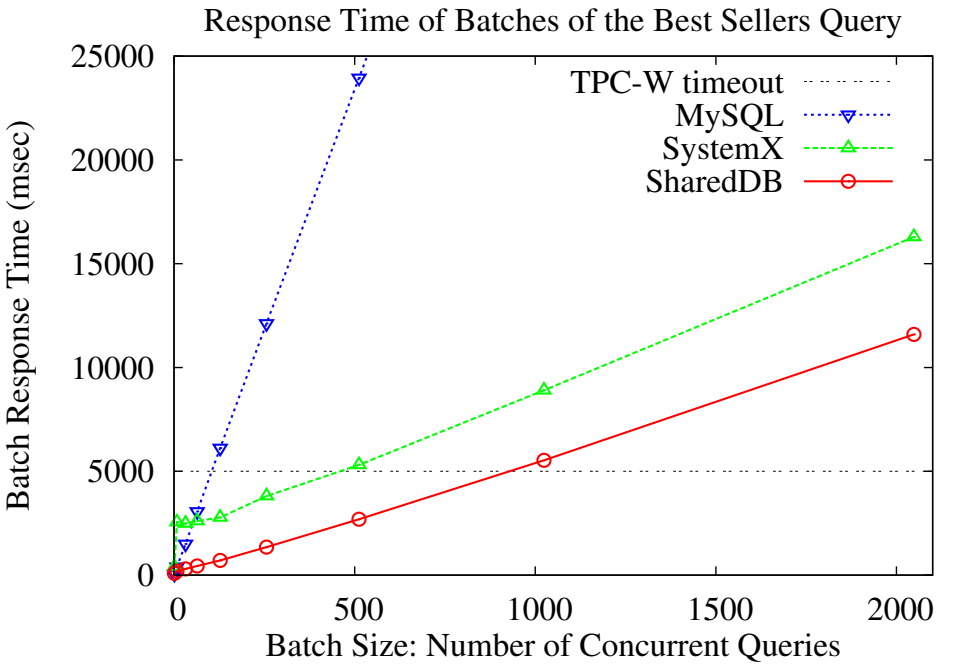
The second query is the "best sellers" query that is part of the `BestSellers` web interaction. This heavy query involves the analysis of the latest 3,333 orders that have been placed by customers in order to retrieve the most ordered items that match a selection predicate provided by the client. It performs three joins over four relations which are followed by a group-by operator and additional sorting of the results:

```
SELECT *, SUM(orderline.quantity) AS val
FROM (SELECT * FROM
        (SELECT * FROM orders ORDER BY date DESC) A LIMIT 3333
     ) AS latest_orders, orderline, item, author
WHERE orderline.ol_o_id = latest_orders.o_id
  AND item.item_id = orderline.ol_i_id
  AND item.subject = ?
  AND item.author_id = author.author_id
GROUP BY item.item_id
ORDER BY val DESC LIMIT 0,50
```

We used batches of an increasing number of such queries and issued a stream of them to the three systems while measuring the time needed in order to complete the whole batch. For SharedDB, the measured time includes the queueing time of the batch, as described in Section 2.4.2. The results are shown in Figure 2.25. With regard to the search item query, the performance of the three systems follows the same trend. SystemX is able to execute the batches of queries faster than SharedDB, which is expected as explained in Section 2.4.2.4. This query is executed so fast that the overhead of batching queries and updates is greater than the gains.



(a) Response Times of TPC-W's `SearchItem`



(b) Response Times of TPC-W's `BestSellers`

Figure 2.25: Heavy Queries vs. Light Queries

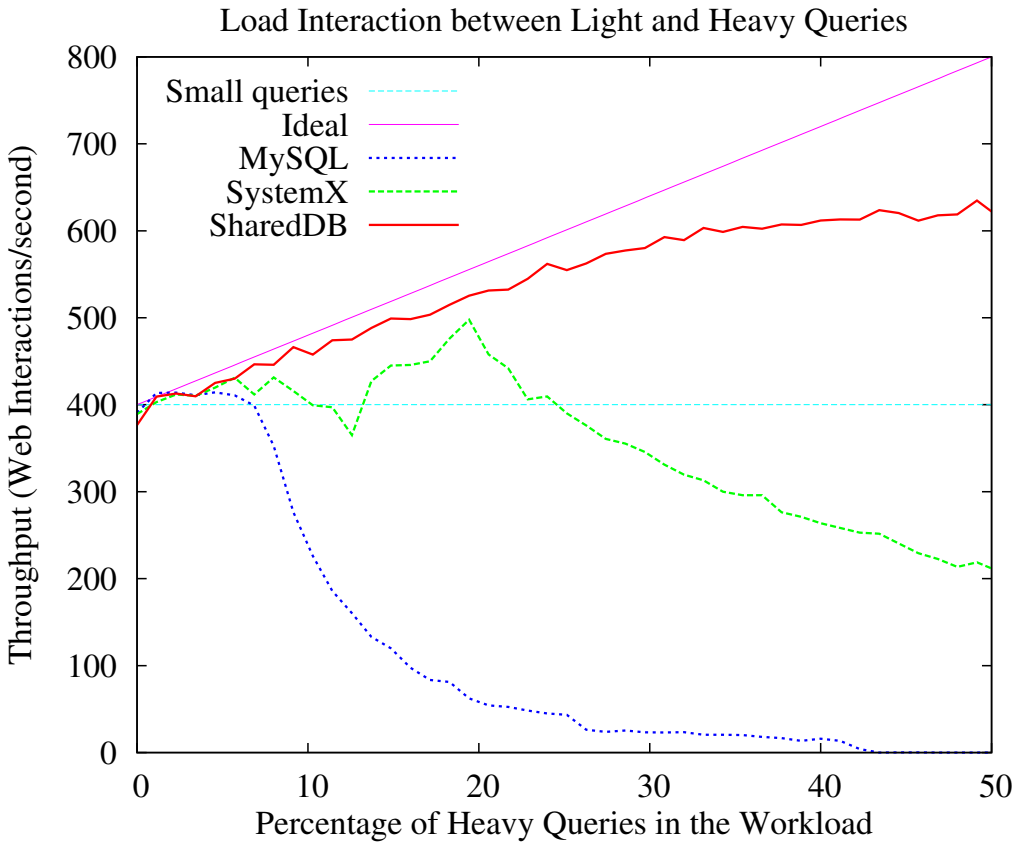


Figure 2.26: Load Interaction

For best sellers queries, the performance of the three systems differs significantly. MySQL's performance is almost linear with the number of queries in the batch. SystemX outperforms MySQL because it is simply the better and more mature system. The best performance, however, was again achieved with SharedDB. Even though SharedDB has a less mature query processor, it outperforms SystemX in this experiment because of sharing.

2.4.2.6 Load Interaction

In our last TPC-W experiment, we explored how heavy queries compete with lighter queries for resources and how this resource contention may impact performance. For this reason we used a synthetic workload that is a mixture of the two queries that were analyzed in Section 2.4.2.5. A constant load of 400 "search item by title" queries per second was sent to the systems under test. The load of these "search item" queries can easily be sustained by all three systems. In addition to these "search item by title", we submitted an increasing number of "best sellers" queries in this experiment. This way, we were able

to study precisely how mixing light and heavy queries affected the performance of the three different systems.

The choice of "search item by title" and "best sellers" queries was not random. In fact, as shown in Figure 2.21, these two queries have a shared join on *Items* and *Authors* in SharedDB. As a result, these two queries compete for resources in SharedDB, too. Of course, an experiment on resource contention between different kinds of queries would not be fair if SharedDB would run the two different kinds of queries on different cores.

The throughput results for all three systems are shown in Figure 2.26. At the beginning (the left part of the figure), the load consists of only the 400 "search item" queries and only a few concurrent "best sellers" queries. Such a workload can be sustained by all three systems. Moving to the right (i.e., increasing the number of concurrent "best sellers" queries), MySQL and SystemsX are not longer able to sustain the throughput. In fact, the throughput of MySQL and SystemX drops below 400 so that the presence of "best sellers" queries hurts the execution of the "search item" queries. Obviously, this problem could be fixed by introducing sophisticated load control mechanisms.

Figure 2.26 shows that such load control mechanisms are not needed for SharedDB. The overall throughput increases monotonically; the more concurrent queries, the more sharing and the merrier. Unfortunately, SharedDB is not a performance panacea either. Starting at about 250 "best sellers" queries per second, SharedDB is not able to handle the full workload either and its throughput diverges from the *ideal* throughput, depicted by the top-most dotted line in Figure 2.26. Since the concurrent "best sellers" queries have different parameter settings, perfect sharing is not possible and there is a per-query overhead in this experiment which limits the scalability of SharedDB with the number of concurrent queries. Nevertheless, SharedDB scales much better than the other systems, beating SystemX by a factor of 3 in throughput in the extreme case of this experiment. More importantly, SharedDB is robust and makes sure that the processing of heavy queries does not have an impact on the performance of light queries, even if no special load control techniques are applied.

2.4.3 TPC-H Evaluation

Finally, the performance of SharedDB was tested under an analytical, decision support workload, the TPC-H benchmark. OLAP workloads are more suited for shared data processing. The bigger run times of queries guarantee that there is always going to be the critical mass of queries required to benefit from sharing. Furthermore, most analytical queries involve scanning of the facts table, an operation that can benefit greatly from sharing.

We implemented a modified set of TPC-H queries in SharedDB. The scale parameter of TPC-H that we used was 10. The modifications regard the top level operators, mostly the `GroupBy` operators. This was necessary, as when we conducted these experiments complex aggregation functions (i.e. `SUM(MAX(price) - MAX(tax))`) were not supported. As a SharedDB storage engine we used the Crescando Storage engine only, which is suitable for heavy full-scan shared workloads.

The baselines for these experiments was PostgreSQL. As with the TPC-W experiments, we tried to make an as fair comparison as possible. We created all necessary indexes for all primary and foreign-key attributes, which allows PostgreSQL to use faster index joins, whenever necessary. We also included indexes on all date attributes for the `Orders` and `Lineitem` relations, which makes up for most of the select predicates on these two biggest relations. As SharedDB is an in-memory database, we allocated all data files in a in-memory filesystem and configured PostgreSQL to allocate a big buffer pool, large enough to store all relations, indexes and intermediate results in memory. Thus, we made sure that PostgreSQL does no disk I/O operations during the experiments. We also configured PostgreSQL to not use query or result caching, as we are interested in the performance of evaluating queries, rather than fetching already processed data from a cache. Still, this is not an apples to apples comparison. PostgreSQL is not an OLAP-specialized database and SharedDB, with Crescando as a storage engine, is far from fully optimized system that can deliver maximum performance for OLAP workloads. Nevertheless, it is a good starting point to see how these two different systems compare when executing a TPC-H based workload.

The query part of the TPC-H workload is a set of 22 complex business intelligence queries, which, typically for OLAP workload, are consisted of expensive select, join, and group aggregate operations. In order to ensure that all the joins will be fully executed in PostgreSQL and certain attributes projected in the joins, we added simple `SUM` aggregators for every query. In SharedDB, since we can control all the attributes projected, we have only added a `LIMIT` operator on top of the join operators. The `LIMIT` operator in SharedDB, unlike in traditional databases, does not halt the execution of the query after the limit has passed, thus we can achieve as close comparison of the two systems as possible.

Apart from removing the complex group aggregations from the queries, we also had to modify or remove some of the predicates to accommodate the restrictions in Crescando. Namely, Crescando does not allow disjunction between predicates, as well as predicates that have multiple attributes as parameters. For instance predicates that have the form `WHERE (PRICE - TAX > ?)` have been replaced with predicates of the form `WHERE (PRICE - 10 > ?)`. We also removed queries, which after being stripped from the unsupported predicates, resulted in unrealistic full table joins between large tables. This left us with a

set of 15 queries, whose SQL syntax form is presented in Appendix A.

The initial plan that we came up with is shown in Figure 2.27. In the figure, dashed lines represent the inner relation, while normal lines the outer relation of join operators. The graph shows the great degree of sharing that exists in TPC-H. A lot of queries share exactly the same execution path. Obviously, the selection predicates are very different, yet a handful of shared operators is able to evaluate all queries.

An important remark on this plan, is that many queries have been extended to include additional operations even if they do not require them. For instance, all queries that access the Customer table will also get a Customer $\bowtie$ Nation $\bowtie$ Region. The reason for this is to eliminate replicating tuples during join operation. For instance, if the Orders $\bowtie$ Customers operator would feed the from both Customers $\bowtie$ Nations and Customers, then tuples from Orders would have to be joined and materialized twice. This would create a domino effect and would double the work for every join operator in the pipeline.

2.4.3.1 Experimental Setup

The hardware that we used for this experiment is a single AMD Magny-Cours machine with 4 $\times$ twelve-core Opteron 6174 processors with 12 $\times$ 512KB dedicated L2 cache and 2 $\times$ 6MB shared L3 cache. The system has 8 NUMA nodes, each with 4 $\times$ 4GB 1333MHz DRAM modules.

We used parallel hash join operators, as described in Section 2.3.6.3. Also, all Storage Engines use multiple cores. As with TPC-W, we performed the assignment of operators to CPU cores by hand. We experimented with different allocation policies. The policy with the highest performance was the one that where storage engines share CPU cores. This may come as a surprise, especially since Crescando is both CPU bound and memory bound. However, the huge runtimes of TPC-H queries (compared to TPC-W), mean that not all storage engines are scanning at the same time. In fact, overlapping the storage engines across a set of 30 cores resulted in the highest storage engine performance. We then used the remaining 18 cores for all the data processing operators. We made sure that operators that may run in parallel never share the same CPU core. For instance, the Lineitem storage engine is not sharing CPU cores with other operators. This is the biggest fact table of the schema and is required to evaluate most of the queries.

We configured SharedDB and PostgreSQL to use all the available resources of the hardware. In PostgreSQL, we set the maximum number of processes to be the number of cores on the machine, which is 48. This means that we have 48 connections from the clients' side. All client threads are queueing up in FIFO fashion to get their queries executed.

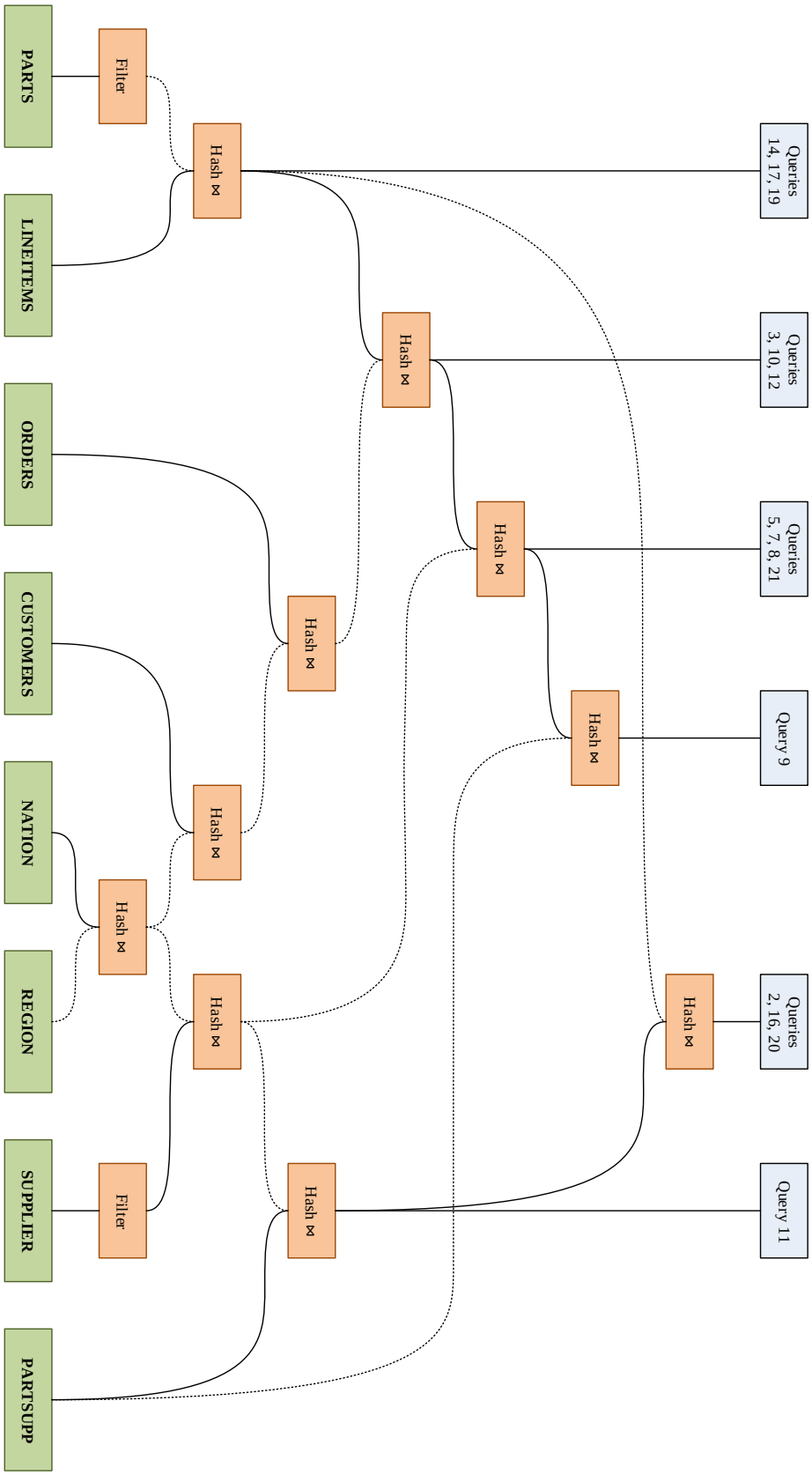


Figure 2.27: Original Global Plan for the TPC-H Benchmark. Updates as well as Selects that access only one relations have been omitted.

The analytical nature of TPC-H and the big runtime of TPC-H queries, cause the 48 connections to be saturated very fast. This is one of the major challenges in query-at-a-time systems.

2.4.3.2 Experimental Results

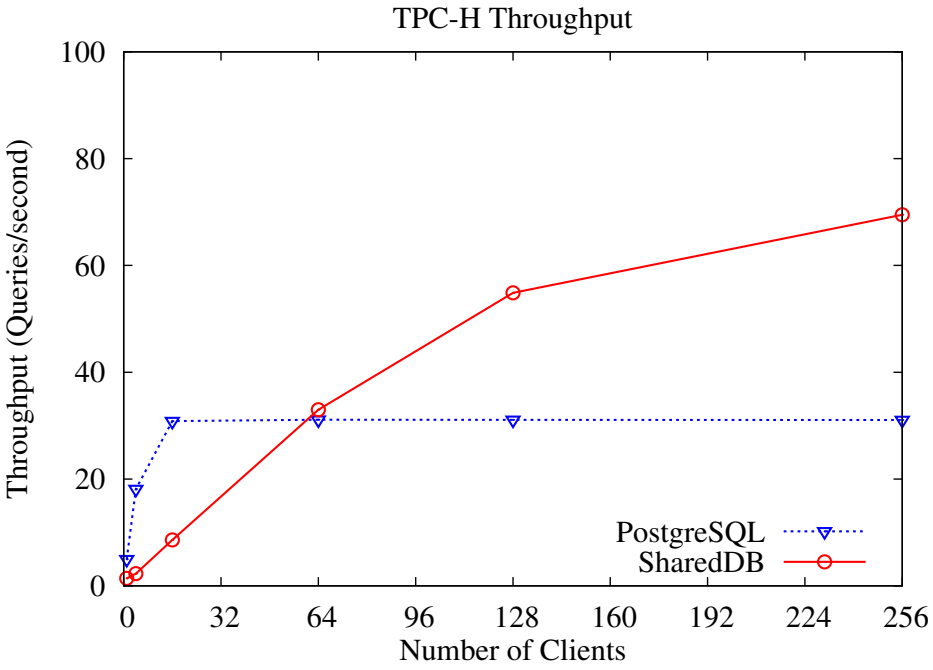
We populated the systems under test with 10GB of TPC-H data and used a variable number of clients that generated load. All clients issued all different TPC-H queries concurrently with randomized parameters, according to the TPC-H specification. This is called a batch. Once the batch finishes, a client is able to issue another batch. Figure 2.28 shows the results. The response time measured corresponds to the time of executing a batch of TPC-H queries.

The plots show that SharedDB outperforms PostgreSQL once a critical mass of queries has accumulated in the system. This is expected, as SharedDB is not using any indexes or optimized algorithms and does not have the maturity of system that has been improving for years, like PostgreSQL. Still, once enough work is accumulated, sharing pays off and compensates for the lower performance with few clients.

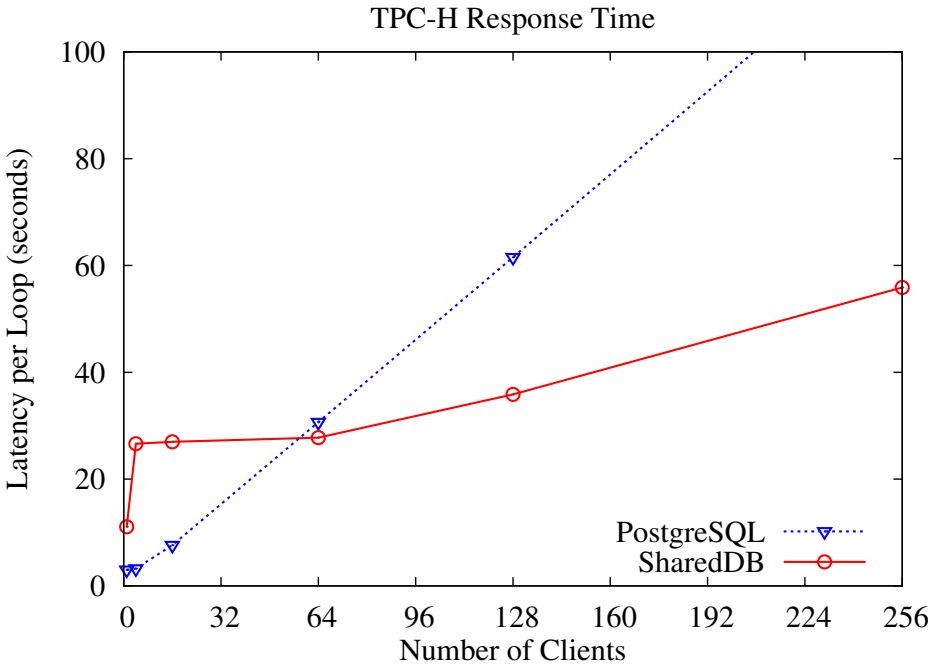
We should point out here that the amount of queries SharedDB executes in parallel is massive. For instance, when there are 256 client threads issuing queries, there are actually $256 \times 15 = 3840$ queries in the system. An interesting feature in SharedDB's graph is the sudden spike when going from 1 client to 4 clients. When there is only one client, the queue in which this client will enqueue its queries will always be empty. When the number of clients is more than 1, the queue will no longer be empty, and in the worst case the enqueued queries will wait for a complete cycle before they are executed. Since this is a closed system, where clients issue their next set of queries after their previous set was executed, we get a situation where for any number of clients $n > 1$, the response time of SharedDB is the response time of executing queries for $n-1$ clients plus the response time of executing queries for 1 client.

This is one of the reasons why SharedDB's performance looks so stable. Another reason is that there is another similar effect inside SharedDB, which unnecessarily replicates work. This is the batch synchronization problem that was explained in Section 2.3.6.4. Combined, these two effects cause major reduction in batch sizes in SharedDB and thus, replicate execution cycles in the operators. Practically, this reduces the benefits of shared execution.

To tackle the issue of batch synchronization, we implemented fused operators in the TPC-H global query plan, as discussed in Section 2.3.6.4. The new Global Query Plan is shown



(a) TPC-H Throughput



(b) TPC-H Response Time

Figure 2.28: TPC-H Performance under Varying Load

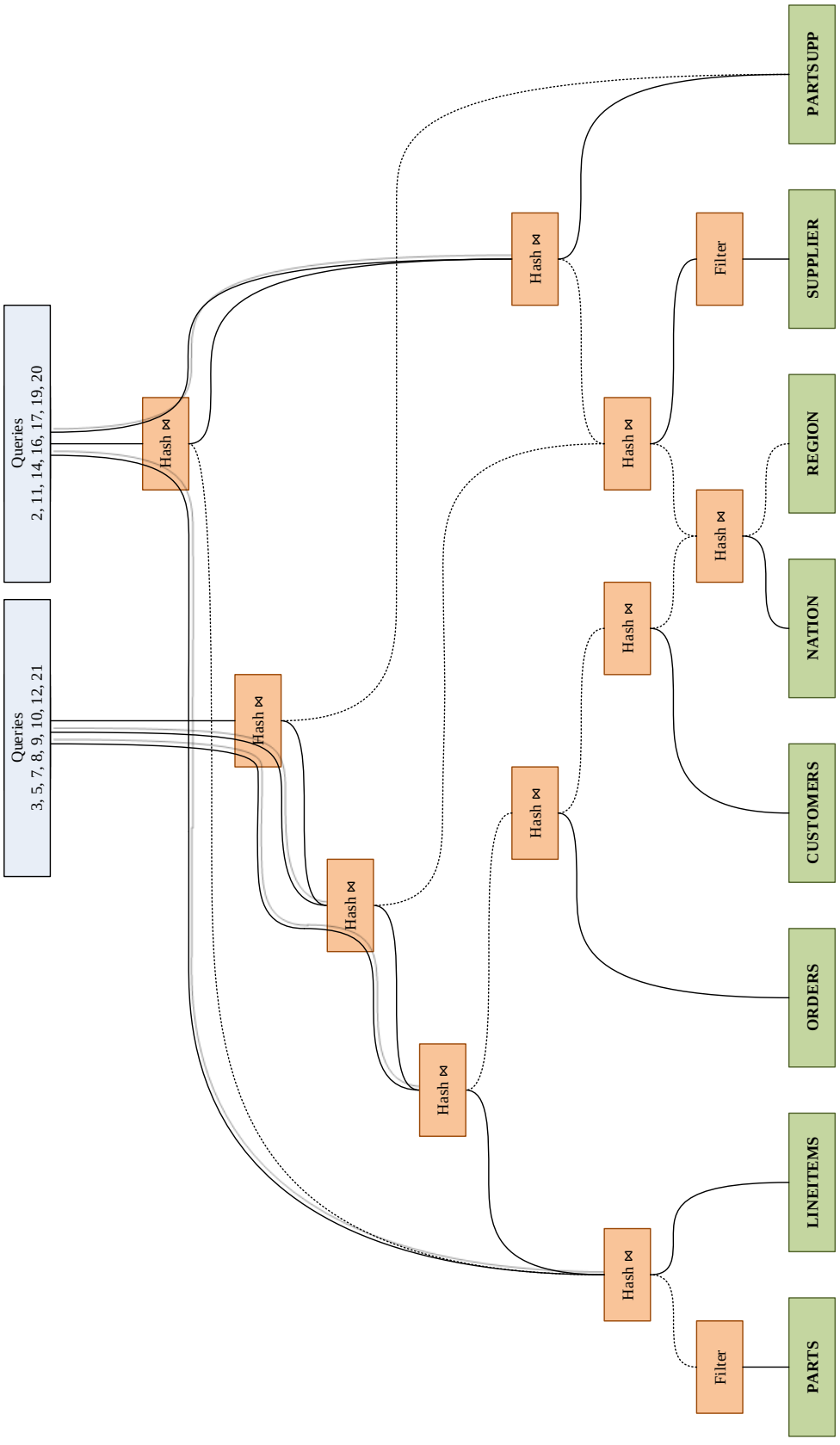
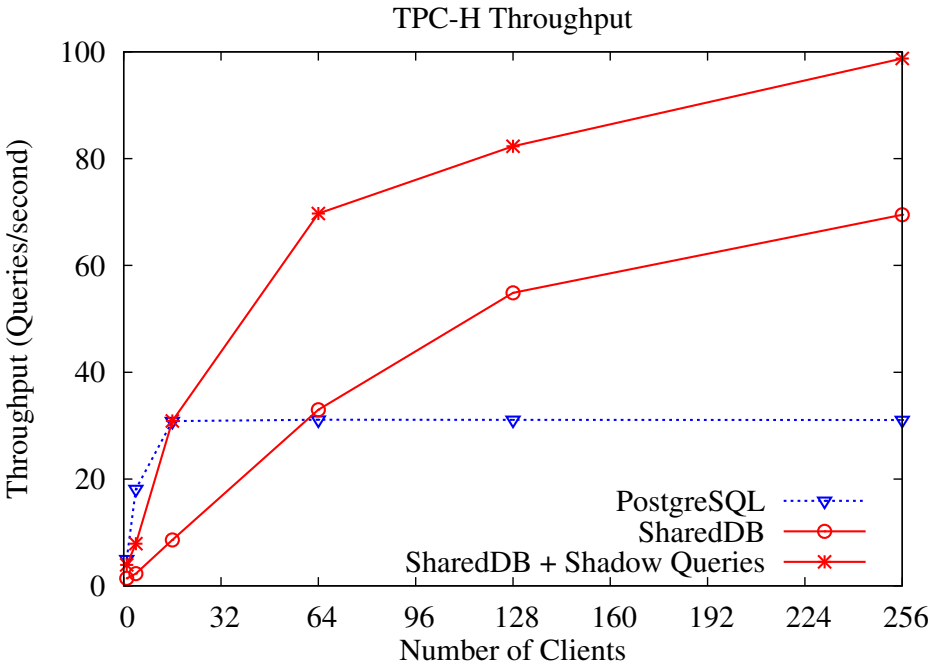
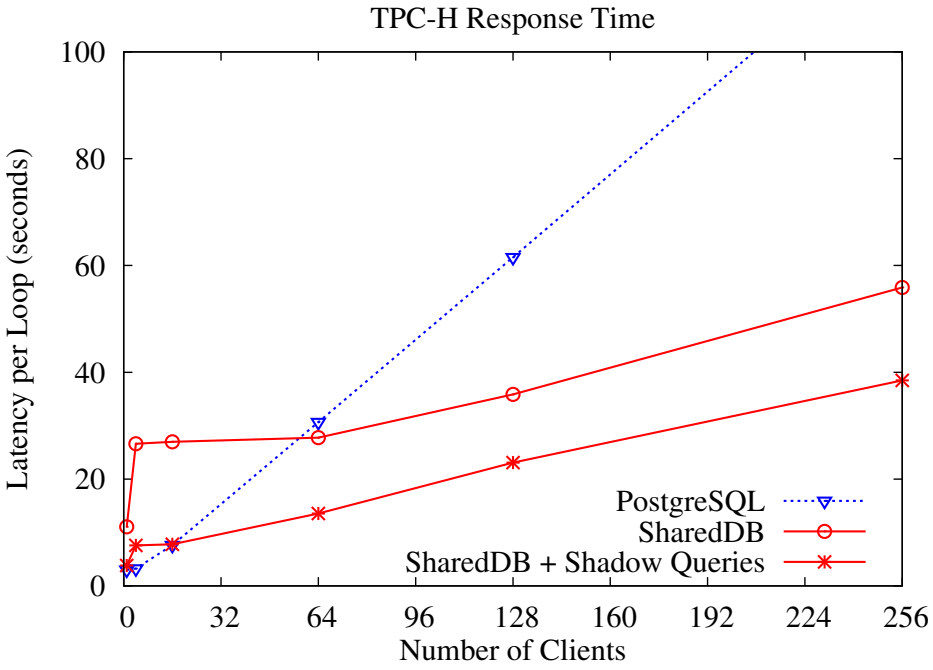


Figure 2.29: Global Plan for the TPC-H Benchmark using Shadow Queries. Updates as well as Selects that access only one relations have been omitted.



(a) TPC-H Throughput



(b) TPC-H Response Time

Figure 2.30: TPC-H Performance under Varying Load when using Shadow Queries

in Figure 2.29. All shadowed lines represent the paths of shadow queries. In order to fully utilize all 14 CPU cores that are allocated for data processing, we increased the degree of parallelism of hash join operators if they were fused. For instance, if a hash join with 2 CPU cores can be fused with another hash join with 3 CPU cores, the resulting fused operator will use all 5 CPU cores.

We repeated the TPC-H experiment, as it was described before. Figure 2.30 shows the results and compares them to the execution plan without the shadow queries. The results show how fusing operators by introducing shadow queries further improves the performance of SharedDB. Most importantly, the point where shared processing becomes better than query-at-a-time processing is shifted considerably in favor of SharedDB. Without fused operators, the critical mass of clients SharedDB requires is 64. Introducing fused operators, this is reduced to just 16.

Finally, we have to repeat that this is by no way a straight comparison of the two system designs. PostgreSQL is a more mature system and is using index lookups and index join operators to evaluate queries. SharedDB in this experiments is only using full table scans (implemented in Crescando) to access tuples. A version of the same global query plan with key-value storage operators could be implemented. Yet, the purpose of this experimental evaluation is not to showcase how fast can SharedDB be, rather than demonstrate that sharing work during data processing is a viable approach for data processing.

2.5 Concluding Remarks

In this chapter, we have described the design and implementation of SharedDB, a push-based, general purpose relational database system that supports all sql operators. At the core of SharedDB is a novel query processing model that is based on batching queries and shared computation. Additionally, we described an extensive experimental study on the prototype of SharedDB that demonstrates its unique properties.

SharedDB does not always outperform traditional database techniques that rely on the "query-at-a-time" processing model. The advantages of SharedDB become apparent for high loads with unpredictable mixes of heavy and light queries and updates. In these situations, the performance (i.e., query latency and sustained throughput) of SharedDB is extremely robust without requiring any special tuning knobs, load control, or other adaptive techniques.

SharedDB's robustness originates from deep aggressive sharing of processing power and resources, as well as isolation of queries. Executing heavy queries in SharedDB has minimal effect on concurrently running smaller queries. In short, each query has a predictable

response time that is defined only by the path in the global query plan and is independent of the concurrent load in the system.

Experimentation on the TPC-W and the TPC-H benchmarks confirmed this robustness along a number of different dimensions: query load, hardware configuration, and query/update diversity. Compared to other related systems that are also based on shared computation, SharedDB wins in terms of generality. For instance, these systems are typically not suitable to process transactional workloads with many small queries and updates.

It becomes clear that the most important part of SharedDB is the global query plan, a single access path used by all queries that are executed in the system. The plans used in this chapter are all handcrafted. Automatically generating such a plan is not a trivial process and has not been studied before. Existing access path optimizers focus in generating a plan that results in the smaller execution time per query. In the following chapter we will present the challenges of generating such a plan, as well as an algorithm that given the whole workload of a system, generates a global query plan with optimized execution time for the whole workload.

3

Work Sharing Optimization

SharedDB is a complete data processing engine that evaluates queries with the goal of reducing the overall work to execute all of them. As described in the previous chapter, a number of similar multi-query-at-a-time data processing systems have been designed over the last decade. The need to reduce the cost of executing a workload is driven by the radical increase of the query workload on database engines. A wide variety of industries such as travel reservation, on-line data services, and real time planning rely on business intelligence for strategic decision making. Furthermore, as the number of internet users and web services increases, databases have to answer an ever-increasing amount of complex SQL queries. A collection of such use cases can be found in [Unt12].

Systems like SharedDB [GUM+10], DataPath [ADJ+10], QPipe [HSA05], Blink [RSQ+08, QRR+08], and Vectorwise [ZHNB07, ZB12] employ cooperative query processing and scan operators that are able to execute multiple queries concurrently. This allows for higher throughput, more predictable response time, and minimal load interaction, as queries are not competing with each other for resources.

Even though these multi query-at-a-time systems are well established and quite sophisticated, most of them do not address the problem of optimizing their execution plans for multiple queries. Some of them use conventional query-at-a-time optimization techniques to generate the execution plans. Others rely on sharing predicates among queries by over-

lapping each individual query access plan and trying to identify commonalities or common sub-expressions, an idea commonly known as Multi Query Optimization (MQO) [Sel88]. Finally, some of these systems rely on manually generated plans, an approach that neither scales nor guarantees optimality. What is sought for is an optimizer capable of looking an entire workload and coming up with a shared execution plan that increases the opportunities for sharing work across queries. All of the aforementioned cooperative systems would benefit from such an optimizer, as the cost of executing a workload would be reduced.

In this chapter we explore the problem of Work Sharing Optimization (WSO), and propose an algorithm that given a set of statements and their relative frequency in the workload, outputs a global plan over shared, always-on operators. Unlike what has been done until now, WSO concerns itself with entire workloads and the simultaneous optimization of all queries in each workload. WSO does not optimize queries individually and does not look for common predicates. Instead it focuses on running queries concurrently over a pool of shared operators and, consequently, must identify which operators to share and how to organize these operators into a global access plan.

Formally, identifying the shared operators and ordering them are not orthogonal decisions, which turns WSO into a bilinear optimization problem. As we will show in the chapter, changing how many statements share the same operator affects the overall selectivity, which might require to reorder the operator. Additionally, the cost function is non-convex, as there are a lot of local optima. For instance, each independent query adds a local optimum, which is the cheapest way to execute it. The non-convex, bilinear nature of the problem renders exhaustive techniques, like brute-force and greedy optimization unsuitable. Exhaustive search will result in huge running times due to the enormous size of the solution space, while greedy optimization will most likely converge to a locally optimum solution. WSO is similar to the stochastic knapsack problem with uncertain weights [KL10], where the stochasticity comes from the fact that the cost or the weight of an item is variable and depends on all previous decisions.

