Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 53

Systems Group, Department of Computer Science, ETH Zurich

Shared, Parallel Database Join on Modern Hardware

by

Darko Makreshanski

Supervised by

Prof. Dr. Gustavo Alonso
Georgios Giannikis

February 2012–August 2012

inf | Informatik
Computer Science

Abstract

The requirements faced by database applications have changed significantly in the last decade. Database users expect a predictable performance, independent of the concurrent load on the system. Additionally, computer hardware has evolved dramatically from single-core CPUs, to multi-core CPUs. However, this evolution is not exploited by database systems which are still based on traditional hardware architectures.

As a solution to this problem, SharedDB was introduced; a database system designed to exploit the vast amounts of memory available to modern computers, as well as the multiple CPU cores. This master thesis investigates new ideas of how to take advantage of modern computer hardware for the purposes of SharedDB.

More specifically, we analyzed and implemented efficient parallel database join operator for SharedDB. Our analysis for the join operation can also be propagated across the whole SharedDB, to allow for better performance. We finally tested our approach by running a join heavy workload based on the TPC-H benchmark and observed that SharedDB performs several times faster than PostgreSQL, a standard off-the-shelf database.

Acknowledgements

I would like to thank my supervisors Georgios Giannikis and Prof. Dr. Gustavo Alonso, as well as Prof. Dr. Donald Kossmann for their invaluable support and guidance during the course of this thesis.

I would also like to thank Cagri Balkesen for the fruitful discussions about join algorithms, and for supplying some of the join implementations.

And finally, special thanks to Tudor Salomie, the rest of the Systems Group, my friends and family for their support during this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The relational join is a fundamental database operation. It combines tuples from two relations to create a new relation. Depending on the way tuples are combined, there are multiple types of joins. However, the most important join type is the equi-join. Equi-join between two relations combines tuples based on equality predicates on a set of attributes.

This thesis investigates ways to efficiently execute equi-joins in SharedDB, a state-of-the-art research relational database system. SharedDB executes queries by sharing computation and memory costs across multiple queries. It also aims to provide predictable performance under varying workload by distributing query execution across independent relational operators. This approach deviates from traditional relational databases, which typically process queries one-at-a-time, trying to optimize individual query execution as much as possible.

We analyzed and optimized two state-of-the-art join implementations, examined their properties and analyzed the effect of the SharedDB's environment on their performance. We also investigated how to efficiently represent tuple-to-query membership in shared query execution, as well as how to coordinate multiple independent database operators in the context of SharedDB.

We concluded that SharedDB favors a simple hash join algorithm, which relies on many random main memory accesses, to a cache-optimized radix partitioning join, which partitions data to avoid random accesses.

We concluded this partly because we found the benefits of partitioning difficult to be achieved in the SharedDB's query execution environment. Another reason was that we managed to optimize the simple hash join up to a level where it performs comparably well even in an environment suitable for the partitioning algorithm.

1.1 Motivation

Join is notoriously expensive, which is why it has been extensively studied in literature [2, 3, 5, 10, 11, 9, 13]. Recently, hardware trends have shifted towards multi-core systems and large inexpensive main memories. This has dictated the research for efficient parallel in-memory joins on modern processors. Furthermore, the increasing speed gap between processor and main memory, as well as increased demands for predictable performance under varying workload,

has led to research into database systems that agressively share data and computation when executing large groups of versatile queries [1, 4, 7, 8, 15].

SharedDB [7] is one of these systems, which offers predictable performance under varying workload by distributing query execution across many independent specialized operators. SharedDB is in its early stage of development and many research questions are still open. This work aims to investigate how to efficiently execute equi-join operations in this environment.

1.2 Problem Statement

The problem that we are trying to solve with this thesis is how to efficiently execute a join operation or a set of join operations by sharing computation and data across multiple different queries. This problem can be divided into several subproblems:

- How to join two relations in shared query execution environment?
- How to efficiently represent and process tuple-to-query membership?
- How to orchestrate multiple joins to work in coordination with themselves and with the rest of the database operators?

1.3 Thesis Contribution

The contributions of this thesis can be summarized as follows:

- We analyzed two existing join algorithms - hash join and radix partitioning join. We also optimized the hash join using several well-known methods and a novel thread synchronization mechanism. Finally, based on this optimized version we implemented a parallel shared in-memory join for SharedDB.
- We analyzed existing techniques and proposed a new method for efficient representation of tuple-to-query membership for shared execution of queries.
- We ran the join workload of the TPC-H benchmark on SharedDB and PostgreSQL and analyzed their performance. Furthermore, we proposed improvements to the execution engine of SharedDB for this workload, and implemented a technique which showed certain benefits of these improvements.

1.4 Thesis Outline

The rest of the report is organized in the following way:

1. Chapter 2 gives a background on the latest join algorithms proposed in literature, and an overview of the underlying systems that will support the join algorithm, namely Crescando [15] and SharedDB.

2. Chapter 3 gives insight on the current state-of-the-art related to this work.

3. Chapter 4 analyzes two existing join algorithms, and proposes an efficient join algorithm suitable for shared join execution in SharedDB.

4. Chapter 5 gives analysis on how to efficiently represent query-to-tuple membership for shared query execution.

5. Chapter 6 examines how to efficiently execute a group of joins, by running the join part of the TPC-H workload on SharedDB and PostgreSQL.

6. Chapter 7 finally concludes this report with final remarks and future work ideas.

Chapter 2

Fundamentals

This chapter describes the algorithms and systems that are directly related to this thesis. It begins with explanation of the join algorithms taken into consideration, and continues with the intrinsics of SharedDB [7] and Crescando [15].

2.1 Relational Join

A join in relational algebra is an operation which combines tuples from two relations. Depending on the way tuples are combined, there are multiple types of joins. The most popular join type is the equi-join which combines tuples based on equality predicate on one or more attributes of the tuples. Equi-join is typically used in joining two normalized relations on their foreign-key attribute.

Due to the nature of the join operation, join algorithms are heavily influenced by the way data is stored. Algorithms vary depending on whether the database is in-memory, disk based, SSD based, and so on. Since our work is related to an in-memory database, we focus only on in-memory joins.

2.1.1 Hash Join

Hash Join matches tuples by populating a hash table using tuples from one of the relations (which we call either build relation, or left-hand side relation), and probing the hash table using tuples from the other relation (which we call probe relation, or right-hand side relation).

A hash table is a data structure which consists of hash buckets, each of which can hold multiple tuples from the build relation. Each tuple can be matched with a hash bucket by calculating a hash of the tuple's join attribute using a deterministic hash function. A modulo of the hash with the number of hash buckets, will give the index of the matching hash bucket.

A build tuple is processed by inserting it into its corresponding hash bucket and a probe tuple is processed by joining it with the build tuples from its corresponding hash bucket. Since hashing and modulo operation cause loss of information, additional equality comparison of the corresponding join attributes is required to ensure correct join.

2.1.2 Partitioning Join

Given that the relations are not ordered by the join attribute, a hash join will cause many random accesses. Thus, as soon as the hash table gets larger than a storage medium with fast random access, the hash join suddenly becomes much more expensive.

A partitioning join will partition both relations on their join attribute, such that partitions of the build relation fit in a storage medium with fast random access.

The fastest partitioning join to date is proposed by Kim et al. [9]. This is a parallelized version of the algorithm proposed by Manegold et al. [10]. This algorithm partitions the relations so that partitions of the build relation fit in the lowest level $L1$ cache. Since partitioning itself can require many random accesses and cause problems with TLB cache, multiple levels of partitioning are needed.

2.1.3 Sort-merge Join

Sort-merge join involves sorting both relations and then running a sequential merge join operation. Sort-merge has worse complexity than the hash join ($O(n \cdot log(n))$ vs $O(n)$), however Kim et al. [9] argued that with recent trends of increasing SIMD instructions' width, the inherent parallelism of sort-merge can make this algorithm faster than the partitioning algorithm.

2.2 SharedDB

Traditional relational database systems execute queries one-at-a-time, trying to minimize the cost of executing each query. SharedDB [7] deviates from this approach by sharing computation and memory costs for multiple queries. Sharing costs for multiple queries has been around for a while in the form of multi-query optimization [12]. While multi-query optimization tries to find common subexpressions in queries, SharedDB's execution engine shares queries even if they do not have common subexpressions. To make this possible, SharedDB extends the relational data model by adding additional attribute to each tuple that keeps the set of queries that the tuple belongs to.

This has significant impact on many relational operators, including the join operator. A join operator in SharedDB has to additionally join the sets of query IDs for all matching pairs of tuples. An example is shown in Figure 2.1, where relations R and S are intermediate result relations for certain queries: A, B, and C, and $R \bowtie S$ is the resulting relation of the join with the join predicate of $R.DID = S.DID$.

SharedDB further deviates from traditional databases by distributing the execution of queries to independent operators, each of which is specialized to execute a certain relational operation. Thus, in order for a query to be executed it has to be split into several subqueries which will be processed by the corresponding operators. An example depicting this approach is shown in Figure 2.2. This example shows a sample execution plan in SharedDB for a schema of two relations *Students* and *Departments*, and 4 queries. The execution plan contains two

Relation R:

CID	CName	DID	Query IDs
1	John	4	A,B,C
2	Nick	4	B,C
3	Peter	1	A
4	Lary	3	A
5	Smith	1	B,C

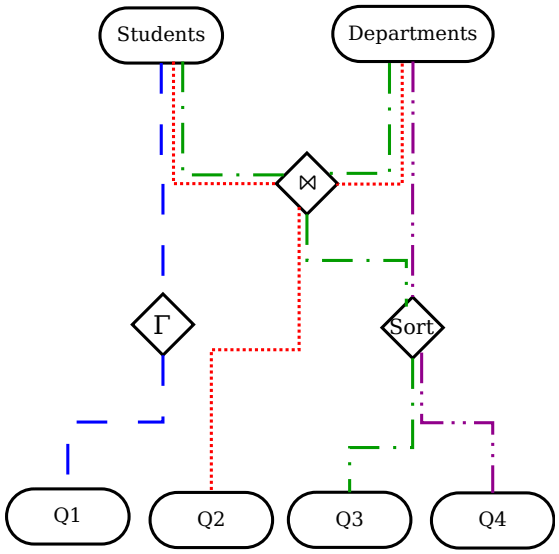
Relation S:

DID	DName	Query IDs
1	D-ITET	C
2	D-MAVT	B
4	D-INFK	A,B

R ⋈ S

CID	CName	DID	DName	Query IDs
1	John	4	D-INFK	A,B
2	Nick	4	D-INFK	B
5	Smith	1	D-ITET	C

Figure 2.1: Sample Relational Join in SharedDB



Query	SQL
Q1	**SELECT** **Count**(∗) **FROM** Students S **GROUP BY** S.DID;
Q2	**SELECT** ∗ **FROM** Students S, Departments D **WHERE** S.DID = D.DID **AND** S.Name = 'John';
Q3	**SELECT** ∗ **FROM** Students S, Departments D **WHERE** S.DID = D.DID **ORDER BY** S.Name;
Q4	**SELECT** ∗ **FROM** Departments D **ORDER BY** D.Name;

Figure 2.2: Sample Execution Plan in SharedDB

storage engine operators for the two relations, one `Join` operator, one `Group By` operator and one `Sort` operator.

In SharedDB operators communicate with each other by exchanging queries and tuples. Two operators that exchange queries and tuples form a so-called parent/child relationship (or equivalently a consumer/producer relationship). An operator A is a parent (or consumer) operator of another operator B if A sends queries to and receives intermediate results (tuples) from B. Consequently B is a child (or producer) operator of A. All operators have two-way relationships with other operators, except for storage engine operators and output operators.

Unlike other operator-centric databases like QPipe, CJoin, and DataPath, which try to execute queries immediately as they arrive, SharedDB's operators execute queries only in batches. This implies that there is a query queue for each operator, where queries queue up and wait to be executed in the next batch, whenever the operator is currently busy.

2.3 Crescando

Crescando [15] is a relational storage engine used to provide predictable performance for business intelligence queries under high update-intensive workload. Unlike traditional databases which index relations to minimize lookup times for individual queries, Crescando indexes queries instead, and always scans over the whole relation. Scans are carried out using the ClockScan algorithm described in [15], which creates batches of queries and updates, and executes them in-order while scanning over the relation. Crescando makes use of multi-core CPUs by doing horizontal partitioning. This is achieved by splitting the data in equal chunks and assigning a dedicated core to each chunk.

Chapter 3

Related Work

3.1 Join Related Work

There are significant number of papers that deal with join algorithms optimized for modern hardware. This section explains related work that focuses on joins for single queries. Related work that focuses on sharing data accross queries is explained in section 3.3

Chen et al. [5] proposed two prefetching methods for hash joins to minimize main memory random access latency. The first method was based on group prefetching and the second was based on pipelined prefetching. They benchmarked both methods in a simulator and showed that their performance was comparable.

Kim et al. [9] presented a very efficient parallel version of the radix partitioning join proposed by Manegold et al. [10]. They argued that radix partitioning join is better than a hash join, due to less main memory bandwidth consumption, however they did not provide comparison of the bandwidth consumption of both approaches. They also argued that with recent hardware trends, which go towards increasing width of SIMD instructions, the inherent paralelism of merge-sort [6] would make the sort-merge join more favorable than the radix partitioning join.

Blanas et al. [3] argued that a simple no partitioning hash join can outperform the cache optimized radix partitioning join, by relying on the hardware to hide cache latencies. However, for their benchmarks they used a small ordered build relation. Furthermore, they argue that random access latencies can be hidden using simultaneous multi-threading (SMT), and do not mention benefits of software prefetching and larger page sizes on random access.

Finally, a work in submition process by Balkesen et al. [2] optimized both the no-partitioning hash join by [3] and the radix partitioning join by [9] and claim that hardware consciousness still matters.

3.2 QID Handling Related Work

So far, in addition to SharedDB we are aware of only two systems that share tuples across different queries: CJoin [4] and DataPath [1]. Both of these systems represent tuple membership in queries by bitsets. CJoin's approach is to have a global mapping from queries to bit locations

in the bitset, 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. SharedDB currently handles query IDs in arrays, which is superior to bitsets in cases of short-running point queriesOLTP, but inferior in cases of long-running range queries.

3.3 Query Execution Engine Related Work

CJoin

The CJoin [4] project proposed a shared join algorithm specialized for star schema databases. A shared sequential scan is employed on the fact relation which selects the tuples and creates a bitset for every tuple. The join operation then consists of probing all the dimension relations' hash tables, and doing a bitwise AND operation on the fact tuples' bitsets and the corresponding dimension tuples' bitsets.