In this chapter we describe an algorithm that tackles the work sharing optimization problem. The algorithm is based on the branch and bound optimization technique and is able to generate a global query plan that answers all queries of a given workload with lower cost than traditional techniques. To reduce the size of the solution space, we introduce two heuristics. The first one makes a quick, yet effective, decision on how to share operators, while the second one gives guidelines on the ordering of these operators.

The implemented algorithm was used to generate global plans of two well known benchmarks, namely the TPC-W and the TPC-H. The runtime of our algorithm is negligible, given that the generated query plan can be used for a long time, possibly the whole life-

time of the system. Experimental results show that the generated global access plans outperform existing query optimization techniques.

In summary, this chapter makes the following contributions to the state of the art of workload optimization:

- a formal definition and analysis of the work sharing problem;
- an extensive analysis of work sharing approaches;
- two empirical heuristics that reduce the solution space of work sharing optimization, as well as an experimental evaluation of their efficiency;
- a novel work sharing optimization algorithm based on branch and bound and the two heuristics; and, finally,
- an experimental evaluation of the WSO algorithm, as well as a comparison to multi query optimization

The remainder of this chapter is organized as follows. Section 3.1 discusses the state of the art and related work on the topic of query optimization. Section 3.2 outlines the key differences between work sharing optimization and multi query optimization. Section 3.3 provides a formal definition of the problem, as well as an analysis of the complexity of problem. Section 3.4 explains the different approaches in work sharing. Section 3.5 introduces two heuristics to reduce the optimization runtime. Section 3.6 presents an algorithm to optimize workloads by work sharing. Section 3.8 shows the results of an extensive experimental evaluation, and finally, Section 3.9 makes concluding remarks.

3.1 State of the Art

Optimizing access plans plays an important role in the performance of database systems. A badly computed access plan may have a huge impact on the performance not only of a single query, but on the overall performance of the system. In this section, we discuss the existing techniques of generating query plans and optimizing the plans for multiple concurrent queries, followed by a discussion on how existing shared work systems handle query plan generation and optimization.

3.1.1 Query-at-a-time Optimization

The problem of optimizing a query plan is not a new one [Pal74, SAC+79]. Most of the early work focused on how to optimize individual query plans. Originally, the goal of the optimizer was to minimize the required processing power and I/O, considering both random I/O and sequential I/O. Even though the metrics for modern optimizers have evolved [MBK02], the key ideas of *single query optimization* remain the same.

Single query optimization boils down to exploring the whole solution space, which gets exponentially bigger as the number of operators involved increases. The solution space has a size of $O(\frac{(2N)!}{N!})$, where N is the number of involved relations. This combinatorial problem is non-convex, meaning that there possibly exist multiple local optima which, in most of the cases, are spread across the solution space.

Since the non-convex nature of the problem renders linear optimization unsuitable and exhaustive search comes with a high runtime, forward dynamic programming became the optimization technique of choice for single query plans [SAC+79]. In dynamic programming, *memoization* is used in order to avoid repeating work while exploring the solution space. The solution space is searched bottom up, by first examining all possible two way joins and then incrementally adding more relations. All intermediate combinations of joins are cached and they are reused whenever possible. Several flavors of dynamic programming have been suggested depending on the nature and complexity of the queries [GM93, KS00]. While dynamic programming comes with certain limitations, especially as the number of relations increases, it has proved to be a successful option for single query optimization. Nevertheless, it cannot be used to optimize multiple queries at the same time, because of the bilinear nature of the problem, as it will be further explained in Section 3.3.

3.1.2 Multi Query Optimization

Multi Query Optimization (MQO) was originally explored in [Sel88]. The main idea is to identify common sub-expressions across the set of running queries. Once common sub-expressions are detected, the system can replace the original sub-expressions with a broader subquery that subsumes all of them or rearrange running queries such that common sub-expressions are executed together.

Even though the idea of MQO was innovative and very promising, it was neglected by most systems for a long time. The reason is that the effort of detecting common sub-expressions across running queries usually surpasses the gains. Additionally, as most database systems are designed to execute queries as fast as possible, there is never the critical mass of commonalities to actually benefit from them. This limits the applicability of MQO to

mostly analytical long-running workloads, where queries execution times are significantly bigger. Additionally, in analytical workloads commonalities are more frequent, especially accesses to the fact relations.

The idea of MQO was extended and further improved in the Volcano optimizer [RSSB00]. This work uses materialized views to further benefit from commonalities across queries. Nevertheless, the additional overhead of maintaining the materialized views limits the applicability of this technique.

An alternative type of multiple query optimization that caches intermediate results has been suggested in [TGO01] and further improved in [IKNG09, MPK00]. Caching (recycling) intermediate results of common operators (i.e. a very frequent join), has been shown to increase the performance as it comes with a number of advantages. Nonetheless, result caching cannot be applied to systems with high update loads, as the caches are invalidated very frequently. In order to allow for high update loads, we seek for a solution that does not rely on result caching at all.

A more recent work by Zhou et al. [ZLFL07] introduces a novel technique for detecting common sub-expressions which allows MQO to be integrated in production systems. Applying the technique to a few queries (less than 20) is extremely efficient, however this approach does not scale to thousands of queries at a time, as it is still based on predicate sharing across running queries.

3.1.3 Work Sharing Systems

In the last years, there is an increasing shift towards database systems and database operators that process multiple queries at a time by sharing work (WS Systems). In Section 2.1.1 we provided an overview of these systems, as well as a description of how sharing is implemented in each one of them. Even though WS systems are quite sophisticated, they all currently rely on single query optimization techniques, MQO, or hand crafted query plans, which provides no guarantees on the optimality of the generated plan. Most importantly, using such techniques to generate query plans does not take full advantage of the capabilities of these systems.

DataPath [ADJ+10] and QPipe [HSA05] rely on predicate sharing and sub-expression matching, similar to MQO. Because these systems rely on temporal overlap for enacting sharing, they are more suitable for analytical workloads where some queries take long enough to allow sharing with other queries. Although intuitively useful, when MQO is applied to work sharing systems, it limits the amount of queries that can share work. The goal of such systems is to process hundreds to thousands of queries concurrently and

we are not aware of any techniques that can detect common sub-expressions for so many queries within a reasonable runtime [ZLFL07]. Additionally, sub-expressions have to be detected every time a new set of queries arrive in the system, as the commonalities of the previous set of queries is independent of the commonalities of the next set of queries. Last but not least, MQO is more effective when used in pull based systems. The asynchronicity of push based systems make it harder to take advantage of commonalities

Our approach to work sharing optimization avoids this limitations by not detecting common sub-expressions at all. Instead it relies on identifying common operators across prepared statements. Moreover, WSO has to be executed only when the workload evolves (i.e. more prepared statements are added). This allows WSO to be applied not only to analytical workloads, but also to transactional workloads, where new queries arrive at a much faster rate. Our experiments on both TPC-H (analytical) and TPC-W (transactional) show that WSO is able to generate efficient plans in both cases. We are not aware of any work on MQO that is able to process the entire TPC-H benchmark, let alone TPC-W which has shorter running queries where MQO is not effective.

CJoin [CPV09] does not implement an optimizer and instead, relies on hand tuned plans. Even though this approach is not very sophisticated, it overcomes the limitations of MQO by requiring more work from a specialized database administrator.

A different type of optimization was originally used in SharedDB in [GAK12] that automates hand tuning of query plans. In this two step optimization, single query optimization is used on each query and then the resulting access plans are overlapped. While this approach simplifies plan generation, it reduces sharing opportunities as executing each query in the best locally way, might create a suboptimal global plan.

To tackle these shortcomings, the query optimization algorithm that is introduced in this chapter is designed specifically for these work sharing systems and aims in increasing the amount of work sharing. The algorithm takes as an input the whole workload and produces an execution plan that minimizes the total amount of work necessary to evaluate it. We will compare the effectiveness of the generated plans by running them on SharedDB. Nevertheless, the algorithm is not specific to SharedDB and can be applied to other WS systems as well.

Finally, some of these WS systems impose additional constraints on the generated pipelines [DSRS01]. For instance, two complementary sort-merge join operator can cause a deadlock if their pipelines are used concurrently. These constraints can be safely integrated in our optimizer. The work of Dalvi et al. presents all the required methods for detecting and resolving a deadlock in such pipelines. Other systems, like CJoin and SharedDB, avoid deadlocks by explicitly not using operators processing two or more streams concurrently.

3.2 Work Sharing Optimization vs. Multi Query Optimization

To illustrate the differences between work sharing optimization and multi query optimization, consider the three queries of Figure 3.1a. Implementing these three queries using MQO results in the execution plan of Figure 3.1b. The expressions of `Q1` and `Q3` are rewritten into a broader expression that selects tuples for both queries. As a result, the cost of accessing `ORDERS` is shared across these two queries. Post-filtering is required in order to separate tuples that answer each query. The filter operator has to execute the predicates of both queries. Yet, this approach allows to share the cost of joining the set of tuples that overlap.

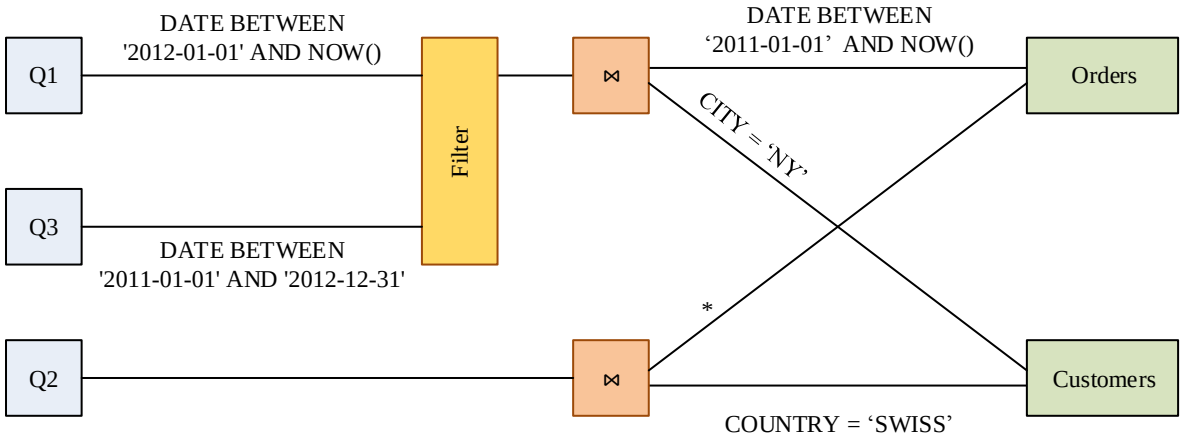
Executing this plan on a work sharing system is not optimal. First of all, the plan generation has to be repeated for every new batch of queries, as the sub-expressions may differ. This is quite common, as most queries are generated from parametric prepared statements. This means that the performance of the system depends on the query parameters; if there is no overlap across query predicates, broader sub-expressions end up being a long disjunction of predicates. Finally, sub-expression detection becomes expensive if hundreds of queries have to be taken into account. CJoin for instance has been shown to easily implement sharing across hundreds of concurrent queries and SharedDB case share work across thousands of queries while both systems maintain stable performance.

A plan that works better for most WS systems is shown in Figure 3.1c. In this case, a single shared join operator processes all three queries at the same time, regardless of commonalities. The predicates are pushed to the storage engines that execute all of them at the same time, while tagging tuples with the IDs of each query. This data processing model is used by SharedDB, as presented in Chapter 2, as well as CJoin [CPV09] and is becoming increasingly popular [PAA13].

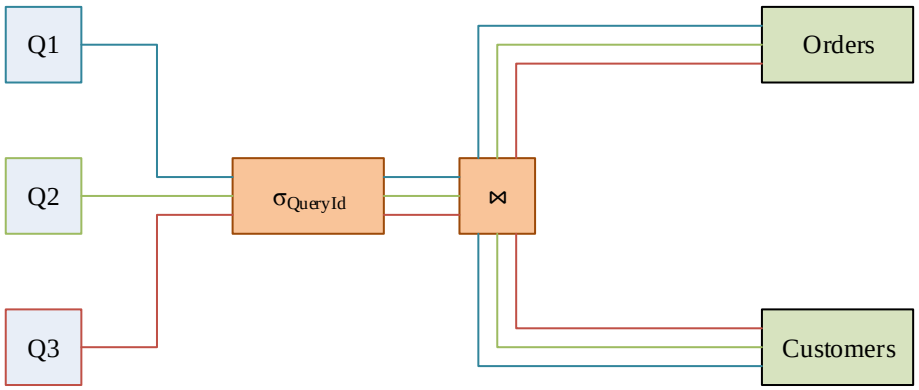
In the plan of Figure 3.1c there is no need to identify common sub-expressions and, as a result, the plan optimizer does not have to examine the query parameters. This allows to create a global plan for all prepared statements of a workload. For instance, if `ORDERS` has to be accessed using full table scans, the same scan will answer all three queries. The lack of commonalities allows this plan to be used by any query that asks for `ORDERS` $\bowtie$ `CUSTOMERS`. Finally, work sharing systems optimize the post filtering part. For SharedDB, this is achieved through the data-query model that was presented in Chapter 2.2.1. The filter groups tuples by the `query_id`s and distributes them to the correct consumers, which is more efficient compared to executing part of the original query.

```
Q1  SELECT * FROM ORDERS NATURAL JOIN CUSTOMERS
    WHERE ORDERS.DATE BETWEEN '2012-01-01' AND NOW()
      AND CUSTOMERS.CITY = 'NEW YORK';
Q2  SELECT * FROM ORDERS NATURAL JOIN CUSTOMERS
    WHERE CUSTOMERS.COUNTRY = 'SWITZERLAND';
Q3  SELECT * FROM ORDERS NATURAL JOIN CUSTOMERS
    WHERE ORDERS.DATE BETWEEN '2011-01-01' AND '2012-12-31'
      AND CUSTOMERS.CITY = 'NEW YORK';
```

(a) Sample Queries



(b) Plan Generated using Multi Query Optimization



(c) Plan Generated using Work Sharing Optimization

Figure 3.1: Differences between MQO and WSO

Work sharing is not always beneficial. Sharing as much as possible may result in more work. To illustrate this consider three queries over two tables with these selections:

	Query 1	Query 2	Query 3
`Customers`	Select 10 Tuples	Fetch all 1,000 Tuples	Select 20 Tuples
`Orders`	Fetch all 10,000 Tuples	Select 100 Tuples	Fetch all 10,000 Tuples

Sharing the `Customer` $\bowtie$ `Order` operation across all queries requires building a hash table of 1,000 `Customer` tuples and probing it with 10,000 Order tuples. In other words, it requires computing the full join. During the build phase, there are a total of 1,030 tuples. The overlapping 30 tuples are processed only once. Similarly, for the probe phase, the processing of the 100 overlapping tuples is shared. The total number of shared tuples for this plan is 130.

An alternative plan is to share the join across `Q1` and `Q3` only. In this case, less tuples are shared. Yet, this plan may be more efficient. The reason is that building a hash table is not as cheap as probing it. This plan requires building two hash tables of $(10 + 20)$ and 100 tuples respectively and probing them with 10,000 and 1,000 tuples. For most WS systems, $build(130) + probe(11K)$ is less expensive than $build(1K) + probe(10K)$. Also, by sharing the join across `Q1` and `Q3`, the probe phase of 10,000 tuples is shared and as a result all tuples are processed exactly once.

3.3 Problem Definition

The problem we are solving in this chapter is query optimization by work sharing, where we search for the globally optimal plan to execute a whole workload. There are two dimensions in this problem: a. ordering of operators and b. sharing of operators. Unfortunately, these dimensions are not orthogonal. For instance, a decision to share a join operator across two queries, means that the selectivity of this operator will be affected and as a result, we should reorder the operators to achieve a better plan.

3.3.1 Operator Ordering

The ordering part of the problem is similar to single query optimization, a problem that has been extensively studied [Pal74]. Given a query that involves n operators, the optimizer has to make n decisions on which order the operators should be executed. For every join operator, the optimizer has to additionally decide on which one is the inner relation and which one is the outer relation. This creates a solution space of $O(\frac{(2n)!}{n!})$ solutions.

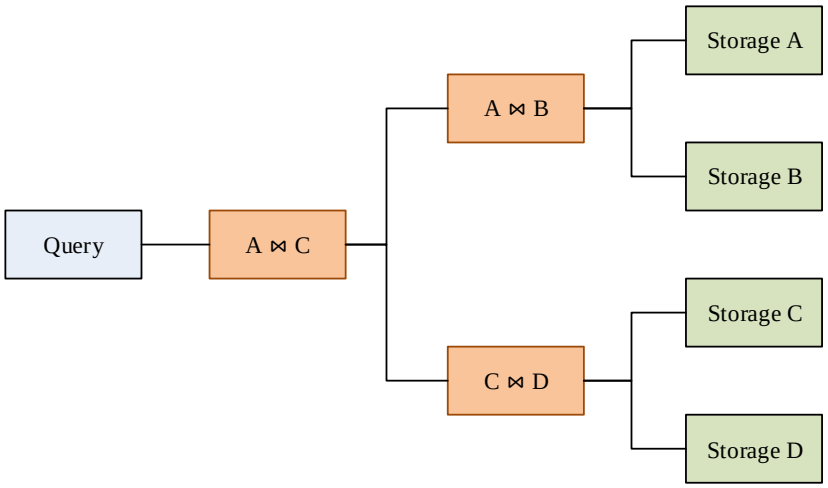
The problem of single query optimization can be mapped into a cost-based packing problem. Thus, we can formulate this as a integer linear programming problem using a two

```
SELECT * FROM A, B, C, D
WHERE A.id1 = B.id
   AND A.id2 = C.id1
   AND C.id2 = D.id
```

(a) Sample Query

	A ⋈ B	A ⋈ C	C ⋈ D
1^{st}	1	0	0
2^{nd}	0	0	1
3^{rd}	0	1	0

(b) Sample *sel* Matrix



(c) Sample Plan

Figure 3.2: Example of Formulization of Single Query Optimization using Integer Linear Programming

dimensional matrix *sel*, to store which operator o was selected on each step s, and a two dimensional matrix $cost_{s,o}$ that contains the cost of selecting operator o at step s. The problem of operator ordering can be mathematically formulated by:

`Minimize:`

$$\sum_{s=1}^{n}\sum_{o=1}^{n} sel_{s,o} * cost_{s,o}$$

`under the constraints:`

$$\sum_{o=1}^{n} sel_{s,o} = 1, \forall s \in [1,n] \tag{3.1}$$

$$\sum_{s=1}^{n} sel_{s,o} = 1, \forall o \in [1,n] \tag{3.2}$$

Here constraint 3.1 enforces that we choose only one operator on each step, while constraint 3.2 enforces that we eventually select all the required operators. An example of a single query plan is shown in Figure 3.2.

This simplified formulization of the problem assumes that there is only one possible implementation for each operator. In most systems there are different flavors of i.e. table access (scans or indexes), or joins (sort-merge and hash join). Adding all the possible flavors of operators in the formulization would only increase the dimension of the operators, without increasing the required steps. On the other hand, the different flavors have a tremendous impact on the complexity of solving the problem with integer programming, which is $O(\frac{(2n)!}{n!})$ as mentioned before. Introducing two additional flavors per operator, i.e. hash join, sort-merge join and partition join, increases the complexity to $O(\frac{(2\cdot 3\cdot n)!}{n!})$.

The "standard" approach to solving single query optimization problems is forward dynamic programming, as pioneered in IBM's System R [SAC+79]. Dynamic programming requires subproblems to be overlapping and independent of each other, which holds for single query optimization. In the case of join ordering, a subproblem is considered as the best way to join n relations. The algorithm considers combining each pair of relations for which a join condition exists, thus solving all subproblems for $n = 2$. Then, it incrementally builds on top of these subproblems to generate the best plans for more relations, until all relations have been joined. In most existing dynamic programming optimizers, the different flavors of accesses and joins are considered when solving each subproblem.

Dynamic programming effectively reduces the complexity of the single query optimization problem, as all subproblems are evaluated once and their results are cached. Yet, even with dynamic programming, a total of $O(3^n)$ joins have to be evaluated [OL90]. As a result, for large values of n the problem of join ordering requires a lot of processing and other techniques should be applied to simplify it. For instance, [KS00] suggests that for n greater than 10 (i.e. a ten-way join), heuristics should be used. Even though queries with ten-way joins are not very common, this indicates a serious limitation of dynamic programming in single query optimization.

3.3.2 Operator Sharing

Shared data processing introduces another dimension to access path optimization, as queries are allowed to share common operators. The problem of operator sharing can be formally defined as:

`Given:` $\mathbb{Q}$, the set of queries that compose the workload, and $|\mathbb{Q}|$ sets of operators, $\mathbb{O}_{1...|\mathbb{Q}|}$, that are used by the queries.

`Find:` A global access plan GP that uses a set of operators selected from all the $\mathbb{O}_{1...|\mathbb{Q}|}$ sets, such that:
- all queries from $\mathbb{Q}$ can be answered, and
- the cost of GP is minimal.

This definition assumes that if multiple queries involve the same operator, then this operator can be chosen only once and serve all queries at the same time. Alternatively, the operator may be spawned multiple times, where each instance of the operator servers only a subset of the queries that involve it.

To handle operator sharing, an optimizer needs to take an additional decision on which queries share which operators. This dramatically increases the decision space of the problem. Additionally, the number of required steps is not fixed. Sharing an operator across s queries means that we require $s-1$ fewer decisions. This three dimensional multi query optimization problem can be formulated as:

`Minimize:`

$$\sum_{s=1}^{S}\sum_{o=1}^{|\mathbb{O}|}\sum_{q=1}^{|\mathbb{Q}|} sel_{s,o,q} * cost(s,o,q)$$

`under the constraints:`

$$\sum_{o=1}^{|\mathbb{O}|}\sum_{q=1}^{|\mathbb{Q}|} \max(sel_{s,o,q}) = 1, \forall s \in [1,S] \tag{3.3}$$

$$\sum_{s=1}^{S}\sum_{q=1}^{|\mathbb{Q}|} \max(sel_{s,o,q}) = \mathbb{O}, \forall o \in [1,|\mathbb{O}|] \tag{3.4}$$

$$\sum_{s=1}^{S}\sum_{o=1}^{|\mathbb{O}|} \max(sel_{s,o,q}) = |\mathbb{O}_q|, \forall q \in [1,|\mathbb{O}_q|] \tag{3.5}$$

In the formulas, s is the current step, o denotes the operator, and q the query. The maximum number of required steps, S, is $\sum_{q=1}^{|\mathbb{Q}|} |\mathbb{O}_q|$, where $\mathbb{O}_q$ is the set of operators required by the qth query. This is the case when no operators are shared across any queries. As a result, the total required operators $|\mathbb{O}|$, is also equal to S, which is the maximum number of operators that can satisfy all the constraints.

Constraint 3.3 enforces that we choose only one operator on each step. This operator might be shared across all or some queries, which introduces the $max()$ function. Constraint 3.4 enforces that we eventually choose all required operators, that is all operators that are

required by all queries. Finally, constraint 3.5 enforces that all queries are answered. This mathematical formalization lacks one more constraint which cannot be described mathematically and limits the sharing of operators. In short, an operator cannot be shared across a set of queries unless all suboperators are also shared across the same set of queries. An analysis of this constraint will be given in Section 3.4.

Last but not least, the cost of subproblems is not fixed. In the two dimensional, single query optimization problem, the cost of each subproblem is independent. Building on top of subproblems is efficient, as results can be reused. This is possible because subproblems are not overlapping, and the problem has optimal substructure. The shared optimization problem does not have optimal substructure. This means that the cost of a join depends on the join itself, and on how many queries are sharing this join and all the underlying joins. In other words, the best way to join n relations depends on how many queries are sharing these $n-1$ joins. More interestingly, the cost changes if one of these joins is shared across some other query (that does not require all n relations). As mentioned before, the problem resembles a knapsack problem with stochastic weights.

Figure 3.3 shows a simplified example of this formulization, as well as the plan that is based on the *sel* matrix. In this example, three queries involving four different joins are combined into a single plan. The plan generation requires more than four steps, as in this example the optimizer decided not to share $A \bowtie B$ with all queries.

Complexity Analysis

As indicated by the previous example, adding sharing on top of ordering, a problem that was already NP-hard [IK84], heavily affects the complexity. A work sharing optimizer has to decide which operator should be executed next, as well as which combination of queries will use it. For every operator, a new dimension of $O(2^{ds-1})$ choices is introduced, where ds represents the degree of sharing, the number of queries that ask for the same operator.

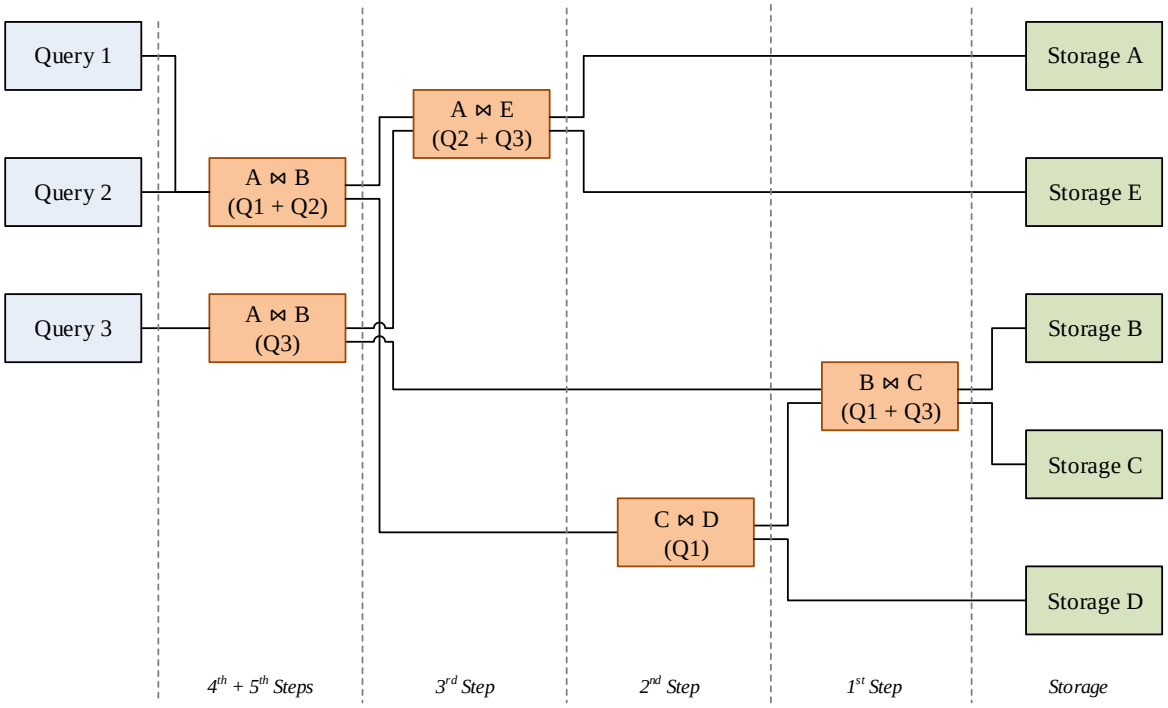
To make matters worse, in WSO we have to consider all operators involved by all queries. For instance, if a workload contains q queries, each of which involving the set of operators $\mathbb{O}_q$, the WSO optimizer will have to consider the union of all $\mathbb{O}_q$ operators. As a result the solution space of the presented problem has a size of $O(2^{ds-1} * \frac{(2*|\mathbb{O}|)!}{(|\mathbb{O}|)!})$, where $\mathbb{O} = \mathbb{O}_1 \cup \mathbb{O}_2 \cup \cdots \cup \mathbb{O}_q$. As explained in Section 3.3.1, traditional techniques, like dynamic programming, are not practical when more than 10 operators (ten-way joins) are considered. In shared workload optimization, we may have way more than 10 operators. For instance, having 10 three-way join queries in a workload, requires examining 30 operators in total. The problem can be classified as a non-convex, bilinear optimization problem, where the non-

```
Q1  SELECT * FROM A, B, C, D
    WHERE A.id = B.id
      AND B.id = C.id
      AND C.id = D.id
Q2  SELECT * FROM A, B, E
    WHERE A.id = B.id
      AND B.id = E.id
Q3  SELECT * FROM A, B, C, E
    WHERE A.id = B.id
      AND B.id = C.id
      AND B.id = E.id
```

(a) Sample Queries

		A ⋈ B	B ⋈ C	C ⋈ D	A ⋈ E
1^{st}	Q1	0	1	0	0
	Q2	0	0	0	0
	Q3	0	1	0	0
2^{nd}	Q1	0	0	1	0
	Q2	0	0	0	0
	Q3	0	0	0	0
3^{rd}	Q1	0	0	0	0
	Q2	0	0	0	1
	Q3	0	0	0	1
4^{th}	Q1	1	0	0	0
	Q2	1	0	0	0
	Q3	0	0	0	0
5^{th}	Q1	0	0	0	0
	Q2	0	0	0	0
	Q3	1	0	0	0

(b) Sample *sel* Matrix



(c) Sample Generated Plan

Figure 3.3: Example of Formulation of Shared Query Optimization

convex nature comes from the fact that there are a lot of local minima and the bilinearity comes from the fact that subproblems are not independent.

In order to solve this problem we considered a number of optimization methods. Exhaustive methods, like brute force or backtracking could not be used, due to the huge dimensions of the solution space. Additionally, greedy optimization methods, as well as convex optimization methods are useless. These methods would most likely converge to a local minimum, which is the best way to execute some query, instead of the best way to execute the whole workload. Furthermore, the dimensions of the problem do not have optimal substructure properties. Solving the ordering problem independently of the sharing problem will result in suboptimal solutions.

A class of optimization methods that is able to solve such problems is stochastic optimization, where generated random variables are applied into the objective function and the best encountered solution is chosen. Recent stochastic methods like Stochastic Optimization using Markov chains [LVLM10] provide guarantees on the optimality of the solution by generating random variables in a smarter way. Yet, the runtime of such methods is quite high which make them not a good solution for this kind of problem.

Next we explored combinatorial optimization methods. Dynamic programming, a method successfully used in single query optimization is not suitable for this problem. In WSO, subproblems are not independent of each other as explained before. As a result, *memoization* is useless and dynamic programming cannot be applied. Branch and bound (B&B), an optimization method used to solve a wide variety of combinatorial problems, fits the constraints of our problem. B&B systematically enumerates a tree of candidate solutions, based on a bounded cost function. However, the huge number of variables of our problem means that branch and bound will need to examine on average $O((|\mathbb{O}| * ds)^{\frac{|\mathbb{O}|}{ds}})$ candidate solutions [TDG09, DZC99]. In this case $|\mathbb{O}| * ds$ is the fan out of the solution tree which is the number of possible solutions to branch, and $\frac{\mathbb{O}}{ds}$ is the average depth of the tree. In order to further reduce the complexity of the problem, heuristics have to be applied.

The algorithm we present in this dissertation is based on branch and bound with heuristics in order to find a globally optimized solution. The heuristics used are presented in Section 3.5. In order to simplify the problem, we will consider queries that only have two operators: hash joins and scans. This does not reduce the effectiveness or applicability of our algorithm, as it can be easily adapted to support more join methods and other operators, like sort and group by. We decided to work with joins, as these are more sensitive in terms of performance when it comes to ordering as well as sharing.

3.4 Work Sharing

The goal of a work sharing optimizer is to find an efficient global access plan for all input queries. As already mentioned, this requires two tasks: a. decide on whether to share a database operator across queries and b. decide on the ordering of the shared and not shared operators. What makes the problem interesting is that these two tasks interact with each other. The task of ordering operators has been extensively studied [Pal74, SAC+79]. In this section we will study how operators can be shared, what are the limitations of sharing, and we will analyze the possible sharing strategies.

As already mentioned in Section 3.1.3, sharing an operator across different statements reduces the amount of work required to execute the workload by not repeating the same work multiple times. However, sharing everything is not always optimal, as shown in Section 3.2. For instance, sharing the execution of a full table join with a much smaller join means that the second statement will have to post filter more tuples on the higher levels of the query plan, as the result stream will contain tuples from both queries.

Based on the constraints of different WS systems, we can enumerate three different types of sharing:

Share Nothing

An example of a share nothing query plan is shown in Figure 3.4. In this approach no work is shared across statements. Queries that are created from the same prepared statement will still share work with each other, however, scans, joins and other operations that are common with queries from other statements will be executed multiple times. Finding the globally optimal plan for this case is easier since traditional (single) query optimization techniques can be applied. Share Nothing is very common in systems that make use of heterogeneous replication. In this case, multiple replicas of the dataset and the query processing engine are present. Each replica is specialized in executing a different prepared statement. Thus, replicas that answer point queries will incorporate the appropriate indexes, while replicas that are dedicated to queries involving full table scans will omit these indexes and as a result avoid the cost of maintaining them. Another example of a share nothing approach is materialized views.

Nevertheless, a plan that shares nothing does not utilize the full potential of WS systems, as there are no sharing possibilities across statements. To make matters worse, share nothing implies that all storage engines have to be replicated for every statement that wants access on them them. This requires additional locking and synchronization to ensure consistency

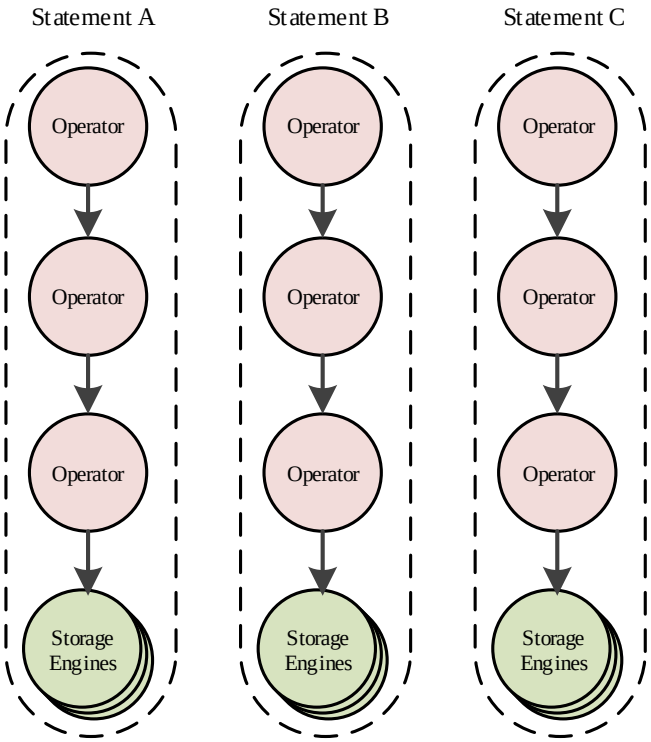


Figure 3.4: No Work Sharing across Prepared Statements

which adds a considerable overhead in case of heavy update loads, and makes the global query plan not scalable to the number of statements. Even though global query plans that share nothing are viable, none of the studied WS systems support them, thus the proposed optimizer is not considering them.

Share Storage Engines

This approach overcomes the issue of storage engine replication by sharing all the storage engines in the system across all statements. An example of such a plan can be seen in Figure 3.5. Database operators are still not shared across statements, while all storage engines are shared. Finding a globally optimal plan is still possible using traditional single query optimization techniques. Each statement has its own pipeline of operators that only interacts with other pipelines in the lowest level, making it possible to reorder operators within each pipeline.

This sharing model is currently employed by a couple of research prototypes, like DataPath [ADJ+10] and StagedDB [HA05]. The big advantage of this model is that any similar operation on the storage layer, which typically is the most expensive one, is not repeated.

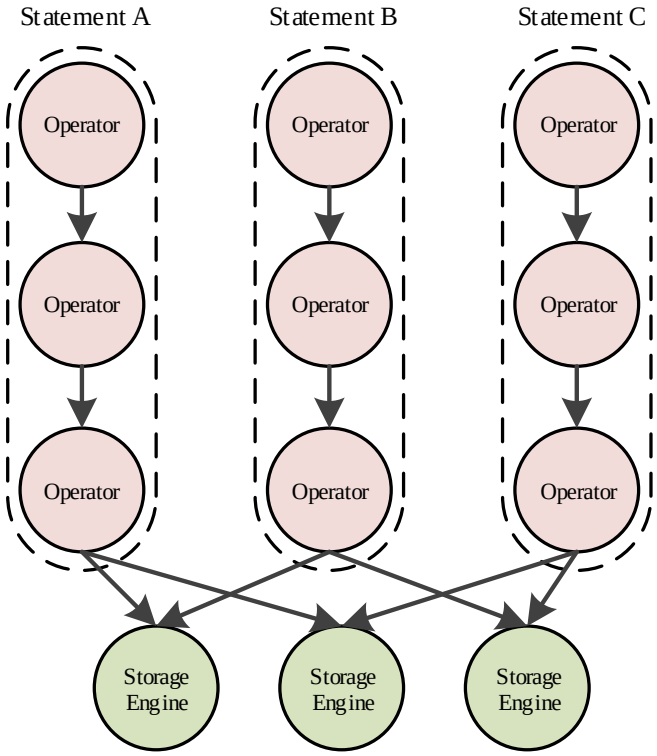


Figure 3.5: Share only Storage Engines across Statements

For instance, a full table scan operator can run once in order to execute as many scan queries are currently pending. This is the main reason why shared (cooperative) scans have been established and implemented in a number of systems, like Blink [RSQ+08, QRR+08] and Vectorwise [ZHNB07, ZB12].

Such query plans are considered in the proposed optimizer even though the amount of shared work is minimal. The optimizer considers building such disconnected pipelines of operators in cases where the statements are very diverse. For instance, consider a workload of two prepared statements, one of which is analytical and long running while the other is a transactional, short running query. Sharing, e.g., a join across these two statements means that the performance of the short running query will be dominated by the shared join which will be slowed down by the analytical statement. The overall cost of executing these two queries will also be greater, as filtering will be required in the smaller access plan in order to isolate the relevant small set of results out of the much larger stream of results.

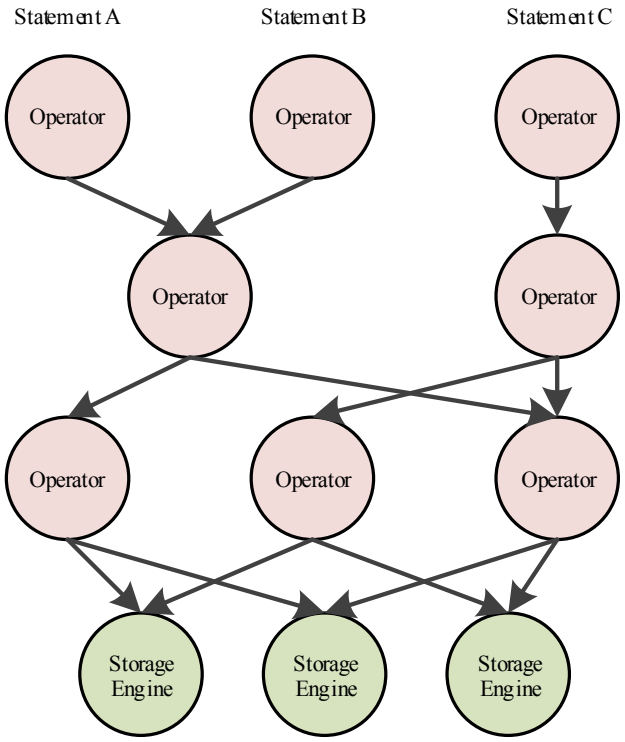


Figure 3.6: Share all common Operators and Storage Engines across Statements

Share Everything

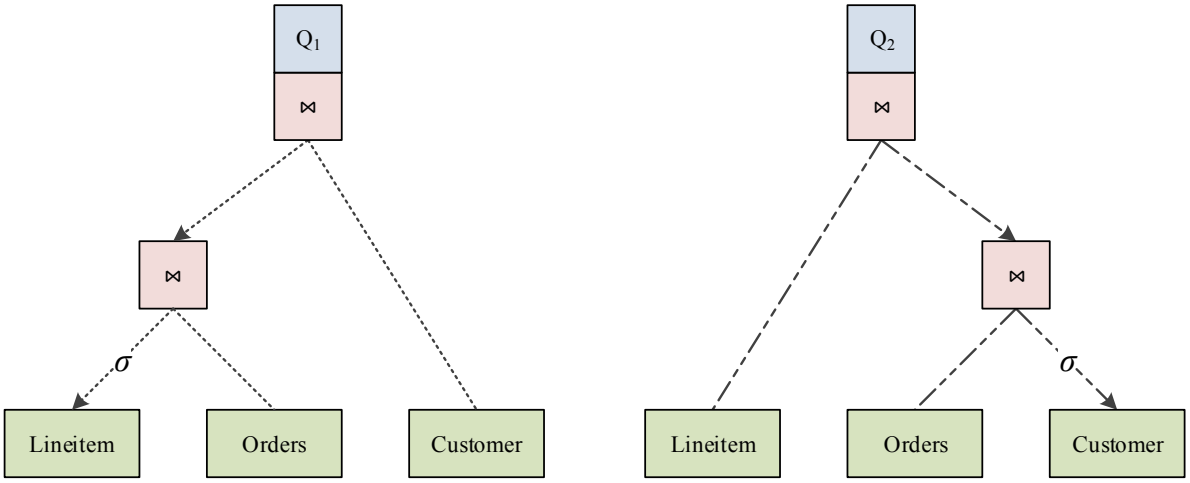
The last type of sharing is to share as much as possible across statements. Such a plan can be seen in Figure 3.6. In this case, each operation that is common across all statements is shared. A number of recent research prototypes fit this model, like CJoin [CPV09]. The Share Everything model achieves maximum sharing which means that work is never repeated. Practically, this approach can minimize the total processing power required to execute a given workload. Nevertheless, if very diverse statements are involved, the pipeline of execution gets slower for the whole workload, and as a result shorter pipelines spend more time waiting for the longer pipelines to finish, rather than actually processing tuples. As a result, in such cases the short queries are penalized and the slowest query dominates the performance of the system.

3.4.1 Work Sharing in Practice

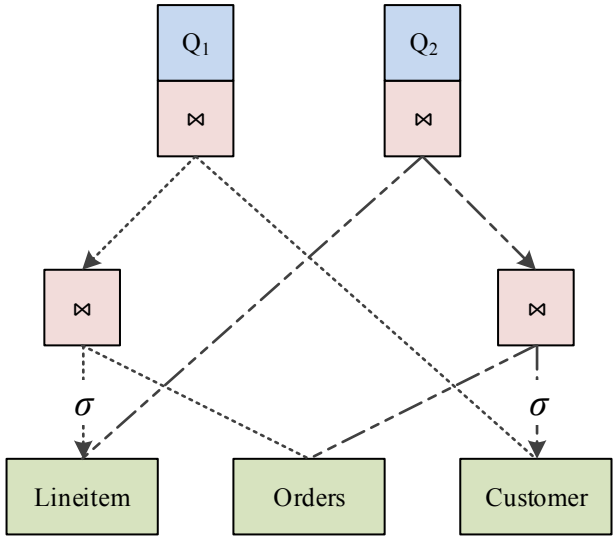
While share everything seems optimal, it adds constraints on the ordering of operators. Specifically, pipelines of shared operators should always have the same sub-operators for

```
Q1  SELECT * FROM LINEITEM JOIN ORDERS JOIN CUSTOMERS
    WHERE LINEITEM.ITEM_ID = '?'
Q2  SELECT * FROM LINEITEM JOIN ORDERS JOIN CUSTOMERS
    WHERE CUSTOMERS.ID = '?'
```

(a) Example Queries with Diverse Joins



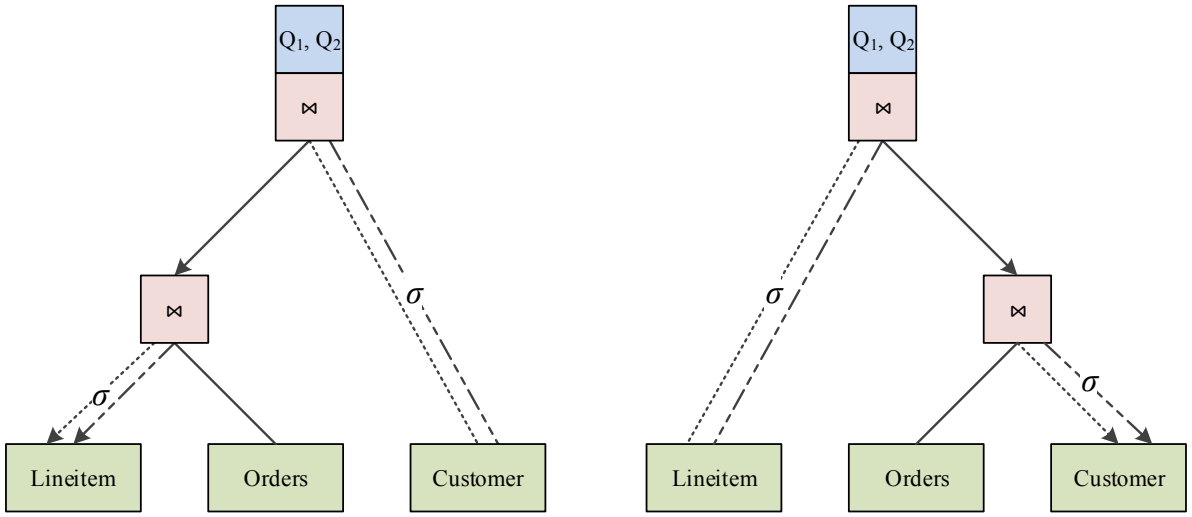
(b) Locally Optimal Plans for Queries



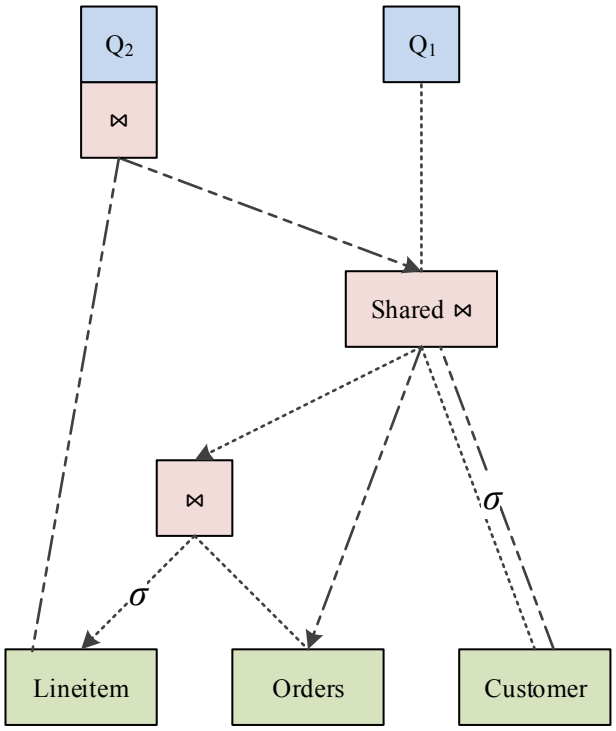
(c) Share Storage Engines

Figure 3.7: Different Approaches to Share Two Query Statements. Arrow lines are build (inner) relations, plain lines are probe (outer) relations.

all the participating statements. To make this clearer, consider the queries of Figure 3.7a. The locally optimal plans for these two queries can be seen in Figure 3.7b. These plans are



(d) Two Possible Shared Plans using Share Everything



(e) An Illegal Sharing of a Join Operator (Source Build Relations are Different)

Figure 3.7: Different Approaches to Share Two Query Statements. Arrow lines are build (inner) relations, plain lines are probe (outer) relations.

generated using single query optimization techniques, and minimize the cost of executing each individual query. There is no work sharing in any operator, and as a result this approach is an example of "Share Nothing". Figure 3.7c illustrates a global plan that

shares only the storage engines. The joins are processed independently and any work that is common, like hashing and probing of tuples as well as materialization, will be performed two times if these queries run concurrently.

Figure 3.7d shows two ways in which these statements can be executed using the Share Everything paradigm. In these cases, a common pipeline of two operators is used to process both queries. The benefit of this approach is clear; a pipeline of just two join operators can perform the work of four operators. Furthermore, common work is performed only once. For example, in both of this plans, the work of probing with the `Orders` relation occurs only once to answer both queries.

Finally, the plan of Figure 3.7e shows a third way that implements work sharing. In this case, the two queries share only one operator that implements the join of `Orders` with `Customers`. The remaining work for each query is not shared. Initially, this approach seems to be more efficient than any of the two plans of Figure 3.7d. The reason is that in this plan the build relations of all join operators are smaller, and the build phase is always more expensive than the probe phase. However, the implementation of such a shared operator will add a considerable overhead. The shared operator is asked to build a hash table with tuples that are of type {`Lineitem` ⋈ `Orders`} and of type {`ORDERS`} at the same time. Even though such a join operator could be implemented, it will require interpretation of every tuple, probably by means of a `switch` statement, in order to distinguish the tuple format. Adding a `switch` in the innermost loop of a hash join, increases the complexity, as well as adds a data dependency in the code. Thus the inner most loop cannot be fully optimized and branch miss-prediction would add a penalty on every tuple. As a result, such a plan is not considered a legal work sharing plan and to our knowledge, none of the studied WS systems allow such graphs in operator graphs.

Out of the three different approaches to work sharing, our algorithm considers mostly hybrid approaches, where the operators that serve similar statements are shared, while operators that serve diverse statements are not shared. The optimizer can also take into account the probability (or the expected probability) of a prepared statement appearing in the workload and decides on whether to share an operator or not. This is not unreasonable even in real-life workloads, as nowadays, most database systems are behind web-services that have well known access patterns. The probability is then used to calculate the expected selectivity at any operator. For instance, in the example of Figure 3.7d, if Query 2 appears with a probability of 0.05 and Query 1 with a probability of 0.95, then the left plan of Figure 3.7d would be more efficient.

3.5 Heuristics

A big difference of work sharing optimization compared to traditional single query optimization is the vast size of the solution space, as explained in the discussion of Section 3.3. A complex workload may require to decide on the order and the share factor of up to hundreds of operators. For instance, the TPC-H, a standardized workload that contains 22 prepared statements, involves 49 join operators. Obviously a global query plan that shares as much as possible can reduce this number to only a handful of operators, like the TPC-H implementation that was presented in Section 2.4.3.

The total number of queries and operators play an important role in the runtime of the algorithm, as the solution space has a size of $O(2^{\mathbb{Q}} * \frac{(2*\mathbb{O})!}{(\mathbb{O})!})$. In order to reduce the size of problem, we introduce two heuristics in this section. The first one simplifies the problem of sharing operators, thus reducing the $2^{\mathbb{Q}}$ factor of the complexity formula. The second heuristic hints the optimizer to order operators in a way that is expected to perform better. This does not reduce the $\frac{(2*\mathbb{O})!}{(\mathbb{O})!}$ fraction, yet the WSO algorithm that is presented in Section 3.6 benefits from such a hint.

The two heuristics have been tested using a set of micro benchmarks in order to verify their correctness. The micro benchmarks will be presented as part of the experimental study in Section 3.8.1.

3.5.1 Sharing Heuristic

The first heuristic simplifies the sharing decision by providing the optimization algorithm with two hints. The first one *suggests* which operators *should not* be shared and the second one which operators *can* be shared across which statements in the global access plan.

The first hint can be easily computed, provided that approximate statistics about the sizes of relations and the cardinalities of all involved attributes exist. The usefulness of this hint is easy to understand: the generated plan should avoid sharing operators across low selectivity and high selectivity subqueries at the same time, as already explained. The second hint requires analysis of each statement. The goal is to find what are the common operators across all statements. Even though this is something that can be computed by the algorithm at runtime, it is better to make this decision at a very early point. The reason is that this information can be reused multiple times during optimization and, as a result, it is better to precompute it. Additionally, it allows the optimizer to explore first the more popular operators. In contrast to the first hint, the second hint should only provide a guideline that can be ignored by the optimizer. For instance, the example of

Figure 3.7e is one of the cases where this *can be shared* hint should be ignored: the heuristic would suggest that the ‘‘`Shared` ⋈’’ operator can be shared, yet the constraints of the system do not allow this.

The sharing heuristic analyzes all operators involved in all statements individually. First, it classifies all storage engine access for each query. We distinguish two classes: `H`igh Selectivity and `L`ow Selectivity. Subqueries that ask for the whole relation or request for big ranges of the relation are classified as `L`, while point queries or subqueries that ask for smaller ranges are classified as `H`. The assignment of a subquery to class `H` or `L` depends on the underlying implementation of the WS System. If the storage engine uses B-Tree lookups to access tuples, then the cost of retrieving and accessing all tree nodes should be taken into account. If the storage engine is implemented using full table scans, like for instance Crescando, then the cost of retrieving 1 tuple or 100 tuples is very similar. In any case, properly identifying the threshold between `L` and `H` requires careful micro benchmarking of the specific WS System and its operators.

In the example of Figure 3.7d, `Lineitem` access is `H` for query `A` and `L` for query `B`. Then, all higher level operators are classified accordingly. This means that three different classes for join operators are taken into account: `H-H`, `L-L` and `L-H`. The symmetrical class of the latter, the `H-L` (i.e. build on a big set of tuples and probe on a smaller) is not taken into account as it is always suboptimal. In this case, we always swap the build side with the probe side. This is the first hint of the sharing heuristic.

The classification of joins always takes into consideration what can be filtered. For example, in query `A` of Figure 3.7d, the `Customers` ⋈ `Orders` is classified as `L-H`, as there is the potential to filter the `Orders` relation, if it is previously joined with `Lineitems`. In order to extend the heuristic to other flavors of join operators besides hash joins, we have to take into account the properties of the specific join algorithm. For instance, sort merge joins should only allow `L-L` or `H-H` classifications, as sort joins perform better if the sizes of the joined relations are similar.

The classification of operators alone is enough to provide a rough guideline of when operators should be shared or not. Operators that belong to different classes should never be shared, as this will result in too much unnecessary work for some queries. For instance, consider the left plan of Figure 3.7d. In this case `Orders` ⋈ `Lineitem` is `L-H` for Query `A` and `L-L` for Query `B`. Our heuristic suggests that this join should not be shared across these queries. Instead, the operator has to be replicated and one instance should be used exclusively for Query `A`, while the second one should be dedicated to Query `B`, as show in Figure 3.7c.

In most cases, the sharing heuristic manages to reduce the solution space by an important

factor. To quantify, consider a "popular" join operator that is part of *ds* different prepared statements. Popular operators are very common in analytical workloads, where the fact tables are always joined with some dimension tables. An exhaustive search optimization algorithm will have to make the decision on which of these *ds* statements should share the operator. This creates a combinatorial problem with a total of $2^{ds-1} - 1$ combinations (the combination that no operator shares the join is void). The Sharing Heuristic groups queries into 3 different groups. Queries that belong to different groups *should not* share the same operator, which reduces the number of combinations to $2^{\frac{ds}{3}-1} - 1$, assuming the average case of uniform distribution.

Even though this heuristic seems quite abrupt, it always hints towards a better solution. In Section 3.8.1 we present experimental results on the effectiveness of this heuristic.

3.5.2 Ordering Heuristic

In addition to the sharing heuristic, our work sharing optimizer uses a second heuristic, the ordering heuristic, which provides the optimizer with a hint on how to explore the solution space. The ordering heuristic uses the same operator classification as the sharing heuristic and it is able to make a draft decision on the ordering of operators in the global query plan.

The heuristic exploits the observation that if a sequence of join operations is requested, executing the smaller joins (in terms of cost and number of tuples) first results in faster runtime. The benefits come mainly from the fact that using smaller sets of tuples first, we essentially perform more filtering at the early stages. This means that fewer tuples have to be joined and materialized.

In order to faster evaluate the heuristic, we reuse the classification of the Sharing Heuristic. Based on it, the optimizer will explore first the plans that have `H-H` hash joins on the lower level, `L-H` joins on the higher and the `L-L` join on the top level of the query plan. This means that before executing any big join (`L-L`), we have filtered as much as possible the inner and outer relations.

The Ordering Heuristic is not reducing the solution space by omitting sub-optimal decisions, but instead gives a greedy direction on how to start exploring the solution space. The worst case is that all solutions need to be explored. Nonetheless, our experimental results show that the ordering heuristic helps in converging towards the best solution faster.

3.6 Work Sharing Optimization Algorithm

In this section we present the work sharing optimization algorithm. The goal of the algorithm is to produce a globally efficient shared access plan, given a whole workload. In this context, globally dictates that we are interested in the cost of executing the whole workload, rather than each query independently.

Work Sharing Optimization is a non-convex bilinear global optimization problem. The objective (cost) function contains a lot of local minima. For instance, the best way to execute a single query is a local minimum. Additionally, subproblems are not independent of each other. The decision to share a hash join across two queries affects the cost of both queries. Consider a join across two relations, `A` and `B`, and two queries. `Q1` asks for a join of the set of tuples $\sigma_1(A)$ with the set of tuples $\sigma_1(B)$, while `Q2` asks for the join of $\sigma_2(A)$ with $\sigma_2(B)$. If the operator is shared, we have to calculate the cost of joining $\sigma_1(A) \cup \sigma_2(A)$ with $\sigma_1(B) \cup \sigma_2(B)$. Because of the union operator, if the selection are the same for both queries, then the shared join will have exactly the same cost as the not shared joins. Reordering join operators also affects the overall cost and may require to un-share certain operators, as already explained in Section 3.4.

The optimizer algorithm is based on the branch and bound optimization method. Branch and bound is used by most optimization solvers due to its effectiveness, especially if the type of optimization is discrete [KL10]. In branch and bound the solution space is greedily explored under the assumption that there is a theoretical bound in the solution. The theoretical bound is usually the solution to the same problem with relaxed constraints. Using this bound as a guide, the method proceeds by calculating the cost of intermediate illegal solutions (i.e. solutions that violate some of the constraints) and modifying the variables until a valid solution is reached.

Before presenting the algorithm, we will define the objective function to be optimized, as well as identify the bounded problem and the algorithm to estimate the cost of bounded solutions.

3.6.1 The Objective Function

The cost or objective function quantifies how good a solution is over other solutions. In most existing single query optimizers, the objective function consists of two variables: the number of I/O operations required and the CPU cost. These two metrics were sufficient for single query systems, where the queries had to be executed as fast as possible.

On work sharing systems these metrics are not applicable as multiple queries are executed concurrently. A shared full table scan for instance, requires a lot of I/O operations, but now it evaluates multiple queries. Adding another query in the load will not increase the number of I/O operations. The same applies to all shared operators in all WS systems that implement them. What is important in WS systems is the number of tuples that have to be processed. Since some of these systems share resources across queries, the cost function is not linear. For example, if a query in the mix requests a full table scan of n tuples while another one requests a single tuple (point query), the total number of tuples fetched is going to be n instead of $n + 1$.

As a result, the objective function of our optimizer is based only on tuple count. The number of tuples is then multiplied by a factor that is different for each operator. For example, full table scans have a factor of 1 per tuple, while key value lookups have a factor that is equal to the number of tuples that fit in a (memory or disk) page. Hash joins have different factors for the build and probe phase. For the hash join implementation used in SharedDB, the build phase has a factor of 4, because four memory reads and writes are required to read and insert a tuple in the hash table. On the other hand, the probe phase has a factor of 1. These factors originate mainly from experimental micro benchmarks, which is part of micro tuning the objective function for a particular WS system. Applying this algorithm to different WS Systems, requires proper analysis of each individual operator.

Finally, the goal is to minimize the objective function. The optimal solution should have the minimum number of processed tuples, while answering the whole given workload.

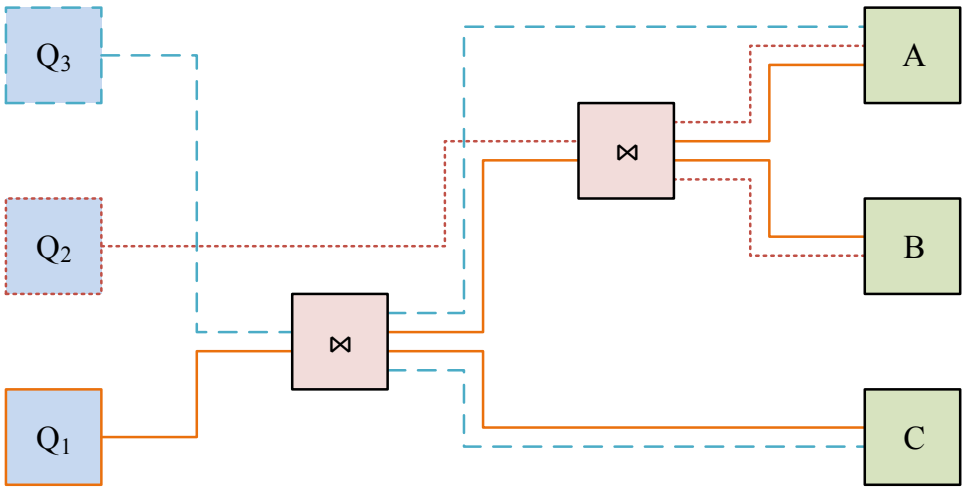
3.6.2 Identifying the Bound

As already explained before, the bounded solution of the B&B method is the optimal solution for a relaxed problem, thus an invalid solution for the actual problem. The bounded solution should have a fast runtime as it is used only as an approximation of how good each step is. Of course, choosing a very relaxed problem increases the number of iterations required, as most of the intermediate steps will be very suboptimal compared to the approximation.

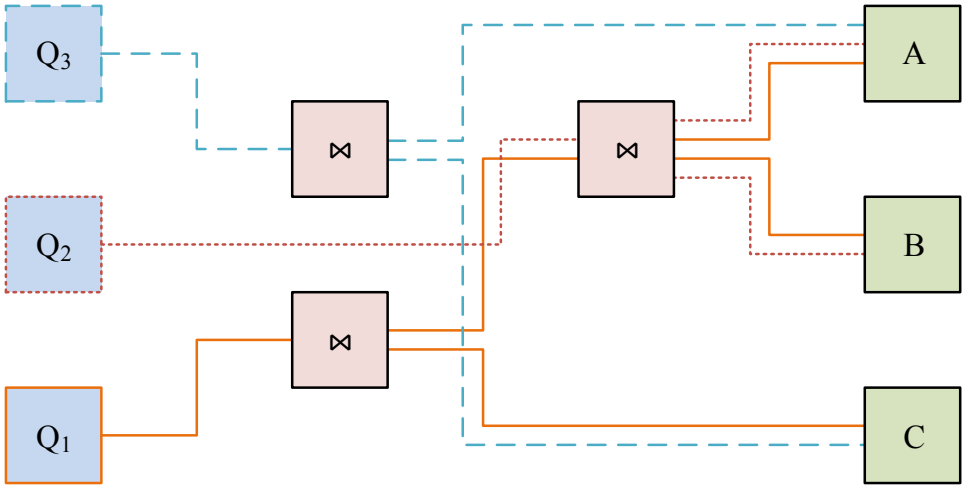
The bounded problem that we used in our optimizer originates from the same optimization problem with the relaxed constraint that every operator can and will be shared across all queries. A solution to the bounded problem violates the constraints that were described in Section 3.4.1. Essentially, the bounded problem is the overlapping of the access plans of all the queries, with no limitations. For instance, consider a workload with three queries,

```
Q1  SELECT * FROM A, B, C WHERE A.id = B.id AND A.id = C.id;
Q2  SELECT * FROM A, B WHERE A.id = B.id;
Q3  SELECT * FROM A, C WHERE A.id = C.id;
```

(a) Sample Queries



(b) A Solution to the Relaxed Problem (Lower Bound)



(c) A Solution to the Work Sharing Optimization Problem. The Cost will be Higher.

Figure 3.8: Example of bounded and unbounded (invalid) problems.

as shown in Figure 3.8a. `Q1` requires two joins, `A ⋈ B` and `A ⋈ C`, `Q2` requires `A ⋈ B` and `Q3` requires `A ⋈ C`. The relaxed problem allows both `A ⋈ B` and `A ⋈ C` to be shared, shown in Figure 3.8b. In contrast, the original WSO problem would only allow one of these joins to be shared. Sharing `A ⋈ B` means that we cannot share `A ⋈ C` (Figure 3.8c), because `Q1` is now interested in the join `(A ⋈ B) ⋈ C`, rather than the join `A ⋈ C`.

In order to reduce the complexity of the bound function, we approximate that an operator that has to execute a set of n queries $\{Q_1, Q_2, ..., Q_n\}$ that ask for $T_1, T_2, ..., T_n$ tuples respectively, has a cost that is equal to $max(T_1, T_2, ..., T_n)$. This does not reduce the effectiveness of the algorithm. The optimal solution can still be reached, as in branch and bound, the bound function is only used to hint on which solution tree should be followed next. Since the bound function calculates the cost of a relaxed problem, the bound cost will always be lower than the actual cost. This means that if two partial solutions are given, we can use the bound function to estimate which of these two is more promising to lead to the valid solution with the lower cost.

The relaxed problem is a combinatorial optimization problem that can be solved with traditional single query optimization techniques, like dynamic programming. Memoization can be applied as the subproblems and decisions are independent of each other. As we explore the solution space of the relaxed problem, we keep the cost of every subproblem. For instance, in a workload that requires 5 hash joins, we will keep the minimum cost of all two-way, three-way and four-way joins. This will help us bound the solution in the branching part of the algorithm. As the B&B visits intermediate (invalid) solutions, i.e. solutions that validate some of the constraints, it can estimate how good this intermediate solution is. The cost of every intermediate solution is equal to the actual cost of the part of the solution that is valid, plus the theoretically limited cost of the invalid solution.

3.6.3 Branch & Bound Algorithm

In this section we present the proposed work sharing optimization algorithm that is based on the branch and bound technique. The algorithm uses "nodes" to keep intermediate states. There are three types of nodes: the solution nodes, the live nodes and the dead nodes. A solution node contains a solution to the problem without violating any constraint. The cost of a solution node comes directly from the cost function. The algorithm may reach multiple solution nodes as it explores the solution space. The solution with the lowest cost is the output of the algorithm.

Live nodes contain the solution to some problem that violates some constraints and they can be expanded into other nodes that violate less constraints. Once expanded, a live node is turned into a dead node, meaning that we do not have to remember it any more. In order to calculate the cost of a live node, we use the cost function for the part of the solution that does not violate the constraints, and the bound cost function for the remaining part of the solution. A feature of branch and bound is that once we have reached a solution node, we can prune all live nodes that have a cost higher than the cost of the solution node. This does not affect the optimality of the algorithm because the cost of a live node

means that as we explore this node and fully expand all children, we will never reach a solution with a lower cost. In other words, the cost of a live node is the theoretical bound of the subtree of nodes.

The algorithm, described in Figure 12, uses a heap to maintain the set of live nodes sorted by their cost. The first node that enters the heap is the root node, a node that contains the solution of the relaxed problem described in Section 3.6.2. The root node dictates that all expanded sub-nodes will have a cost that is equal or higher to the cost of the root node. The algorithm proceeds by removing the first node of the heap, and testing if it is a valid solution or not. In case it is a solution, and it is cheaper than any solution we have seen before, we keep it. In the case that the active node is not a solution, it is further expanded and all child nodes are inserted into the heap. As already explained, we can prune a node and the whole subtree if it has a cost that is higher than the best solution seen so far. This is because expanding a node will result in nodes with a cost equal or higher to the one of the parent node.

The algorithm implementation uses a two dimensional matrix to store the state of a node. The first (horizontal) dimension describes which operator is used, while the second one the queries that share this operator. Additionally every node remembers which part of the bitmap is the valid solution and which has yet to be expanded. The root node has no valid operators, as constraints are violated.

Figure 3.9 shows an example of how our algorithm explores the solution space by expanding nodes. The root node holds a matrix that describes which queries use which operators. The values of the matrix are tristate. A value of '`1`' means that the respective query is interested for this operator, and in this node it is using it. A value of '`0`' means that the query is interested for the operator, but in this step it is not using it. Finally, a value of '`-`' means that the query is not interested in this operator.

In Figure 3.9b expanding the root node results in three child nodes. These child nodes have the first operator *set*, meaning that as they expand, this part of the solution will remain constant. Their cost is calculated by combining the cost of the set operators using the cost function, with the cost of the remaining operators using the bound function.

During node expansion all constraints must be taken into account. In our example the expansion of the root node creates only 3 child nodes (Figure 3.9b). The `D` $\bowtie$ `E` operator is not part of the permutation, because it is a `L-L` operator and the ordering heuristic suggests that the selective operators should be executed first. Also, as node #2 expands into node #2.1, the `A` $\bowtie$ `B` operator is duplicated (Figure 3.9c). This is done in order to adhere to the sharing constraint that was presented in Section 3.4.1. The first replica serves only `Q1` and `Q3` and it is able to join tuples of `(BC)` with tuples of `(A)`, while the

Algorithm 12: Multi Query Optimizer Algorithm

```
Data: Heap heap;                                  /* a cost-sorted heap of all live nodes */
Data: Node boundSolution;                         /* the relaxed problem solution */
Data: Node e;                                     /* the currently expanded node */
Data: Node solution;                              /* the solution node */
Function BranchAndBoundWSO():
    e ← boundSolution;                            /* Create the Root Node */
    e.validOperators ← 0;
    e.cost ← BoundCostFunc(boundSolution);
    Push(heap, e);
    solution.cost ← ∞;
    while ¬ IsEmpty(heap) do
        e ← Pop(heap);
        if IsValidSolution(e) then
            /* The current node is a valid solution. Keep it only if it is the cheapest
               so far.                                                              */
            if e.cost < solution.cost then
                solution ← e;
            end if
        else if e.cost < solution.cost then
            /* If the current node is cheaper than the best solution, expand it and
               look for better solutions.                                           */
            Expand(e, h);
        end if
    end while
    return solution;
end
Function IsValidSolution(Node e):
    if e.validOperators = e.totalOperators then
        /* If the sequence of all operators is valid, we have reached a solution   */
        return true;
    end if
    else
        return false;
    end if
end
```

Algorithm 12: Multi Query Optimizer Algorithm (continued)

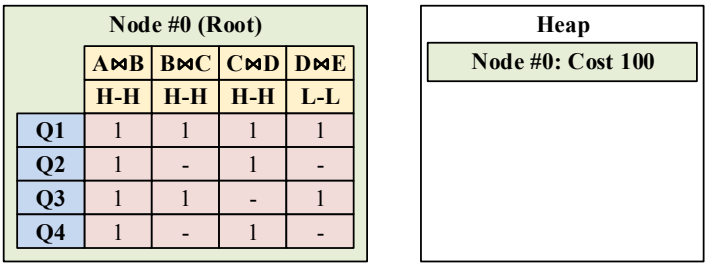
```
/* Branch into a Node and calculate the costs of all Sub-Nodes */
Function Expand(Node e, Heap h):
    Node[] children ← ChildrenOf(e);
    foreach Node c ∈ children do
        /* On each branch, the number of valid operators increases. */
        c.validOperators ← e.validOperators + 1;
        c.cost ← CalculateCost(c);
        Push(heap, c);

Function CalculateCost(Node e):
    Result: Cost cost ;                    /* The cost of the given node */
    /* Find the cost of the valid part */
    Cost c1 ← CostFunction(c, c.validOperators);
    /* Find the bounded cost of the invalid part */
    Cost c2 ← BoundFunction(c, c.validOperators);
    return c1 + c2
```

second replica serves `Q2` and `Q4` and it is able to join tuples of `(B)` with tuples of `(A)`. For reasons of simplicity, we have not taken into account different build and probe orders in this example.

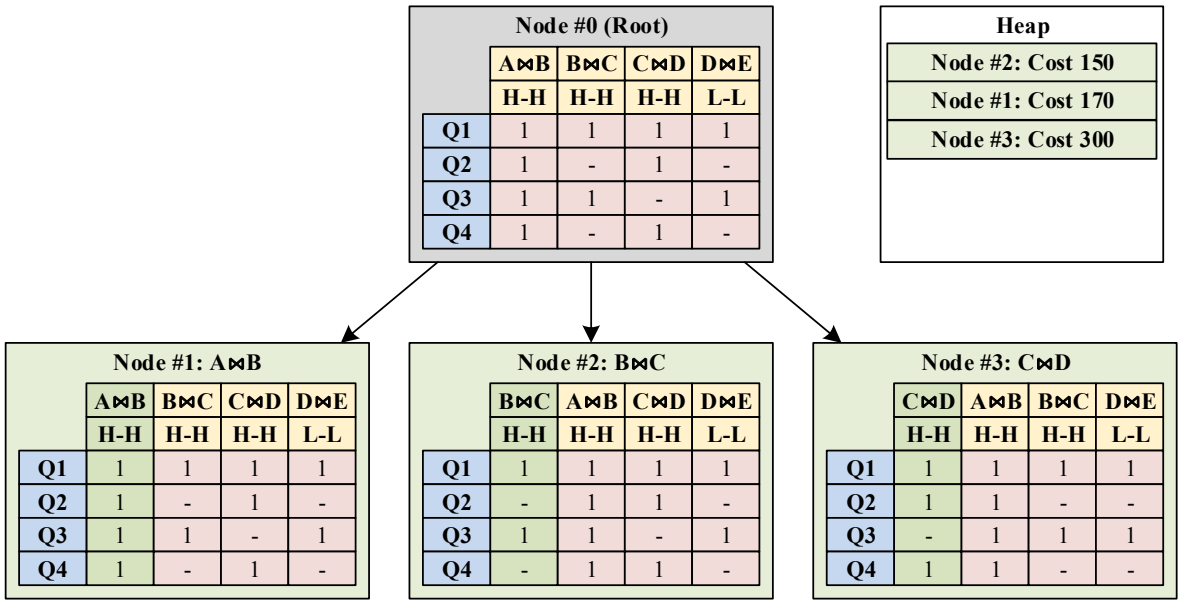
Finally, at this point we should notice the importance of the heuristics. Without the sharing heuristic, node #1 would be split into 8 different nodes that contain all the possible values of the bitmap $\{1, 1, 1, 1\}$. The optimizer would have had to consider instantiating the `A` $\bowtie$ `B` operator just for query 1 ($\{1, 0, 0, 0\}$), then just for query 2 ($\{0, 1, 0, 0\}$) and so on, for a total of 2^4 nodes. The benefits of the ordering heuristic is less obvious. The root node expands only to 3 nodes instead of 4, as the hint says that the `L-L` joins should be executed after the `H-H` joins.