An important characteristic of CJoin is that it starts processing queries immediately as they arrive. More specifically, when a query arrives, all hash tables must be updated for the query, before the new fact tuples can be processed. Furthermore, when all fact tuples for a query have been processed, all its changes in the hash tables have to be undone. This becomes problematic in schemas different than star schemas, for instance when there are two fact tables. Thus, we argue that the system can benefit more in the case when queries are processed in batches rather than immediately as they arrive. Some of our arguments are:

1. When queries are executed in a batch, the system can create a better decision on how to organize the queries' execution. For example, query to bit location can be optimized to minimize the bitset's size.

2. The drawback of batching with respect to the CJoin's technique is that enqueued queries will have to wait for the current batch to finish with execution. In a perfect system where performance is constant with respect to the number of queries in the system, a query's response will be at worst twice as much. However, in a real world system, increasing the number of queries increases the cost of doing the join operation, thus reducing the batching effect on response time.

3. Analytic workloads typically contain group aggregators and sorts, which are fully blocking operators and if shared execution in them is beneficial, then they have to employ batched execution. As a result, the overhead of handling queries as they arrive in the join operators is not necessary.

DataPath

DataPath [1] proposed a push-based system, 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.

DataPath uses a single hash table for all join operations. One reason is to allow asynchronous query admission, similarly to CJoin. The paper, however, does not mention how tuples in the hash table are flushed when their queries have finished.

Furthermore, we believe that the approach of trying to push data as much as possible from the storage engine hinders opportunities for sharing. For joins, it is beneficial to match the batch sizes of the build input relation and the probe input relation.

QPipe

QPipe [8] proposes a system where intermediate results and disk pages of concurrently running queries can be reused. It follows a similar operator-centric approach, where one relational operator (sort, join, aggregate) can be responsible for many queries, however unlike the other approaches, computation work in QPipe is shared only for queries that share subplans. Furthermore, QPipe does not compute optimal plan for a group of queries, but rather tries to adapt the query plan for new queries to the plan of the queries that are already being executed. This can lead to suboptimal plans, as the optimal plan to execute a group of queries can be different than the individual optimal query plans.

Chapter 4

The Join Operation

This section explains the specifics of a single join operation on two relations, excluding how query IDs are handled. First, we describe the set of join algorithms we considered. Then, we analyze the theoretical aspects of the algorithms, taking into consideration the properties and constraints put forward by SharedDB. Afterwards, we put the conclusions of the theoretical part to the test, and see if they hold in practice, by taking well optimized implementations of two algorithms, and finally we implement the chosen algorithm in SharedDB and see how it performs.

4.1 Join Algorithm Alternatives

Since SharedDB in its current form is targeted only at in-memory databases, we are considering only in-memory joins. We considered a no-partitioning hash join by Blanas et al. [3], Balkesen et al. [2], and a radix-partitioning join by Kim et al. [9]. Kim et al. argued that in the near future a sort-merge join based on bitonic sort implementation using SIMD [6] will perform better than the radix-partitioning join, however we believe that sort-merge join and radix-partitioning join share very similar characteristics when investigated as a join option for SharedDB in terms of blocking property, and memory access and thus we decided to only consider one of the two.

4.1.1 Hash Join (HJ)

The simple hash join is well studied in literature, including optimizations for disk databases, and in-memory databases. Due to the simple nature of an in-memory hash join, relying mostly on random memory accesses, optimizations are mainly focused on reducing memory access latency, and optimizing for lower DRAM bandwidth consumption. Optimizing for both often involves partitioning of the data so that it fits in cache, but we will explain this types of algorithms in section 4.1.2

Other optimizations include using software prefetching to utilize modern processors' ability to handle multiple outstanding cache misses. The inherent parallelism of a hash join, allows for multiple addresses of hash buckets to be calculated and prefetched before they are accessed, thus

hiding the DRAM access latency. Chen et al. [5] first proposed software prefetching techniques for hash joins. They proposed two techniques, group prefetching and pipelined prefetching and benchmarked both of them, showing that their performance is comparable. However, their benchmarks were only done using simulations which considered constant time lookups for the TLB cache. We will later show that for large datasets TLB misses are the primary factors for degraded performance of random main memory access.

Optimizing for memory bandwidth, typically involves minimizing number of cachelines accessed during the build and probe phases. This is usually done by trying to fit all information required for one build and one probe operation in as few cachelines as possible. Balkesen et al. [2], optimizing the no-algorithm of Blanas et al. [3], proposed a hash join algorithm where a build and probe operation accesses only a single cacheline of $64B$, given that hash collisions are rare. Minimizing usage of memory, makes it harder to saturate the memory bandwidth, which is why we decided to combine this technique with prefetching to obtain optimal performance for this type of join.

4.1.2 Radix Partitioning Join (RPJ)

Partitioning the dataset so that it fits in a CPU cache was proposed by Shatdal et al. [13]. Manegold et al. [10] observed that the partitioning stage can cause many TLB and cache misses and proposed multi-pass radix partitioning, and Kim et al. [9] proposed a parallel version of the algorithm and compared it to sort-merge join. Kim et al. argued that the regular hash join is inferior to cache optimized radix partitioning join, because it utilizes more memory bandwidth, however they did not provide analysis on the memory consumption by both algorithms.

4.2 Memory Bandwidth Consumption

Both HJ and RPJ are without a doubt read/write memory intensive operations. Both are consisted of simple either random or several sequential accesses to data that does not fit into cache, and both are expected to saturate the memory bandwidth when run on several cores.

This is why we first decided to investigate the impact of these two algorithms on the memory bandwidth. In Table 4.1 we calculated the memory bandwidth used per build and probe tuple for both the algorithms mentioned above for the synthetic join setup as described in [9]. In their setup, they assume tuple size of 8 bytes, where 4 bytes are for the join attribute and 4 bytes are for the index of the tuple in the relation. Furthermore, they do not do any materialization of the output. For our calculations, we assume that cachelines are $64B$ long, and the cache has $write\text{-}allocate/fetch\text{-}on\text{-}write$ writing policy. For the HJ, we assume the optimal case where one random cacheline needs to be accessed per build and per probe tuple. We provide data for both 1-pass and 2-pass RPJ, since we believe that with the modern CPU support for very large pages, for instance $1GB$ pages for the AMD Bulldozer micro-architecture, 1-pass partitioning will be faster than 2-pass. Since in certain cases 4 bytes are not enough to represent a join attribute, we also provide data for 8-byte attributes, or $16B$ tuples.

Table 4.1: Memory Bandwidth Consumption for HJ and RPJ for Column Store Join with Tuples of $4B$

	$8B$ Tuples		$16B$ Tuples	
	Bytes per Build Tuple	Bytes per Probe Tuple	Bytes per Build Tuple	Bytes per Probe Tuple
1-Pass RPJ	32	32	64	64
2-Pass RPJ	64	64	128	128
HJ	132	68	136	72

Based on the table, we can deduce that RPJ is not superior to HJ with respect to DRAM bandwidth consumption. Consumption per probe is comparable, and although consumption per build tuple is larger, we expect that this does not have much weight, since probe relations are expected to always be larger (sometimes by a significant margin) than build relations.

4.2.1 Impact of the SharedDB Environment

Next, we investigated how a database system like SharedDB will affect the consumption of memory bandwidth of these two algorithms. A SharedDB-like database shares tuples for different queries and executes them in a pipeline of operators working in parallel.

We can not provide exact quantitative analysis of DRAM bandwidth usage, because the current implementation of SharedDB does not represent an implementation that consumes the optimal or close to optimal DRAM bandwidth consumption. Furthermore, the optimal case is not easily quantifiable due to too many parameters and tuning knobs that can have impact. Nevertheless we will present the properties of a SharedDB-like database that affect the bandwidth consumption of the join algorithms, and we will afterwards make a general conclusion on what would be a better fit for a join algorithm for SharedDB.

The properties of a SharedDB-like database that affect a join operation are:

1. Executing a group of queries together requires projecting a set of attributes, which is a union of the sets of attributes required by each query individually.

2. In a similar manner, executing a group of queries together requires executing a set of joins, which is a union of all joins required by individual operations.

3. Tuples might be pushed to multiple consumer operators, so we can no longer expect that data will be always consecutive.

4. Since the accompanying QIDs for every tuple and the record itself might be of variable size there is a need for a meta-data structure per tuple, which keeps pointers and sizes of these objects.

5. Blocking operators obstruct parallelism in the pipelined execution of SharedDB. Furthermore, in some cases we would not like to keep all data from the larger relation in memory until the join is finished.

6. In-between probe and materialization step, there is the step of joining query ID sets. This step is equal to the key comparison check and is unavoidable even in the case of post-materialization.

The impacts that these properties make on the memory bandwidth for a join operation are the following:

- From properties 1. and 3.: the join attributes might be further apart, thus increasing the bytes accessed when scanning only over the join attributes.
- From properties 2., 4. and 6.: while executing a join, we will need to access more data from the build relation, which makes partitioning to fit this data in cache more expensive. Furthermore, we need to access more data for the probe relation, hence accessing it several times is also more expensive.
- From point 5.: since radix partitioning join blocks on both relations, in order to "de-block" operations on the probing relation, we have to separate it into several blocks, which will cause additional memory bandwidth in order to read the build relation once for each block.

Judging from the above, we can deduce that the execution paradigm of SharedDB hurts the benefits of RPJ over HJ. We can find the reason for this by looking at the properties of the algorithms and SharedDB. Benefits of RPJ over HJ come mostly in two aspects:

- To minimize the usage of DRAM bandwidth, data is made as compact as possible and only data required for the join, i.e. tuple's join attribute are accessed. This is difficult to achieve in SharedDB, since data becomes more generic as it is shared across queries. Thus a penalty is payed when accessing a single tuple in terms of bytes accessed, which in turn pays off when more queries are sharing the tuple. Furthermore, the drawback of HJ over RPJ is that a whole cacheline of $64B$ has to be accessed from memory just to access one or a couple of tuples that are $8B$ each. In the case of SharedDB, we need to access more data, including larger tuples and QIDs, which could fill up the rest of the cacheline.
- The second aspect is that during the probing operation, the data from the build relation is read once and kept in cache for the whole process. The penalty paid in this case is reiterating and partitioning the larger relation. This optimization can be hard to achieve not only in SharedDB, but also in a any database system. One reason is that the partitioned data will be larger in a real world setting and especially in SharedDB. This might be due to tuples being larger than 8 bytes, or due to having multiple relations being joined at the same time with a single large relation. Another reason is that it might be more expensive to scan the larger relation several times, especially in SharedDB when this scan will cause additional data access due to meta-data that needs to be accessed.

Finally, we can conclude that when we compare HJ's and RPJ's DRAM bandwidth consumption, HJ is a better choice for join algorithm for SharedDB. Although having the build relation in cache is also beneficial for SharedDB, the cost of partitioning it, and the cost of multiple scans over the probe relation would outweigh those benefits.

4.3 Join Benchmarks

Based on the conclusions from the previous part, we put the algorithms to the test, to make sure that DRAM bandwidth consumption is the key issue. We will evaluate them against the following three platforms:

1. 4 AMD Opteron 6174 (Magny-Cours) processors, each with 12 cores. The processors are running at $2200MHz$ with $12 \cdot 64KB$ $L1$ data and instruction caches, $12 \cdot 512KB$ dedicated $L2$ cache, and $2 \cdot 6MB$ shared $L3$ cache. There are two NUMA nodes per processor, hence 6 cores per NUMA node, and $4 \cdot 4GB$ DDR3 $1333MHz$ DRAM modules per NUMA node.

   Unless otherwise noted, experiments were carried out on this platform.

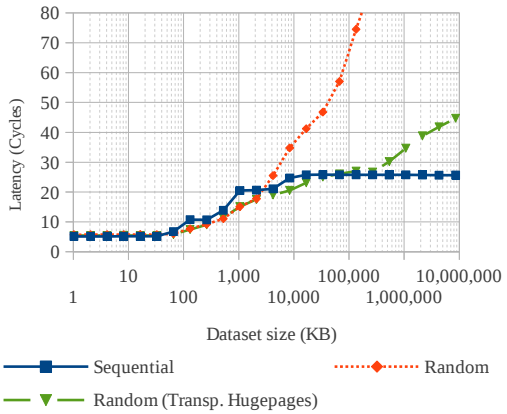
2. 4 AMD Opteron 6276 (Bulldozer) processors, each with 16 cores. The processors are running at $2300MHz$ with $8 \cdot 64KB$ shared instruction caches, $16 \cdot 16KB$ dedicated $L1$ data cache, $8 \cdot 2MB$ shared exclusive $L2$ caches, and $16MB$ shared $L3$ cache. There are two NUMA nodes per processor, and $32GB$ DDR3 memory at $1333MHz$ per NUMA node.

3. 4 Intel Xeon $E5-2600$ (Sandy Bridge) processors, each with 8 hyperthreaded cores. The processors are running at $2400MHz$ with $8 \cdot 32KB$ $L1$ data and instruction caches, $8 \cdot 256KB$ $L2$ caches, and $20MB$ shared $L3$ cache. There are 4 NUMA nodes in total in the system, with $8 \cdot 16GB$ DDR3 $1600MHz$ modules per NUMA node.

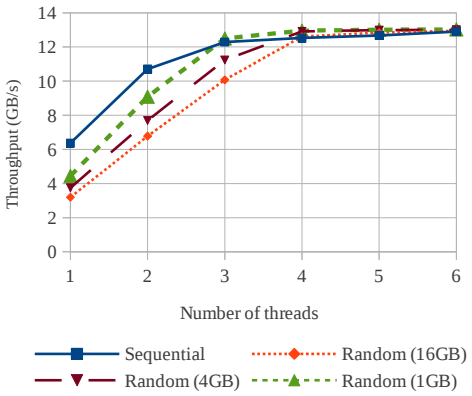
4.3.1 DRAM Random Access Latency

The major difference of the two algorithms is that one of them relies on random accesses while the other on multiple sequential accesses over data. We will first test random and sequential read performance of the platforms.

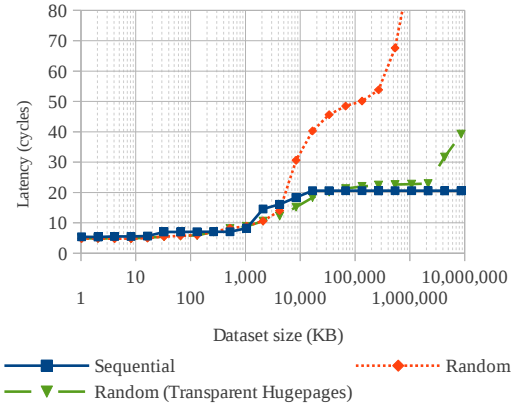
First, we performed tests to evaluate how does sequential access of cachelines compare to random access with prefetching. The experiments included having a single thread measuring the time it takes to access a single integer from random or sequential cachelines. The addresses of the cachelines to be accessed are precomputed and stored in a separate array which is read by the thread. The memory accessed by the thread is always located on the local NUMA node. The results can be seen in Figure 4.1. We were hoping that we can hide the random memory access latency by executing prefetch instructions early enough. However, we realised that this did not happen, and that prefetching did not help at all. The reason was that the loops were simple enough that with the compiler's loop unrolling and hardware's out-of-order execution, the cache misses were already hidden.