3.7 Implementation and Integration with SharedDB

In this section, we give an overview of our reference implementation of the work sharing optimizer as well as details on how the optimizer integrates with SharedDB. The optimizer has been developed in Java, mostly because of the automatic memory management and garbage collection that Java provides. This is a necessity, as the WSO algorithm may visit a couple of thousands state nodes before converging to a solution node. The algorithm



(a) Root node. A `'1'` marks which queries are interested in which operator.

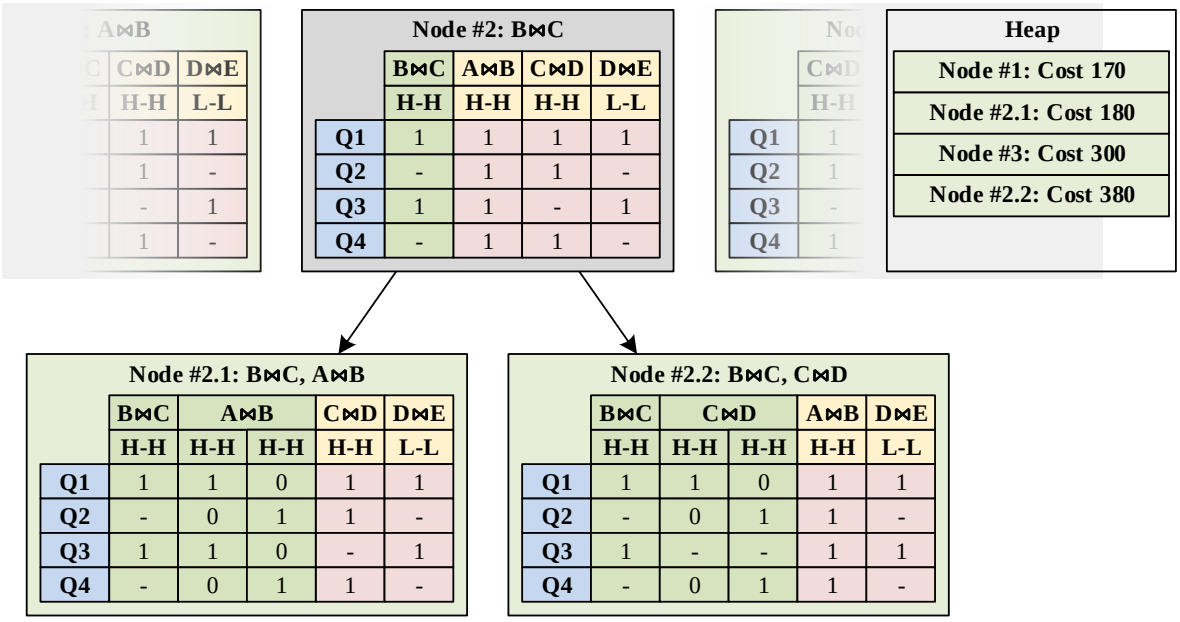


(b) Root node expands into 3 child nodes and is marked as dead. The new nodes have the first operator fixed. A fourth node (`D ⋈ E`) is not created, as it is classified as `L-L`. All `H-H` operators have to be used first. The cost of these nodes is calculated using $costFunction(fixedPart) + boundFunction(remaining)$. A heap maintains all live nodes in increasing cost. The next node that should be explored is node 2.

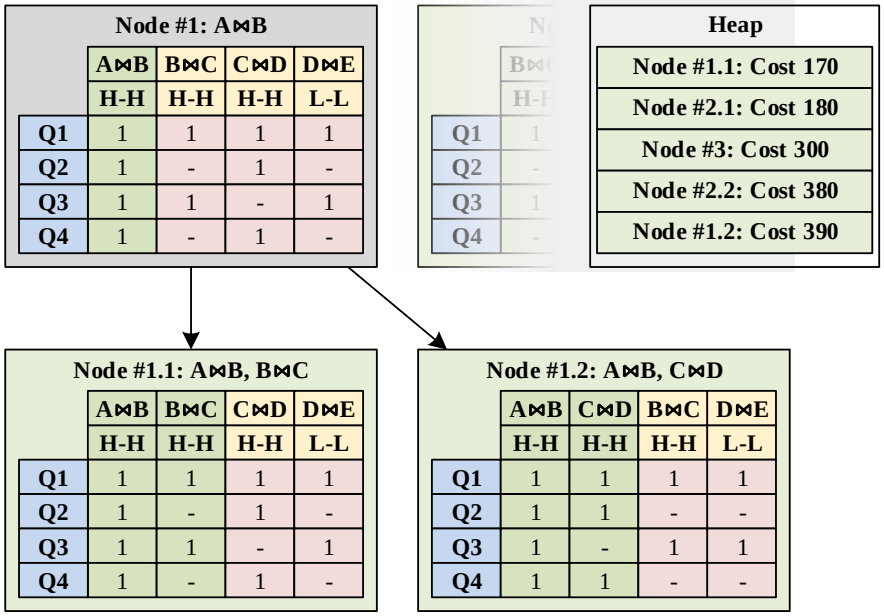
Figure 3.9: Work Sharing Optimization Algorithm at Runtime.

itself was implemented in roughly 24,000 lines of code. The codebase of the implementation is substantially larger than the pseudo-code in the previous section might suggest, because a lot of abstractions were used in the pseudo-code.

Since this is an optimizer that parses the whole workload and generates a plan, the input of the optimizer is a set of statements. The optimizer is able to parse SQL commands, analyze them and create a Java representation for each statement. This internal Java representation of a statement is a tree of operators with a set of parameters for each operator. The tree representation allows to easily reorder operators within each statement.



(c) Node 2 expands into 2 child nodes and is marked as dead. Both children require replication of the new operator. A '0' in the node state represents that the operator is used, but will not answer the query.



(d) Node 1 (the most promising node) expands into 2 child nodes and is marked as dead. The process continues until a branch reaches a valid solution (i.e. all operators marked green), and no other (leaf) node has a lower cost.

Figure 3.9: Work Sharing Optimization Algorithm at Runtime (continued).

The statements are classified using the two heuristics that were presented in Section 3.5 and then the branch and bound WSO algorithm is executed to generate the globally optimized query plan. Similar to the statements, the global query plan is represented as a graph of operators. Then the statements are adapted to describe a path in this graph of operators. After this step, the task of optimizing the workload is complete and the global query plan is communicated and implemented in the SharedDB engine.

The SharedDB core is implemented in C++ and makes heavy use of template metaprogramming. All SharedDB operators are implemented as C++ templates in order to take full advantage of compile time optimizations. For instance, a sort operator can be implemented with the following template:

```
typedef SortOperator<int, 0, true> SortCustomersOperator;
```

In this example, the `SortCustomersOperator` will process integers that are positioned in the 0th byte of the input record. The operator will sort the result set in ascending (`true`) order. SharedDB implements most of the storage engine and data processing operators using such templates. This is the equivalent of hard-coding these parameters into the source code of a sort operator, yet they allow to create different types of sort operators (i.e. on a date), by declaring different template types.

Similar compile time optimizations are implemented in a number of commercial systems and research prototypes [Neu11]. Compile time optimizations minimize the data dependencies in the code, allowing for less branch mispredictions and higher instruction locality. Consider for instance a generic sort operator that is not specialized for integers at byte offset 0. Such an operator will have to use a `switch` statement as part of processing every incoming tuple in order to decide on the comparison operator. Alternatively, it has to make a virtual function call to a comparison function pointer for every incoming tuple.

Compile time optimizations have significant gains, however they can only be applied when the software is compiled. To allow our optimizer to take advantage of compile time optimizations, we implemented two complementary components for the optimizer; a code emitter and a code compiler.

The code emitter component takes as input the optimized global query plan and generates a set of C++ source and header files that implement the given plan. This requires translation of operator parameters to hard coded values. For instance, an `Aggregate` operator that calculates the `SUM(ITEMS.value * ITEMS.tax)` will have the whole aggregate expression expressed as a template. Thus, no interpretation will take place during runtime.

Once the required source files have been generated, the code compiler component calls the system-wide C++ compiler and generates a set of dynamic libraries for all the operators

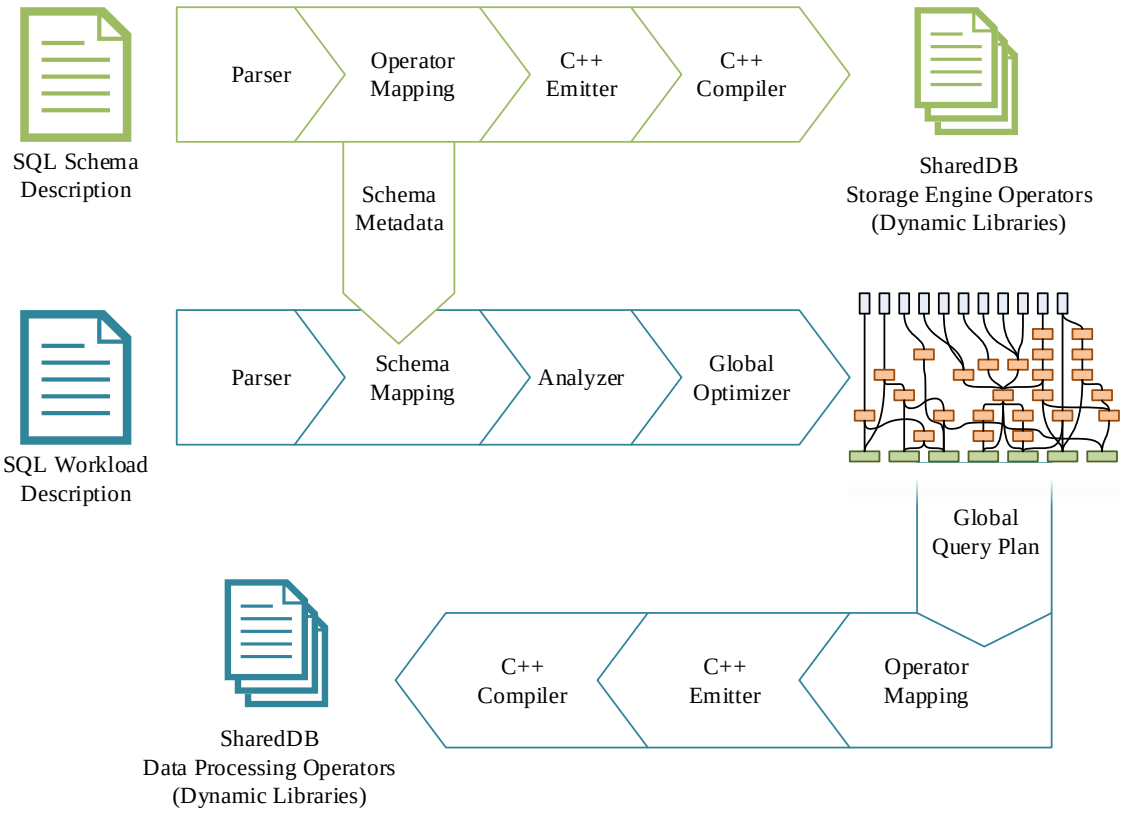


Figure 3.10: WSO Optimization Process and Integration with SharedDB

that are part of the plan. This component takes care of all the fine details that are required for compilation. For instance, dependencies across operators are handled automatically.

Finally, the SharedDB core has been modified to allow to dynamically add or remove operators at runtime by loading and unloading the generated libraries. This allows to setup a schema first and then load all operators required to evaluate a workload. Another application of this feature is that it enables dynamic modifications of the global query plan. If the dataset or the workload evolves, a database administrator can trigger a reoptimization. The new global query plan can be applied to a running instance of SharedDB, without taking down the system. Figure 3.10 shows the whole WSO optimization process, as well as the integration with SharedDB.

3.8 Experimental Analysis

In order to evaluate the performance of our work sharing multi query optimization algorithm, we carried out a series of performance experiments. First we present experimental results to validate the correctness of the two heuristics of our algorithm and then we present results on the generated plans of two well known workloads, the TPC-W and the TPC-H benchmarks.

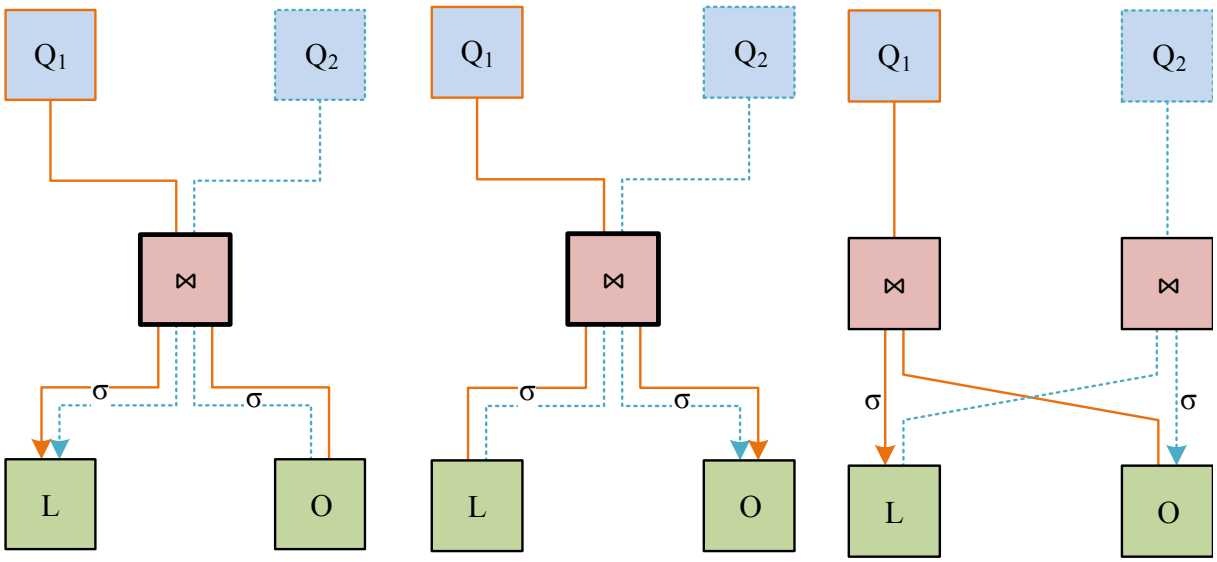
The baseline we compared against is an optimization technique based on multi query optimization, where only sub-expressions of the same prepared statements are executed together (predicate sharing). Executing the same workload on a query-at-a-time database system, like MySQL, would be fruitless. WS systems outperform query-at-a-time systems, especially under high load, as already shown in the experiments of Section 2.4 and related work [GAK12, HA05]. We decided not to perform any tests with a query-at-a-time system, as we seek to optimize entire workloads info an globally efficient access plan.

The generated plans, as well as the baseline plans were implemented in SharedDB. For the baseline plans, we introduced the additional constraint that each batch can contain only queries that originate from the same prepared statement (i.e. they have common sub-expressions). The sharing that occurs in baseline plans resembles the one of Figure 3.7c, while the WSO algorithm shares more work by creating plans similar to the ones of Figure 3.7d. In order to generate the baseline plans, we used dynamic programming to optimize each individual query. Then we combined the locally optimal plans to create the global query plan. This two step optimization approach is how traditional multi query optimization can be implemented in work sharing systems, as suggested in [GAK12].

The global query plans were generated on a dual core laptop, as the runtime of the optimization algorithm is very small. We discuss the exact optimization time of each experiment as we present the results. The same laptop was used to generate the baseline query plans. In all experiments reported in this section, we used a multi core machine to evaluate the efficiency of the generated plans. This machine features four eight core Xeon CPUs. Each core has a 2.4 GHz clock frequency and is hyper threaded. The hyper threaded cores were used to run the clients. This did not affect the performance in any benchmark, as clients mostly wait for the results to arrive and perform no additional work.

3.8.1 Heuristic Tests

The first part of our experimental study focuses on the correctness of the heuristics used in our algorithm. We used two small micro benchmarks, one that involves a two way join



(a) Shared: Build L, Probe O (b) Shared: Build O, Probe L (c) Not Shared

Figure 3.11: Different Plans to Evaluate Two Statements That Join Two Relations

across two relations, and one that involves two overlapping three-way joins. In both cases, we used two prepared statements with very diverse selectivities.

3.8.1.1 Two Way Joins

For the first micro benchmark, we used two relations, `L` and `O`. `L` was loaded with approximately 1 GB of data that were generated from the `LINEITEM` relation of TPC-H, while `O` was loaded with approximately 230 MB of the `ORDER` relation. We considered a natural join across these two relations and issued a workload of the two following prepared statements:

PS1 `SELECT * FROM L JOIN O on L.o_id = O.id WHERE L.part = ?;`

PS2 `SELECT * FROM L JOIN O on L.o_id = O.id WHERE O.id = ?;`

The query parameters were also generated based on the TPC-H specification. We used a total of 160 client threads that issued queries generated from these prepared statements in a closed loop.

Based on our operator classification presented in Section 3.5.2, these two statements have a `L-H` join. We considered three different global plans to execute this workload, as shown in Figure 3.11. The first plan creates an operator that builds a hash table with tuples from the `L` relation and probes with tuples from `O` and shares the operator across both statements. The second one builds on `O`, probes on `L` and shares the operator across both statements as well. Finally, the third possible plan uses both of these join operators, one

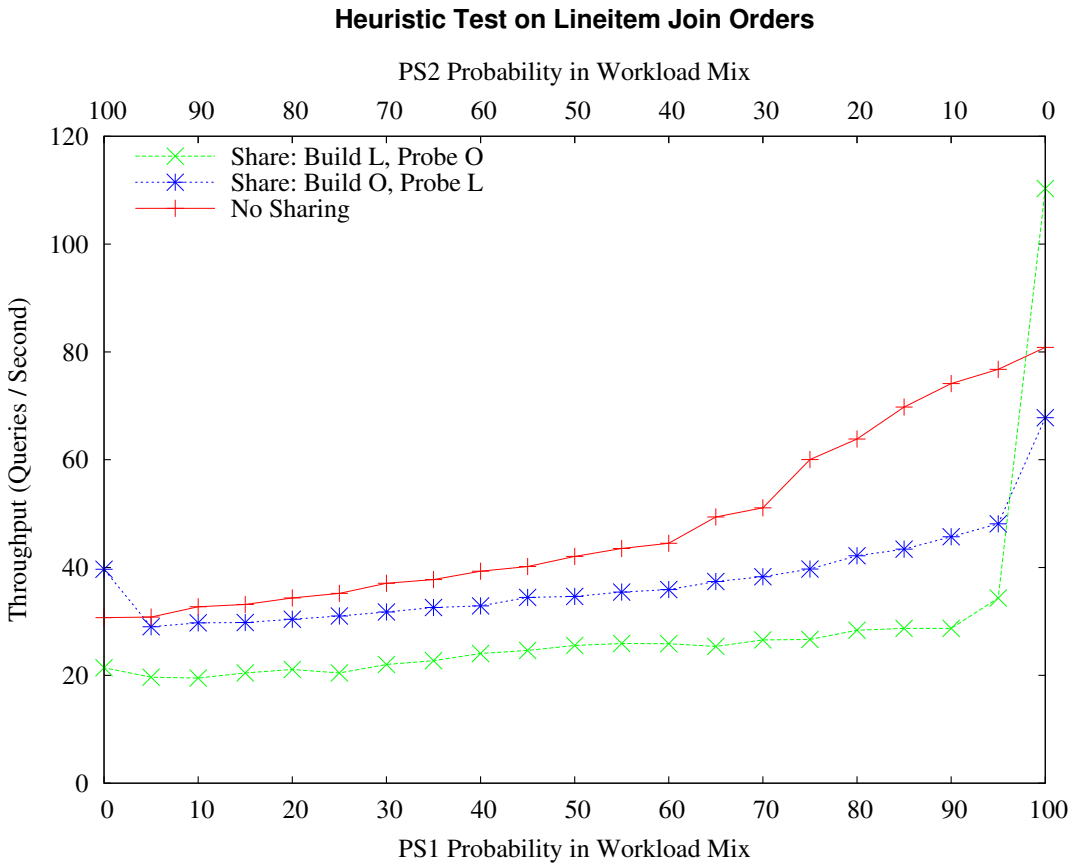


Figure 3.12: Experimental Results on a 2-Way Join to Evaluate the Sharing Heuristic.

PS1	PS2	Maximum	Optimal	Actual
0	10	13,500,000	13,500,000	13,500,000
5	5	78,005,035	30,004,860	30,004,860
10	0	54,010,935	54,010,935	54,010,935

Figure 3.13: Amount of Shared Work for 10 Queries for a 2-Way Join.

for `PS1` and one for `PS2`. In order to have a fair comparison across these three plans, we used the same amount of resources in all plans, in terms of number of CPU cores and memory buffers.

We varied the probability of a prepared statement appearing in the workload and measured the achieved throughput on all possible plans. Figure 3.12 shows the results. In the left-most extreme, where no queries generated from `PS1` were executed, the second plan is better, while in the right-most extreme, where only queries of type `PS1` were executed, the first plan is better. However, in these extreme cases we would never consider work

sharing. If the probability of PS1 is 0, there is no need to create a shared operator that would serve PS1 and PS2.

The area of the plot between the two extremes is the interesting region. The "No Sharing" plan has a higher performance in this area. This means that whenever two queries with diverse selectivities are present, the optimizer should never attempt to share the operator across them. This is exactly the hint that the sharing heuristic provides, as the join in the two statements belongs in different classes: the first one is L-H, while the second one is the complementary H-L.

Finally, the table of Figure 3.13 shows the number of shared tuples while executing a batch 10 queries. We used three scenarios based on how many of these queries originate from PS1 and how many originate from PS2. We evaluated the optimal sharing by computing all possible plans and their respective shared tuples. This is possible as the solution space of this microbenchmark is very small. We see that for this specific micro benchmark, the decision taken by the optimizer is equal to the optimal.

3.8.1.2 N-Way Joins

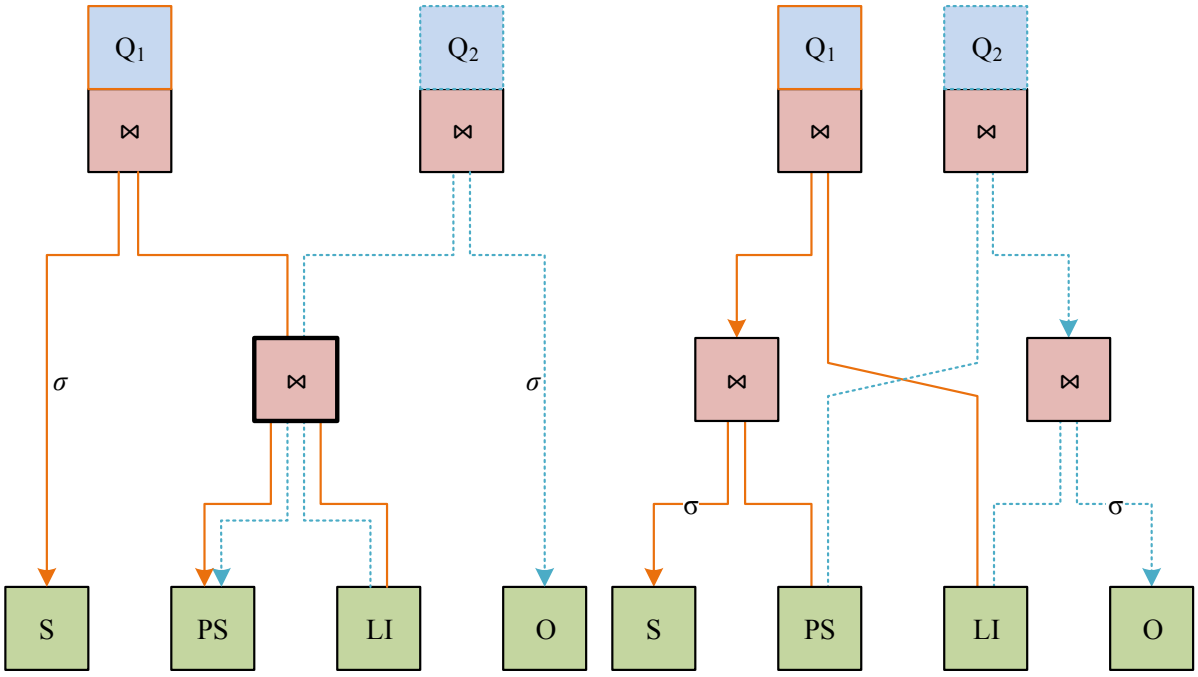
The next micro benchmark studies the correctness of the heuristic in more complicated scenarios. We used a schema consisting of four relations taken from the TPC-H benchmark, Supplier, PartSupp, LineItem and Orders. Similar to the 2-way experiments, we used a workload of the following two statements:

PS1
```
SELECT * FROM S JOIN PS on S.id = PS.s_id
                JOIN LI on PS.ps_id = LI.ps_id WHERE S.id = ?;
```

PS2
```
SELECT * FROM O JOIN LI on O.id = LI.o_id
                JOIN PS on LI.ps_id = PS.ps_id WHERE O.id = ?;
```

To answer this workload, we considered the two global plans of Figures 3.14a and 3.14b. The first plan uses a shared operator to evaluate Lineitem ⋈ PartSupp. This is against the sharing heuristic, as in the first prepared statement, tuples from PartSupp can be filtered, while in the second prepared statement, tuples from LineItem can be filtered. Also, the ordering heuristic would instruct the optimizer to execute LI ⋈ PS at the end, as it is a L-L join. In order to ensure fairness in our comparisons, we assigned twice the amount of resources (CPU cores, buffer size) to the shared operator of Figure 3.14a.

We measured the throughput of the two global query plans, as we modified the probability of a statement appearing in the mix. The results are presented in Figure 3.15. The results show that not sharing the LI ⋈ PS operator and executing it after filtering as many tuples as possible is better for all cases, except for the extreme case where only queries of the



(a) Plan with Shared `PS` ⋈ `LI` (b) Plan with no Sharing

Figure 3.14: The Two Considered Plans for the N-Way Join Micro Benchmark.

second statement are executed. In this case, the two cores that were assigned to the shared operator, give an advantage to the shared plan. Yet, if the optimizer knows that `PS1` will never appear in the workload, there will be no need for a decision of sharing this join or not. Additionally, the number of shared tuples when executing 10 queries can be seen in Figure 3.16. As with before, the measured shared tuples match the optimal, something that validates the correctness of the heuristics.

3.8.2 TPC-W Analysis

As seen in the previous experiments, greedily sharing work sharing is not optimal. The two heuristics of our algorithm make sure that we do not share too many operators. In most realistic workloads, removing these cases still leaves enough opportunities for sharing. To demonstrate this, we used the TPC-W workload with 10,000 items which contains a total of 11 prepared statements that use a total of 17 join operators. The prepared statements were issued based on the frequencies defined by the TPC-W Browsing mix. The query parameter were generated as defined in the TPC-W specification. What makes this workload interesting is that it contains queries that have deep pipelines of operators (analytical queries) as well as short running queries that join only a couple of tuples every

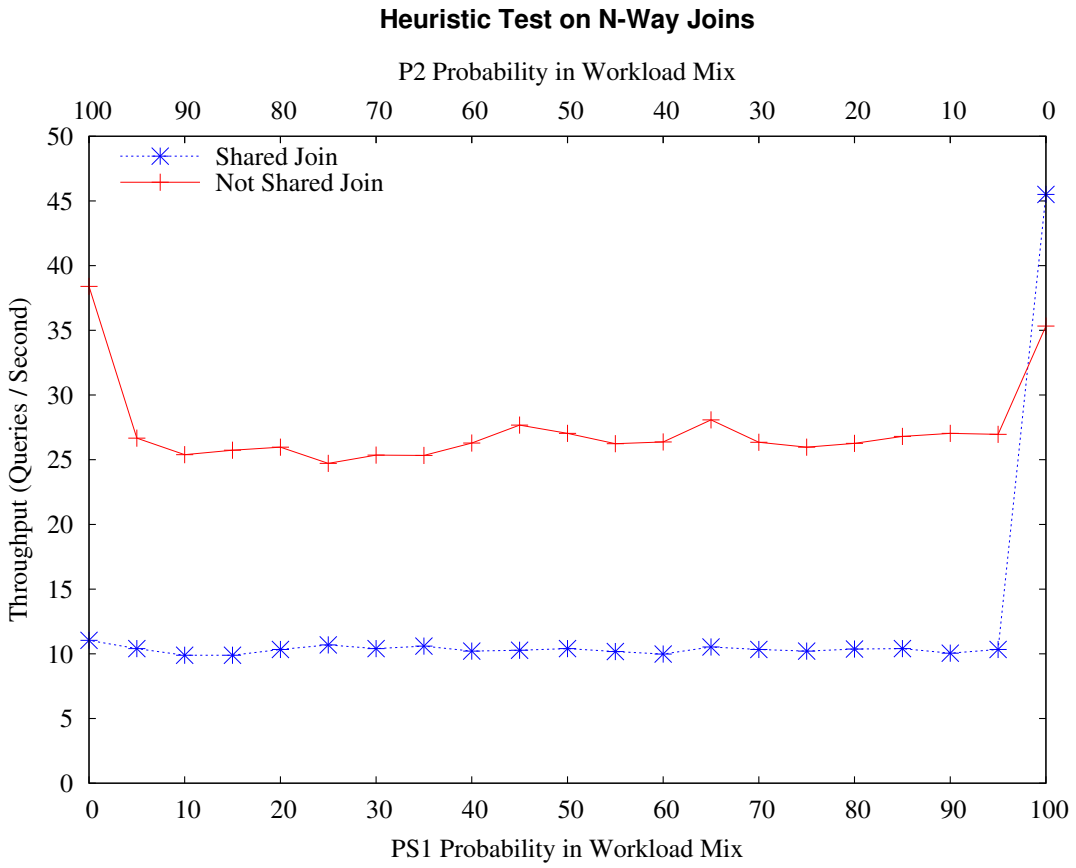


Figure 3.15: Experimental Results on a N-Way Join to Evaluate the Heuristics.

PS1	PS2	Maximum	Optimal	Actual
0	10	61,210,935	61,210,935	61,210,935
5	5	122,421,870	54,409,720	54,409,720
10	0	61,210,935	61,210,935	61,210,935

Figure 3.16: Amount of Shared Work for 10 Queries for a N-Way Join.

time. In order to focus only on the query plan efficiency, we used only the prepared statements and skipped the web interface that is part of the TPC-W specification.

Since our implemented algorithm focuses on hash join and scan operators, we removed all other relational operators from the prepared statements. All selections were executed as part of the scans. Yet, this does not limit the applicability of our algorithm. The same technique can be easily applied to all relational operators, like `SORT` and `GROUPBY`. We limited our implementation to join operators because of their sensitivity to optimization decisions. Additionally, adding different storage access (i.e. index lookup) and different

flavors of joins is orthogonal to our algorithm and has been extensively examined in prior work [SAC+79, KS00].

The runtime of our work sharing optimizer for the whole TPC-W was less than 3 seconds, which is quite fast, given it requires analysis of 17 join operators across 11 statements. To put it into perspective, a brute force optimizer would have to consider a total of $2^{16} * \frac{(2*11)!}{11!} = 4.39 * 10^{16}$ possible solutions. The generated plan contained only 10 join operators, some of which were shared across multiple statements. The baseline plan, where operators are shared across queries of the same statements only, was generated in 500 msec and requires a total of 17 join operators. In traditional MQO, this plan has to be generated every time a new set of queries arrives in the system. This means that if more than 6 batches of queries are going to be executed, the time spent in detecting commonalities and generating a plan using MQO is greater than the time of WSO. We did not include the overhead of plan generation in our measurements. The reported latency and throughput account only for the query execution.

We implemented the two plans in SharedDB and used the same amount of hardware resources for both generated plans. A variable number of TPC-W clients were used in a closed system with no thinking time, as we measured the throughput and latency of the system. Each experiment was run for one hour. The results, shown in Figure 3.17, clearly demonstrate that the Work Sharing Optimization algorithm outmatches the traditional multi query optimization techniques. Sharing operators across statements independently of whether there are common sub-expressions or not is beneficial both in terms of throughput and in terms of latency. For the latter, we can also observe that work sharing reduces the variance of the response time. The latency plot illustrates the average response time, as well as, the 90th percentile.

3.8.3 TPC-H Analysis

Next, we used the TPC-H benchmark with a scale factor of 1.0 to measure the quality of the work sharing global query plans. As with TPC-W we simplified the queries by removing all relational operators except joins and full table scans. We used all 22 statements that are part of the specification, even the ones that require just a full table scan and no join operators, like `Q1` and `Q6`. A total of 49 hash join operators were considered.

The WSO algorithm needs less than 30 seconds to optimize the whole TPC-H workload, which is acceptable, given the size of the problem and the fact that the generated plan can be used for the whole lifetime of the system. The generated plan contained only 34 operators, some of which were shared. The baseline plan for a set of 22 queries requires

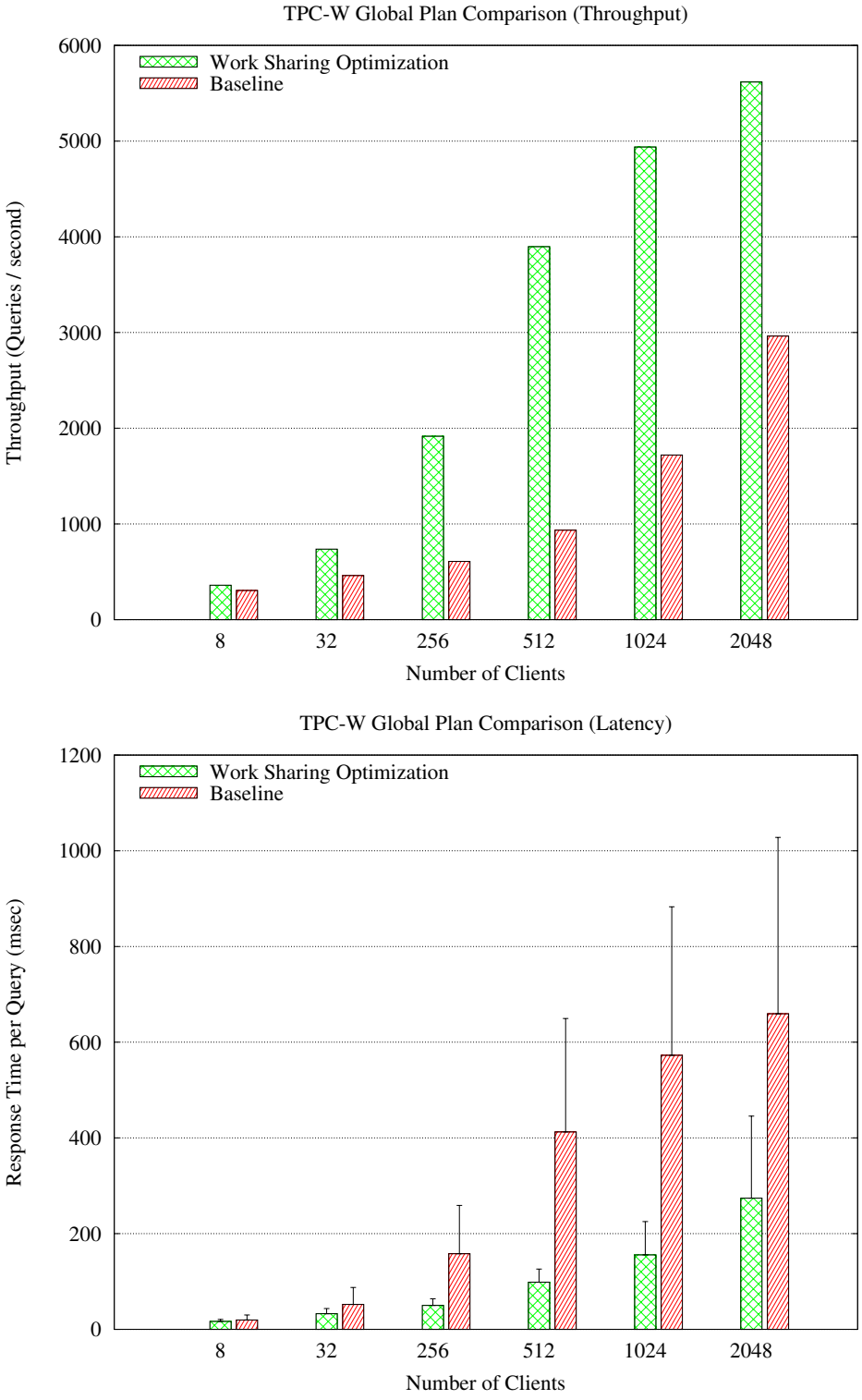


Figure 3.17: Testing the TPC-W Browsing Workload

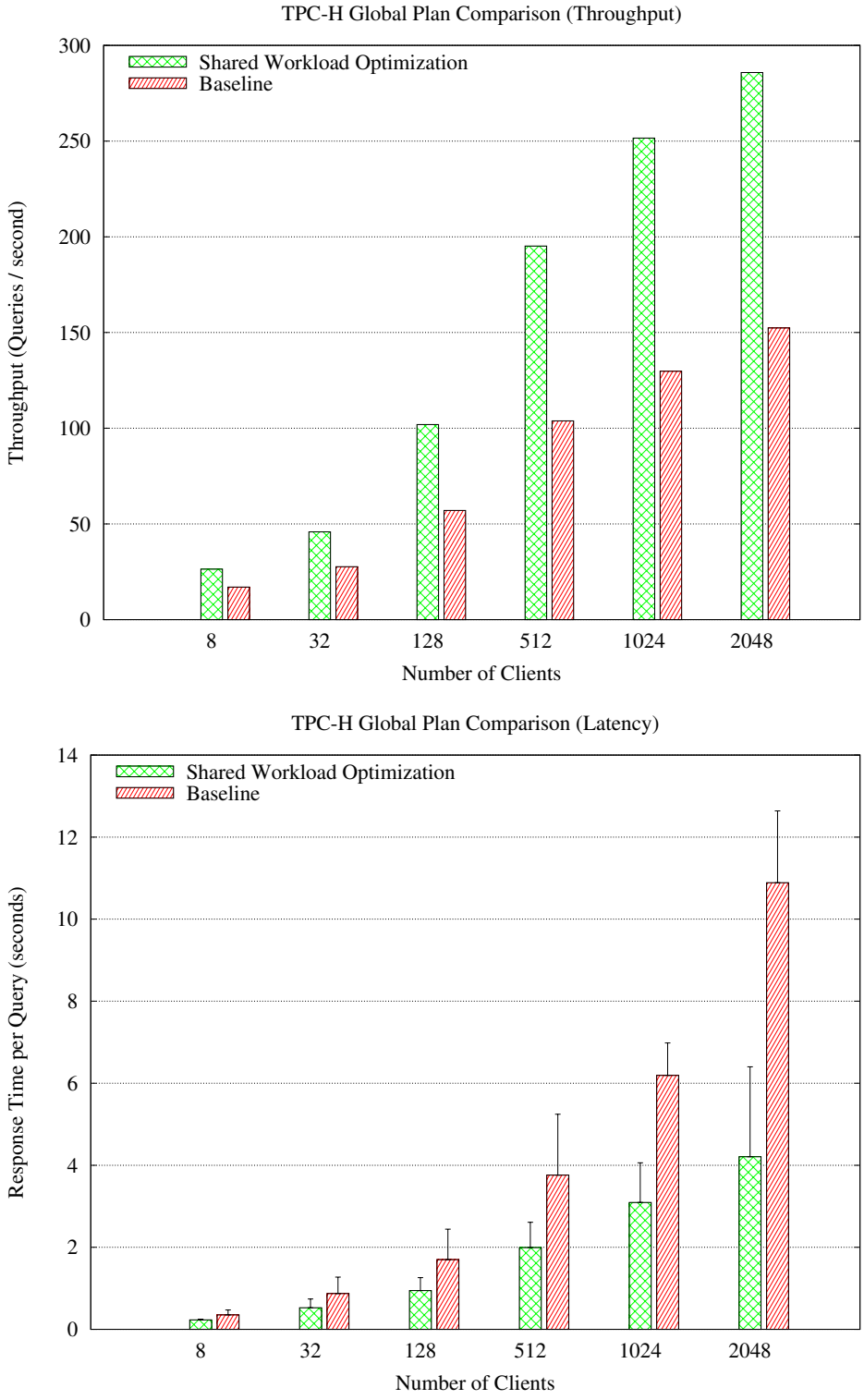


Figure 3.18: Testing the TPC-H Workload

on average 3 seconds to be generated. As with the TPC-W benchmark, our metrics do not include the overhead of generating a plan every time a new set of queries arrives.

The results of our experiment are shown in Figure 3.18, and clearly show the advantages of work sharing. Any work that is common across statements is executed only once, which explains the higher throughput and the lower response time of the work sharing plan.

As already explained, a query-at-a-time system database would not be able to handle such a load. Since, all table accesses are full table scans, the storage engine overhead would dominate the performance of such a system. Additionally, the number of clients is prohibitive. Executing 2,048 queries concurrently would either require queuing, or queries will fight with each other for resources. In both cases, performance would be significantly lower compared to the performance of a WS system.

3.9 Concluding Remarks

In this chapter we presented a global query optimization algorithm that can be used to optimize the whole workload on modern cooperative database systems, like SharedDB. The problem is twofold. The optimizer has to order the relational operators as well as decide which queries should share them. The generated plan does not necessarily answer each query in the best possible way. Nevertheless, the cost of executing the whole workload is optimized.

What makes the problem interesting is that state-of-the-art query optimization techniques cannot be applied. The solution space is so big that exhaustive techniques cannot be used. Additionally, the problem does not have optimal substructure and as a result, it cannot be broken into smaller subproblems. To tackle these challenges, our optimizer uses the branch and bound optimization method. In order to reduce the solution space, we introduce two heuristics that prune suboptimal solutions and guide the optimizer to explore first the most promising part of the solution space. Our experimental results on the generated plans of different workloads, show that work sharing outperforms traditional multi query optimization techniques that share work only across common sub-expressions.

There are a lot of extensions that can be incorporated in the presented algorithm. Smart assignment of cooperative operators to CPU cores as part of the optimization can further increase performance. This requires to combine the hardware topology with the generated access plan, in a way that operators have minimal distances in the physical layout. Furthermore, the algorithm can be executed multiple times during the life cycle of the system, allowing for dynamic plan generation. In this case, as the workload or dataset evolves, so will the global access plan. Finally, an interesting extension to our algorithm

is query prioritization, by means of SLA. Introducing certain constraints during the plan generation, we could generate a plan that provides certain response time guarantees. All these extensions are not currently implemented in the algorithm, but introducing them does not require any change in the core of the algorithm that was presented in Section 3.6.

4

SharedDB/TX

An important feature of database systems is the ability to execute transactions, a set of read and write operations with all-or-nothing semantics. Modern workloads make heavy use of database transactions, as more and more users insert, modify and delete data on database systems. Ensuring that each user accesses a consistent view of the data is crucial, especially in businesses like banking and reservation systems, where inconsistencies can lead to revenue loss.

Nowadays, the vast majority of relational database systems support Atomic, Consistent, Isolated and Durable (ACID) transactions. While the basic techniques and ideas behind transactions have been established in the Seventies [Pap79], the implementation details evolve to match the current hardware trends and workload requirements. For instance, the early database systems expected transactions to access data on disks, and were designed for concurrent accesses from a small number of users [MLO86]. Modern database systems support hundreds of concurrent users and consider main memory as the primary storage medium [DFI+13].

SharedDB natively supports snapshot isolation per individual request, as well as write monotonicity guarantees. This means that all accesses to the data read a consistent snapshot, yet across two requests, data may be inconsistent. While this is enough for applications that require session consistency (i.e., a user should see his own writes only),

like the TPC-W workload, it is not sufficient to cover the needs of most modern workloads.

To this end, this chapter presents the design and implementation of concurrency control in SharedDB, that supports atomic, isolated read-write transactions. The implementation uses existing techniques, adapted to the context of a work sharing processing system that processes batches of queries concurrently. For this reason, transactions never block during normal processing, but they have to wait before commit to ensure correct serialization ordering. The implemented Work Sharing Transaction Manager uses Multi Version Concurrency Control (MVCC) to handle read and write conflicts, and is designed for main memory storage. Most importantly, the amount of modifications that are required are minimal. The operator model is preserved and all shared data processing operators can be used without any modifications. Finally, SharedDB, the Work Sharing Optimizer and the Work Sharing Transaction manager can be merged into a concrete database system, SharedDB/TX.

We tested the implemented concurrency control mechanism using a modified TPC-W workload that creates conflicts across reads and writes, and compared the performance against the implementation of Chapter 2. The overhead of implementing transactions is justified by the amount of extra work required to detect and resolve conflicts.

The remaining of this chapter is organized as follows: Section 4.1 gives an overview of related work in the context of transaction processing; Section 4.2 presents background details on transaction processing that are taken into consideration when designing SharedDB/TX; Section 4.3 presents the design and implementation of transaction handling in SharedDB; Section 4.4 shows the results of our experimental evaluation of SharedDB/TX and compares it to SharedDB; and finally, Section 4.5 concludes the chapter.

4.1 State of the Art

Concurrency control has a long and rich history going back to the beginning of database systems. Several excellent surveys and books on concurrency control are available [BHG87, GR92, Kum96, MBKB07, Car83].

In single machine systems, two concurrency control mechanisms are widely used: locking and multi version concurrency control [BG83]. Each approach has different advantages and disadvantages. Locking is pessimistic and preemptively prevents access to data that may cause conflicts, by using locks or latches. A number of optimizations have been proposed in order to increase performance, by minimizing lock overhead [Car84, JASA09, JPH+09]. Nevertheless, locking is not a good candidate for SharedDB, or batch processing, push-based systems, as we will explain in Section 4.2.

Multi Version Concurrency Control (MVCC) methods also have a long history. In MVCC multiple versions of the data are maintained, and each reader chooses which version should be accessed. Bernstein et al. introduces three different methods for MVCC: timestamp ordering (MVTO), two-version two-phase locking (2V2PL) and a hybrid of these two [BHG87]. 2V2PL uses at most two versions: last committed and updated uncommitted. The mixed method uses MVTO for read-only transactions and Strict 2PL for update transactions.

Kung et al. introduced the optimistic approach to concurrency control, using a single-version database [KR81]. Based on this, a number of MVCC schemes have been proposed, like Multiversion Serial Validation [CM86] and Multiversion Parallel Validation [ABG+87], which support only repeatable read isolation levels. Our approach is also based on optimistic multi version concurrency control, yet it supports all possible isolation levels defined in SQL/92.

Snapshot Isolation (SI) is a MVCC method used by many database systems [BBG+95], to isolate read-only transactions from updates. Oracle, PostgreSQL, SQL Server, and possibly others are amongst the RDBMS that support it. However, SI is not serializable and many papers have considered under what circumstances SI is serializable or how to make it serializable. SharedDB natively supports Snapshot Isolation, as each read always sees a consistent view of the data. This is a property of the two implemented storage engines, as presented in Section 2.3.4.

A complete and practical solution on how to guarantee serializable isolation with SI data stores has been published in [CRF08]. This technique requires that transactions check for read-write dependencies. The implementation focuses on disk databases and uses a standard lock manager and transactions acquire locks and check for read-write dependencies on every read and write. The locks are non blocking and used only to detect read-write dependencies. While, this is efficient when a small number of transaction is executed, locks are a big overhead when hundreds of transactions are active. This is the case for batch processing systems, like SharedDB. Thusm our approach is based on a validation scheme that does not use any locks and instead is based on repeating part of the transaction.

Techniques that validate isolation by checking the repeatability of reads have already been used in the past [BHEF11]. Oracle TimesTen, IBM's solidDB, SAP HANA [SFL+12] and Hekaton [DFI+13] are a couple of commercially available main-memory DBMSs that use such techniques. The most recent one, Hekaton, requires no locking during normal execution. Additionally, transactions are never blocked, except during the validation phase [LBD+11].

Our approach extends these ideas and applies them to a work sharing, main memory

system. One of the contributions of our implementation is how to efficiently handle transactions in batches and validate the isolation, without halting the batch. Furthermore, our system uses a novel garbage collection approach that is designed specifically for push-based systems and allows efficient removal of obsolete versions, without examining the whole dataset.

4.2 Background: Concurrency Control Methods and Transactions

Concurrency control is the activity of coordinating the actions of independent operations that work in parallel, access shared data, and therefore potentially interfere with each other. Concurrency control problems arise in the design of hardware, operating systems, real time systems, communications systems, and database systems, among others.

In a database system, the abstraction of concurrency control is a transaction, a set of read and write operations that access a dataset. The goal of concurrency control is to ensure that transactions are executed atomically. Formally, this requires two things:

- each transaction is able to access the dataset without interfering with other, concurrently executed transactions, and
- each transaction that successfully terminates has *all* of its modifications applied permanently the dataset; otherwise, the transaction should not have any effect on the dataset at all.

A transaction in practice is a set of operations issued by the user. A typical example of a transaction is the following:

```
Transaction TR1
1: BEGIN transaction ON ERROR ABORT transaction;
2: SELECT balance AS :old_balance FROM CUSTOMERS WHERE ID = ?;
3: UPDATE CUSTOMERS SET balance = :old_balance + ? WHERE ID = ?;
4: SELECT balance AS :new_balance FROM CUSTOMERS WHERE ID = ?;
5: COMMIT transaction;
```

Obviously, different transactions have different consistency requirements. For instance, in banking systems, transactions should have access to only consistent, up-to-date data. In other cases, for instance in social networks this is not a necessity. For example, it

makes very little difference whether a famous twitter user is displayed to have 100,000 or 100,001 followers. In such cases, data may be outdated for brief periods of time, without hurting the model of the system. In fact, in such cases where a big user base can issue dataset updates, it is beneficial to have a consistency model that does not fully isolate transactions.

The ANSI/ISO standard SQL 92 specifies four different degrees to which one transaction must be isolated from data modifications made by other transactions: read uncommitted, read committed, repeatable read and serializable. Read uncommitted is a transaction isolation level that makes no guarantees on the read data. For instance, in the previous example, it may happen that another transaction executes at point `4` and reads `new_balance`, while `TR1` aborts at step `5`. Read committed guarantees that all read data are permanent. Yet, this does not guarantee that repeating the read will give the same result. For instance, a concurrent transaction may read balance at point `2` (thus reading `old_balance`) and then read balance again at point `4`. As long as `TR1` commits at step `5`, this is valid for the repeatable read isolation level. Repeatable read guarantees that re-accessing data from the same transaction, returns the same tuples. However, it does not solve the problem of reading phantom tuples. For instance, if another transaction wanted to count how many `CUSTOMERS` have a balance over some threshold, then depending on the order of execution and the parameters of `TR1`, the customer of `TR1` may be counted or not. Finally, serializability ensures that none of these anomalies are present [GR92].

To formulate this, a transaction T is serializable if the following two properties hold:

Read Stability If T reads some version V_1 of a tuple during its processing, we must guarantee that V_1 is still the version visible to T at the end of the transaction. This means that V_1 has not been replaced by another committed version V_2.

Phantom Avoidance If T reads a range of tuples during its processing, we must guarantee that at the of the transaction no new tuples that have been added to this range by another committed transaction.

Read stability ensures that no tuple has disappeared while phantom avoidance ensures that no tuples have appeared in the dataset during the runtime of the transaction. Figure 4.1 shows all different isolation levels, as well as the read anomalies that are allowed in each one of them.

Transaction isolation levels have different impact on a database. Uncommitted reads require no synchronization before reading a tuple, thus allow for higher performance. On the other hand, serializable requires synchronization before reads, consistency across multiple reads and checking of range reads.

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
Read Uncommitted	•	•	•
Read Committed		•	•
Repeatable Reads			•
Serializable			

Figure 4.1: Transaction Isolation Levels and the allowed Anomalies

There are two widely used concurrency control methods in database management systems: locking and multiple versioning [BHG87]. Locking is a pessimistic concurrency control method, where data should be locked before they are read or written. This allows a single transaction to have exclusive access to all locked tuples. Other transactions have to wait for the tuples to be unlocked before they can be accessed. To allow for better read performance, shared locks can be used. Shared locks allow multiple transactions to access the same data, as long as the data is only read and not modified. Since locking is a pessimistic approach, all the accessed data is consistent. However, deadlocks are possible and certain mechanisms to detect them and take care of them are necessary. Nevertheless, locking is simple to implement and as a result it has been adopted by a number of database systems. A lot of research has focused on optimizing locking by, among others, minimizing locking overhead [PJHA10], special handling of "hot locks" [JASA09], and using intra-transaction parallelism [CASM05].

Multi Version Concurrency Control (MVCC) is an optimistic approach to concurrency control where multiple transactions are able to access the same data. To ensure consistency, data is immutable. Updates are supported by creating a newer version of the data, which if the transaction successfully completes, replaces the older version. Multiple versions can exist at any time, and a transaction can choose which version it should read.

Locking compared to MVCC has a smaller footprint, since keeping multiple versions requires additional storage space. Also, tuples are tagged with their version which adds a constant storage overhead per tuple. Most importantly, locking is simpler to implement than MVCC. MVCC has to provide version management and add more logic to read and write data. These are the main reasons that made locking a popular choice for DBMS, especially in the Eighties and Nineties where storage capacity was limited. On the other hand, MVCC scales nicely to the number of concurrent transactions. There is no transaction stalling due to locking, and no need to detect and handle deadlocks.

Shared query processing favors a concurrency control mechanism based on MVCC. Systems like SharedDB handle thousands of queries concurrently. Since these queries are actually executed in a single batch, locking and eventually stalling some of these queries means that

the whole batch will have a higher processing time. To make matters worse, this breaks the predictability guarantees of SharedDB, as now the response time of a query is affected by the number of collisions across other queries. Finally, deadlock detection requires coordination of storage engines. In a shared processing environment that uses push-based staged storage operators, this introduces an additional message passing channel across these operators.

SharedDB/TX implements concurrency control using multi versioning, for the aforementioned reasons. All ANSI/ISO SQL'92 isolation levels are supported. Using MVCC, requires no deadlock detection mechanism, and as a result no additional communication across storage engine operators. In fact, the SharedDB global query plan needs no modifications in order to support MVCC.

4.3 System Design

This section describes how MVCC is implemented in SharedDB/TX. The presented techniques can be used to any other push-based work-sharing system in order to support handling of demanding transactional workloads.

4.3.1 Transaction Flow

The simplest and most widely used MVCC method is Snapshot Isolation (SI). SI alone does not guarantee serializability, since reads and writes logically occur at different times. More specifically, in SI, all reads occur in the beginning of the transaction and all writes at the end of it. In order to provide serializability guarantees on a system that implements SI, a transaction should see exactly the same data if all its reads were repeated at the end of the transaction. This means that in order to guarantee read stability, SharedDB/TX has to validate that each read tuple has not been replaced with a newer version at the end of the transaction. Similarly, in order to guarantee phantom avoidance, all range accesses to the dataset have to be repeated, while validating that no new tuples have been added.

4.3.1.1 Versioning and Transaction Visibility

The implemented MVCC system uses monotonically increasing Version Identifiers (`VID`) to keep track of the different versions. Transactions are uniquely identified with similar Transaction Identifiers (`TID`). In order to reduce storage overhead, SharedDB/TX combines

both identifiers into a single 32-bit integer, where the first bit defines if it is a VID or a TID. This combined TVID field is part of all storage level tuples.

All storage level tuples use two TVID fields. The first one, the $\texttt{TVID}_{\texttt{From}}$, marks the version in which this tuple was created, while the second one, the $\texttt{TVID}_{\texttt{To}}$, marks the version in which the tuple expires. This means that tuples in the implemented system are immutable. In order to update an existing tuple, a new physical copy with a higher version identifier is created and inserted into the dataset, while the original tuple is marked as expired. Tuples that have not expired, have their $\texttt{TVID}_{\texttt{To}}$ set to positive infinity. These two TVID fields allow a reader to access any version of the dataset. For instance, a reader that wants to access version v, should access all tuples that have been created before v and have not been deleted before v. This constraint is expressed with the formula $\texttt{TVID}_{\texttt{From}} \leq v < \texttt{TVID}_{\texttt{To}}$.

All storage engines maintain a separate VID, which is the latest version of the dataset. Transactions also contain a VID alongside their TID. The VID is used to determine which version of data the transaction should access. This enables a transaction to read an older version of the dataset, allowing non-blocking concurrent updates to the dataset, a property which is crucial for the performance of SharedDB/TX.

A newly created transaction is assigned the next available TID and the latest storage engine VID, meaning that it would see the most fresh data. All newly inserted tuples by a transaction receive special values for their $\texttt{TVID}_{\texttt{From}}$ and $\texttt{TVID}_{\texttt{To}}$ fields. The $\texttt{TVID}_{\texttt{From}}$ is set to the TID of the transaction, while TVID-To is set to positive infinity, as the tuple has not expired. At this point, all other concurrently executed transactions will not see this newly inserted tuple, because $\texttt{TVID}_{\texttt{From}} > T_v$. In order to maintain the snapshot isolation property for the current transaction, the extra constraint of $\texttt{TVID}_{\texttt{From}} = \texttt{T}_{\texttt{TID}}$ should be added to the visibility formula.

Finally, transactions maintain a set of all tuples that are deleted during their execution. No modifications to the tuples is made, until the transaction is committed. This renders the final version of the visibility formula:

$$((\texttt{TVID}_{\texttt{From}} \leq \texttt{T}_v < \texttt{TVID}_{\texttt{To}}) \vee (\texttt{TVID}_{\texttt{From}} = \texttt{T}_{\texttt{TID}})) \wedge \texttt{Rec} \notin \texttt{T}_{\texttt{Deleted}}$$

Figure 4.2 shows an example with all possible cases for tuple visibility.

4.3.1.2 Transaction Validation (PREPARE)

Since MVCC is optimistic regarding transaction collisions, a validation phase is required before commit. In this phase, the system checks if the transaction's writes can be atomically and consistently applied to the dataset, as if no other modification has been applied

TX	TID	VID	Statement
T_0	256	1	`SELECT * FROM CUSTOMERS;`
T_1	257	10	`INSERT INTO CUSTOMERS ...;` `SELECT * FROM CUSTOMERS;`
T_2	258	10	`DELETE FROM CUSTOMERS WHERE CID = 10;` `SELECT * FROM CUSTOMERS;`
T_3	259	12	`UPDATE CUSTOMERS SET ... WHERE CID = 7;` `SELECT * FROM CUSTOMERS;`

(a) Sample Running Uncommitted Transactions

CUSTOMERS Relation

CID	$TVID_{From}$	$TVID_{To}$	Name	...	Visibility
1	0	255	John Smith	...	T_0, T_1, T_2, T_3
3	2	255	Kate Johnson	...	T_1, T_2, T_3
4	2	8	George Dallas	...	-
7	5	255	Jim Simpson	...	T_1, T_2, ~~T_3~~ (Updated)
7	259	255	Jim Simpson	...	T_3
10	0	11	Karl Smith	...	T_0, T_1, ~~T_2~~ (Deleted)

(b) Sample Dataset and Visibility

Figure 4.2: Visibility of Tuples in SharedDB/TX

in the meantime. Additionally, depending on the transaction isolation level, the validation phase checks if the data that was read by the transaction is still consistent. For this reason, each transaction keeps track of two *ReadSet*s, a *PointReadSet* and a *RangeReadSet*, in addition to the *DeletedSet*. If the validation fails, then the transaction cannot commit, and the issuing client is informed for the reason.

The validation phase begins by examining all writes made by a transaction `T`. The writes of a transaction are consistent, if the following two properties hold:

Delete Stability If a tuple was deleted during the execution of `T`, it should not have been persistently deleted by another transaction.

Integrity Constraints If a tuple was inserted during the execution of `T`, all integrity constraints should hold for the new tuple.

The first property is checked by revisiting the $\texttt{T}_{\texttt{Deleted}}$ set and verifying that all tuples marked as deleted by T, were not deleted by some other transaction in the meantime. This is done by checking if the record's $\texttt{TVID}_{\texttt{From}}$ version is not positive infinity:

$$\forall\ \texttt{Rec} \in \texttt{T}_{\texttt{Deleted}} : \{\texttt{TVID}_{\texttt{To}} = +\infty\}$$

Integrity constraints is a more complicated check that most of the times depends on the used-defined database schema. The system has to verify that all constraints are still valid, if all transaction's writes are applied. For the simple case of unique key constraints, the system has to check if the key of the newly added tuples has been inserted by some other transaction.

Once the writes have been successfully validated, the reads are examined. For read stability, all tuples that were accessed as part of a lookup, should not have been deleted:

$$\forall\ \texttt{Rec} \in \texttt{T}_{\texttt{PointReadSet}} : \{\texttt{TVID}_{\texttt{To}} = +\infty\}$$

For phantom avoidance, the system has to repeat all read operations and verify that no tuples were inserted or deleted in the meantime. That is, all tuples should have been deleted before T or not at all, and they should have been created before T or during T:

$$\begin{aligned}\forall\ \texttt{Rec} \in \texttt{ReadSets} : \{\quad & (\texttt{TVID}_{\texttt{To}} \leq \texttt{T}_{\texttt{VID}}\ \wedge\ \texttt{TVID}_{\texttt{To}} = +\infty)\ \vee \\ & (\texttt{TVID}_{\texttt{From}} > \texttt{T}_{\texttt{VID}}\ \wedge\ \texttt{TVID}_{\texttt{From}} = \texttt{T}_{\texttt{TID}})\}\end{aligned}$$

4.3.1.3 Transaction Commit

If the validation phase is successful, the transaction can be persistently committed by applying the writes to the dataset. The transaction receives another version identifier, $\texttt{VID}_{\texttt{Commit}}$, which is the version on which the transaction will be applied. Different steps are followed to apply inserts and deletes. All tuples inserted by T were initially inserted with the $\texttt{VID}_{\texttt{From}}$ equal to the TID of the transaction. To apply the inserts, the following transformation has to be applied:

$$\forall\ \texttt{Rec} \in \texttt{InsertSet} : \left\{\texttt{Rec}_{\texttt{TVID}_{\texttt{From}}} \leftarrow \texttt{T}_{\texttt{VID}_{\texttt{Commit}}}\right\}$$

Finally, deletes can be applied in a similar way, by visiting $\texttt{T}_{\texttt{Deleted}}$, the set of tuples that T has deleted, and modifying the expiration of the tuples:

$$\forall\ \texttt{Rec} \in \texttt{DeleteSet} : \left\{\texttt{Rec}_{\texttt{TVID}_{\texttt{To}}} \leftarrow \texttt{T}_{\texttt{VID}_{\texttt{Commit}}}\right\}$$

Dataset Modification Operations

Operation	Action
Insert tuple t	Insert t in dataset with $\texttt{TVID}_{\texttt{From}} = \texttt{TX}_{\texttt{TID}}$.
Delete tuple t	Add t in $\texttt{TX}_{\texttt{DeleteSet}}$.
Update tuple t	Perform an insert and an update.
Prepare	Validate that $\forall\ t \in \texttt{DeleteSet} : \{t.\texttt{TVID}_{\texttt{To}} < \texttt{TX}_{\texttt{VID}_{\texttt{Commit}}}\}$. Validate that inserts match dataset constraints.
Commit	$\forall\ t \in \texttt{InsertSet} : \{t.\texttt{TVID}_{\texttt{From}} \leftarrow \texttt{TX}_{\texttt{VID}_{\texttt{Commit}}}\}$. $\forall\ t \in \texttt{DeleteSet} : \{t.\texttt{TVID}_{\texttt{To}} \leftarrow \texttt{TX}_{\texttt{VID}_{\texttt{Commit}}}\}$.
Abort	Clean up `InsertSet`.

Read Operations

Isolation	Constraint
Read Uncommitted	None.
Read Committed Repeatable Read Serializable	$((\texttt{TVID}_{\texttt{From}} \leq \texttt{T}_v < \texttt{TVID}_{\texttt{To}}) \vee (\texttt{TVID}_{\texttt{From}} = \texttt{T}_{\texttt{TID}})) \wedge \texttt{Rec} \notin \texttt{T}_{\texttt{Deleted}}$

Read Validations

Isolation	Constraint
Read Uncommitted	None.
Read Committed	None.
Repeatable Read	$\forall\ \texttt{Rec} \in \texttt{T}_{\texttt{PointReadSet}} : \{\texttt{TVID}_{\texttt{To}} = +\infty\}$
Serializable	$\forall\ \texttt{Rec} \in \texttt{T}_{\texttt{PointReadSet}} : \{\texttt{TVID}_{\texttt{To}} = +\infty\}$ $\forall\ \texttt{Rec} \in \texttt{ReadRange} : \{\ (\texttt{TVID}_{\texttt{To}} \leq \texttt{T}_{\texttt{VID}} \wedge \texttt{TVID}_{\texttt{To}} = +\infty) \vee (\texttt{TVID}_{\texttt{From}} > \texttt{T}_{\texttt{VID}} \wedge \texttt{TVID}_{\texttt{From}} = \texttt{T}_{\texttt{TID}})\}$

Figure 4.3: Operations of a Transaction `TX` on the Dataset

After these modifications, the transaction is considered committed and the storage engines have their latest `TVID`s set to $\texttt{T}_{\texttt{Commit}}$. Any future transaction will view all changes of `T`. Since tuples are never physically deleted during the commit phase, a garbage collector is required in order to remove obsolete versions of tuples. We will discuss the specifics of the garbage collector in Section 4.3.4.3

$\texttt{TVID}_{\texttt{From}}$	$\texttt{TVID}_{\texttt{To}}$	**State**
$< \texttt{S}_{\texttt{TVID}}$	$< \texttt{S}_{\texttt{TVID}}$	Deleted
$< \texttt{S}_{\texttt{TVID}}$	$2^{31} - 1$	Live data
$\geq 2^{31}$	$2^{31} - 1$	Inserted, but not committed
$> \texttt{S}_{\texttt{TVID}} \wedge < 2^{31} - 1$	$2^{31} - 1$	Inserted and committing
$< \texttt{S}_{\texttt{TVID}}$	$> \texttt{S}_{\texttt{TVID}} \wedge < 2^{31} - 1$	Deleted and committing

Figure 4.4: Possible States of a Storage Level Tuple in SharedDB/TX. $\texttt{S}_{\texttt{TVID}}$ is the latest version of the storage engine.

4.3.1.4 Transaction Abort

In case the validation phase is not successful, its effects have to be reversed in order to keep the dataset in a consistent state. An advantage of the described MVCC approach is that no modifications are required to rollback a transaction. All inserted tuples, have a $\texttt{TVID}_{\texttt{From}}$ value that makes it invisible to other transactions. These tuples are safe to delete when the transaction is aborted, as it does not hurt the snapshot isolation properties of other readers. Alternatively, they can be communicated to the garbage collector, who will clean them up whenever it is necessary.

Deletes require no rollback, as they make no modification to the data itself. All delete operations are accumulated in the transaction's `DeleteSet` which is freed when the transaction is aborted.

Figure 4.3 shows the different operations of a transaction and the effects it has on the data.

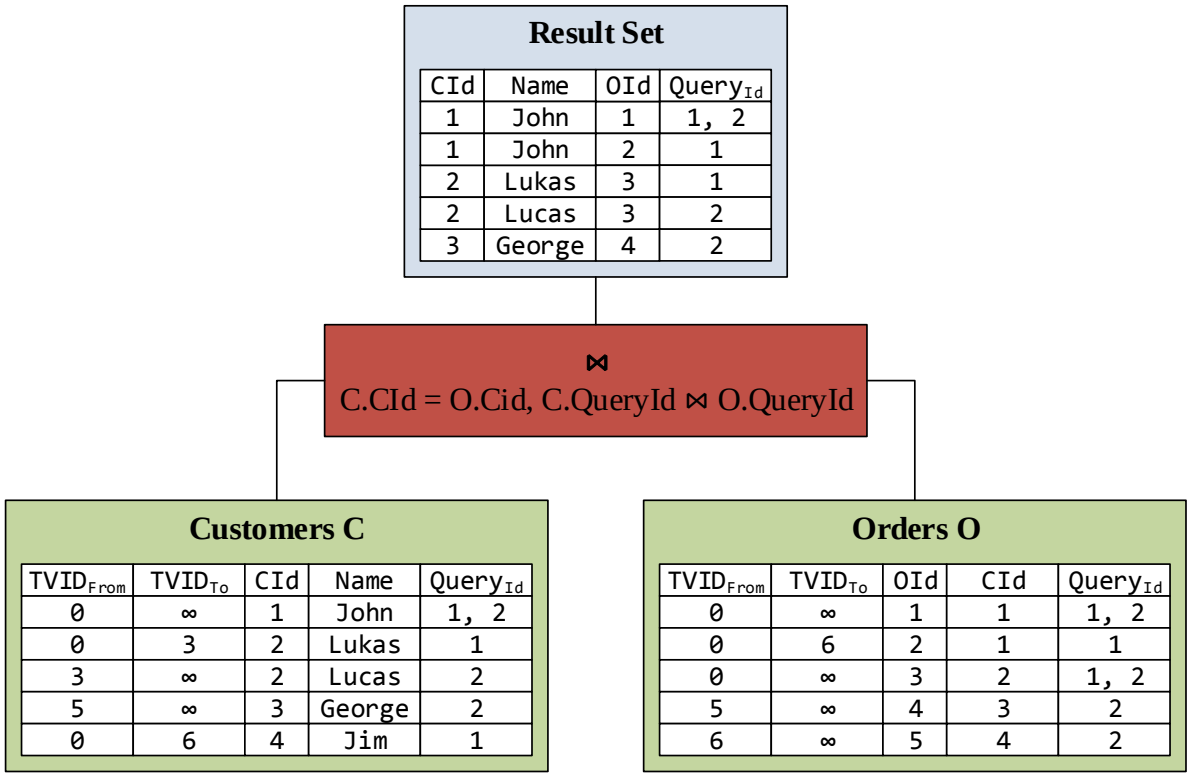
4.3.1.5 Tuples

In practice, `TVID` is implemented as a 32-bit integer, where the most significant bit defines if it is a version or a transaction identifier. In this case, positive infinity is the value of $2^{31} - 1$, the first transaction identifier is 2^{31} and the maximum possible transaction identifier is $2^{32} - 1$. Figure 4.4 summarizes all possible states in which storage layer tuples can be.

An important property of the design of SharedDB/TX is the fact that intermediary tuples do not require version or transaction identifiers, as shown in Figure 4.5. In the example of this figure, the record of Jim is not part of the result set, even though an order with a matching customer Id exists. Yet, the storage level query will tag the customer tuple of

TX	TID	VID	Query
T_1	1	0	SELECT * FROM CUSTOMERS JOIN ORDERS;
T_2	2	10	SELECT * FROM CUSTOMERS JOIN ORDERS;

(a) Running Transactions



(b) Sample Runtime

Figure 4.5: Version Identifiers in Shared Data Processing

Jim with `query_id` 1 only, as it was deleted as of version 6 and transaction 2 is reading version 10.

In fact, all data processing operators that were presented in Section 2.3.5 can be used in SharedDB/TX without any modifications, which shows the flexibility of the model of SharedDB. In order to implement transactional properties, no modifications are required in the global query plan, or the data processing architecture. Figure 4.6 shows a global query plan that supports transactional operations. In this figure, shaded components are unmodified SharedDB components.

Another advantage of this design, is that it allows both versioning storage engines and non-transactional, snapshot-isolation storage engines to be used at the same time. In the

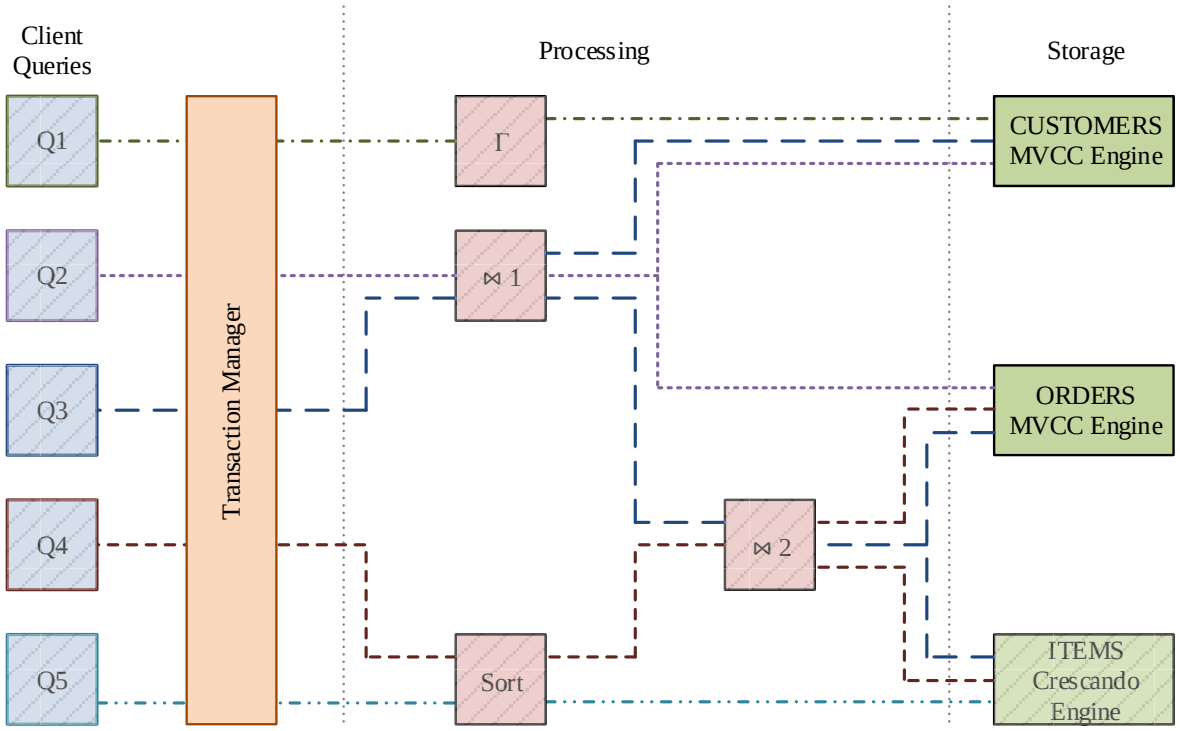


Figure 4.6: Global Query Plan of SharedDB/TX

example of Figure 4.5, the `ITEMS` operator uses the Crescando storage engine. Reads and writes to this table provide only snapshot isolation guarantees for each query. Yet, this may be a good choice for tables that are updates by a single writer. For instance, the online store administrator may be the only client allowed to sequentially edit the `ITEMS` table.