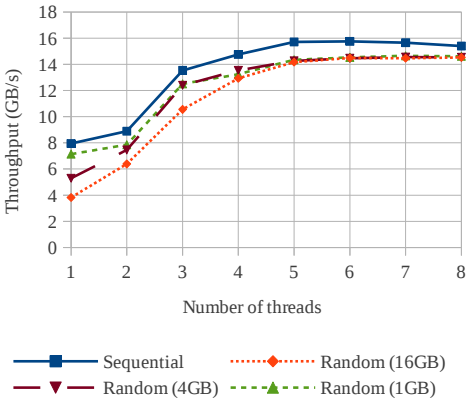
(a) Latency - Magny-Cours



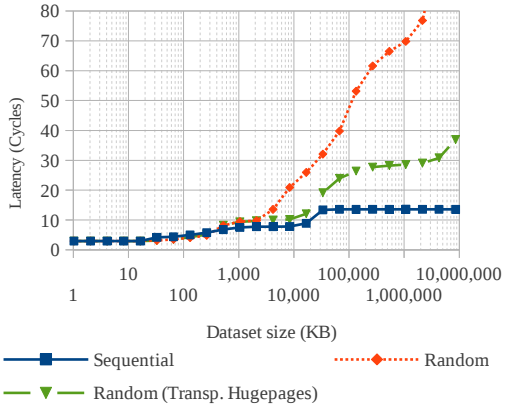
(b) Bandwidth - Magny-Cours



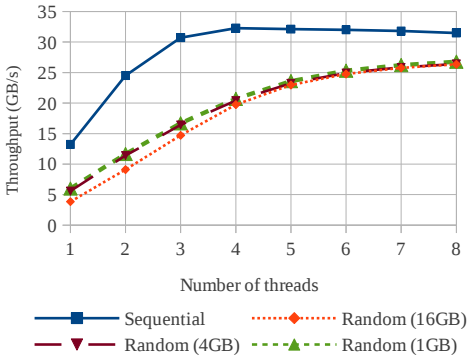
(c) Latency - Bulldozer



(d) Bandwidth - Bulldozer



(e) Latency - Sandy Bridge



(f) Bandwidth - Sandy Bridge

Figure 4.1: Random vs. Sequential Read Access Latency and Bandwidth on the Different Platforms

Additionally, there was another problem why performance dropped significantly with increased dataset and it was related to TLB. Namely, accessing too many TLB entries is prohibitive to the processor to be able to handle multiple cache misses efficiently. Luckily, modern processors support larger page sizes. E.g. the AMD Bulldozer micro-architecture supports $4KB, 2MB$ and $1GB$ pages. The Linux kernel, currently supports transparently only $2MB$ pages (option: transparent hugepages), which is not enough for our multi-gigabyte sized dataset to fit in the fastest TLB caches. However, it still proved much more effective than using the default $4KB$ sized pages. In the Figure, we can see that the corresponnding micro-architectures from Intel and AMD have quite different characteristics. Intel's machine is twice as fast for sequential reads than for random reads, while AMD's machines perform almost equally for both.

4.3.2 DRAM Random Access Bandwidth

The next experiment was to determine how random and sequential read access patterns influence the performance of memory bandwidth on these platforms. The experimental setup was similar to the latency experiment, and included running multiple threads trying to access a single integer from random or sequential cachelines whose addresses were precomputed and stored in a separate array. All threads were forced to run on a single NUMA node, hence the maximum number of threads for a machine is the number of cores per NUMA node. All threads accessed separate memory regions on the local NUMA node. Although, threads accessed only one integer from a cacheline, the bandwidth was calculated assuming a whole cacheline is transferred for each integer access.

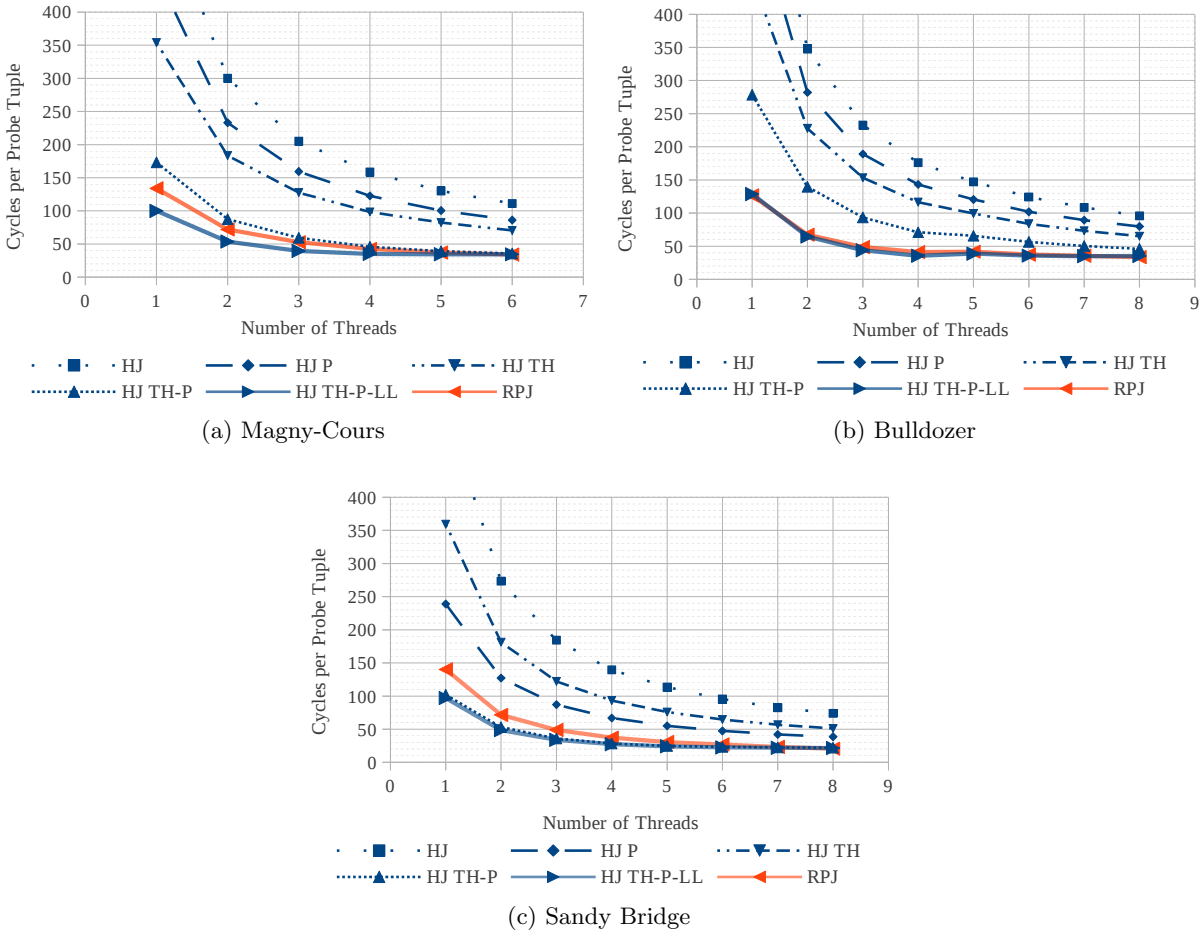
The results from Figure 4.1 show that for the AMD machines random and sequential read bandwidth were comparable, similarly to the latency measures. For the Intel machine, then maximum sequential read bandwidth was again higher than maximum random read bandwidth, however the difference was no longer a factor of two, but smaller.

4.3.3 HJ and RPJ performance

Regarding HJ we used a version of the no-partitioning algorithm of Balkesen et al. [2] which was an optimized version of the algorithm of Blanas et al. [3]. The optimizations to the original no-partitioning algorithm included minimizing the random memory access of the algorithm to a single cacheline. This was a perfect baseline for our experiments, since its memory bandwidth consumption coincides with our expectations for the HJ algorithm. Regarding the RPJ algorithm, we used the algorithm of Balkesen et al. [2] which is an optimized version of the algorithm of Kim et al. [9]. The optimizations in this case were only done to the final join phase of the algorithm and do not change its memory bandwidth consumption.

We tested these algorithms on the three platforms using the default experimental setup of Kim et al. [9]. This setup assumes a column-store database join, where both build and probe relations have 128 million $8B$ tuples. The join in this setup does not create an output relation. We noticed the following:

- Transparent hugepages made a similar impact to performance as with the synthetic benchmark from the previous section.
- Unlike previously, prefetching made significant improvement since the loops were more complex. This causes a lot of data dependency in the code that prevents the processor from doing out-of-order execution.
- For the AMD micro-architectures in test we noticed that the synchronization mechanism of spinlocks used to synchronize the threads during the building of the hash table caused significant overhead. Spinlocks use atomic compare and exchange instructions, which showed to be expensive on these platforms. Thus, we devised a synchronization mechanism that does not rely on locked atomic instructions. We call this mechanism Lazy Lock. A description is provided in section 4.4



(a) Magny-Cours

(b) Bulldozer

(c) Sandy Bridge

Figure 4.2: HJ and RPJ Performance on various Platforms for $128M \cdot 128M$ Join of $8B$ Tuples. TH : Transparent Hugepages, P : Prefetching, LL : Lazy Locks

We can conclude from the results from Figure 4.2 that:

- RPJ requires more computation than HJ and when fully optimized HJ is faster than RPJ when run on fewer cores.
- When running on maximum number of cores, RPJ only slightly outperforms HJ. This does not correspond to our DRAM bandwidth consumption measurements, so the explanation is either there is more memory accessed by RPJ or since it is a multi-stage algorithm, some of the stages are still compute-bound at maximum number of threads.

Impact of Tuple Size

The current setup assumes a join in a column-store database where the join attribute is $4B$, and together with the virtual OID of $4B$, which is used to address tuples from the build relation make tuples to be $8B$. In certain systems this is not enough, so attribute and OID are $8B$, which makes the tuples to be $16B$. This certainly has a big impact on RPJ since it will double the memory used. This effect can be seen in Figure 4.3 where the performance of HJ remained almost the same, while the performance for RPJ degraded by a factor close to 2.

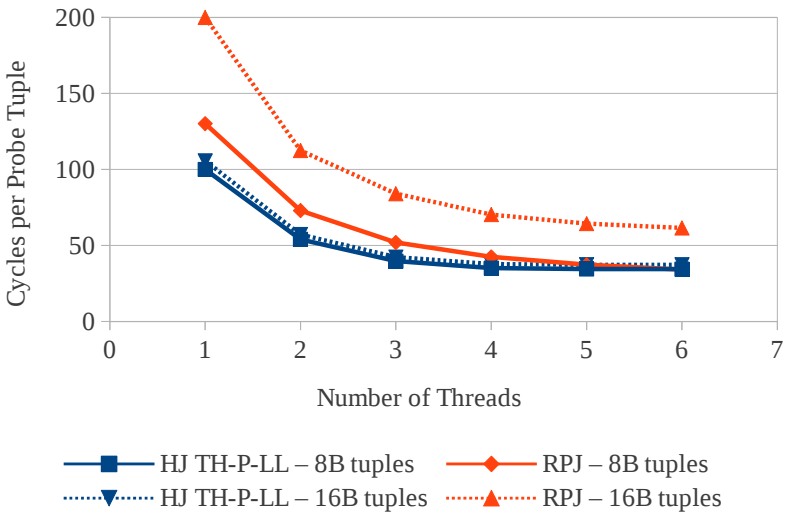


Figure 4.3: Effect of Larger Tuple Sizes on HJ and RPJ

Impact of Build Relation Size

In our bandwidth consumption analysis from Table 4.1 we predicted that HJ consumes almost twice bandwidth per build tuple, than per probe tuple. Our experiments so far assumed equal-sized build and probe relations. In our next experiment we vary the size of the build relation to confirm our predictions.

Figure 4.4 shows the performance of HJ and RPJ when the size of the build relation varies, as well as the theoretical performance based on the calculations from Table 4.1. These results confirm our predictions about the different bandwidth consumption of HJ per build and per probe tuple.

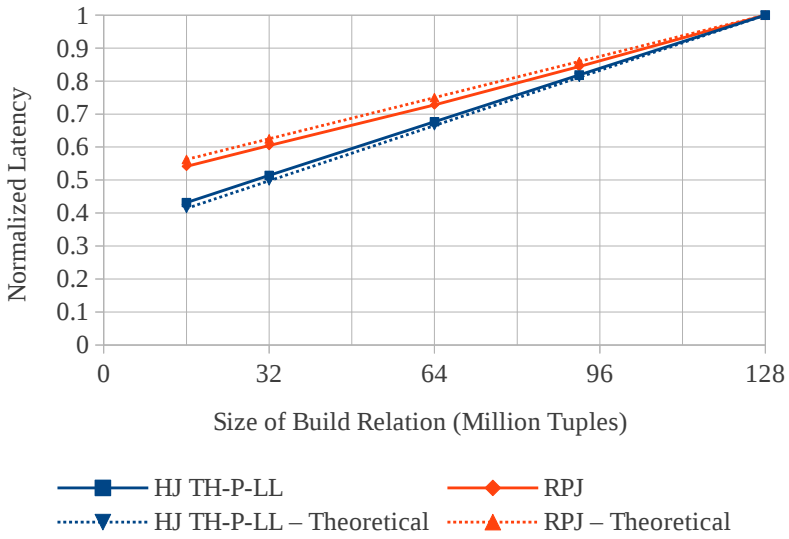


Figure 4.4: Effect of Build Relation Size on HJ and RPJ

Page Size Impact on Partitioning

RPJ uses 2−pass partitioning to avoid the cost of accessing too many TLB page entries. Since, we saw that using larger page sizes significantly reduced TLB miss penalty for HJ, we could also use them to improve RPJ. Technically, if the data fits in the fastest TLB cache, we can partition it efficiently in a single pass. Unfortunately, the Linux kernel currently has transparent support for only $2MB$ page sizes and not the larger $1GB$ page sizes. This turned out to be decisive setback of using 1−pass partitioning instead of 2−pass. Figures 4.5 and 4.6 show that transparent hugepages do manage to improve the performance of the 1−pass partitioning RPJ, but not enough to make a considerable improvement over the 2−pass partitioning RPJ.

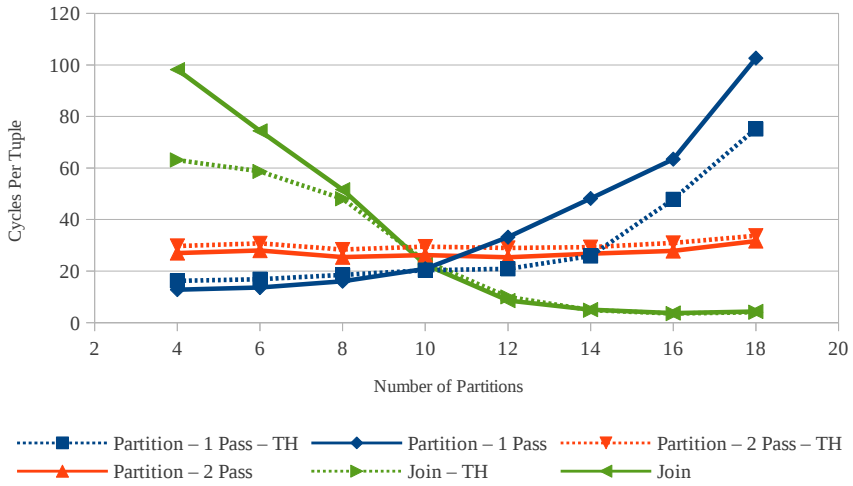


Figure 4.5: Effect of Transparent Hugepages on Partition and Join Phases for RPJ

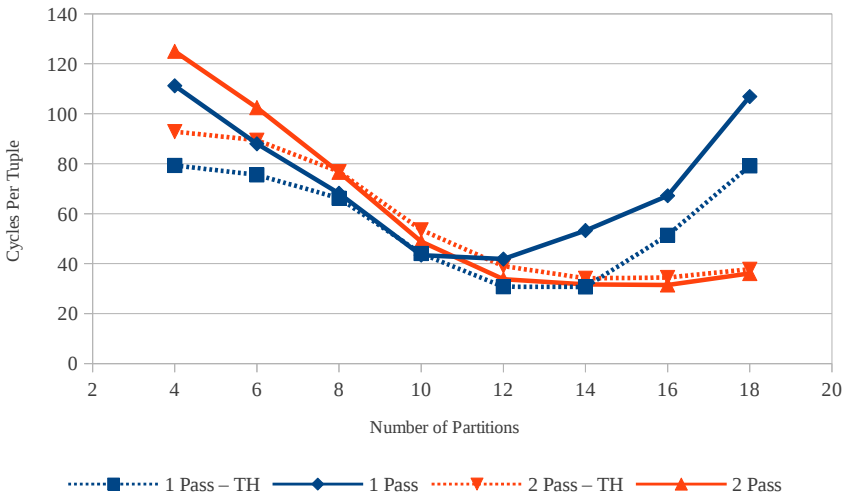


Figure 4.6: Effect of Transparent Hugepages on RPJ

4.4 Lazy Synchronization Technique

The lazy synchronisation technique is suitable for scenarios where there are large number of locks which threads are constantly acquiring and releasing. In this case, performance of lock and unlock method is crucial and it is very unlikely that two or more threads will contend for the same lock at a certain point in time. The usual synchronization method in this case is based on spin-locks. Spin-locks are based on locked atomic instructions, which are expensive in certain micro-architectures as we saw earlier.

The algorithm of lazy locking can be seen in Algorithm 1. Every lock is consisted of as much bytes as there are threads. Then each thread when it tries to acquire the lock writes 1 to its byte, thus setting one bit in its byte. Afterwards the thread reads all the bytes from the other threads, including its own and checks how many bits were set. If there is more than one bit set, it means that there are at least two threads trying to acquire the lock at the same time. If there is only one bit set, the thread can safely execute the critical section of the code.

Algorithm 1 Lazy Synchronization Method

```
lock.bytes[myThreadId] ← 1
bytesSet ← countSetBits(lock.bytes)
if bytesSet = 1 then
    criticalSection
else
    postponeCriticalSection
end if
lock.bytes[myThreadId] ← 0
```

This synchronization mechanism guarantees that at most one thread will enter the critical section, but does not guarantee that at least one thread will enter. This is why when a thread reads that more than one bits were set, it will postpone the work for later when all threads will

be synchronized with a traditional synchronization mechanism.

Lazy locks rely on modern processors' ability to atomically read and write to aligned variables of size $8bit, 16bit, 32bit$ and $64bit$. Only issue can be memory ordering, as different architectures have different memory ordering principles. In any case, if a processor has very weak memory ordering such that the code can not be used without memory fences, which can be as expensive as locked instructions, several locks and unlocks can be grouped together, and only one fence instructions executed for many tuples. This approach can be seen in Algorithm 2.

Algorithm 2 Grouped Lazy Synchronization Method

```
for i = 1 → n do
    locks[i].bytes[myThreadId] ← 1;
end for
memoryBarrier
for i = 1 → n do
    bytesSet ← countSetBits(locks[i].bytes)
    if bytesSet = 1 then
        criticalSection(i)
    else
        postponeCriticalSection(i)
    end if
end for
memoryBarrier
for i = 1 → n do
    locks[i].bytes[myThreadId] ← 0;
end for
```

Obviously the biggest drawback of this approach is that it requires one byte per lock per thread. At least a byte is required, because that is the smallest unit of memory that a modern processor can atomically write to. This drawback is actually not an issue in our case, since we are already saturating the bandwidth with $3-4$ cores. Hence, using $3-4$ bytes per lock does not create problems, especially since we are anyways fetching a cacheline of 64 bytes for each hash bucket.

4.5 Materialization

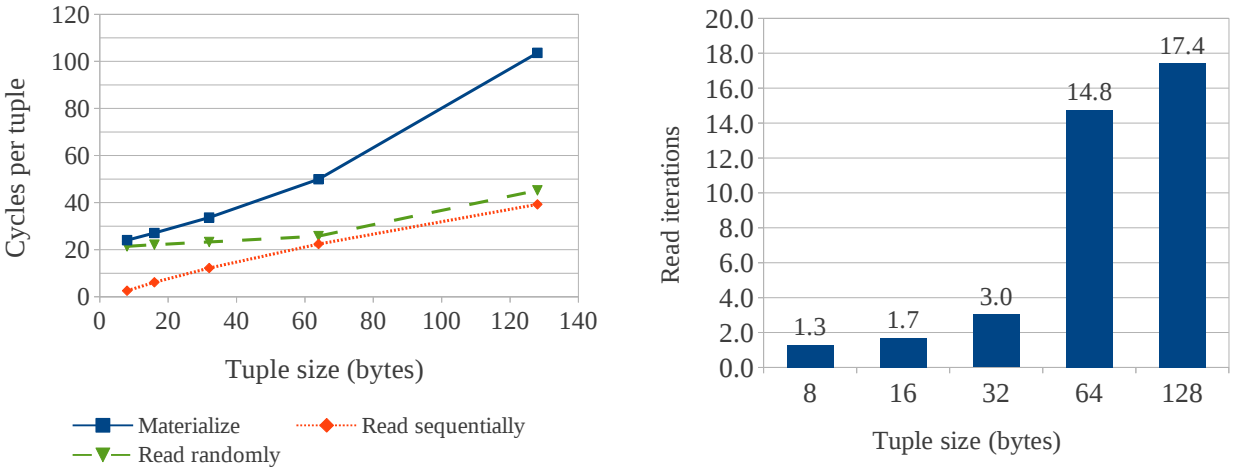
Materializing or projection of the output tuple is also challenging in join algorithms. Similarly to RPJ, partitioning based projection was proposed by Manegold et al. [11]. Projection however is trickier than the join operation, since partitioning the probe relation is very restrictive. The solution proposed in [11] is to partition the join index generated by the join operation twice, once for the probe relation and once for the build relation. The problem in that approach is that for every output tuple, the projected attributes from the build relation have to be copied twice. This approach is more attractive under the assumptions given in the paper that random access is approximately 10 times more expensive then sequential access, however as we have

seen earlier, this is no longer the case. Hence, we decided to go with random access projection based on prefetching.

Given that random memory access is comparable to sequential access, we can raise the question of whether we need to do any materialization. So we performed a series of experiments where we wanted to see at which point it is better to materialize?

We measured time to copy randomly placed variables into sequential order, as well time to just read them randomly or sequentially, while we varied the size of these variables. The results can be seen in Figure 4.7a.

The next question is how many read iterations of the join's output relation are required so that materialization pays off. This is a trivial question for a regular database system, since data is not shared among queries, but for SharedDB it is important. The output relation of a join can be read by many operators. The results are presented in Figure 4.7b.



(a) Materialization vs Random read vs Sequential read cost (b) For how many Read Iterations Materialization Pays of

Figure 4.7: To Materialize or not to Materialize?

4.6 SharedDB Implementation

Based on our analysis and results, we decided that HJ is the best fit for SharedDB. We implemented the algorithm as a SharedDB operator. The micro-benchmarks so far were performed on a join operation that only built the hash table and probed it without doing output tuple materialization. Apart from that, SharedDB requires that query ID sets from build and probe tuples are joined and that tuples are pre-materialized for each join operator.

4.6.1 Hash Bucket Structure

Before the implementation of the algorithm, we had to decide on the structure of a hash bucket. Our primary goal was to minimize the number of random cachelines accessed while processing

build and probe tuples. This requires keeping all meta-data related to a hash bucket in a single cacheline. This meta-data consists of:

1. The join attribute
2. The pointer to the tuple
3. The pointer to the tuple's QIDs (and size of QIDs object, if necessary)
4. The pointer to the next bucket in case of a colision
5. The Lazy Lock object
6. The last session ID

Given that these objects are grouped and aligned properly, they can easily fit in a 64 byte cacheline, and there would still be some space left. The remaining space can be used to cache the QIDs if they are few, or the tuple itself if it is small enough. Currently, only the QIDs are cached.

4.6.2 Hash Table Reusing

A SharedDB join operator is executing batches of queries non-stop, which renders creating and initializing a hash table for each batch of queries expensive. Thus, we prefer to reuse the hash table when we process a new batch of queries. Previously in the SharedDB's join operator this was achieved by using a separate bitset which contained one bit for every hash bucket. The bitset was used to mark if a hash bucket is used or not for the current batch, and it was reset after each batch. The problem with this approach is that it requires additional random access when a corresponding bit for a hash bucket needs to be checked. And if the bitset was removed, then the huge data structure of the hash table needs to be reset to zero to avoid data corruption for the next batch.

The solution is to include a variable for each hash bucket, namely the last session ID mentioned above. The session ID for a join is incremented for each batch of queries. In this way, when a thread accesses a hash bucket, it first checks for the session ID to see if this bucket is active for this batch or not.

Resetting the whole hash table in this case is still necessary but only after all possible values for a session ID have been used out. Thus, for a 2-byte session ID, the hash table needs to be reset once for every $16'384$ batches.

4.6.3 The Algorithm

An important decision about the algorithm that we made is whether to use a grouped prefetching technique or a pipelined prefetching technique. Since both techniques were shown to perform equally well by Chen et al. [5], we decided to go with the solution that is simpler and is more appropriate for modern processors.

An algorithm like hash join that can make use of software prefetching is typically consisted of iterating over a large amount of items, each of which needs processing that requires one or more random accesses to memory. We will call these random accesses stages.

- Normal execution without prefetching means iterating over the data, and executing all stages for each item.
- Pipelined prefetching works by iterating over the data normally, but executing each stage for different items. At each stage we issue a prefetch instruction for the next stage.
- Grouped prefetching works by executing stage by stage, and executing prefetch instructions for the next stage, but for groups of items.

In the case of an algorithm with fixed number of stages, pipelining and group prefetching would be similar. Group prefetching would require as much `for` loops as there are stages, and pipelining would require as many states for different items as there are stages.

For SharedDB's join algorithm, we have multiple, some of them repeatable, and data dependent stages. This seriously hinders pipelined prefetching performance, since it causes branch mispredictions while executing a stage for a single item. Hence, we decided to go with grouped prefetching.

Build Phase

The build phase is simpler than the probe phase, in terms of that it requires only one random access per tuple, which is the corresponding hash bucket itself. As a result, grouped and pipelined prefetching will both work for the build phase. A pseudo-code is presented in Algorithm 3

Probe Phase

The probe phase is more complicated and consists of several stages. Each stage prepares the data and executes prefetch instructions for the next stage. The stages are the following:

1. Calculate addresses of hash buckets, prefetch them and prepare the array of {tuple, bucket} pairs for the next stage.
2. Process supplied {tuple, bucket} pairs:
   (a) Check if the buckets' session IDs are valid, check if keys are equal, and if both of these hold, prefetch the query IDs and add the tuples to next stage's array.
   (b) If the buckets' next bucket pointers are not zero, prefetch them and add them to the list of colision buckets.
3. Calculate QID intersections for tuples from last stage and for those tuples, whose QID intersections are not empty, add them to the array for the next stage, and prefetch the tuples' records.

Algorithm 3 Build Algorithm

```
procedure BUILD
    while t ← FETCHTUPLE do
        ADDTUPLETOGROUP(t)
        if groupSize > groupMaxSize then
            PROCESSGROUP
            CLEARGROUP
        end if
    end while
    PROCESSGROUP
end procedure
procedure PROCESSGROUP
    for tuple ∈ group do
        bucket ← CALCULATEHASHBUCKETADDRESS(tuple)
        PREFETCH(bucket)
    end for
    for tuple ∈ group do
        bucket ← CALCULATEHASHBUCKETADDRESS(tuple)
        if !TRYLAZYLOCK(bucket, myThreadId) then
            ADDTOLAZYTUPLES(tuple)
            CONTINUE
        end if
        if bucket.lastSessionId = currentSessionId then
            colisionBucket ← GETNEXTCOLISIONBUCKET
            SWAPNEXTBUCKETPTRS(bucket, populateBucket)
            POPULATEBUCKET(tuple, colisionBucket)
        else
            POPULATEBUCKET(tuple, bucket)
        end if
        RELEASELOCK(bucket, myThreadId)
    end for
end procedure
```

Algorithm 4 Probe Algorithm

```
procedure PROBE
    while t ← FETCHTUPLE do
        ADDTUPLETOGROUP(t)
        if groupSize > groupMaxSize then
            PROCESSGROUP
            CLEARGROUP
        end if
    end while
    PROCESSGROUP
end procedure
procedure PROCESSGROUP
    checkKeysGroup ← Array()
    colisionGroup ← Array()
    for tuple ∈ group do
        bucket ← CALCULATEHASHBUCKETADDRESS(tuple)
        PREFETCH(bucket)
        ADDTOARRAY(checkKeysGroup, {tuple, bucket})
    end for
    while checkKeysGroup.size > 0 do
        checkQIDsGroup ← Array()
        for {tuple, bucket} ∈ checkKeysGroup do
            if bucket.lastSessionId = currentSessionId
AND MATCHKEYS(tuple, bucket.key) then
                ADDTOARRAY(checkQIDsGroup, {tuple, bucket})
                PREFETCH(bucket.qids)
            end if
            if bucket.nextBucket ≠ null then
                ADDTOARRAY(colisionGroup, {tuple, bucket.nextBucket})
                PREFETCH(bucket.nextBucket)
            end if
        end for
        materializeGroup ← Array()
        for {tuple, bucket} ∈ checkQIDsGroup do
            resultQIDs ← JOINQIDS(tuple.qids, bucket.qids)
            if resultQIDs.size > 0 then
                ADDTOARRAY(materializeGroup, {tuple, bucket, resultQIDs})
                PREFETCH(bucket.tuple)
            end if
        end for
        for {tuple, bucket, qids} ∈ materializeGroup do
            MATERIALIZETUPLE(tuple, bucket.tuple, qids)
        end for
        SWAP(checkKeysGroup, colisionGroup)
    end while
end procedure
```

4. Materialize all tuples that passed the previous stage and push them to the output.

5. If the colision array from stage $2a$ was not empty, go to stage 2 with {tuple, colision bucket} pairs.

A pseudo-code version of the above algorithm is shown in Algorithm 4. One thing to note is that even though the algorithm contains many `if` statements, most of them are avoided in the actual implementation.