4.3.2 Results

SharedDB/TX introduces one different result type, along with the result tuples and end-of-stream results that were presented in Section 2.3.3. The new result type carries information about the status of a transaction. Obviously, the transaction results can be safely ignored by data processing operators. These results are generated by the Transaction Manager operator that will be described in the next section.

There are two different types of transaction results: successful and unsuccessful. The successful results are the outcome of completed transactional operations, such as a successful commit or abort operation. The unsuccessful results are the outcome of a failed operation and carry more information, which is the reason why the operation has failed. This way

the client is informed why a commit failed and can decide if the transaction should be repeated, or the modify the parameters before resubmitting it.

4.3.3 Transaction Manager

Figure 4.6 introduces a new special operator in the global query plan, the Transaction Manager (TM) operator. The TM operator handles all the complicated versioning logic, which is transparent to all processing operators, performs all consistency and stability checks, and manages the commit or abort of running transactions.

The TM operator implements the same operator model that was presented in Section 2.3.1. It uses two queues, one for incoming SharedDB queries, and one for incoming results. Similarly to all SharedDB processing operators, the TM maintains a sets of producing operators, i.e. operators that will send results to the TM, as well as, a set of consuming operators, which is the set of clients that issued the currently running queries. Internally, the logic of the operator is slightly different, as no data processing takes place. All results that arrive from producing operators are simply forwarded to the consumers.

There are four different types of queries that the TM operator handles:

Begin Transaction The Begin Transaction query asks the TM operator to initiate a new transaction with the provided isolation level. This involves initializing all the metadata that are used during the execution of the transaction, such as the `ReadSet`s and `DeleteSet`. The client receives a token for the newly created transaction, which for simplicity, is the unique transaction identifier (`TID`). All further operations related to this transaction contain this `TID`.

Internally, the `TID` is mapped to the transaction metadata, as well as, the transaction's `VID`. This allows the system to decide which version of the dataset should be read for each query, given a `TID`.

Use Transaction This type of query contains a `TID`, as well as, a chained subquery, as defined in the query model of Section 2.3.2. The Transaction Manager process the Use Transaction query by modifying the chained subqueries, appending information related to the transaction itself. Subqueries that will be executed on data processing operators, or on non-transactional storage engines are not modified.

This query helps in keeping all transaction details transparent to the client. The client does not have to keep track of the `VID` of the transaction, or the `ReadSet`s and `DeleteSet`.

Commit Transaction The Commit Transaction query is issued by the client to persistently commit the modifications of a transaction. The TM operators asks all transactional storage engines to prepare for commit. This is done by forwarding the Commit Transaction query to the query queues of the storage engine operators.

The storage engine operators will validate whether the transaction can commit or not, and will send their response to the Transaction Manager via the TM's result queue. If all responses are positive, the TM will ask the storage engines to commit the transaction through the same procedure. Alternatively, if at least one storage engine detects a collision with some other transaction, then the TM aborts the transaction. Finally, the client is informed by receiving a result containing the outcome of the Commit query.

This 2-phase commit protocol is only used if more than two storage engines were used in a transaction. For instance, if a transaction inserts tuples in `Customers` and also reads data from `Customers`, there is no reason to prepare the transaction. Similarly, in a read uncommitted transaction that writes tuples in `Customers` and reads data from any storage engine, there is no reason to perform validation on all storage engines, thus the prepare phase is not necessary.

Abort Transaction Finally, the client can explicitly ask the TM operator to abort a transaction. Once the TM operator receives such a query, it forwards it to all transactional storage engines that were involved in the transaction.

Figure 4.7 illustrates the flow of all TM queries.

As with all SharedDB operators, the TM operator handles queries in batches. In order to ensure high performance, TM queries should not block other queries unless it is necessary. The described system design requires only a single synchronization point: the 2-phase commit procedure. That is, when one transaction is preparing to commit (validating the required isolation level), no other transaction can either prepare or commit, as that may break write and read stability.

For instance, consider the example of Figure 4.8. In this example. two clients issue two transactions that conflict with each other. While deleting the tuple, there is no collision, as the tuple is still visible by both transactions. Both clients are informed that one tuple is modified, a piece of information that may be used in the logic of the client. Once the clients issue a `COMMIT`, the TM asks the storage engine to prepare, validate and check for conflicts. If there no synchronization on the `PREPARE` phase, then no conflict will be detected. Both `PREPARE` operations will see a $\texttt{VID}_{\texttt{To}}$ that is equal to positive infinity, thus, from the point of view of each prepare, no other transaction tried to delete the specific tuple. As a result,

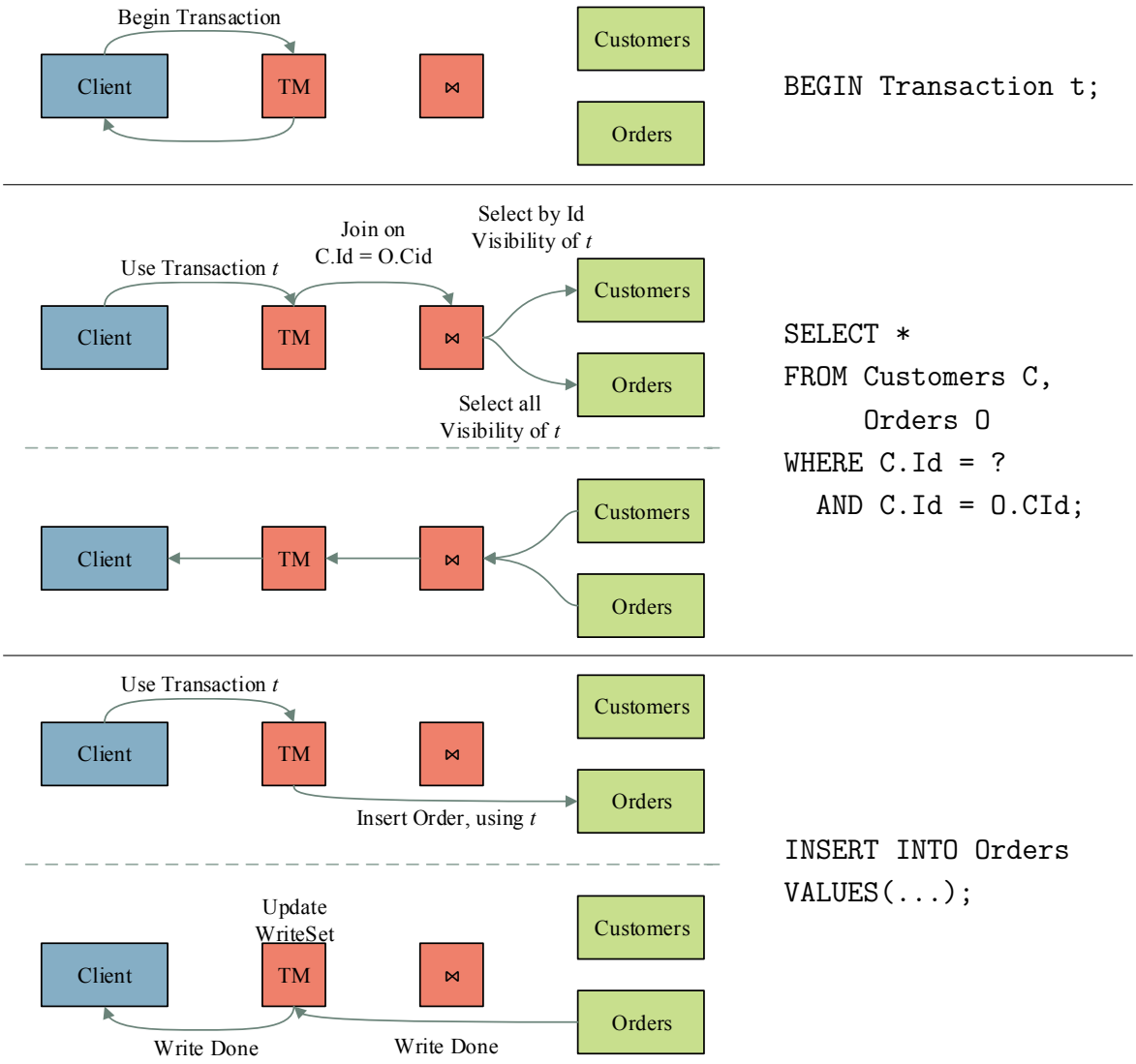


Figure 4.7: Transaction Flow in SharedDB/TX

the TM operator will ask the storage engine to `COMMIT` both transactions (even though there is conflict), and inform the clients that their transactions were committed.

Synchronizing the `PREPARE` phase solves this problem. All `PREPARE` operations are serialized, as shown in the last part of Figure 4.8. First t_1 prepares, while t_2 is waiting. Since there is no conflict, t_1 commits, modifying the $\texttt{VID}_{\texttt{To}}$ of the affected tuple. Then, t_2 attempts to prepare, however the conflict is correctly detected now, and as a result, t_2 is aborted.

Implementing the TM operator using a single thread, similarly to the skeleton Operator of Algorithm 1, means that no other TM query can be executed while another transaction

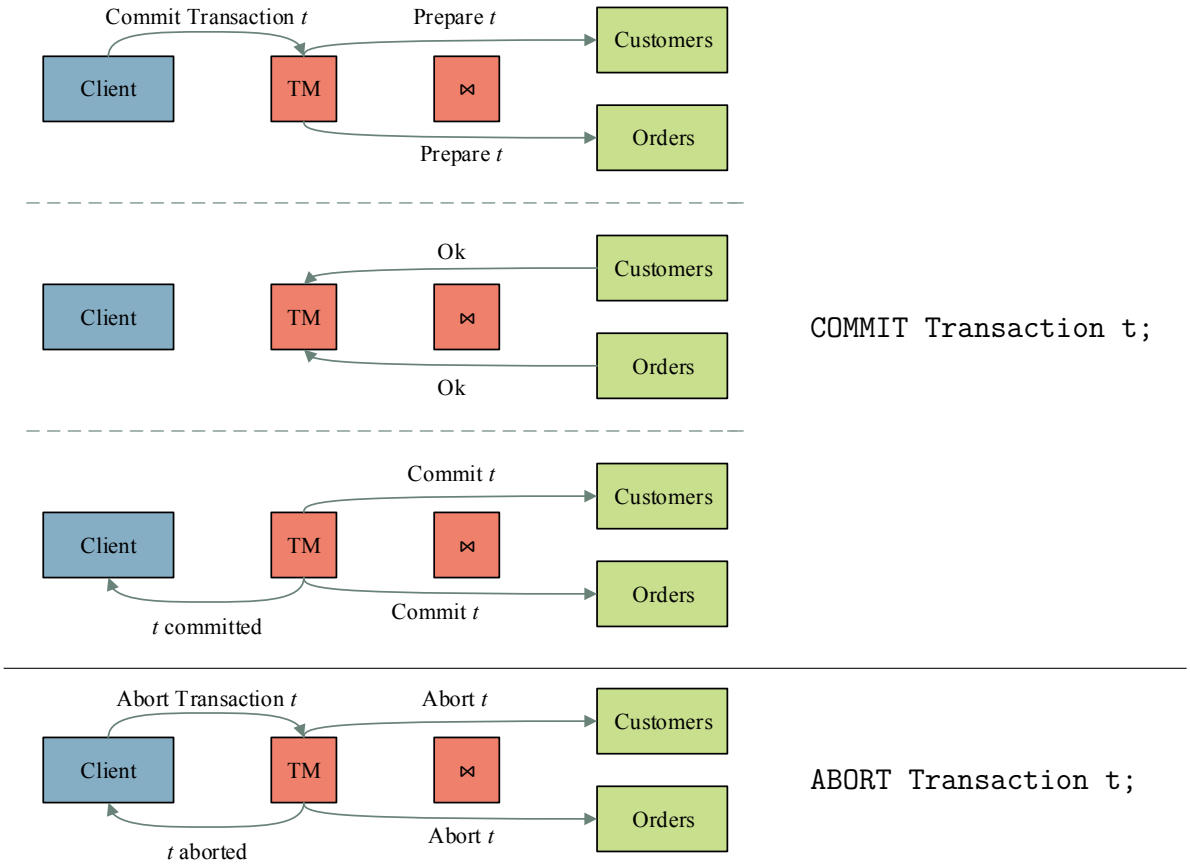


Figure 4.7: Transaction Flow in SharedDB/TX (continued)

is preparing. This includes `BEGIN`, `USE` and `ABORT` queries, which do not have to be synchronized. In order to provide higher performance, and allow other transactions to run in parallel, the TM operator is implemented as two different SharedDB operators, as shown in Figure 4.9. The first SharedDB operator executes the `BEGIN`, `USE` and `ABORT` queries, while the second one executes the `COMMIT` queries.

The two operators use shared state to keep track of all running transactions, which is implemented as a hashmap of `TID`s to the transaction metadata. Even though this hashmap is shared across two running threads, no synchronization mechanism is required. This is because, a transaction is accessed by only one of the two threads at a given time. I.e. there is no way that a transaction is used and committed at the same time.

We also considered a model where each type of TM query is processed in a dedicated SharedDB Operator, requiring a total of four different operators. While this approach is valid, it adds small gains to the system. Typically, `BEGIN`, `USE` and `ABORT` queries are very lightweight and, most importantly, require no synchronization. This means that the

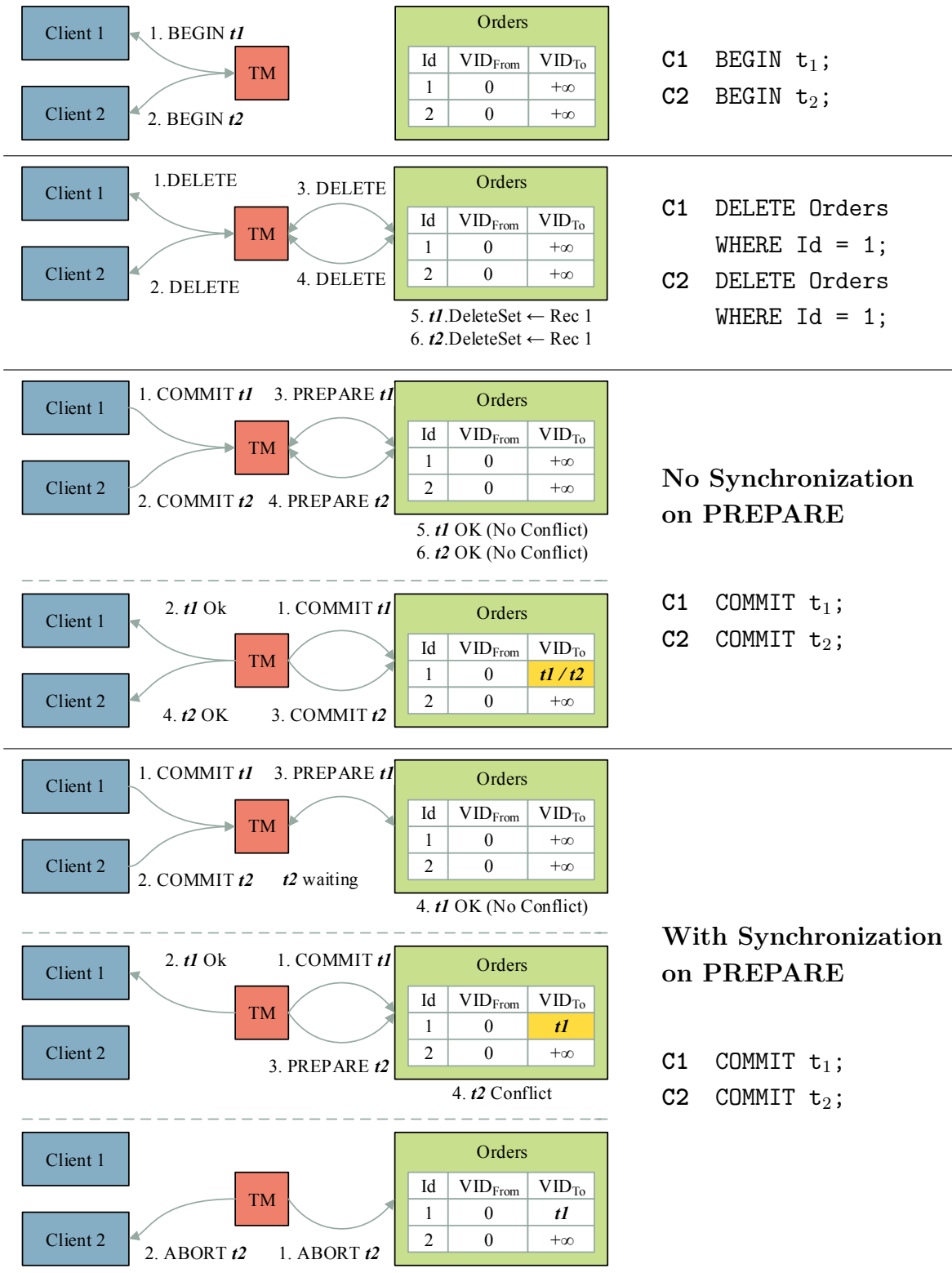


Figure 4.8: Transaction Conflicts and Prepare Synchronization

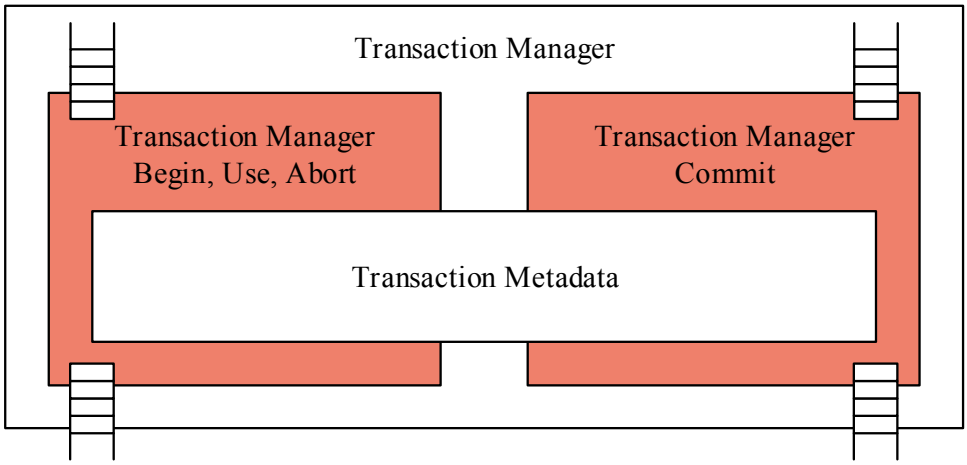


Figure 4.9: Implementation of Transaction Manager using two SharedDB Operators

operator that processes them, is able to process them in batches, similarly to all SharedDB Operators.

Algorithm 13 shows the skeleton of these two operators. They are built around the SharedDB operator model, and each one of them uses two incoming queues: one for queries and one for results. The result queues are used only for `USE` queries and `COMMIT` queries, since `BEGIN` and `ABORT` generate no results inside SharedDB/TX.

Finally, the TM operator can take advantage of the shadow queries optimization that was presented in Section 2.3.6.4. In cases where high numbers of results are generated, the result queue of the `USE` operator can become the bottleneck. We can take advantage of the fact that no data processing is applied in the transaction manager, and override the pipeline of operators. Figure 4.10 shows a global query plan that uses this approach to create a shortcut in the pipeline from the join operator to the client.

In addition to shadow queries we considered other approaches. For instance, the transaction manager can spawn multiple `USE` SharedDB operators, that process only `USE` queries and their results. While this reduces the effects of the bottleneck, the `USE` operators waste system resources. With high loads, TM `USE` operators are mostly forwarding results, rather than processing queries. The approach of shadow queries overcomes this issue. Furthermore, even if the result queue is not a bottleneck, using shadow queries removes one unnecessary step in the pipeline of result processing.

Algorithm 13: Transaction Manager Operators Algorithm

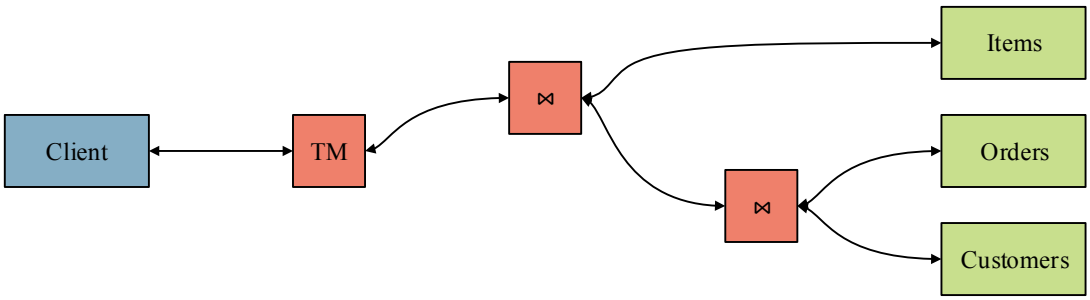
```
/* Shared State: a map of all running transactions                    */
Data: Map[TVID → TransactionMeta] transactions;
Operator TM_Begin_Use_Abort:
    Data: TVID v_last;                          /* Latest committed version */
    Data: TVID t_last;                  /* Last TID assigned to a transaction */
    Data: SyncedQueue iqq;                      /* Incoming queries queue */
    Function Loop():
        Array aq ← ∅ ;                          /* Array of active queries */
        Array consumers ← ∅;          /* Operators that will receive the results */
        /* Activate all queries in the incoming queue                     */
        while ¬IsEmpty(iqq) do Put(aq, Get(iqq)) ;
        /* Enqueue subqueries to underlying operators                     */
        foreach Query q ∈ ag do
            switch GetType(q) do
                case BEGIN
                    /* Create new transaction and send the TID to the client */
                    Transaction t ← CreateTransaction(t_last, v_last);
                    Put(transactions, GetTID(t), t);
                    SendResult(GetProducer(q), New TMResultOk(GetTID(t)));
                case USE
                    /* Add transaction metadata to subquery and forward   */
                    Transaction t ← Get(transactions, GetTID(q));
                    SetMetadata(q, t);
                    PushToProducers(q);
                case ABORT
                    Remove(transactions, GetTID(q));
                    PushToProducers(q);
                    SendResult(GetProducer(q), New
                    TMResultAbort(GetTID(t)));
            endsw
        end foreach
        /* Process results, similarly to the Operator Model (Algorithm 1) */
    end
end
```

Algorithm 13: Transaction Manager Operator Algorithm (continued)

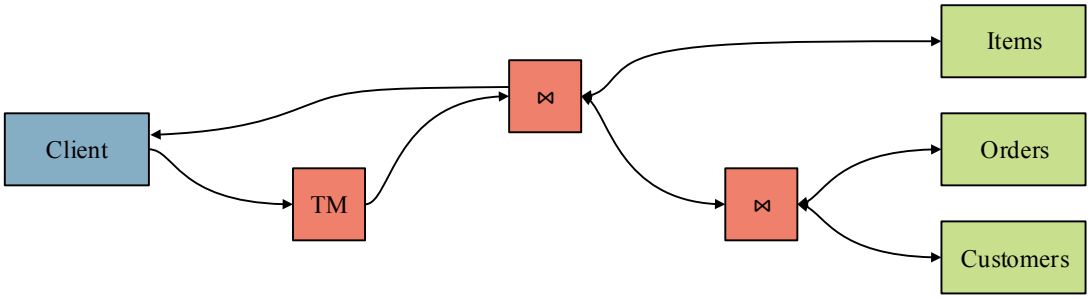
```
Operator TM_Commit:
    Data: SyncedQueue iqq;                          /* Incoming queries queue */
    Data: SyncedQueue rq;                           /* Incoming results queue */
    Function Loop():
        Query q ← GetFront(iq);                     /* Get one COMMIT query */
        /* Retrieve the metadata of the transaction                          */
        Transaction t ← Get(transactions, GetTID(q));
        /* Validate isolation and prepare to commit                          */
        foreach Operator op ∈ GetStorageEngines(t) do
            EnqueueQuery(op, New PrepareQuery(q, t));
        end foreach
        /* Wait for all validation results to arrive                         */
        Array results ← WaitForValidationResults(rq);
        if AllResultsAreOK(results) then
            /* Prepare is OK. Commit                                         */
            foreach Operator op ∈ GetStorageEngines(t) do
                EnqueueQuery(op, New CommitQuery(q, t));
            end foreach
        else
            /* Abort                                                         */
            foreach Operator op ∈ GetStorageEngines(t) do
                EnqueueQuery(op, New AbortQuery(q, t));
            end foreach
        end if
        /* Clean up the transaction                                          */
        Remove(transactions, GetTID(q));
    end
end
```

4.3.4 MVCC Storage Engine

SharedDB/TX introduces a new type of MVCC storage engine operators to SharedDB, since versioning is necessary to ensure consistency and isolation across queries. The requirements of all MVCC storage engine operators were presented in Section 4.3.1. In short, a SharedDB/TX storage engine should maintain multiple versions of tuples, and al-



(a) Transaction Manager without Shadow Queries
The whole result stream passes through the Transaction Manager Operator



(b) Transaction Manager with Shadow Queries
The result stream is diverged directly to the client

Figure 4.10: Using Shadow Queries to Circumvent the Transaction Manager

low queries to access a specific snapshot of the dataset, without using any synchronization mechanism, like locks or latches.

SharedDB/TX uses a B+-Tree based, main-memory key-value storage engine, that implements a relational table, and processes incoming queries and updates in batches. The design is based on a number of existing storage engines and ideas [BM70, DFI+13], adapting these ideas to shared work processing. The storage engine is wrapped inside a SharedDB/TX operator and exposes the same enqueue query/dequeue result interface as all SharedDB operators do.

Figure 4.11 shows a high level overview of the implemented Work Sharing Multi Version (WSMV) storage engine operator. The operator contains three components: the storage manager component which is responsible for storing the physical tuples, the indexes which allow for key and range lookups on a specific attribute, and the garbage collector which frees up space by removing obsolete versions of the data.

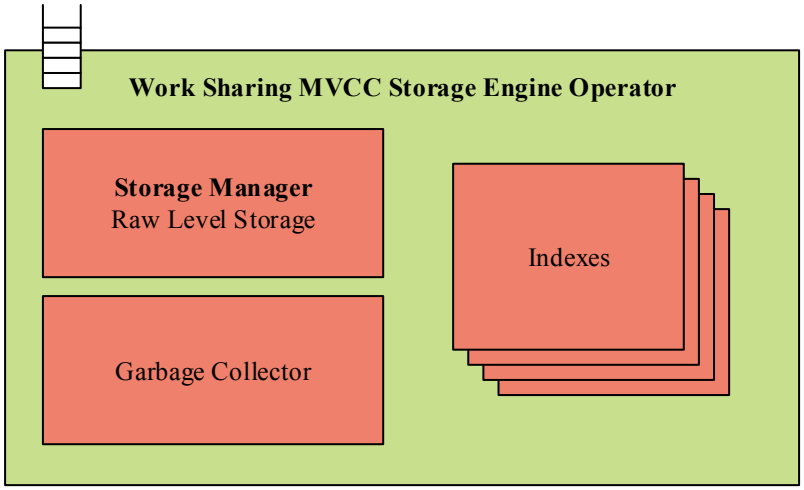


Figure 4.11: High Level Overview of the MVCC Storage Engine of SharedDB/TX

4.3.4.1 Storage Manager

The Storage Manager component is responsible for the raw storage of tuples in main memory. It allocates a variable number of continuous fixed-sized memory areas, and uses them to store a new tuple. Tuples are accessed using either a Record Identifier `RID`, or using record pointers. The latter is possible since allocated memory is not resized (`realloc`).

The stored tuples include a number of attributes, besides the relation's data. Each tuple contains the `RID`, the versioning identifiers $\texttt{TVID}_{\texttt{From}}$ and $\texttt{TVID}_{\texttt{To}}$, and a variable number of pairs of record to other records. These record pointers create double linked chains of records, where each chain corresponds to an index. Thus, the number of pairs is equal to the total number of indexes that exist on the storage engine.

Figure 4.12 shows an example of the raw tuple layout in the storage manager. In this example, the storage operator uses two indexes, the first on the `City` attribute and the second on the `Name` attribute. The two chains connect all the leaf level records to allow for iterating accesses, like range scans. Tuples that have expired are also part of the chain, as running transactions might ask for an older version of the dataset.

This layout is used in many existing implementation of MVCC storage engines. For instance, Hekaton [DFI+13] is using exactly the same layout, even though they limit the system to hash indexes only. This layout has a number of advantages, compared to a layout that does not include the chain pointers in the raw tuple. First of all, it allows switching from one chain to another without probing another index. For instance, in the example of Figure 4.12, given the tuple with `RID` 3, we can chose if we want to access the next tuple when ordered by `Name`, or by `City`. This is extremely useful during the

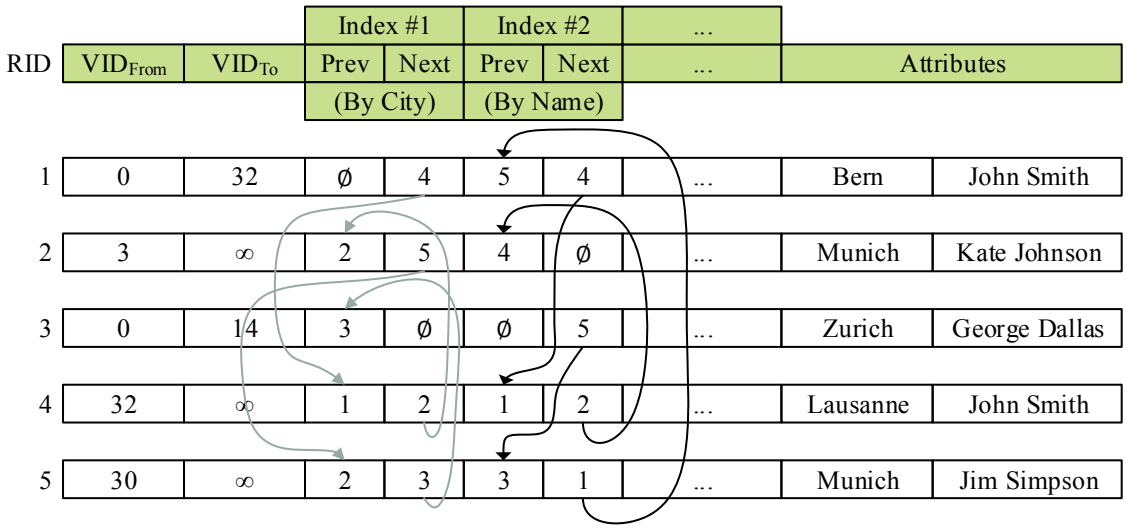


Figure 4.12: Raw Tuple in Storage Manager

validation (`prepare`) phase, where storing only the `RID`s (or the record pointers) in the `ReadSet` and `WriteSet`, is sufficient to do a constraint check. For example, if `Name` is a unique indexed attribute of the schema and a new tuple is inserted during a transaction, it is enough to visit the neighbors of the `Name` index in order to verify the uniqueness of the new tuple's `Name`.

Furthermore, including the chained pointers in the raw tuple, allows for faster and more efficient updates. For instance, consider a transaction that wants to delete a tuple by `Name`. The `Name` index will be probed and the corresponding tuple(s) will have their $\texttt{TVID}_{\texttt{To}}$ attributes updated. There is no need to propagate this modification to the `City` index, or any other indexes that the engine might have. Finally, this layout can be used by any index type, like B-Tree, R-Tree and hash indexes. In the case of a hash index, the chain of records is used to link all the tuples that have a collision on the hash function, similarly to Hekaton's implementation [DFI+13].

This layout has also some disadvantages, but for the specific use case, they do not surpass the gains of the advantages. Memory locality is not optimal, as the leaf node chains are not part of the index itself. This requires one additional read for each visited tuple in order to iterate a chain. Furthermore, any modification results in a large copy cost, as the whole tuple has to be copied, rather than the delta of it. Also, modifying the number of indexes at runtime is more tedious compared to a traditional approach.

Since the Storage Manager is accessed by a single thread (the operator thread), there is no need to enforce a concurrency mechanism. There are no tuple level (or table level) locks in the Storage Manager, minimizing the effects of lock contention that are observed

in most database systems [PJHA10, JPA09].

4.3.4.2 Indexes

SharedDB/TX implements only B+-Tree based indexes. Yet, the design of the system does not exclude other indexes, like R-Tree or hash indexes. We chose B+-Tree indexes as they support range queries, which are not supported in hash indexes, and are slower on R-Tree indexes.

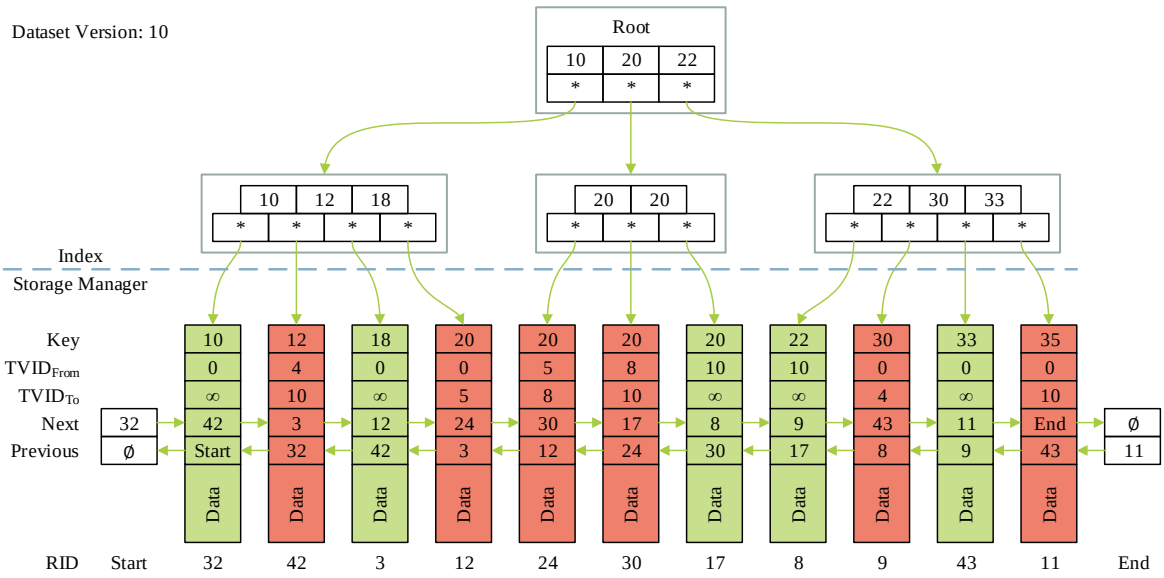
The implemented indexes are based on existing research and techniques – we do not claim novelty here. The big difference to existing implementations is that, similarly to the Storage Manager, no locks are required to synchronize access to inner or leaf nodes, as the WSMV Storage Operator uses a single thread that executes all queries and updates. This renders modern implementation, like the BW-Tree [LSL13], unsuitable for SharedDB/TX.

To increase performance, the B+-Tree allocates inner and leaf nodes that are small enough to fit just into a couple of cache lines. Figure 4.13a shows an instance of the implemented B+-Tree. The inner and leaf nodes of the tree hold only the key of the tuple, while the version identifiers are stored in the raw tuple.

Indexes in the WSMV Storage Engine operator are only used during reads and writes. The validation phase does not have to access any of the indexes, since it relies on the `ReadSet` and `WriteSet` of each transaction. As explained in Section 4.3.3, both sets use tuple pointers to keep track of read tuples and accessed ranges of tuples. Figure 4.13b shows an example. The same applies to the `COMMIT` phase, in which all tuples of the `WriteSet` are visited and their $\texttt{TVID}_{\texttt{From}}$ and $\texttt{TVID}_{\texttt{To}}$ attributes are modified.

The fact that the validation and the commit phases do not access the indexes and that do not modify the chains of records, allows for an interesting optimization: the validation and commit queries can be handled by a different thread that access the same data as the main operator thread. Figure 4.14 shows exactly this optimization.

No synchronization mechanism is required to implement this optimization. The chain of records is only modified when a new tuple is inserted or removed in the chain. Because validation always accesses tuples in a forward fashion, the main thread should ensure that the order of insertion or removal operations do not break the point at any time. Figure 4.15 shows this procedure for both an insertion and a removal of a record in the chain. Records are only removed as part of a garbage collection, a process that will be explained in Section 4.3.4.3.