We can notice from the algorithm that the number of tuples that progresses through each stage is filtered. Thus, the group size of each prefetching stage will be reduced and at some point prefetching might be no longer effective. This is true and in fact with this algorithm we can never avoid the case when at a certain point there is only a few tuples in the stage and prefetching is issued to late to be effective.

However, if we set the initial group size to be significantly larger than the minimum group size required for prefetching to be effective, then we can minimize the effect of the penalty of a few late prefetches in the further stages.

4.6.4 Performance

We tested the performance of the new SharedDB join operator, and the results can be found in Figure 4.8. The figure compares the algorithm to a parallelized version of the previous SharedDB join operator.

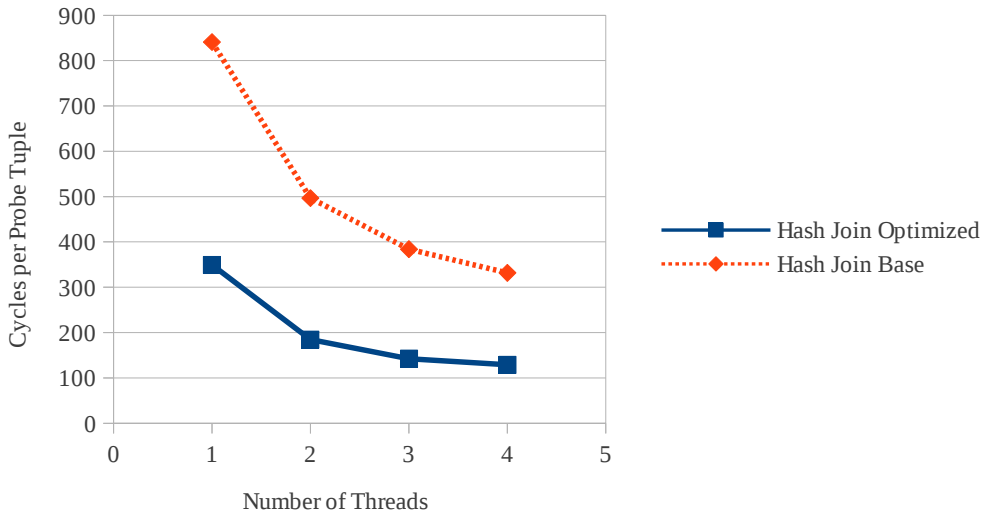


Figure 4.8: Performance of the proposed SharedDB Join Algorithm, Compared to the Previous SharedDB Join Algorithm

The join operation is a full-table join of two TPC-H relations: `Part` and `Partsupp`, where only the join attributes are projected. The scale of the TPC-H data is 10, which means that the

size of relations are $2M$ and $8M$ tuples respectively. Both relations are completely randomized to avoid sequential access.

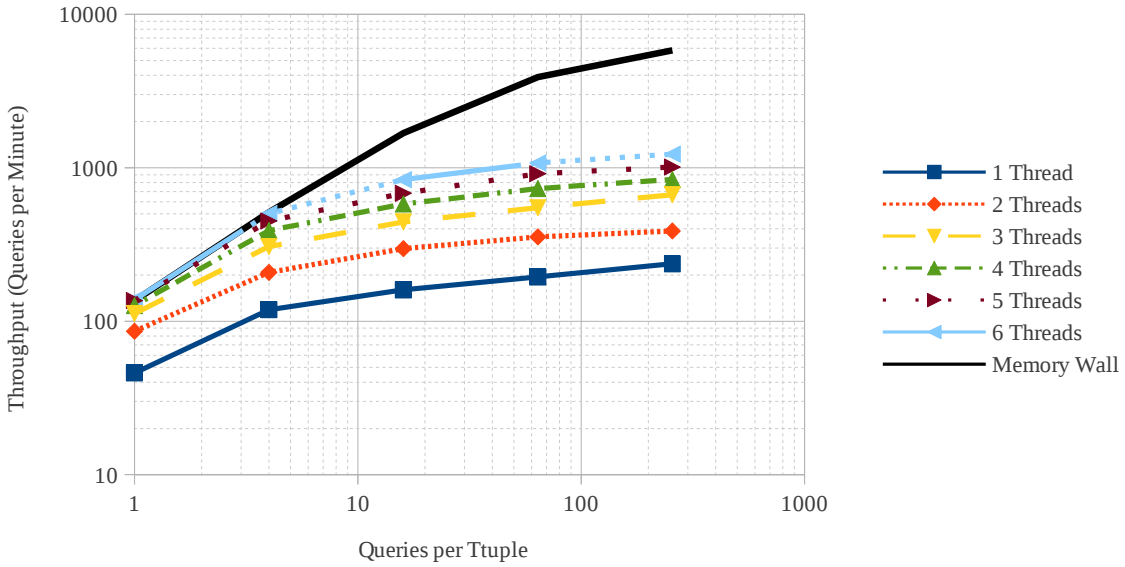


Figure 4.9: Performance of the SharedDB's Join Algorithm with Multiple Queries per Tuple

An important thing to note from this graph is that unlike in the synthetic environment where the performance of the hash join was around 30 cycles, in the SharedDB environment it is above 100 cycles. The reason is that the meta-data per tuple in SharedDB currently is 64 bytes. In this setup, the producers for the join have to write this metadata to memory before the join reads it. Furthermore, since it is a full table join, the join operator also writes a 64-byte metadata object for every tuple, which is read by its consumer. All in all, this causes more transfer to main memory and back than the join in the synthetic setup used previously. Hence, we can conclude that inter-operator communication in SharedDB needs to be optimized. We discuss this further in Chapter 6.

The experiment from Figure 4.8 showed the performance of a full-table join, where there is only one query per tuple. This is equivalent to executing only one query in SharedDB, which is meant to execute multiple queries at once. These queries are likely to share tuples so next we tested how does this join perform when tuples are shared across queries. The results are presented in Figure 4.9, where we observed performance while varying the queries-per-tuple parameter. Additionally, we show the calculated maximum performance if the join were to be memory bound. Thus, we can conclude that the QID processing part of the join is too computationally expensive and requires optimization.

In the next Chapter, Chapter 5, we will describe the current method of how query IDs are represented and handled, and we will propose different techniques that can speed up the part of processing QIDs.

Chapter 5

Handling of Query IDs (QIDs)

One of the benefits of a database like SharedDB is that tuples can be shared across different queries. However, as a tuple is shared across many queries, the cost of processing all the queries for the tuple increase. A simplified equation of the processing speed up for a join operation in SharedDB over a normal database is:

$$\frac{Time_X}{Time_{SDB}} = \frac{n \cdot t_X}{t_{SDB_{Tuple}} + n \cdot t_{SDB_{QID}}} \tag{5.1}$$

where n is the average number of queries per tuple, t_X is the time required to process a tuple for DBMS X, $t_{SDB_{Tuple}}$ is the time required to process a tuple for SharedDB, and $t_{SDB_{QID}}$ is the time required to process a single query ID for a tuple. In chapter 4 we covered the cost of t_X and $t_{SDB_{Tuple}}$ for a join operation, and in this chapter we will focus on $t_{SDB_{QID}}$.

We will present three ways of how to store and handle query IDs for a tuple in sections 5.1, 5.2, and 5.3, as well as how to compress the QID space in section 5.4.

Different kinds of operators process QID sets in a different way, hence different techniques of storing QID sets need to have good performance characteristics in several aspects. Since this work is focused around the join operation, and the join operation has to process QID sets by creating intersection between QID sets from build tuples and probe tuples, we will mostly pay attention to the calculation of intersection between two QID sets. Nevertheless, we will also cover other standard properties which would be beneficial for any database operator.

5.1 Arrays

The first technique for storing QIDs that we analyze is the currently implemented technique in SharedDB. In this case, the QID set for a tuple is stored in an array of 4 byte integers. This approach is more favorable than approaches like bitsets in the case when the ratio $r_{\frac{QpT}{QpB}}$ is reasonably low. This ratio is defined as:

$$r_{\frac{QpT}{QpB}} = \frac{QpT : Number\ of\ queries\ per\ tuple}{QpB : Number\ of\ queries\ in\ the\ batch} \tag{5.2}$$

The variable QpB is specific to bitsets and will be discussed later. Advantages of using arrays are that in cases of low values for $r_{\frac{QpT}{QpB}}$, memory consumed is low ($QpT \cdot 4bytes$).

Disadvantages of this approach are that in cases of high values for $r_{\frac{QpT}{QpB}}$, memory consumed is high, and the time required to do intersection or union is linearly proportional to QpT.

5.1.1 QID Set Intersection Performance

Set intersection for two arrays of QIDs can be seen as a regular join operation between two relations. The first implementation in SharedDB was based on hash-join, where a hash set was populated from one of the arrays, and then subsequently probed with QIDs from the other array.

One thing that differentiates sets of QIDs with sets of tuples is that QIDs are generated during execution of a query, thus their order can be manipulated at tuple creation time. As a result, a merge join is more attractive in this case.

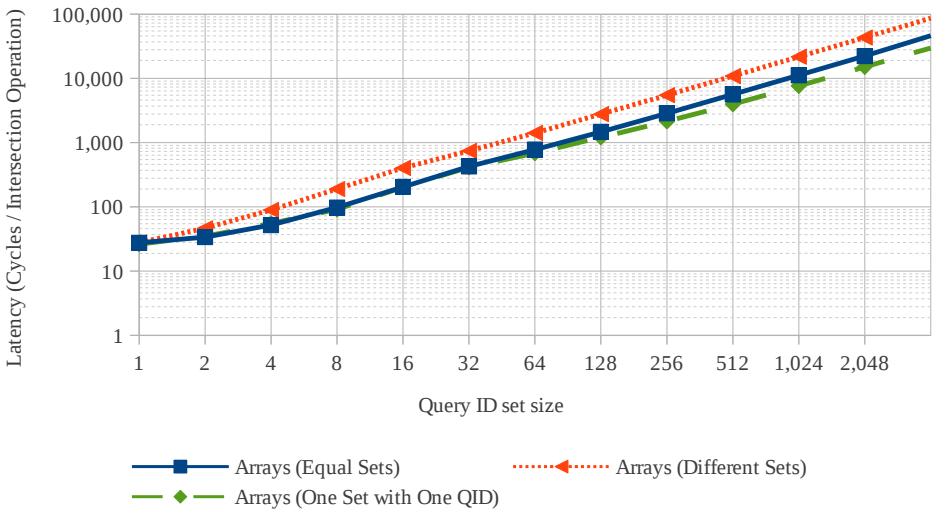


Figure 5.1: Merge-join Performance for Sorted QID Arrays

In Figure 5.1 we see the performance of merge-join on QID arrays

5.2 Bitsets

Bitsets have the natural advantage over arrays that they can compress the amount of data significantly. Furthermore, the set intersection operation for bitsets can be done using a bitwise `AND` which, if bitsets are aligned properly, can be executed in one instruction for up to $64bit$ regular integers or $128bit$ vectorized integers with SSE instructions.

5.2.1 Managing query-to-bit mapping for bitsets

A problem with bitsets is that every query must have a bit assigned to it for every tuple, even if the query is not interested for the tuple. The implication is that if the ratio $r\frac{QpT}{QpB}$, defined in equation 5.2, is very low and QpB is significantly higher than 64 or 128, then bitsets will not only consume more space than arrays, but will also take more time to execute the set intersection operation.

Query-to-bit mapping is an important part for an implementation of the bitsets approach, since in order to maintain high performance for operations such as intersection, or union, the mapping must be consistent throughout the system.

This problem has been seen in a number of other systems. CJoin [4] uses global query-to-bit mapping, and explanation of how queries are mapped to bits is not provided for DataPath [1]. The problem of global mapping is that bitsets will be as large as the number of queries in the system. Hence, the sparsity in bitsets will not only be influenced by the type of workload, but also on the size and complexity of the system.

What we propose as an alternative to global query-to-bit mapping is batch-specific query-to-bit mapping. This means, that each operator before it starts processing a batch of queries, decides how each of these queries will be mapped to bit locations. Immediately, one can make a remark that this will create different mappings for separate operators, creating the need for expensive shuffle operations when operators exchange tuples. This is true, however we will show that with some tricks, expensive bit shuffling operations will not be needed.

A demonstrating example for this approach is depicted in Figure 5.2. In this example, operator X has received 7 queries from operator A, and 6 queries from operator B. Of all the 13 queries it has to process, 7 need to be further enqueued to operator 1, and 6 need to be enqueued to operator 2.

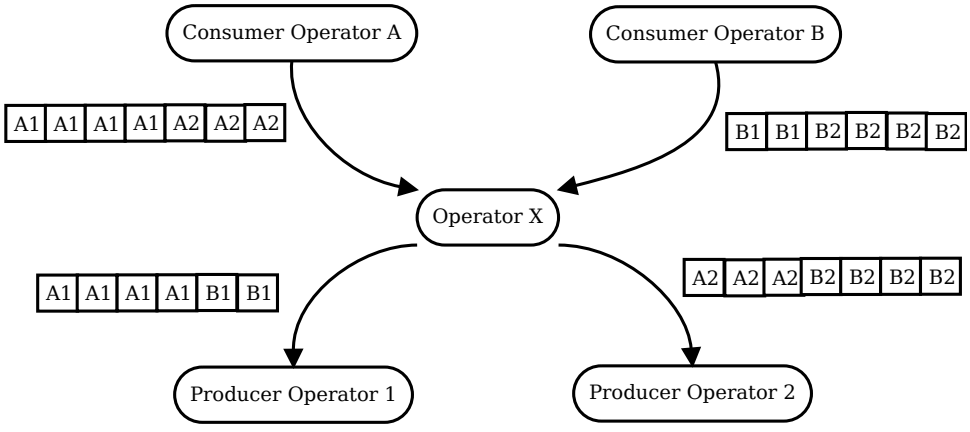


Figure 5.2: Batch-specific Query-to-bit Mapping Example

When operator X has received enqueued queries from operators A and B, it has also received the query-to-bit mapping that these operators want. Thus, when operator X pushes results to A and B it has to make sure that the query-to-bit mapping is the same as the one that these

operators have requested it to be. This can be achieved either by creating two separate bitsets for A and B for each tuple, or creating one shared bitset and informing the A and B operators, which regions in the bitset correspond to the regions that they are interested in.

In a similar manner, X enqueues all $A1$ and $B1$ queries to operator 1, and all $A2$ and $B2$ queries to operator 2. Subsequently, when X receives tuples from e.g. operator 1, it will have to separate the received bitset into two parts, one for A and one for B. It is easy to note here that the received bitset will be easy to be separated, since the original query-to-bit mapping that X has received from A and B have already been nicely implemented such that queries' mappings are sorted into groups for operator 1 and 2 respectively.

Thus, in order to ensure simplified shuffling when tuples are forwarded from producing operators to consumer operators, top-level operators have to be aware of the complete plans for all queries, so that they can be ordered in an optimal way.

5.2.2 Bitsets Performance

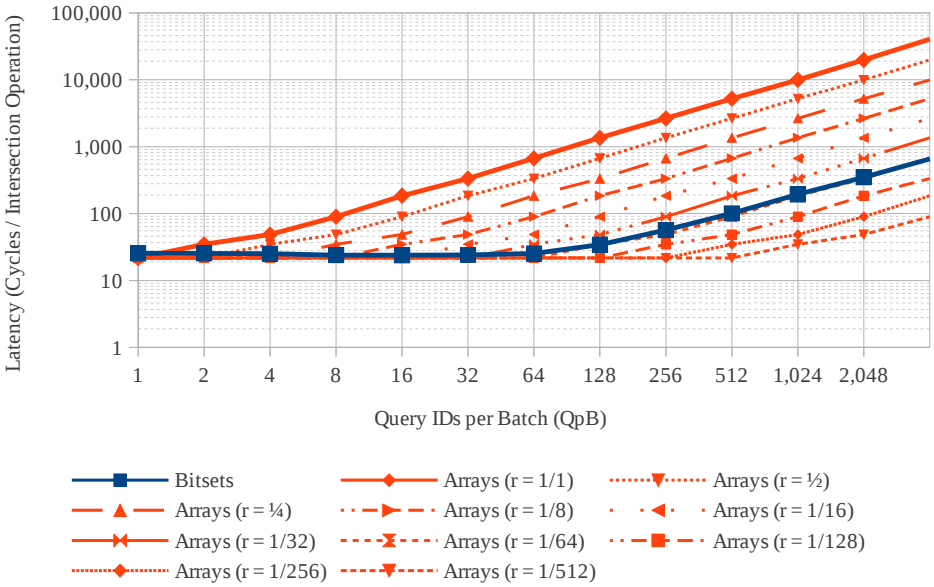


Figure 5.3: Set Intersection Throughput for Bitsets

In Figure 5.3 we see how bitsets perform for set intersection when we vary the queries per batch parameter. We also compare it to the performance of arrays for different values of $r_{\frac{QpT}{QpB}}$. Intuitively, the bitsets' and arrays' performance is comparable when $r_{\frac{QpT}{QpB}} = \frac{1}{64}$. In Figure 5.4 we see a similar comparison of the two approaches, but this time with respect to memory consumption. Naturally here the threshold is $\frac{1}{32}$, since we are using 4 byte integers as QIDs.

Apart from set intersection performance, we also have to examine three additional operations on bitsets:

1. Time to set all bits to zero.

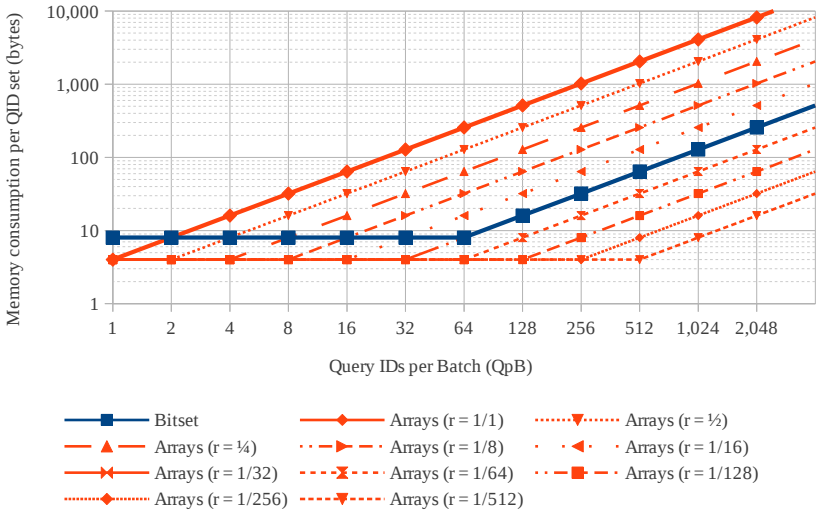
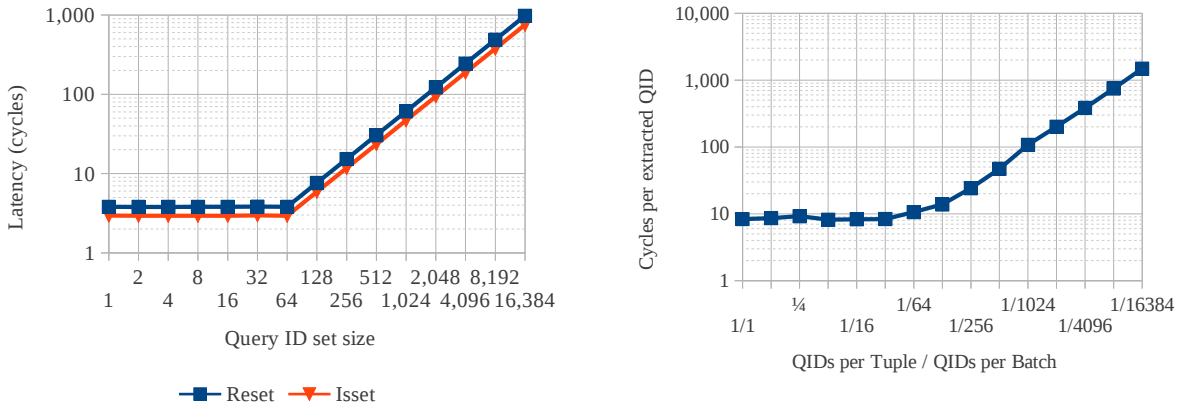


Figure 5.4: Memory Consumption per QID Set for a Tuple for Bitsets and Arrays

2. Time to test if there is at least one bit set.

3. Time to extract indexes of all bits that are set.



(a) Performance of Reset and Isset Operations

(b) Bitsets to Arrays Conversion Performance

Figure 5.5: Performance for Reset, Isset and To Array Conversion for Bitsets

The first operation is needed by any operator that creates new tuples. The second operation is needed for any operator that filters the bitset (does bitwise AND) including the join operator, and the third is required by operators that have specific state per query and need to iterate over all QIDs in the set.

Performance of all these operations is presented in Figure 5.5. As expected performance is constant until the magical number of 64 which is our unit operation size.

5.3 Combined

We presented the two methods of arrays and bitsets for handling QIDs, as well as their advantages and disadvantages. The following approach tries to combine the two to get advantages from both of them, minimizing the disadvantages at the same time.

The basic idea is to also use bitsets, but only to write those regions of bitsets that have at least one bit set. E.g. bitsets would be divided in regions of 24 or 56 bits, and each region will be annotated with 8 bit integer which will define the offset of the region in the bitset.

The key idea that would make this work reasonably fast is that a region together with its offset will fit in a 32 bit or 64 bit integer. In Figure 5.6 we present pseudocode algorithms for the intersection operation for all the three methods discussed.

Algorithm 5 Bitsets

```
for i = 0 → QpB64 do
    outQIDs ← leftQIDs[i]&rightQIDs[i]
end for
```

Algorithm 6 Arrays

```
li ← 0
ri ← 0
oi ← 0
while li < left.cnt AND ri < right.cnt
do
    if left[li] = right[ri] then
        out[oi] ← left[li]
        oi ← oi + 1
        li ← li + 1
        ri ← ri + 1
    else if left[li] < right[ri] then
        li ← li + 1
    else
        ri ← ri + 1
    end if
end while
```

Algorithm 7 Combined

```
li ← left.start
ri ← right.start
oi ← out.start
while li ≠ left.end AND ri ≠ right.end
do
    if li.offset = ri.offset then
        oi.bits ← li.bits & ri.bits
        if oi.bits > 0 then
            oi.offset ← li.offset
            oi ← oi + 1
        end if
        li ← li + 1
        ri ← ri + 1
    else if li.offset < ri.offset then
        li ← li + 1
    else
        ri ← ri + 1
    end if
end while
```

Figure 5.6: Set Intersection Pseudocode for the Three Methods

In Figure 5.7 we show intersection performance for what would the worst case for the combined method, compared to the arrays method. In this case, there would be one bit set for every bitset block. Memory consumed in this case would be the same or twice the one of arrays, given that we use 32bit or 64*bit* blocks. The performance we see in this case is with 32bit blocks and we can conclude that the subtle changes in code do not affect performance when compared to arrays.

In Figure 5.8 we show intersection performance for what would be the best case for the combined method and the bitsets method. Here it is certainly unavoidable that the combined method is slower than bitsets, however by increasing the block size to 64bits causes significant improvement and closer performance to bitsets.

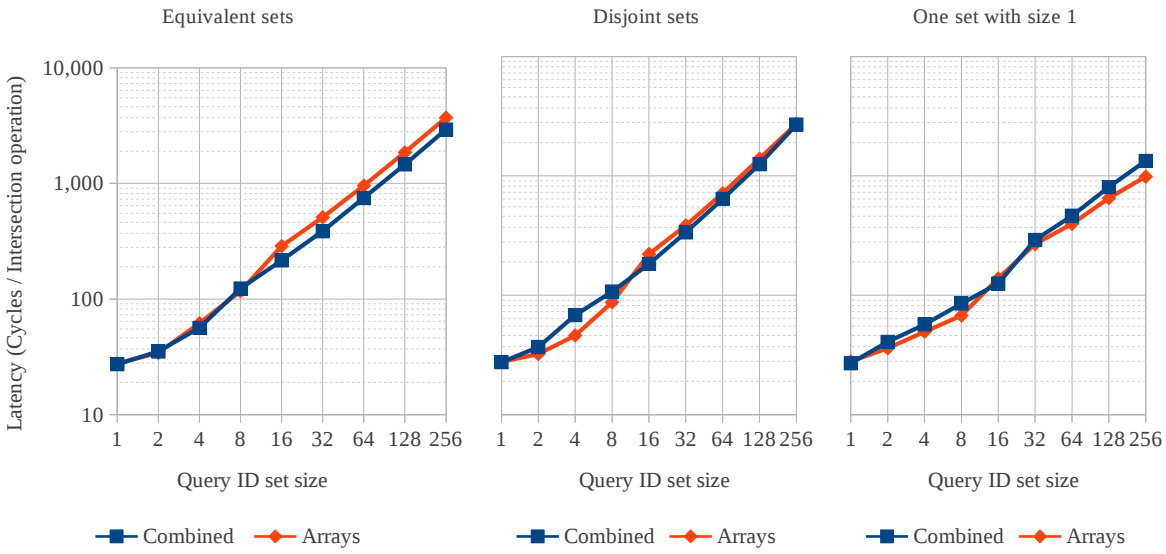


Figure 5.7: Intersection Performance for Worst Case Scenario for Combined Method Compared to Performance of Arrays

In SharedDB, we could easily avoid the worst-case scenario by doing some tricks with query-to-bit mappings. For instance, queries with equality predicates on certain attributes can be grouped together based on the predicates' values. In cases of comparison predicates, queries can be sorted based on the predicates' values.

On the other hand, the best-case scenario is not quite realistic, or at least not for the majority of tuples. If many queries share many tuples, they will likely be queries equal to each other, in which case can be grouped and treated as a single query. We will discuss this in section 5.4.

Another important note to make is that even though the simple bitset implementation is faster in this synthetic environment, we expect that similarly to the join operation it will easily saturate the DRAM bandwidth of the system. In the end, the amount of bytes saved by the combined approach together with smart query-to-bit mapping will dictate the performance of the system.

5.4 QID Compression

So far analyzed techniques of how to store and process QIDs. All of these techniques scale linearly with the number of queries. Some of them, like bitsets, actually scale constantly until

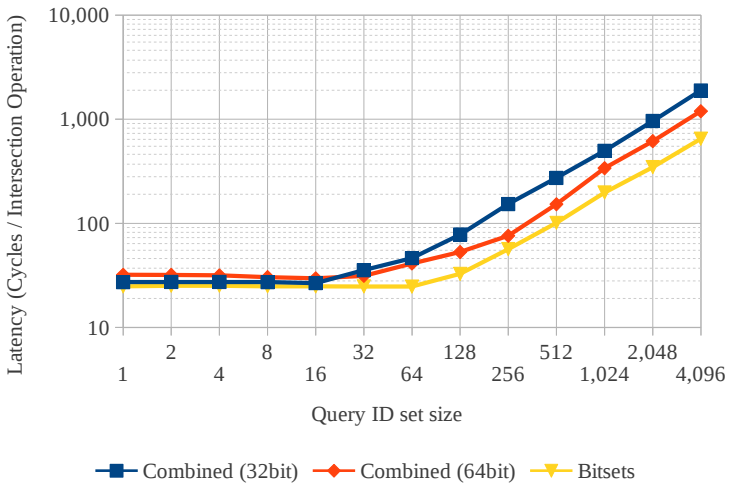


Figure 5.8: Intersection Performance for the Best Case Scenario for Combined Method Compared to Performance of Bitsets

a certain point.

Even though these techniques are fast, we can expect that DRAM bandwidth would become a bottleneck at some point. In that case any reduction in the size of the of QID set representation would proportionally improve performance.

One obvious characteristic of a usual workload is that in many cases joins have `SELECT ALL` queries as build or probe subqueries, or even both. For example, in the `TPC-H` [14] workload, for the cases when join operations are required, 25% of the joins related to the largest `Lineitem` table, have `SELECT ALL` on `Lineitem`. The same number is 100% for the `Partsupp` table and 33% for the `Orders` table.

We can take advantage of this, together with other scenarios where queries share common subqueries to combine them into a single query and share both data and computation costs. This would resemble multi-query optimization in traditional database systems.

Chapter 6

Execution Engine

In the previous Chapters, we presented how to execute a single join operation, and how does SharedDB change the requirements of it. We also analyzed how to efficiently handle the tuple-to-query membership representation. In this chapter we will take this a level higher and see how can we execute and schedule multiple join operations together with a storage engine that is supplying data to the joins.