(a) Overview of a WSMV Storage Engine's Index

Operation	VID	RIDs	Explanation
Read Key 20	9	[3–9]	To verify that the read was ok, the range between the previous of Record 12 and the next of Record 30 has to be checked. At VID 9, when this operation is executed, Records 17 and 8 did not exist.
Read Range 11–30	5	[32 – 43]	To guarantee that no tuples have been inserted or deleted in the range, the range between Record 32 and 43 has to be revisited. Record 32 has the largest key to the beginning of the range, while record 43 has the smallest key to the end of the range.
Delete 20	10	[17]	To ensure no collisions, the deleted tuple (`RID` = 17) should still have the $\texttt{TVID}_{\texttt{To}}$ equal to infinity.
Insert 22	10	[8]	The `WriteSet` in this case contains only the new tuple. Validation should take into account any possible constraints in the schema, i.e. unique or referenced keys.

(b) `Readsets` `WriteSets` in SharedDB/TX. `RIDs` are based on the example of Figure 4.13a

Figure 4.13: `ReadSets` and `WriteSets` in a WSMV Storage Engine Index

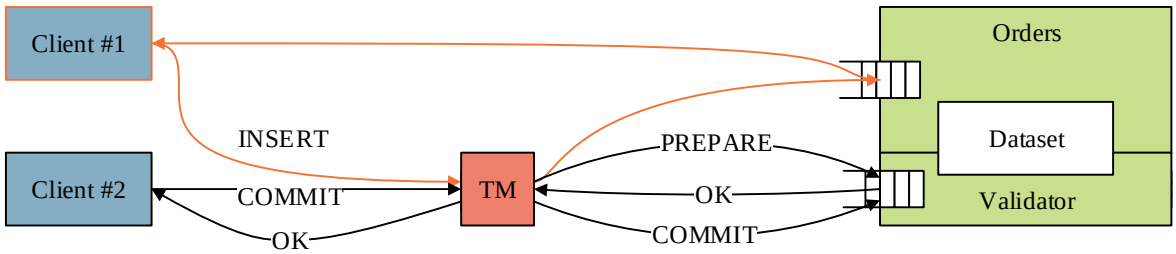
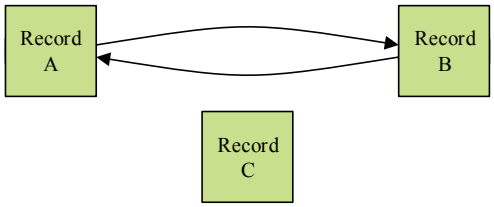


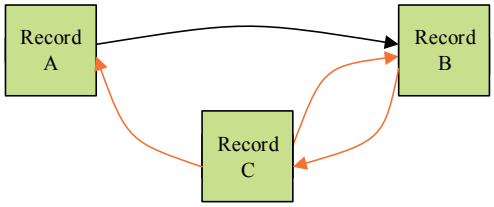
Figure 4.14: Global Query Plan using Dedicated `PREPARE` and `COMMIT` Threads

Insert Tuple C



Initial State

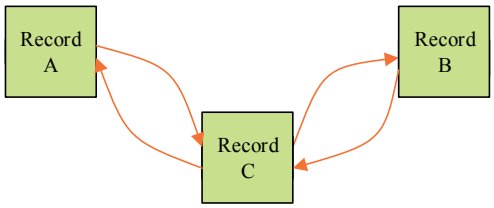
Update Backward Chain




```
C.next = B;
C.prev = A;
B.prev = C;
```

The operator thread modifies the chain, while the validation thread is running. The validation thread is accessing only the forward chain, so this part is not affecting the validation process.

Update Forward Chain



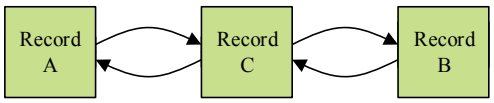

```
sfence();
A.next = B;
```

Modify the next of A to update the forward chain. If the validation reads the old value, it will continue on Record B, while if it reads the new value it will continue on Record C. In any case, Record C can never affect the validation process. A newly inserted record always has $TVID_{From}$ set to a transaction identifier, thus it is not visible by all other transactions. The store fence command is required to guarantee the order of writes (it is explicit on modern architectures).

Figure 4.15: Modifying the Record Chain without Affecting any Concurrent `PREPARE` or `COMMIT` Operations.

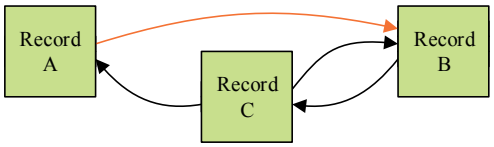
Durability At the time of writing this dissertation, there is no durability mechanism implemented in SharedDB/TX, yet this is not a limitation of the system. In this section

Remove Tuple C
(Garbage Collect)



Initial State

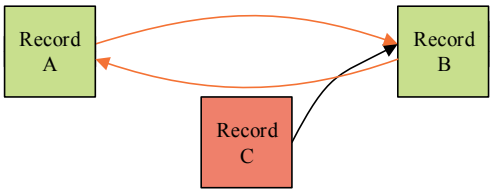
Update Forward Chain



`A.next = B;`

The operator thread modifies the chain, while the validation thread is running. If the validation thread reads the old value, it will continue on Record C. In the case it reads the new value, it will continue on Record B.

Update Backward Chain



`B.prev = A;`

Fix the backward chain by modifying the previous record of B. Record C is not actually deallocated at this point and the forward pointer to B is still kept. That will allow the validation thread to move forward if it currently checks Record C. Record C is only deallocated when a new batch of Transaction Manager queries is received.

Figure 4.15: Modifying the Record Chain without Affecting any Concurrent `PREPARE` or `COMMIT` Operations. (continued)

we will discuss a possible implementation of a durability mechanism for SharedDB/TX. The design of the systems favors fuzzy checkpoints and logical logging of operations. This approach has been extensively studied in literature [GR92]. Crescando/RB [Unt12], a work sharing relational table, implements the same approach which introduces a minimal overhead to the overall performance.

Checkpointing is performed by issuing a select all query that fetches all records to a temporary memory buffer. Once all of the tuples have been fetched, a dedicated thread flushes the buffer to a file in persistent storage. In order to ensure durability, two files are required: one that contains the last complete checkpoint, and a second one that is possibly an incomplete checkpoint. The process of selecting tuples and flushing them to disk repeats, alternating the checkpoint file.

For bigger datasets, where it is not possible to create another copy of the data in main memory, implicit horizontal partitioning is used. Each partition is flushed to a different file, in incremental order. A total of $p + 1$ files is required, where p is the number of horizontal partitions. Additional metadata in the checkpoint files hold the current `VID` of

the dataset, in order to allow for proper recovery.

In order to ensure that a transaction that committed between two checkpoints is durable, physical logging is used. Transactions are logged once committed but before the client is informed about the commit. This assumes that if the system crashes and the client misses a confirmation, the client will reissue the transaction. Finally, the size of the log should be big enough to hold all the transactions that are executed between two checkpoints.

Recovery is performed by reading the p checkpoint files, except the latest modified file, which is probably incomplete. Once the dataset is recovered, the transaction log is replayed, bringing the system in a consistent state.

4.3.4.3 Garbage Collection

Finally, the last component of the MVCC Storage Engine is the garbage collector. In multi versioning systems, tuples are never removed from the system during query execution, as they may still be used by other concurrent transactions that access older snapshots of the data. The garbage collector takes care of cleaning up the old versions of tuples, making sure that no transaction requires access to them.

There are different approaches to cleaning up old versions. The most common across multi versioning systems, is scan-and-remove. In this approach, all running transactions and their versions are monitored, keeping track of the oldest required version to answer them. One or more dedicated garbage collecting threads periodically issue scan queries that search for any tuples that have been deleted prior to that lowest version. These tuples are then deallocated by the same threads, adding the necessary synchronization mechanisms to ensure thread-safety.

Scan-and-remove is a viable approach for SharedDB/TX's garbage collector. Yet, we can take advantage of the fact that only a single thread (the operator thread) has access to each relation. This means that this thread can gather all information about which tuples have been deleted and which transactions have started or finished.

Our approach is based on an ordered data structure that allows random access to elements. This has been implemented as an ordered N-ary tree, but any other data structure with the same properties is appropriate. Every element in this data structure is called Version Garbage Element (VGE) and is associated with one specific version of the dataset, including past versions, aborted versions, as well as future versions (i.e. uncommitted versions). Additionally, the VGE uses a reference counter to keep track of how many transactions are interested in the specific version. Finally, each VGE contains a linked list of pointers

to tuples that have been deleted in that version. Figure 4.16 shows an example of this data structure.

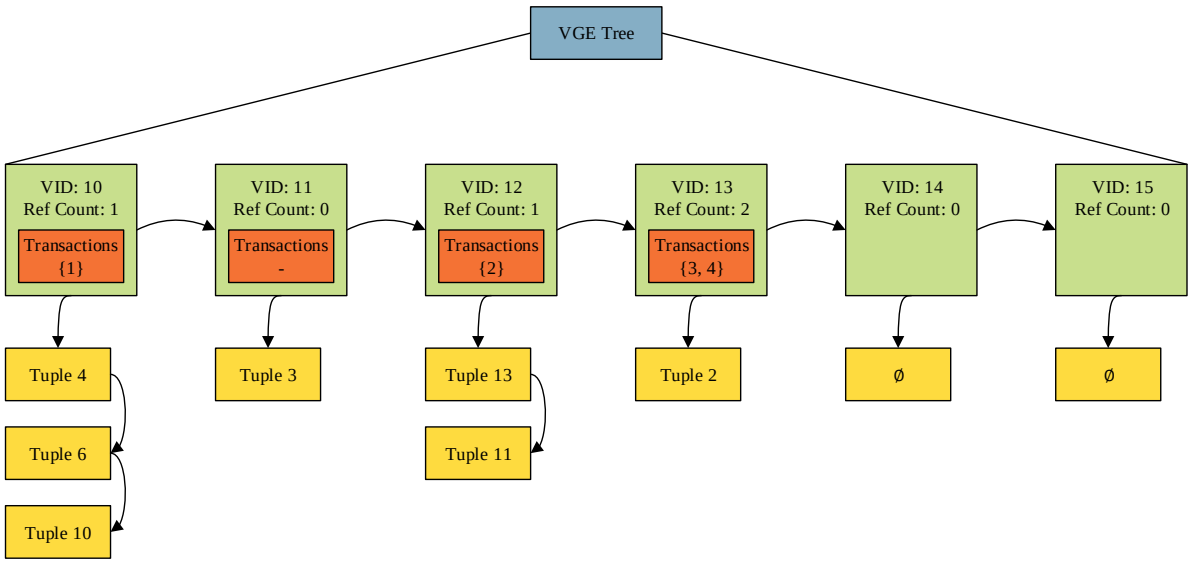
When a new transaction t starts, it is given the latest `VID` of the dataset, T_v which is used to read the most recent version of the data. This means that t has access to all tuples that have been created before T_v except for the tuples that have been already deleted before T_v. Thus, if t is the only transaction in the system, any tuple with $\texttt{TVID}_{\texttt{To}} < T_v$ can be deallocated safely. In contrast, any tuple with $\texttt{TVID}_{\texttt{To}} \geq T_v$ is still visible. In order to maintain this information in the garbage collector, the VGE associated with version T_v has its reference counter incremented.

All dataset updates caused by transaction t, are stored in the transaction's `WriteSet`, as explained in Section 4.3.1. Before committing a transaction, a new `VID`, $T_{\texttt{CommitVID}}$ is assigned to it. Any modified tuples will use this `VID` to either mark that they are visible from this point on, or mark that they have been deleted in this version. Deleted tuples are also inserted in the linked list of the VGE that is associated with $T_{\texttt{CommitVID}}$. This means that if no other transaction is interested in this version, the chained tuples can be safely deallocated. Finally, when the transaction is committed or aborted, the reference counter of the initial version t_v is decremented, since this transaction has finished and is no longer interested in that version.

A VGE with reference counter equal to 0 means that no transaction is interested in the specific version. However, other transactions may require access to earlier versions. As a result, VGEs with a zeroed reference counter are not considered for garbage collection, unless the VGE is the first one in the sorted data structure (i.e. lowest `VID`). In this case, all chained tuples can be deallocated. Additionally, this process is repeated as long as the first VGE has reference counter equal to 0.

To visualize this, consider the VGEs of Figure 4.16 and the order of operations shown in Figure 4.16c. In this case T_2 commits first and as a result the VGE that belongs to version 12 has zero references. However, the associated tuples, tuple #13 and #11, cannot be deallocated, as transaction T_1 is still able to read them. Transaction T_1 has started at version 10 and has not committed yet. This means that all tuples that were deleted in version 12 should appear for T_1. T_3 and T_4 are committed and again, no tuples are deallocated. Finally, T_1 commits at step 4. This decrements the counter of VGE 10, and since this is the first VGE in the tree, the linked tuples can be freed. Once VGE 10 is removed, the next VGE is examined. In this case VGE 11 has already a zeroed reference counter, thus tuple #3 can be freed. The process continues until the first VGE has a positive reference counter.

Finally, if a transaction is aborted, all tuples that were inserted by this transaction can be



(a) Garbage Collector Meta-Data

Step	Action	Result
1	Commit T_1	Decrement Ref Count of `VID` 10
		Clean `VID` 10: Deallocate Tuples 4, 6, 10
		Clean `VID` 11: Deallocate Tuple 12
2	Commit T_2	Decrement Ref Count of `VID` 12
		Clean `VID` 12: Deallocate Tuples 13, 11
3	Commit T_3	Decrement Ref Count of `VID` 13
4	Commit T_4	Decrement Ref Count of `VID` 13
		Clean `VID` 13: Deallocate Tuple 2

(b) Sample Execution #1: `COMMIT` T_1, T_2, T_3, T_4

Step	Action	Result
1	Commit T_2	Decrement Ref Count of `VID` 12
2	Commit T_3	Decrement Ref Count of `VID` 13
3	Commit T_4	Decrement Ref Count of `VID` 13
4	Commit T_1	Decrement Ref Count of `VID` 10
		Clean `VID` 10: Deallocate Tuples 4, 6, 10
		Clean `VID` 11: Deallocate Tuple 12
		Clean `VID` 12: Deallocate Tuples 13, 11
		Clean `VID` 13: Deallocate Tuple 2

(c) Sample Execution #2: `COMMIT` T_2, T_3, T_4, T_1

Figure 4.16: Garbage Collection in SharedDB/TX.

deallocated at any point. Yet, in order to reduce the complexity of the `ABORT` operation, they are instead added to the chain of tuples that belongs to the first VGE. This means that whenever the transaction that is interested in the oldest version finished, these tuples can be deallocated.

4.4 Experimental Evaluation

In order to evaluate the performance of SharedDB/TX, we carried out a series of performance experiments that are based on a modified version of the TPC-W benchmark. The modifications were necessary to introduce conflicts across different clients.

As stated in Section 2.4.2, the TPC-W benchmark requires only session consistency, as the writesets of all clients are isolated. This means that a write by a client will never conflict with another concurrently running client. This is common in e-commerce systems, where each customer places orders that are not accessible by other customers.

To artificially create conflicts, we allow customers to use multiple clients concurrently. This is the equivalent of a customer ordering an item from one device while simultaneously checking the pending orders on a second device. In this case, in order to ensure serializability, the second device should always access a consistent snapshot.

In all experiments reported in this section, we used a 48 core machine as a database server. This machine features four twelve-core AMD Opteron 6174 ("Magny-Cours") sockets and is equipped with 128 GB of DDR3 1333 RAM. Each core has a 2.2 GHz clock frequency, 128 KB L1 cache, 512 KB L2 cache, and is connected to a shared 12 MB L3 cache. The operating system used in all experiments was a 64-bit SMP Linux. The emulated browsers were executed on up to eight client machines, each having 16 CPU cores and 24 GB of DDR3 1066 RAM. The clients also ran the application logic; that is, the clients issued queries directly to the database server. This is a slight simplification of the TPC-W set-up and justified because we were interested in the performance of the database system under high load. The client machines were connected to the database server machine using a 1 Gbps ethernet.

4.4.1 Baseline

To put the performance of SharedDB/TX into perspective, we compared it against the version of SharedDB that does not implement transactions, as it was presented in Chapter 2. While the two systems have the same design, they have a number of differences in implementation.

First of all, SharedDB/TX has to serialize all operations before preparing and committing them. This introduces a single point of serialization, which even though it is heavily optimized, it can become a bottleneck under high loads. Additionally, operations in SharedDB/TX have to be repeated if a conflict is detected. In order to be compliant with the TPC-W benchmark, an operation with any necessary repetitions has to be executed in the required SLA.

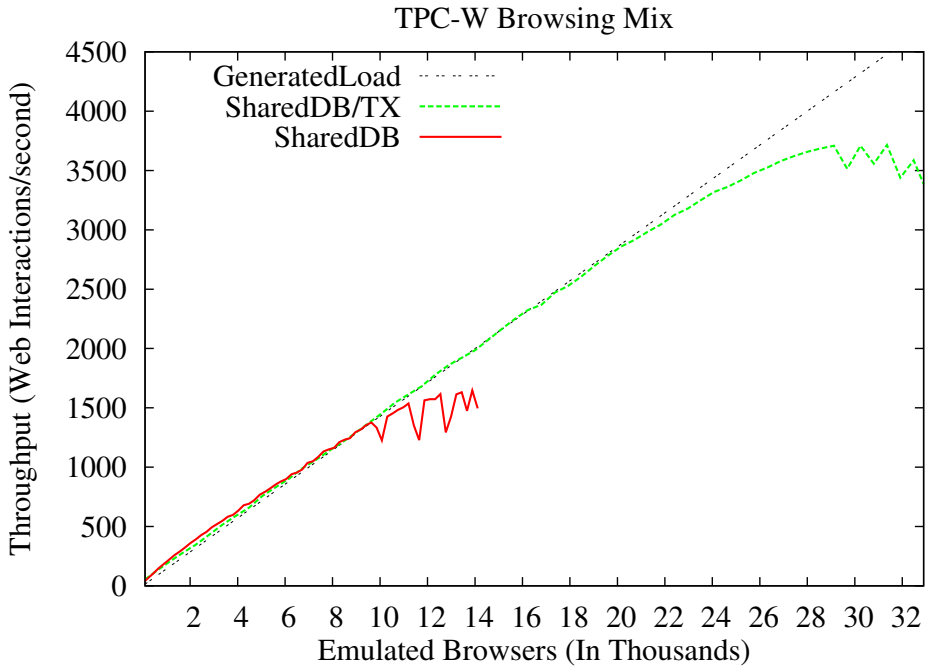
On the other hand, SharedDB/TX is more optimized while processing intermediate results. While SharedDB has to materialize all intermediate results before processing them, SharedDB/TX can skip all materialization and only materialize before shipping the results to the clients. This is possible because storage level tuples are immutable; all updates are reflected as a newer version of the tuples.

4.4.2 Performance under Varying Load

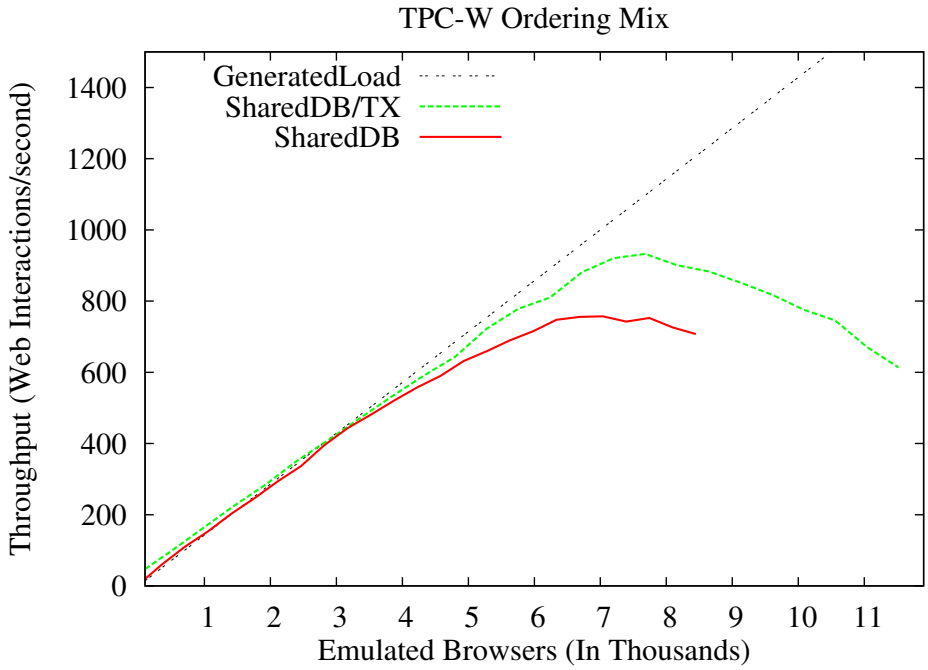
In the first set of experiments, we compared the performance of SharedDB and SharedDB/TX on the three workload mixes of TPC-W. We varied the load of the system by increasing the number of emulated browsers (clients) and measured the web interactions that were successfully answered by the system in the response time limit that is defined by the TPC-W specification. All web interactions that exceeded this limit were not accounted as successful. For SharedDB/TX we used up to 10 different clients per customer, in order to artificially inflict transaction conflicts. For this experiment, we configured the database system to use 24 CPU cores. We used the same global query plan for all three mixes. Taking into account the probabilities of queries appearing in the mix, would generate a better, more optimized global query plan. Nevertheless, the goal of this Chapter is not to optimize the query plan, rather than demonstrate that transactions can be efficiently implemented in work sharing data processing systems.

Figure 4.17 shows the results. As expected, the Browsing mix favors SharedDB/TX as it is dominated by read intensive queries. Furthermore, these read queries include a significant portion (11%) of heavy, analytical queries that have to process high number of tuples. SharedDB/TX achieves more than two times higher performance compared SharedDB due to the lack of intermediate result materialization. Furthermore, since the amount of updates in the Browsing mix is small, few collisions and rollbacks occur.

The performance of the two systems is fairly similar in the Ordering mix, a write-intensive mix of queries. The amount sharing work is very small, as the queries are very small. Furthermore, SharedDB/TX performance drops as the load increases. This is the result of two factors. First, all writes have to be serialized before commit, which under high loads

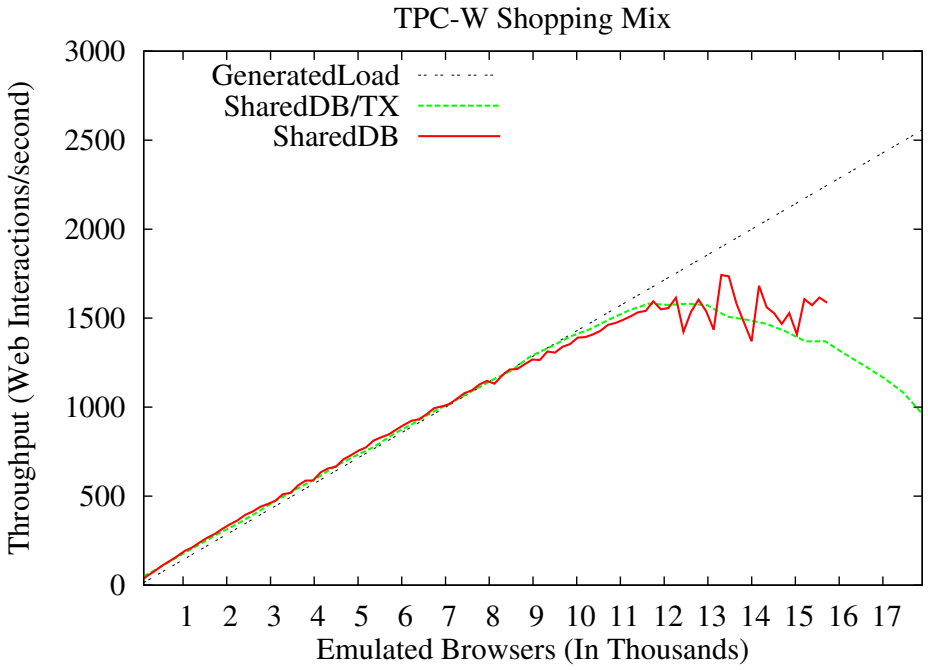


(a) TPC-W Browsing Mix



(b) TPC-W Ordering Mix

Figure 4.17: TPC-W Throughput: Varying Load, All Mixes



(c) TPC-W Shopping Mix

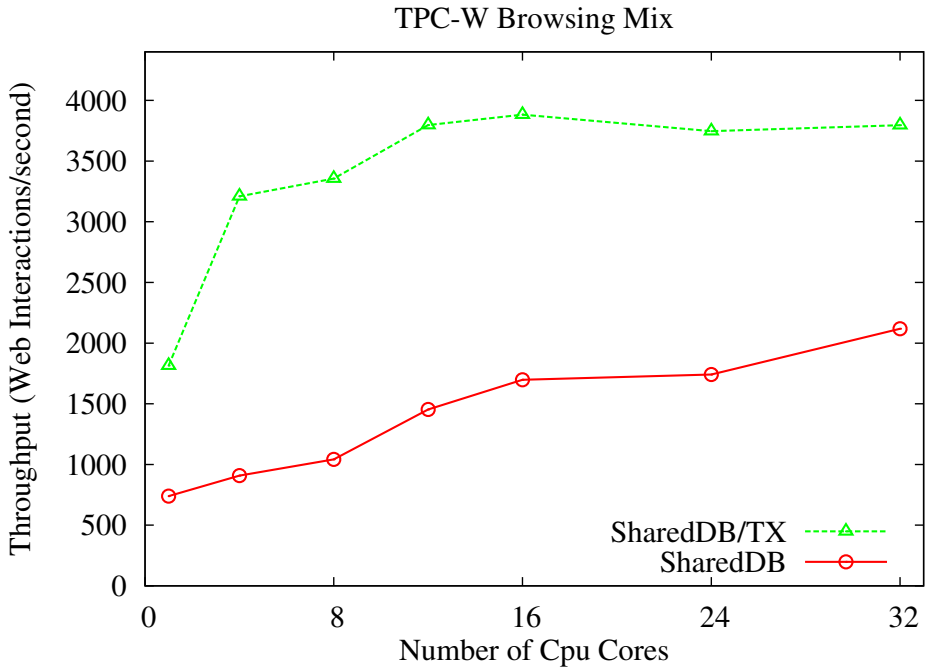
Figure 4.17: TPC-W Throughput: Varying Load, All Mixes

results in queueing of operations. In addition to serialization, writes may conflict and as a result the whole transaction has to be repeated which further decreases the performance of the system.

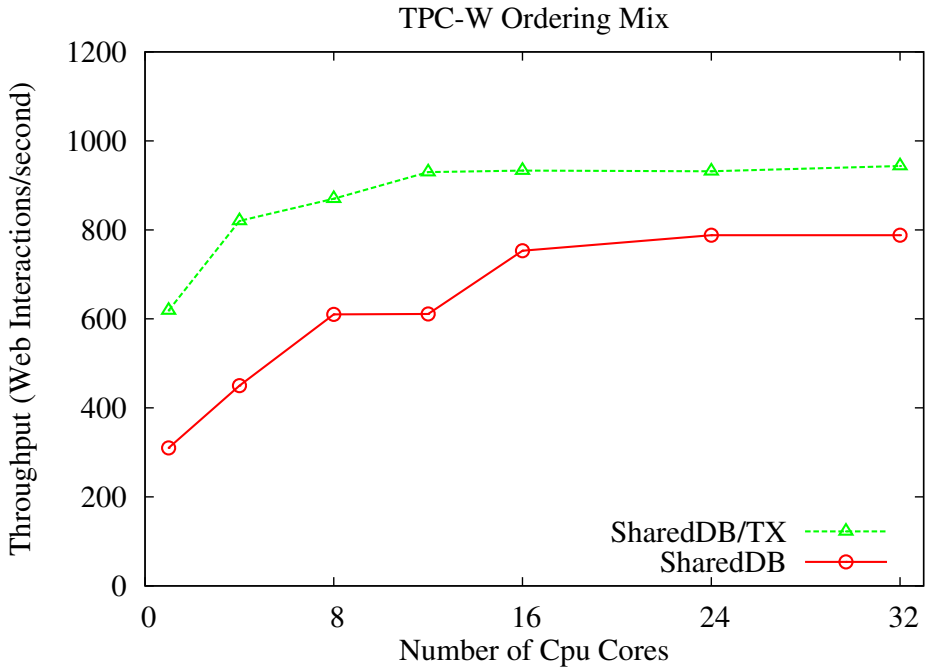
Finally, the Shopping mix has a similar performance on both systems. The Shopping mix contains a balanced amount of heavy analytical queries and writes. While SharedDB/TX should benefit from the lack of result materialization for the heavy read queries, write conflicts require repetitions of both read and write operations. Nevertheless, this benchmark shows that introducing transactions in a data processing system has a minimal overhead in the performance. On the contrary, under read intensive workloads, performance is improved.

4.4.3 Scaling with the Number of Cores

In the second set of experiments, we explored the impact of having additional CPU cores on the database server. For this reason, we varied the number of available CPU cores of the database server machine from 1 to 32 and repeated the experiments of Section 4.4.2 for each configuration. We measured the maximum number of successful web interactions

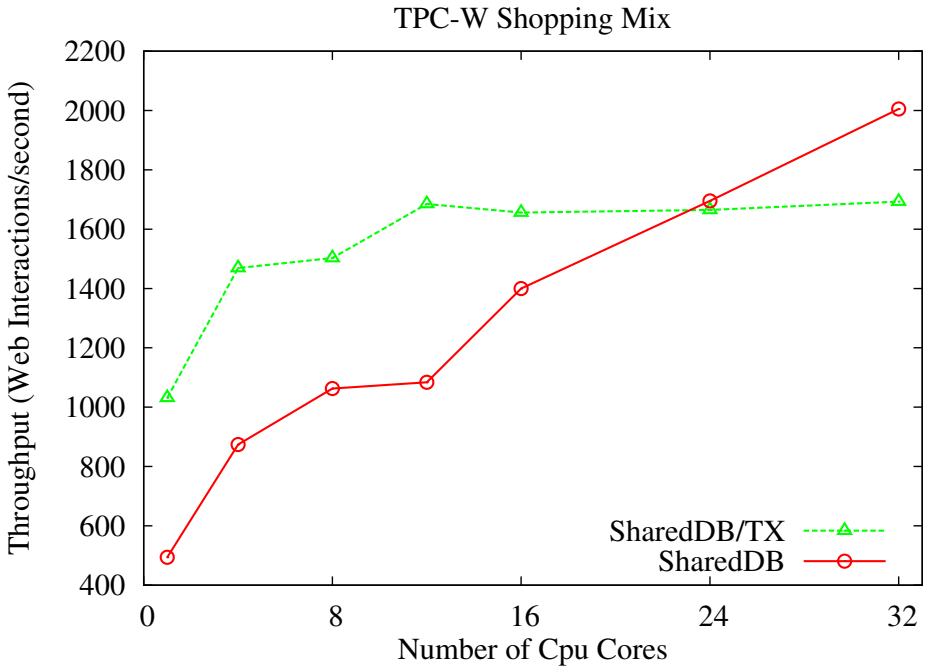


(a) TPC-W Browsing Mix



(b) TPC-W Ordering Mix

Figure 4.18: Max. Throughput: Vary # Cores, All Mixes



(c) TPC-W Shopping Mix

Figure 4.18: Max. Throughput: Vary # Cores, All Mixes

per second each system could achieve.

Figure 4.18 shows the results. Introducing transactions with multi version concurrency control, increases the performance of SharedDB by a factor of 2 to 3.5 for read-intensive workloads, like the Browsing mix. With just 12 CPU cores, the system is saturated. This is because of the prepare phase that is required before committing a transaction that has to repeat all reads to validate that no new data have been introduced in the working snapshot. Obviously, the performance can be further increased if the validation phase is parallelized. Yet the goal of this experiment is to show what is the peak performance of single core operators.

The Ordering mix results, depicted in Figure 4.18b, shows that SharedDB/TX is slightly better for write-most workloads as well. Even though transactions may conflict and they have to be repeated, the overall performance is better, which is mostly because of the small percentage of lightweight read queries in the mix. These queries have a smaller overhead due to late result materialization and if a conflict is detected, repeating the queries introduces only a small penalty. Finally, the Shopping mix results validate the experiment of Section 4.4.2. When the workloads consists of both analytical queries and a significant percentage of writes, transaction handling becomes a bottleneck. This is

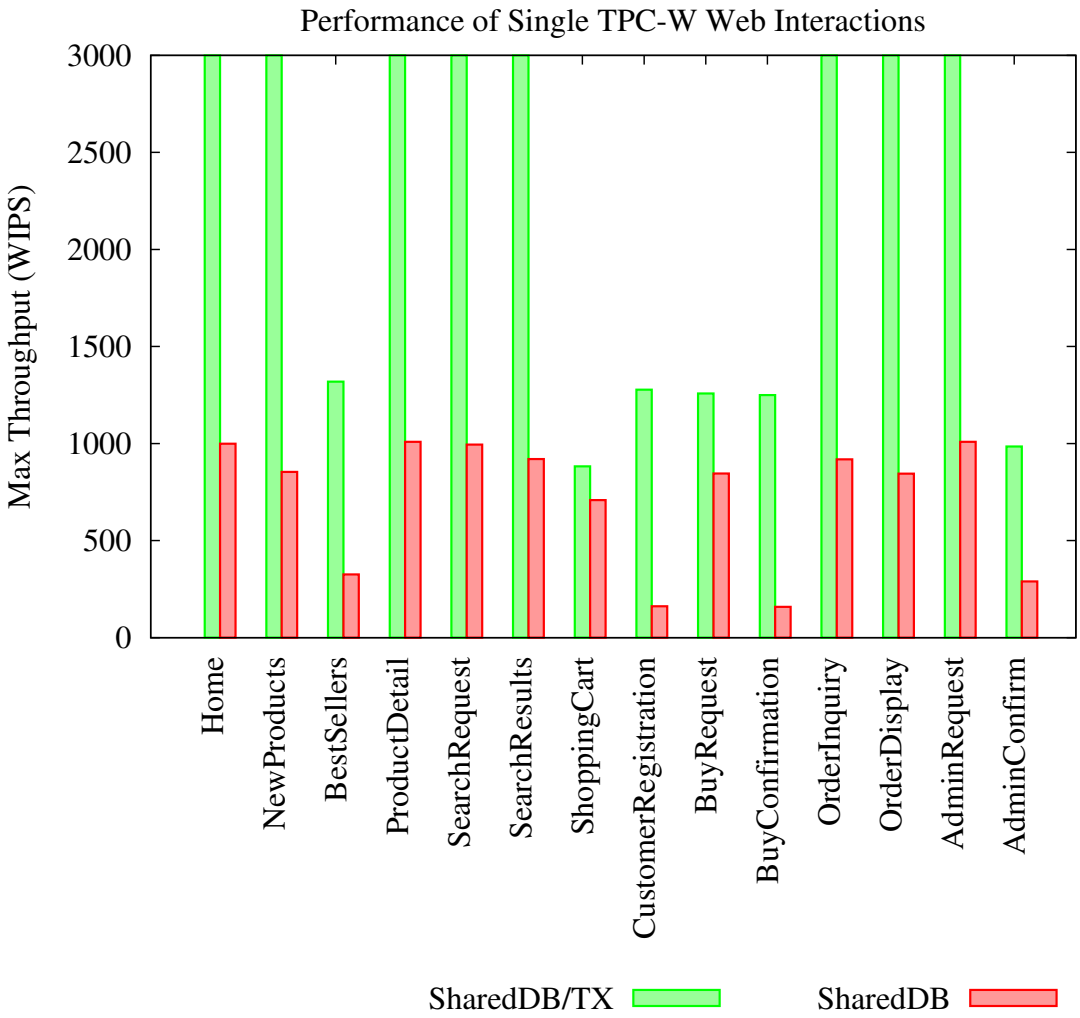


Figure 4.19: Analysis of Individual Web Interactions

due to transaction conflicts that may require repetition of the analytical queries. Since the analytical queries of TPC-W include full table scans, any committed updates require repetition re-execution of the queries and as a result more full table scans.

4.4.4 Analysis of Individual Web Interactions

Next we examined the performance of the two systems for each TPC-W web interaction. The TPC-W benchmark involves a variety of different web interactions, each involving a different set of queries. For instance, the home web interaction involves two simple point queries (fetching promotion articles and a user's profile). Other web interactions involve point queries and several updates. Finally, there are also web interactions that involve

heavy, analytical queries with multiple joins, grouping, and sorting. Figure 4.19 shows the maximum throughput that each of the two systems can achieve if the clients are configured to issue only queries that correspond to a single web interaction. These experiments were carried out in a configuration with 24 cores for the database server.

SharedDB/TX performs better for every single web interaction. This is expected for read only web interactions, like the best sellers web interaction. For these web interactions, the gains surpass a 3x factor. Surprisingly, SharedDB/TX is better for web interactions that include updates.