To achieve this, we will use the SharedDB's execution engine, with the join algorithm described in Chapter 4. The workload we will use, is a modified set of `TPC-H` read-only queries. From QID handling perspective, we will use the old array method of handling QIDs, and leave the wide-spread SharedDB implementation of the other more optimized techniques for future work.

To ideally compare our shared join approach with a standard database join, we have to isolate the join operator from the storage engine and other database operators. Since the work required was not enough for the timeframe of this project, we decided to include the effects of the storage engines. As a SharedDB storage engine we will use Crescando [15] which is suitable for heavy full-scan shared workload. As a baseline, we will use PostgreSQL, supplied with 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. Since SharedDB is an in-memory database, we gave a big buffer pool to PostgreSQL, 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.

This setup is not ideal to compare the two approaches of shared joins and query-specific joins. 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 like workload.

6.1 The TPC-H Workload

The read-only 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. Since we are testing only the performance of the storage engines and join operators, we removed the group aggregations from the queries. 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've only added a limit operator on top of the join operators. The limit operator, unlike in traditional databases, does not stop the execution of the join after the limit has passed, thus we can achieve as close comparison of the two systems as possible.

Apart from removing the 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. After applying all these transformations to the queries, we removed certain queries which either had no join operation in them, or the join type is unsupported by SharedDB at the moment. 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.

6.1.1 Execution Plan in SharedDB

The initial plan that we came up with is shown in Figure 6.1. At the bottom of this graph are the numbered exit points of the corresponding queries. Some queries share exit points, since they are being fed from the same operator. At the top of the graph are the storage engine operators and in the middle are the hash join operators. The lines between operators depict how data is flowing between operators. Hash join operators have two types of lines: full lines and dotted lines. Dotted lines represent the left-hand side of the join or the build input, and full lines represent the right-hand side of the join or the probe input.

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` join. The reason for this is to eliminate replicating tuples during join operation. For instance, if the `Orders` $\bowtie$ `Customer` join operator would feed the from both `Customer` $\bowtie$ `Nation` and `Customer`, 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.

Another remark is that the join operators materialize only attributes that would be required for group aggregators afterwards.

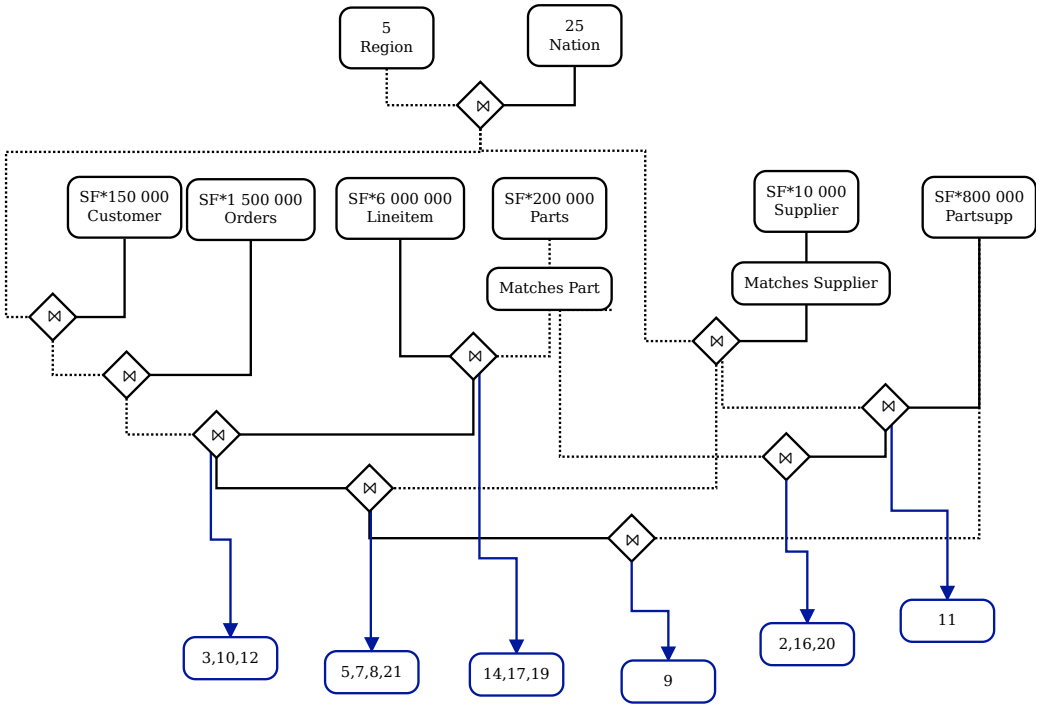


Figure 6.1: SharedDB Query Execution Plan for the TPC-H Workload

6.2 Evaluation

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

6.2.1 Core Allocation

SharedDB currently does not have an optimizer that automatically allocates cores to operators, hence we had to do this job manually. Our first approach was to keep all expensive Crescando and join operators on isolated cores. We were especially careful with allocated cores to Crescando operators, since Crescando is parallelized by partitioning the data evenly across all operators, hence affecting only one Crescando core, might slow down the whole operator.

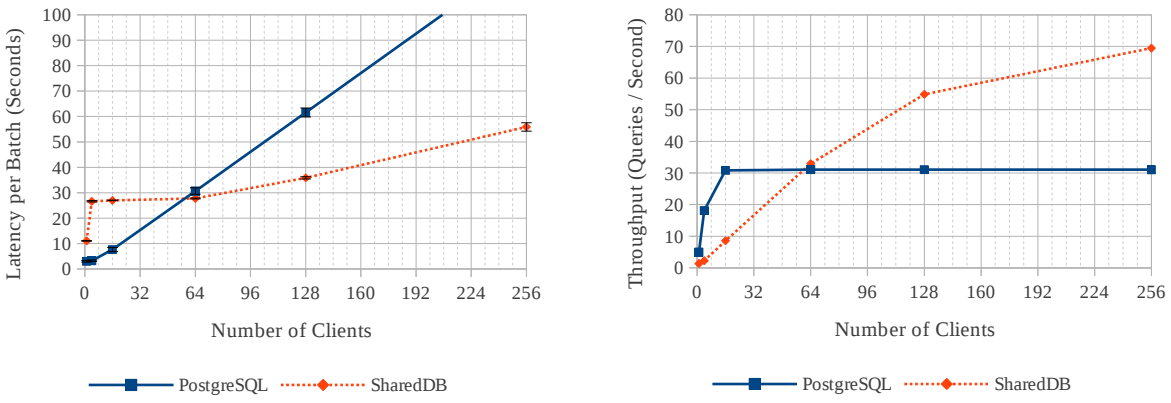
This approach, however, lead to great under utilization of the system, which is why we abandoned the idea of isolating the operators and decided to allocate many cores to all operators and have them overlap. We only kept in mind that the Crescando storage engines' cores are kept separate from the join operators' cores.

Thus, we ended with a scheme where all Crescando operators were overlapping on a set of 30 cores, and the rest of the SharedDB operators, including join, were running on the rest of the 48 cores of the system. This ratio of 30 to 18 for Crescando cores to join cores was decided on a trial and error basis.

Since, currently SharedDB does not expose individual cores of operators outside of the operator, we could not connect cores from different operators. Hence, we could not create a NUMA aware system that minimizes bandwidth consumption across nodes, so we did all core allocation in a round-robin fashion having each operator span across many NUMA nodes.

6.2.2 Results

In figure we show how SharedDB performed against PostgreSQL. The setup for this experiment is that each client measured the time it takes to execute once the batch of 15 queries with parameters randomized according to the TPC-H specification. Furthermore, for both PostgreSQL and SharedDB each client tries to execute all 15 queries at once.



(a) Response time

(b) Throughput

Figure 6.2: Latency and Throughput for the TPC-H Workload Compared on SharedDB and PostgreSQL

For PostgreSQL, we set the maximum number of processes to be the number of cores on the machine, i.e. 48. This means that we have 48 connections from the client's side for which the client threads are queueing up in FIFO fashion to get their queries executed. And since each client tries to execute all 15 queries at once, we notice that these connections are easily saturated after 16 client threads, since at that point there are already $16 \cdot 15 = 240$ threads trying to execute queries at the same time.

For SharedDB, this means that the number of queries in the system is massive. For instance, when there are 256 client threads issueing queries, there are actually $256 \cdot 15 = 3840$ 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 we have 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. We call this effect a batch synchronization problem and we explain it in more detail in section 6.3.1.

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. In the following section we discuss these effects together with other important aspects of the SharedDB's execution.

6.3 Discussion

In this section we will provide discussion for the drawbacks of the current execution engine, as well as give ideas on how to improve on each point. In the next section, we will present couple techniques that we've used to mimic the proposed ideas in this section and look at the performance improvements that they have brought.

6.3.1 Operator Spread-out

When looking at how to process batch of queries which can share tuples, we can look at two extremes, in terms of number of independent processors (queues) in the system:

1. The first extreme is that there is a single generic operator which takes all queries and executes them in some magical way. It is not really important how they are executed, it is just that there is only one queue where queries are enqueued

2. The other extreme is that there are as many as possible specialized operators all of which are independent and are responsible for doing certain type of operations. In this case, in order to execute a query, it must be split into several subqueries and then enqueued to proper operators.

Both extremes, naturally have disadvantages. The problem with the first extreme is that the system will not process new queries until all current queries are executed. Thus, if there is one very long running complicated query in the system, it will prevent the execution of other possibly simpler short-running queries. The disadvantages of the other extreme are that information exchange between operators will become a major burden and that since there are much more queues in the system, a complete query will have to wait at many queues before it is executed. Currently SharedDB is leaning more towards the second extreme, and in the following paragraphs we will show examples where these disadvantages impact the performance for our workload. In order to improve on this we will have to group operators in SharedDB together, as long as it does not impact too much the predictability of the system.

Batch Synchronization / Too Many Independent Operators

This paragraph deals with the issue that queries have to wait at many independent operator queues and its effect in our execution plan. If we look at the execution plan in Figure 6.1 we can notice that all of the joins that operate on the largest `Lineitem` relation, are receiving queries from (and pushing results to) two parent operators. The first is the parent join operator in the pipeline, and the second is the `Output` operator for the corresponding queries. The effect that this has on the execution of the queries is that when one of these join operators finishes executing a cycle, its parent `Output` operator will be faster to enqueue the next batch of queries, than its parent `Join` operator. And since we have 4 `Join` operators of this sort in the pipeline, this implies that the last one, which is the `Lineitem` ⋈ `Partsupp` join operator, would have to wait for three cycles of the `Lineitem` storage engine operator, before it could start processing `Lineitem` tuples.

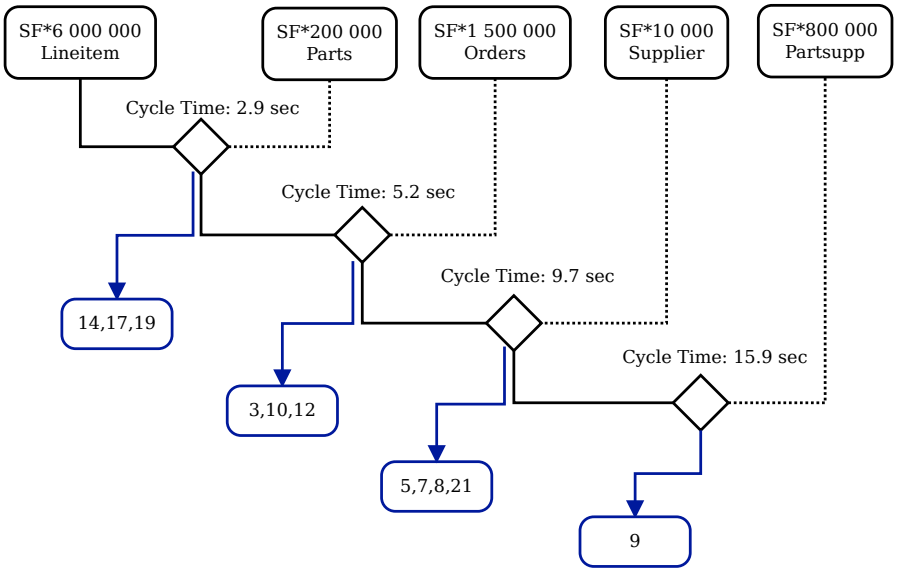


Figure 6.3: Batch Synchronization Problem

We show this effect in Figure 6.3 which shows the corresponding subgraph of the query execution plan, together with sample numbers for cycle times for the `Join` operators.

By combining all of these `Join` operators together we could eliminate this problem since they will receive queries from only one queue and will not enqueue queries to each other. Combining, of course could bring us the unwanted characteristics of the first extreme. That is if one of our join operators has a very expensive build part, then queries that do not use this join would be hurt. In this case the join operator for this expensive query could be separated from the other join operators.

One might argue that if we reach the point where performance is linear with the number of queries, reducing the number of queues in the system will not affect performance. This is true, however if we reduce the number of queues (independent operators), we would increase batch

sizes in the system, which can give us possibility to find more sharing opportunity and break the linearity.

Missed Possible Data Sharing Across Operators

As we have mentioned earlier, overqueuing is not the only disadvantage of having too many operators. If we were to combine the previously mentioned join operators into one, we could gain the advantage of sharing the read and materialization cost of the `Lineitem` probe records for all 4 joins, and remove the cost for pushing these records across the join operators. At the end, this might prove to be more beneficial than reducing the number of queues.

6.3.2 Underutilization of the System

This issue is related more to the technical aspect of execution in the operators. Currently, operators are designated to work at some specific cores when they are started. This type of execution creates a problem with our pipelined style of execution. This requires that before we start the system we assign certain cores to operators, or have a mechanism which adds or removes cores to operators. It is usually very difficult to estimate the amount of workload per operator before the system starts. Although this can be solved by looking at executions from previous batches and adding/removing cores accordingly, the actual problem is that the workload varies during execution of a single batch. This variation affects different operators, but for instance, the join operator's workload is different during building and probing, the group aggregator's and sort operator's workload is different before and after all tuples have been received, and so on. If we were to change the current approach into just defining work packages per group of intermediate results, and assign them to specific workers we could gain two benefits:

1. Easier workload balancing among operators.
2. Enforce intermediate results being processed before they are written to main memory.

This approach again represents sort of a shift towards the first extreme described earlier, in terms of taking some of the independence of the operators.

Although trying to always enforce the second benefit mentioned above could hurt predictability, the scheduler could abandon this property for certain part of the pipeline if it slows down other operators.

This is very related to the idea of combining operators described above. A scheduler can combine certain operators together in order to optimize their performance and synchronize their batches, but can also split them up if some operator is too heavy on the pipeline and slows down the others.

Crescando's compatibility with SharedDB

Crescando was designed to be independently running on a machine. Before it is started, data is partitioned into predefined number of partitions and each partition is assigned a predefined core.