The case of the Shopping Cart web interaction is the most interesting. This web interaction contains the following queries:

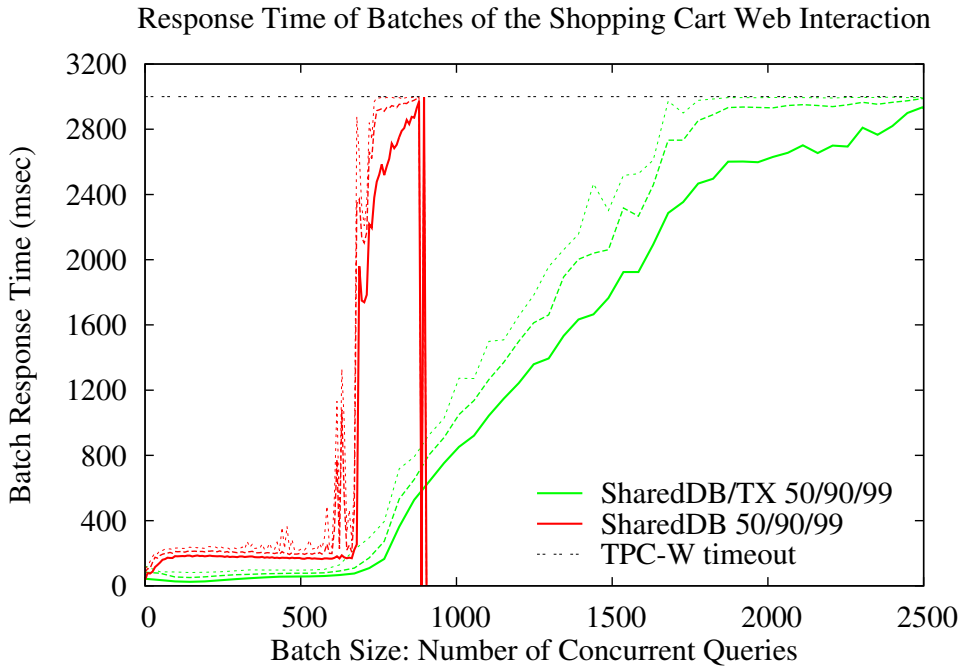
```
Clear Cart:
DELETE FROM shopping_cart_line WHERE scl_sc_id = ?;

List Cart:
SELECT * FROM shopping_cart WHERE sc_id = ?;
SELECT * FROM shopping_cart_line, item
WHERE scl_sc_id = ? AND scl_i_id = item.i_id;

Update Cart Time:
UPDATE shopping_cart SET sc_time = NOW() WHERE sc_id = ?;
```

Since multiple client devices can access the same shopping cart, the number of conflicts in this web interaction is very high. In fact, only one transaction per customer can be committed at a time, since there is always a conflict between the `UPDATE` and the `SELECT`. Still, SharedDB/TX is able to perform slightly better than vanilla SharedDB.

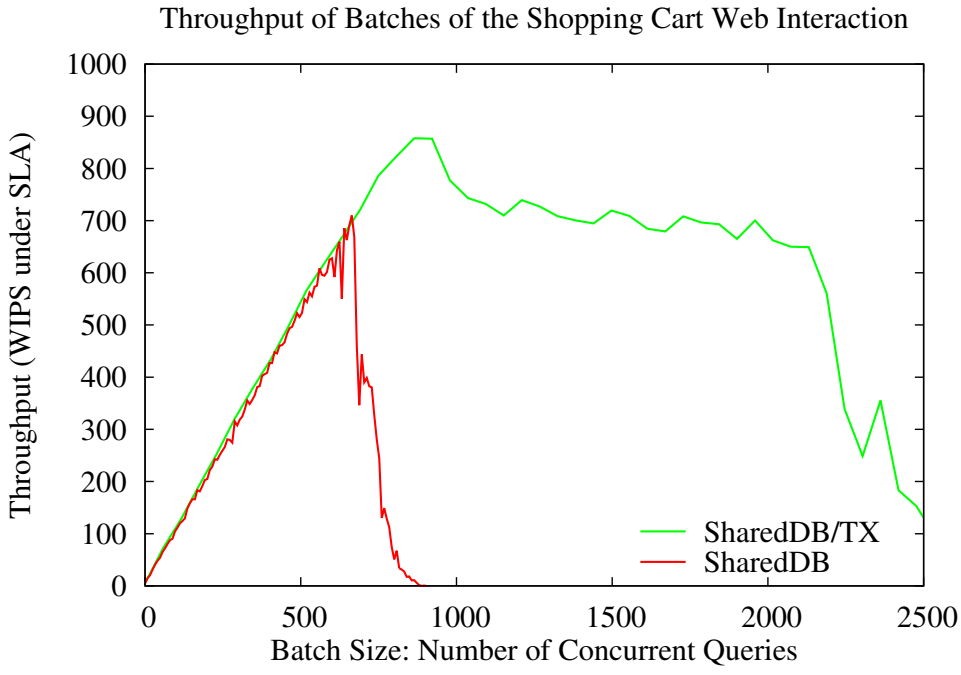
Figure 4.20 shows the throughput and response time of both systems under a workload composed of Shopping Cart Web Interactions only. While the response time of SharedDB/TX is much lower than the TPC-W timeout value, throughput does not scale more than 900 WIPS. This is because 90% of the web interactions have to be repeated due to conflicts. Finally, we observe that in both systems the 50th, 90th and 99th percentile are relatively close, showing the robustness of work sharing data processing.

We repeated the same analysis for the best sellers web interaction which contains a single query:

```
SELECT *, SUM(order_line.ol_qty) AS val
```

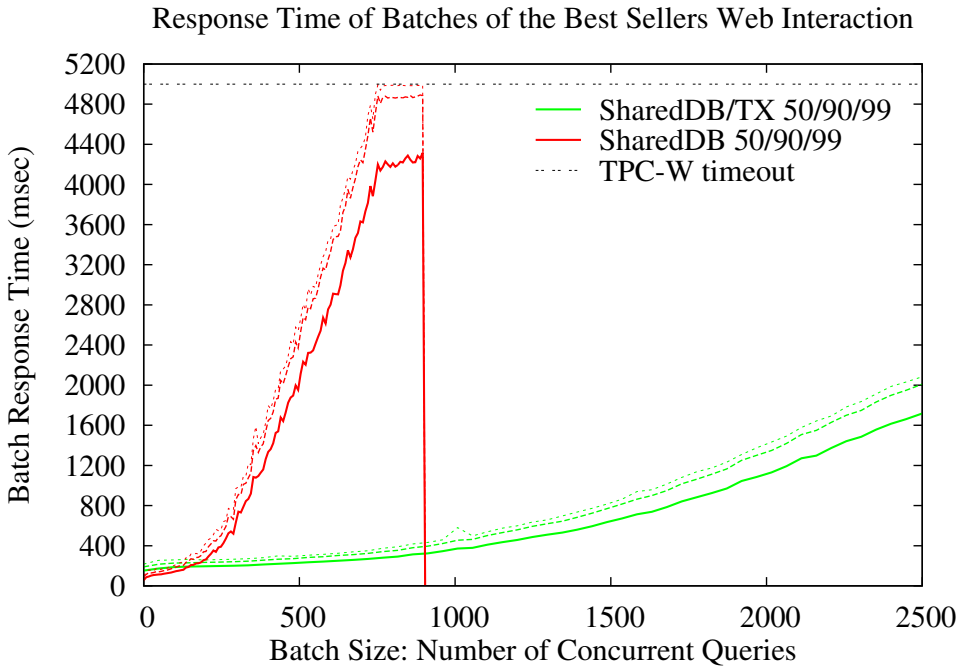


(a) Response Time

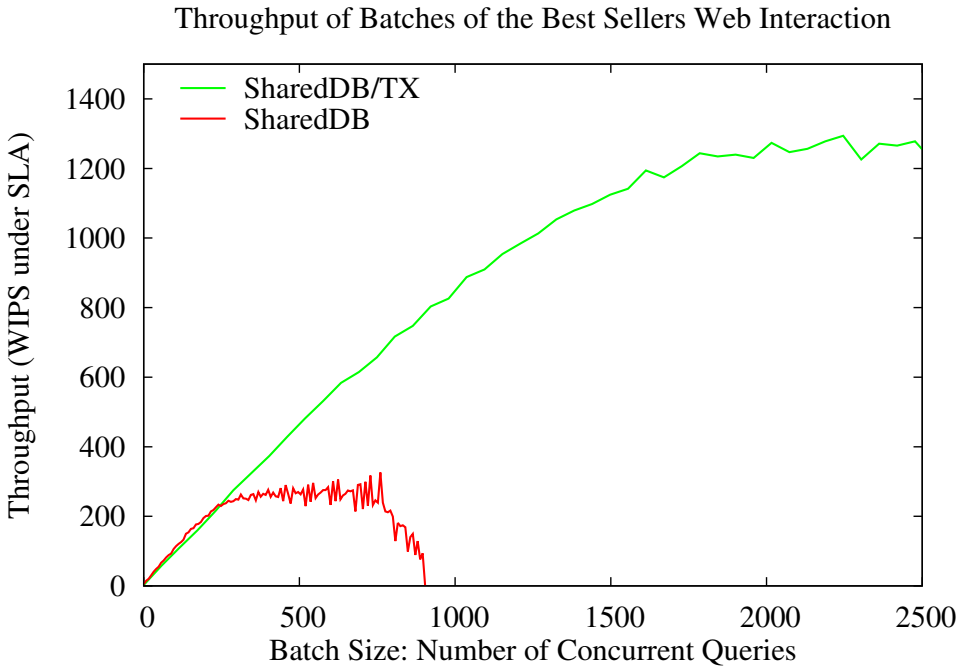


(b) Throughput

Figure 4.20: Analysis of the Shopping Cart Web Interaction



(a) Response Time



(b) Throughput

Figure 4.21: Analysis of the Best Sellers Web Interaction

```
FROM
  (SELECT * FROM orders ORDER BY o_date DESC LIMIT 0,3333) AS latest,
  order_line, item, author,
WHERE order_line.ol_o_id = latest.o_id
  AND item.i_id = order_line.ol_i_id
  AND item.i_subject = ?
  AND item.i_a_id = author.a_id
GROUP BY i_id
ORDER BY val DESC LIMIT 0,50
```

The throughput and response time of both systems are presented in Figure 4.21. SharedDB/TX scales more gracefully to bigger loads and achieves almost 6 times higher performance, which is justified by the lack of result materialization. Nevertheless, performance does not scale beyond 1,300 WIPS, even though response time is considerably smaller than the required SLA. This is because of the validation phase, which as explained before is a serial operation.

4.5 Concluding Remarks

This chapter presented an extension to SharedDB that supports ACID transactions on a work sharing data processing architecture. The implementation, SharedDB/TX, is based on multi version concurrency control, as it requires no blocking in case of concurrent updates.

The implementation requires very few modifications to the SharedDB code. In fact, all data processing operators can be used without any modifications. A new type of storage engine operators is required though, in order to support multiple versions of tuples. This multi-version storage engine is based on the same abstract operator model of SharedDB. Furthermore, the transactional logic is handled by a dedicated Transaction Manager operator which maintains the read sets and write sets of all active transactions, as well as handles the commit procedure.

SharedDB/TX uses a 2 phase commit approach, similarly to most MVCC based systems. In the first phase inserted data violations are checked and read accesses are repeated in order to verify that no tuples have been modified in the accessed snapshot. If all storage engines validate the isolation guarantees, the second phase of committing the writesets takes place. This adds a single serialization point to SharedDB/TX: no transactions are allowed to commit while another transaction is either validating or committing.

Nevertheless, experiments on SharedDB/TX demonstrate that this serialization point has a very small overhead to the performance of the system. In fact, SharedDB/TX is able to outperform vanilla SharedDB when the workload is not write intensive by a factor of 2–6. The reason of this performance gain is that SharedDB/TX requires no intermediate result materialization since storage level tuples are immutable (all updates are reflected with a newer version of the tuple). Vanilla SharedDB has to materialize all intermediate results, as concurrent updates are applied in-place.

In write-most workloads, SharedDB/TX has a very small advantage compared to vanilla SharedDB. The performance gain comes mostly from any read queries that are present in the workload. Finally, in workloads that combine heavy analytical queries with high number of updates, SharedDB/TX has a slightly worse performance. The reason is that any conflict across writes, forces a re-evaluation of the analytical queries.

Last but not least, this chapter introduced a novel garbage collection mechanism for multi version storage engines in SharedDB. The garbage collector relies on knowledge of all concurrently executed queries, as well as their writesets. This information is very hard to collect and maintain in a query-at-a-time system, where queries arrive and are executed asynchronously. Yet, in a batch processing system, where multiple queries arrive and finish at the same time, it is trivial to collect and more importantly, requires no synchronization mechanism.

5

Conclusions

To support modern workloads, database systems must be able to execute high loads of small transactional queries, as well as long running, heavy analytical queries. More importantly, they should exhibit a robust performance, both to the amount of queries and to their complexity. This is the case for most systems that provide real-time analytics and support services like travel reservation, insurance, and social networking.

Traditional relational database solutions fail to meet these requirements. Most of these systems evaluate each request individually, and a result when multiple requests are executed concurrently, contention arises, leading to unpredictable performance. To make matters worse, this affects all running queries, rather than the ones that cause the contention.

To this end, this dissertation presented SharedDB, a system that implements an alternative data processing approach. SharedDB uses a push-based data flow network of always-on operators that process queries and updates in batches. Deep, aggressive sharing in terms of both computation and resources is the key component of the design. Any common work across all concurrently running queries is evaluated only once, leading a more predictable performance. Additionally, instead of assigning resource to individual queries, SharedDB assigns resources to operators. This allows heavy queries that use more processing operators to be isolated from lightweight queries. In general, SharedDB is able to support a high throughput of queries, while ensuring system robustness and proper query isolation.

SharedDB benefits greatly from sharing work across queries, yet sharing too much work may result in a worse performance. For this reason, this dissertation presented a work sharing query optimizer (WSO) that generates a global query plan, able to evaluate the whole workload of the system. The generated query plan may not result in best performance per individual query. Yet, it results in a better performance, and most importantly, more predictable performance, for the whole workload. The task of identifying the best plan is not trivial, as it involves two decisions that affect each other's solution space. The first part of the problem is to define the order of operators, while the second part is to define which queries will share an operator. The presented algorithm is based on the branch and bound optimization technique and generates such optimized global query plans in a couple of seconds.

Finally, the last part of the dissertation presented how existing transaction processing techniques can be applied in a work sharing system like SharedDB. Transaction processing requires very small modifications to the core of SharedDB, which shows the flexibility of the system design. The implementation is based on multi version concurrency control. The used techniques originate from query-at-a-time processing and have been extended to the support the design of SharedDB.

SharedDB, WSO and the work sharing transaction manager, form a fully functional relational database system, SharedDB/TX, that consists of roughly 118,000 lines of C++. These numbers do not include a wide range of tools, unit tests, and benchmark drivers written in various languages.

Because SharedDB/TX is a complete, practical system, it touches many deep technical issues, from NUMA-aware and cache-aware algorithms to query optimization techniques. For some of these topics, we feel that we have barely touched the surface. For example, SharedDB/TX will benefit greatly from a runtime NUMA optimizer that distributes operators to the appropriate NUMA nodes. Also, a number of well-studied, optimized data processing algorithms can be implemented in SharedDB, to further improve the performance. Finally, since all processing operators are independent, heterogeneous hardware may increase the performance of the system even more. For instance, certain CPU-heavy operators can be offloaded to a dedicated FPGA chip, or even a GPU. While all these ideas have not been studied during this dissertation, the design of SharedDB/TX allows such optimizations.

In this dissertation, we presented the described novel work sharing approach to data processing, as well as, the system design and the implementation details of the concrete implementation of SharedDB/TX. Additionally, this dissertation contributes to, among other things, our scientific understand of how to efficiently share resources and common

work across all queries, without sacrificing the applicability or robustness of the system. Most importantly, this dissertation shows that work sharing is a viable and efficient solution solution to data processing, and is able to support modern workloads better than existing solutions.

We hope that, in the future, others will continue the research where we had to move on in the interest of building a complete system, and we hope that SharedDB/TX will inspire many more exciting systems and services in the domain of work sharing data processing.

Modified TPC-H Queries used in Section 2.4.3

Listing A.1: Modified TPC-H Queries

```
-- Query 2 --
select
  sum(ps_supplycost)
from
  part,
  supplier,
  partsupp,
  nation,
  region
where
  p_partkey = ps_partkey
  and s_suppkey = ps_suppkey
  and p_size = $1
  and p_type like $2
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = $3;

-- Query 3 --
select
  sum(l_extendedprice)
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = $1
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < $2
  and l_shipdate > $2;

-- Query 5 --
select
  sum(l_extendedprice)
from
  customer,
  orders,
  lineitem,
  supplier,
  nation n1,
  region r1,
  nation n2,
  region r2
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = n2.n_nationkey
  and n2.n_regionkey = r2.r_regionkey
  and r2.r_name = $1
  and s_nationkey = n1.n_nationkey
  and n1.n_regionkey = r1.r_regionkey
  and r1.r_name = $1
  and o_orderdate >= $2
  and o_orderdate <
      $2 + interval '1' year;

-- Query 7 --
select
  sum(l_extendedprice) as revenue
from
  supplier,
  lineitem,
  orders,
  customer,
  nation n1,
  nation n2
where
```

```
  s_suppkey = l_suppkey
  and o_orderkey = l_orderkey
  and c_custkey = o_custkey
  and s_nationkey = n1.n_nationkey
  and c_nationkey = n2.n_nationkey
  and n1.n_name = $1
  and n2.n_name = $2
  and l_shipdate between
      date '1995-01-01'
      and date '1996-12-31';

-- Query 8 --
select
  sum(l_extendedprice)
from
  part,
  supplier,
  lineitem,
  orders,
  customer,
  nation n1,
  nation n2,
  region
where
  p_partkey = l_partkey
  and s_suppkey = l_suppkey
  and l_orderkey = o_orderkey
  and o_custkey = c_custkey
  and c_nationkey = n1.n_nationkey
  and n1.n_regionkey = r_regionkey
  and r_name = $2
  and s_nationkey = n2.n_nationkey
  and n2.n_name = $1
  and o_orderdate
      between date '1995-01-01'
      and date '1996-12-31'
  and p_type = $3;

-- Query 9 --
select
  sum(s_acctbal + ps_supplycost)
from
  lineitem,
  part,
  supplier,
  partsupp
where
  l_partkey = p_partkey
  and l_suppkey = s_suppkey
  and p_name like $1
  and l_partkey = ps_partkey
  and l_suppkey = ps_suppkey;

-- Query 10 --
select
  sum(l_extendedprice) as revenue
from
  orders,
  lineitem
where
  o_orderkey = l_orderkey
  and o_orderdate >= $1
  and o_orderdate <
      $1 + interval '3' month
  and l_returnflag = 'R';

-- Query 11 --
select
  sum(ps_supplycost)
from
  partsupp,
  supplier,
  nation
where
  ps_suppkey = s_suppkey
  and s_nationkey = n_nationkey
  and n_name = $1;

-- Query 12 --
select
  count(*)
from
  orders,
  lineitem
where
  o_orderkey = l_orderkey
  and o_orderpriority = '1-URGENT'
  and l_shipmode = $1
  and l_receiptdate >= $2
  and l_receiptdate <
      $2 + interval '1' year;

-- Query 14 --
select
  sum(l_extendedprice)
from
  lineitem,
  part
where
  l_partkey = p_partkey
  and l_shipdate >= $1
  and l_shipdate <
      $1 + interval '1' month
  and p_type like '%PROMO%';

-- Query 16 --
select
  count(*)
from
  part,
  (select
    *
   from
     partsupp
```

```
    where
      ps_suppkey in
       ( select
           s_suppkey
         from
           supplier
         where
           s_comment not like
               '%Complaints%'
       )
  ) as suppcnttbl
where
  p_partkey = ps_partkey
  and p_brand <> $1
  and p_type not like $2
  and p_size = $3;

-- Query 17 --
select
  sum(l_extendedprice)
from
  lineitem,
  part
where
  p_partkey = l_partkey
  and p_brand = $1
  and p_container = $2;

-- Query 19 --
select
  sum(l_extendedprice)
from
  lineitem,
  part
where
  ( p_partkey = l_partkey
  and p_brand = $1
  and p_container like '%SM%'
  and l_quantity >= $2
  and l_quantity <= $2 + 10
  and p_size between 1
  and 5
  and l_shipmode = 'AIR'
  and l_shipinstruct =
      'DELIVER IN PERSON');

-- Query 20 --
select
  sum(ps_supplycost)
from
  part,
  supplier,
  partsupp,
  nation
where
  ps_suppkey = s_suppkey
  and s_nationkey = n_nationkey
  and n_name = $2
  and ps_partkey = p_partkey
  and p_name like $1;

-- Query 21 --
select
  count(*)
from
  supplier,
  orders,
  nation,
  lineitem
where
  s_suppkey = l_suppkey
  and o_orderkey = l_orderkey
  and o_orderstatus = 'F'
  and s_nationkey = n_nationkey
  and n_name = $1;
```

List of Figures

List of Algorithms

Bibliography

[ABG+87] D. Agrawal, A. J. Bernstein, P. Gupta, S. Sengupta, and D. A. I. Currently, "Distributed optimistic concurrency control with reduced rollback," *Distributed Computing*, vol. 2, pp. 45–59, 1987.

[ADJ+10] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez, "The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 519–530. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807224

[AH00] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/342009.335420

[AKN12] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems," *Proceedings of VLDB Endowment*, vol. 5, no. 10, pp. 1064–1075, Jun. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2336664.2336678

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," *SIGMOD Rec.*, vol. 24, no. 2, pp. 1–10, May 1995. [Online]. Available: http://doi.acm.org/10.1145/568271.223785

[BG83] P. A. Bernstein and N. Goodman, "Multiversion concurrency control – theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, Dec. 1983. [Online]. Available: http://doi.acm.org/10.1145/319996.319998

[BHEF11] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete, "One-copy serializability with snapshot isolation under the hood," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 625–636. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2011.5767897

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BLP11] S. Blanas, Y. Li, and J. M. Patel, "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989328

[BM70] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '70. New York, NY, USA: ACM, 1970, pp. 107–141. [Online]. Available: http://doi.acm.org/10.1145/1734663.1734671

[BPS+09] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs, "Your Computer is Already a Distributed System. Why Isn'T Your OS?" in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, ser. HotOS'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855568.1855580

[BTAO13] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu, "Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, ser. ICDE '13. Washington, DC, USA: IEEE Computer Society, April 2013, pp. 362–373. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2013.6544839

[CAB+94] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, "Readings in database systems (2nd ed.)," in *Readings in Database Systems (2Nd Ed.)*, M. Stonebraker, Ed. San Francisco, CA, USA: Morgan Kaufmann

Publishers Inc., 1994, ch. A History and Evaluation of System R, pp. 54–68. [Online]. Available: http://dl.acm.org/citation.cfm?id=190956.190963

[CAGM07] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving Hash Join Performance Through Prefetching," *ACM Trans. Database Syst.*, vol. 32, no. 3, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272743.1272747

[Car83] M. J. Carey, "An abstract model of database concurrency control algorithms," *SIGMOD Rec.*, vol. 13, no. 4, pp. 97–107, May 1983. [Online]. Available: http://doi.acm.org/10.1145/971695.582211

[Car84] ——, "The performance of concurrency control algorithms for database management systems," in *Proceedings of the 10th VLDB Conference*, 1984, pp. 107–118.

[CASM05] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry, "Optimistic intra-transaction parallelism on chip multiprocessors," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05. VLDB Endowment, 2005, pp. 73–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083592.1083604

[Cat11] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Record*, vol. 39, no. 4, pp. 12–27, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1978915.1978919

[CF02] S. Chandrasekaran and M. J. Franklin, "Streaming Queries over Streaming Data," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 203–214. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287388

[CM86] M. J. Carey and W. A. Muhanna, "The performance of multiversion concurrency control algorithms," *ACM Trans. Comput. Syst.*, vol. 4, no. 4, pp. 338–378, Sep. 1986. [Online]. Available: http://doi.acm.org/10.1145/6513.6517

[CN97] S. Chaudhuri and V. R. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 146–155. [Online]. Available: http://dl.acm.org/citation.cfm?id=645923.673646

[CPV09] G. Candea, N. Polyzotis, and R. Vingralek, "A scalable, predictable join operator for highly concurrent data warehouses," *Proceedings of VLDB Endowment*, vol. 2, no. 1, pp. 277–288, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687627.1687659

[CPV11] ——, "Predictable Performance and High Query Concurrency for Data Analytics," *The VLDB Journal*, vol. 20, no. 2, pp. 227–248, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1007/s00778-011-0221-2

[CRF08] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 729–738. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376690

[DAF+03] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path Sharing and Predicate Evaluation for High-performance XML Filtering," *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 467–516, Dec. 2003. [Online]. Available: http://doi.acm.org/10.1145/958942.958947

[DFI+13] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1243–1254. [Online]. Available: http://doi.acm.org/10.1145/2463676.2463710

[DFJ+96] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 330–341. [Online]. Available: http://dl.acm.org/citation.cfm?id=645922.673462

[DSRS01] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan, "Pipelining in Multi-query Optimization," in *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '01. New York, NY, USA: ACM, 2001, pp. 59–70. [Online]. Available: http://doi.acm.org/10.1145/375551.375561

[DSTW02] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm," in

Proceedings of the 28th International Conference on Very Large Data Bases, ser. VLDB '02. VLDB Endowment, 2002, pp. 299–310. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287396

[DZC99] L. Devroye and C. Zamora-Cura, "On the Complexity of Branch-and Bound Search for Random Trees," *Random Struct. Algorithms*, vol. 14, no. 4, pp. 309–327, Jul. 1999. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1098-2418(199907)14:4⟨309::AID-RSA2⟩3.0.CO;2-1

[Fer94] P. M. Fernandez, "Red Brick Warehouse: A Read-mostly RDBMS for Open SMP Platforms," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '94. New York, NY, USA: ACM, 1994, pp. 492–. [Online]. Available: http://doi.acm.org/10.1145/191839.191947

[Fin82] S. Finkelstein, "Common Expression Analysis in Database Applications," in *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '82. New York, NY, USA: ACM, 1982, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/582353.582400

[FJL+01] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '01. New York, NY, USA: ACM, 2001, pp. 115–126. [Online]. Available: http://doi.acm.org/10.1145/375663.375677

[FK05] P. M. Fischer and D. Kossmann, "Batched Processing for Information Filters," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 902–913. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2005.25

[GAK12] G. Giannikis, G. Alonso, and D. Kossmann, "SharedDB: Killing One Thousand Queries with One Stone," *Proceedings of VLDB Endowment*, vol. 5, no. 6, pp. 526–537, Feb. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2168651.2168654

[GM93] G. Graefe and W. J. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in *Proceedings of the Ninth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 209–218. [Online]. Available: http://dl.acm.org/citation.cfm?id=645478.757691

[GMAK13] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann, "Workload Optimization Using SharedDB," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1045–1048. [Online]. Available: http://doi.acm.org/10.1145/2463676.2463678

[GMAK14] ——, "Shared Workload Optimization," *Proceedings of VLDB Endowment*, vol. 7, no. 6, pp. 526–537, Feb. 2014.

[GR92] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[GUM+10] G. Giannikis, P. Unterbrunner, J. Meyer, G. Alonso, D. Fauser, and D. Kossmann, "Crescando," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1227–1230. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807326

[HA05] S. Harizopoulos and A. Ailamaki, "StagedDB: Designing Database Servers for Modern Hardware," *IEEE Data Eng. Bull.*, vol. 28, no. 2, pp. 11–16, 2005. [Online]. Available: ftp://ftp.research.microsoft.com/pub/debull/A05june/stavros.ps

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible query processing in starburst," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '89. New York, NY, USA: ACM, 1989, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/67544.66962

[HM97] S. Helmer and G. Moerkotte, "Evaluation of main memory join algorithms for joins with set comparison join predicates," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, ser. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 386–395. [Online]. Available: http://dl.acm.org/citation.cfm?id=645923.673667

[HNZB07] S. Héman, N. Nes, M. Zukowski, and P. Boncz, "Vectorized data processing on the cell broadband engine," in *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, ser. DaMoN '07. New York, NY, USA: ACM, 2007, pp. 4:1–4:6. [Online]. Available: http://doi.acm.org/10.1145/1363189.1363195

[HSA05] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: A Simultaneously Pipelined Relational Query Engine," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 383–394. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066201

[IK84] T. Ibaraki and T. Kameda, "On the Optimal Nesting Order for Computing N-relational Joins," *ACM Trans. Database Syst.*, vol. 9, no. 3, pp. 482–502, Sep. 1984. [Online]. Available: http://doi.acm.org/10.1145/1270.1498

[IKNG09] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves, "An Architecture for Recycling Intermediates in a Column-store," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 309–320. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559879

[JASA09] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki, "A new look at the roles of spinning and blocking," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, ser. DaMoN '09. New York, NY, USA: ACM, 2009, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/1565694.1565700

[JPA09] R. Johnson, I. Pandis, and A. Ailamaki, "Improving oltp scalability using speculative lock inheritance," *Proceedings VLDB Endowment*, vol. 2, no. 1, pp. 479–489, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687627.1687682

[JPH+09] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-mt: A scalable storage manager for the multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '09. New York, NY, USA: ACM, 2009, pp. 24–35. [Online]. Available: http://doi.acm.org/10.1145/1516360.1516365

[KFDA00] D. Kossmann, M. J. Franklin, G. Drasch, and W. Ag, "Cache investment: Integrating query optimization and distributed data placement," *ACM Trans. Database Syst.*, vol. 25, no. 4, pp. 517–558, Dec. 2000. [Online]. Available: http://doi.acm.org/10.1145/377674.377677

[KKL+09] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. Hash Revisited: Fast

Join Implementation on Modern Multi-core CPUs," *Proceedings VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687553.1687564

[KL10] S. Kosuch and A. Lisser, "Upper bounds for the 0-1 stochastic knapsack problem and a B&B algorithm," *Annals of Operations Research*, vol. 176, no. 1, pp. 77–93, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10479-009-0577-5

[KR81] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/319566.319567

[KS00] D. Kossmann and K. Stocker, "Iterative dynamic programming: A new class of query optimization algorithms," *ACM Trans. Database Syst.*, vol. 25, no. 1, pp. 43–82, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/352958.352982

[Kum96] V. Kumar, *Performance of concurrency control mechanisms in centralized database systems*. Prentice Hall, 1996. [Online]. Available: http://books.google.ch/books?id=WNhQAAAAMAAJ

[LBD+11] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance concurrency control mechanisms for main-memory databases," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 298–309, Dec. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2095686.2095689

[LBMW07] C. A. Lang, B. Bhattacharjee, T. Malkemus, and K. Wong, "Increasing buffer-locality for multiple index based scans through intelligent placement and index scan speed control," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1298–1309. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325999

[LJ03] H. K. Y. Leung and H.-A. Jacobsen, "Efficient matching for state-persistent publish/subscribe systems," in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '03. IBM Press, 2003, pp. 182–196. [Online]. Available: http://dl.acm.org/citation.cfm?id=961322.961352

[LSL13] D. B. Lomet, S. Sengupta, and J. J. Levandoski, "The bw-tree: A b-tree for new hardware platforms," in *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ser. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 302–313. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2013.6544834

[LVLM10] A. Lecchini-Visintini, J. Lygeros, and J. M. Maciejowski, "Stochastic Optimization on Continuous Domains with Finite-Time Guarantees by Markov Chain Monte Carlo Methods," *IEEE Transactions on Automatic Control*, vol. 55, no. 12, pp. 2858–2863, Dec 2010. [Online]. Available: http://control.ee.ethz.ch/index.cgi?page=publications;action=details;id=3702

[Mak12] D. Makreshanski, "Shared, Parallel Database Join on Modern Hardware," Master's thesis, Systems Group, Department of Computer Science, ETH Zurich, 2012. [Online]. Available: http://systems.ethz.pubzone.org/pages/publications/showPublication.do?pos=1&publicationId=2035180

[MBK02] S. Manegold, P. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 191–202. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287387

[MBKB07] S. K. Madria, M. Baseer, V. Kumar, and S. Bhowmick, "A transaction model and multiversion concurrency control for mobile database systems," *Distrib. Parallel Databases*, vol. 22, no. 2-3, pp. 165–196, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10619-007-7019-7

[MLO86] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM Transactions on Database Systems*, vol. 11, pp. 378–396, 1986. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.8871

[MPK00] S. Manegold, A. Pellenkoft, and M. Kersten, "A Multi-Query Optimizer for Monet," in *Advances in Databases*, ser. Lecture Notes in Computer Science, B. Lings and K. Jeffery, Eds. Springer Berlin Heidelberg, 2000, vol. 1832, pp. 36–50. [Online]. Available: http://dx.doi.org/10.1007/3-540-45033-5_4

[MRS+04] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *Proceedings*

of the 2004 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 659–670. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007642

[Neu11] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proceedings VLDB Endowment*, vol. 4, no. 9, pp. 539–550, Jun. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002938.2002940

[OL90] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization," in *Proceedings of the 16th International Conference on Very Large Data Bases*, ser. VLDB '90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 314–325. [Online]. Available: http://dl.acm.org/citation.cfm?id=645916.671976

[PA11] D. Pasetto and A. Akhriev, "A comparative study of parallel sort algorithms," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. SPLASH '11. New York, NY, USA: ACM, 2011, pp. 203–204. [Online]. Available: http://doi.acm.org/10.1145/2048147.2048207

[PAA13] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki, "Sharing data and work across concurrent analytical queries," *Proceedings VLDB Endowment.*, vol. 6, no. 9, pp. 637–648, Jul. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2536360.2536364

[Pal74] F. P. Palermo, "A Data Base Search Problem," in *Proc. of the 4th Symposium on Computer and Information Science 1974*, J. T. Tou, Ed. Springer US, 1974, pp. 67–101. [Online]. Available: http://dx.doi.org/10.1007/978-1-4684-2694-6_4

[Pap79] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/322154.322158

[PFL+00] J. a. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha, "Publish/Subscribe on the Web at Extreme Speed," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 627–630. [Online]. Available: http://dl.acm.org/citation.cfm?id=645926.671690

[PJHA10] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 928–939, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920959

[QRR+08] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman, "Main-memory Scan Sharing for Multi-core CPUs," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 610–621, Aug. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1453856.1453924

[RSQ+08] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-time query processing," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ser. ICDE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 60–69. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2008.4497414

[RSSB00] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00. New York, NY, USA: ACM, 2000, pp. 249–260. [Online]. Available: http://doi.acm.org/10.1145/342009.335419

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79. New York, NY, USA: ACM, 1979, pp. 23–34. [Online]. Available: http://doi.acm.org/10.1145/582095.582099

[SC05] M. Stonebraker and U. Cetintemel, ""one size fits all": An idea whose time has come and gone," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 2–11. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2005.1

[Sel88] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, Mar. 1988. [Online]. Available: http://doi.acm.org/10.1145/42201.42203

[SFL+12] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in sap hana database: The end of a column store myth," in *Proceedings of the 2012 ACM SIGMOD*

International Conference on Management of Data, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 731–742. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213946

[Sha86] L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Trans. Database Syst.*, vol. 11, no. 3, pp. 239–264, Aug. 1986. [Online]. Available: http://doi.acm.org/10.1145/6314.6315

[SSGA11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso, "Database Engines on Multicores, Why Parallelize when You Can Distribute?" in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966448

[TDG09] N. Thakoor, V. Devarajan, and J. Gao, "Computation Complexity of Branch-and-Bound Model Selection," in *Computer Vision, 2009 IEEE 12th International Conference on*, Sept 2009, pp. 1895–1900. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5459420

[TGO01] K.-L. Tan, S.-T. Goh, and B. C. Ooi, "Cache-on-Demand: Recycling with Certainty," in *Data Engineering, 2001. Proceedings. 17th International Conference on*, 2001, pp. 633–640. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=914878

[UGA+09] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable performance for unpredictable workloads," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 706–717, Aug. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687627.1687707

[Unt12] P. Unterbrunner, "Elastic, Reliable, and Robust Storage and Query Processing with Crescando/RB," Ph.D. dissertation, ETH Zürich, 2012. [Online]. Available: http://e-collection.library.ethz.ch/view/eth:5677

[WCY04] K.-L. Wu, S.-K. Chen, and P. S. Yu, "Interval query indexing for efficient stream processing," in *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, ser. CIKM '04. New York, NY, USA: ACM, 2004, pp. 88–97. [Online]. Available: http://doi.acm.org/10.1145/1031171.1031188

[ZB12] M. Zukowski and P. Boncz, "From x100 to vectorwise: Opportunities, challenges and things most researchers do not think about," in *Proceedings of*

the 2012 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 861–862. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213967

[ZHNB07] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Cooperative scans: Dynamic bandwidth sharing in a dbms," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 723–734. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325934

[ZLFL07] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 533–544. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247540