The problem comes when Crescando needs to be integrated as a storage engine in a database like SharedDB. Once started, performance of Crescando can not be affected since there is only one core per partition. The problem with this is that depending on the workload Crescando will either be the bottleneck or will run too fast and other operators will become bottleneck. Furthermore, even if only one of the partitions is slowed down for some particular reason, all partitions will suffer. Ideally Crescando would partition the data only for a predefined number of partitions, (i.e. number of NUMA nodes or number of machines) and would leave to the database to decide how many cores should be allocated to each partition.

6.3.3 Replication and Partitioning

Databases specialized for OLAP workloads which work on a cluster of machines usually have to partition and replicate relations. Large relations might not fit in RAM of one machine, but fit in RAM in several machines. This requires partitioning of large relations across machines and replication of small relations. A NUMA machine can be thought of as a cluster of machines as well. Bandwidth between NUMA nodes is limited and might even be less than bandwidth on the local NUMA node. While Crescando supports partitioning and our join is multithreaded, the current SharedDB execution engine does not connect parallel threads from separate operators. Hence join threads can not be coupled to a certain partition to avoid unnecessary bandwidth across NUMA nodes.

6.3.4 Existing Implementation Details

Currently Crescando calls `malloc` for creating every intermediate tuple and its QIDs. Furthermore it always projects complete tuples. Ideally it would either provide just pointers to the original tuples, or at least project only certain set of requested attributes.

6.4 Improvements

In the last section we analyzed some of drawbacks of our current execution engine and execution plan for the TPC-H workload and also propose new techniques to overcome them. Implementing all of the proposed solutions would require work beyond the scope of this thesis, however we will try to overcome some of the drawbacks by using the following techniques.

1. Simulate the first extreme by creating only one entry queue for queries from outside clients. This will not bring optimizations from combining operators, however will make the system behavior more predictable, in the sense that we can predict at each point in time, which operators of the system will be idle and which will be working in parallel. Thus, overlapping operators will no longer fight for resources as much.

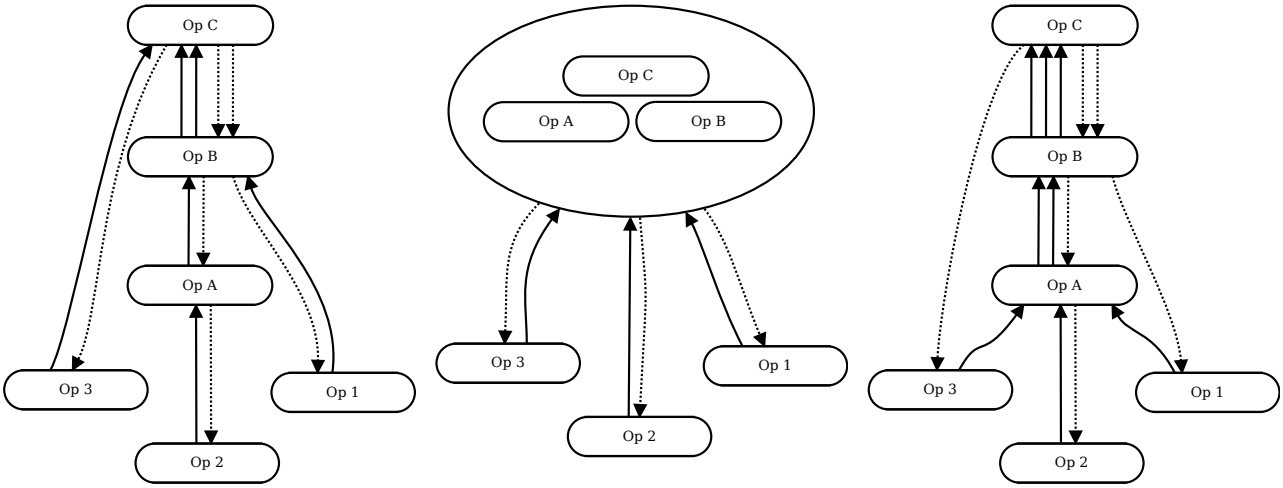
   As we have explained before this will equalize response time of all queries, which will be dictated by the response time of the most expensive query or set of queries. In our case,

it is not a big problem, partly because our queries are simple and do not require any other operation beyond joins of relations.

2. Simulate batch synchronization across operators. We are introducing a technique called shadow queries which will synchronize batches for certain groups of operators. Again, this will not bring us the optimizations of combining operators, but will bring us possible performance improvement by reducing number of queues and increasing batch sizes, as well as make the system more predictable in the same sense as described above.

6.4.1 Shadow Queries

In this section we will describe how we synchronize the batches for a group of operators while still have them operate independently of each other. We call this technique shadow queries and show the approach in Figure 6.4



(a) Initial Approach (b) Combined Operators (c) With Shadow Queries

Figure 6.4: Diagrams Showing Different Query Batching Methods. Full Lines : Query Flow, Dotted Lines : Intermediate Tuples Flow

Shadow queries are queries used internally in the system to synchronize batches for operators that work in a pipeline. This is achieved by having all queries that are meant to be enqueued to any of the operator of a certain pipeline of operators, to actually be enqueued to the first operator in the pipeline. They will be enqueued in a special form of queries - shadow queries. Shadow queries will not be processed by the operator that receives them but will only be enqueued to the next operator together with the other normal queries in the batch. The following are the properties and differences of shadow queries compared to normal queries in SharedDB:

- A shadow query must have a child query. The child query can be either a Shadow query or a regular query.

- A shadow query must have a parent query. The parent query can be either a Shadow query or a regular query.
- An operator that receives a shadow query should not process the query, but instead enqueue the query's child query to the corresponding operator.
- An operator that receives a regular query whose parent is a shadow query must process the query and push results to the operator who is responsible to the first query up the chain which is not a shadow query.

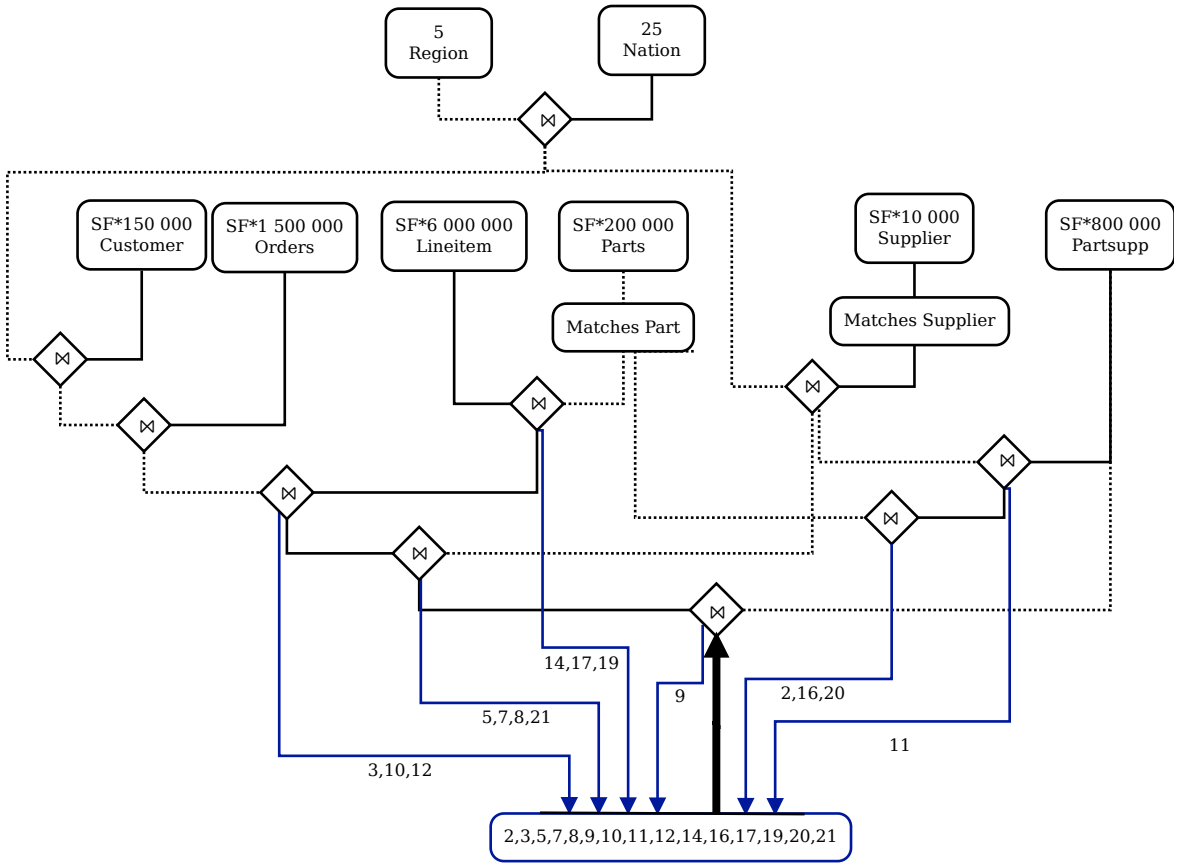


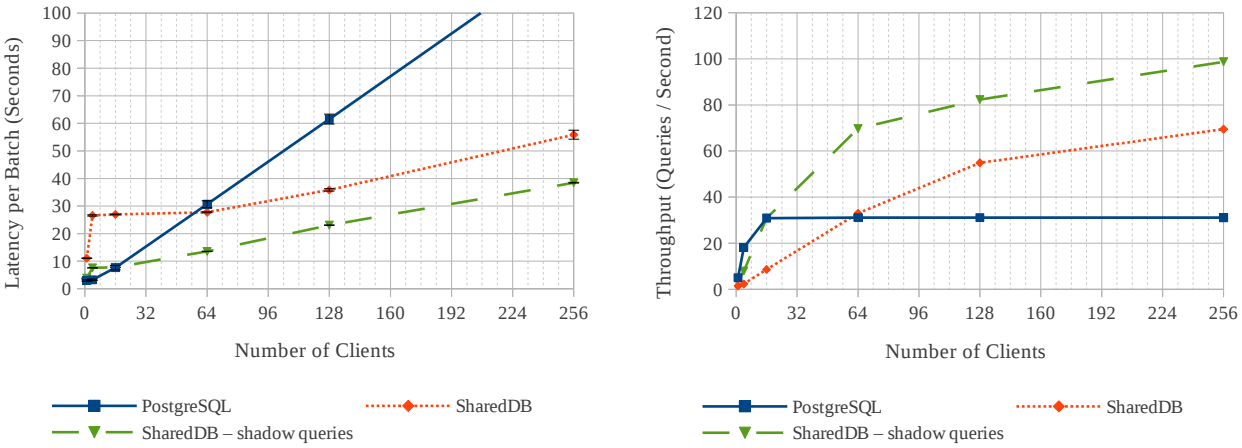
Figure 6.5: Execution Plan with Shadow Queries

In Figure 6.5 we see how does the query exeucution plan look after we applied both improvements described above.

6.4.2 Results

In Figure 6.6 we can see the improvements that shadow queries have made to SharedDB in this case. Biggest performance benefit can be seen when there are only few clients running and this is expected since minimizing the queueing and increasing batch sizes internally in the system is

most beneficial when they are actually small. Performance improvement is less when there are 256 clients, and mostly comes from the difference of the performance when there is 1 client.



(a) Response time

(b) Throughput

Figure 6.6: Latency and Throughput for the TPC-H Workload with Shadow Queries

Chapter 7

Conclusion

A major conclusion from this work is that a simple hash join without partitioning is a better match than the radix partitioning join for an environment like SharedDB. This is due to SharedDB introducing too much generality that hurt the subtle optimization techniques used in the radix partitioning join. Furthermore, we showed that the simple hash join can give performance comparable to the one of the radix partitioning join, even in the synthetic environment suitable for the latter algorithm. We can also conclude that software prefetching for hash joins, to hide main memory access latency, is effective and increases performance, but only when TLB misses are not a big problem.

Another important conclusion is that when handling of query IDs in shared execution needs to be optimized, batched query execution would be a key requirement. This is an additional argument for SharedDB's distinct approach of batching queries for every operator, against ad-hoc addition of queries proposed by SharedDB's closest competitors [1, 4].

Finally, we can also conclude, that although batching is beneficial, we need to be careful about the number of independent operators (queues) in the system. Too many queues increase waiting time of queries and decrease batch sizes. The other extreme of only one queue (one operator) will cause unpredictable performance, so we need to find a compromise between the two extremes.

7.1 Future Work

Possible topics for future work are:

- Since we have only done microbenchmarks for the proposed QID handling methods, we plan to implement them across all SharedDB operators, and test their impact on performance.
- The TPC-H benchmarks were executed unaware of the NUMA effects of the platform, so the next step would be to distribute and replicate operators to minimize inter-node communication.

- We ran benchmarks based on expensive join operations from the TPC-H benchmark to compare our shared join approach to the PostgreSQL' query-at-a-time approach. This is not a fair comparison, since we did not just measure the performance of the join operation, but also the performance of the storage engine. As a future work, we might create a synthetic environment with pre-generated intermediate results from the storage engine, where we would just run shared and individual joins on these results to compare performance of the two approaches.

- Shared query execution has separate impact on different relational operators. For the hash join, it additionally requires a computation of query ID set intersection for every joined tuple. For other operators, for instance aggregators or group aggregators, it requires separate state for each query. An important part of future work would be to investigate efficient ways to share query execution for all relational operators, as well as create a detailed performance analysis of the impact of shared query execution.

- As we have seen, too many independent operators hurt performance. An important future work would be to devise a mechanism for SharedDB where operators could be combined and split dynamically. Combining would be done to improve performance, while splitting would be done to ensure predictability of performance for certain queries.

Bibliography

[1] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez, *The DataPath system: a data-centric analytic processing engine for large data warehouses*, Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (New York, NY, USA), SIGMOD '10, ACM, 2010, pp. 519–530.

[2] C Balkesen, J Teubner, and G Alonso, *Efficient Main-Memory Hash Joins on Multi-Core CPUs : Does Hardware Still Matter ?*, (Under Submission).

[3] Spyros Blanas, Yinan Li, and Jignesh M. Patel, *Design and evaluation of main memory hash join algorithms for multi-core CPUs*, Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (New York, NY, USA), SIGMOD '11, ACM, 2011, pp. 37–48.

[4] George Candea, Neoklis Polyzotis, and Radek Vingralek, *A scalable, predictable join operator for highly concurrent data warehouses*, Proc. VLDB Endow. **2** (2009), no. 1, 277–288.

[5] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry, *Improving Hash Join Performance through Prefetching*, Proceedings of the 20th International Conference on Data Engineering (Washington, DC, USA), ICDE '04, IEEE Computer Society, 2004, pp. 116–.

[6] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey, *Efficient implementation of sorting on multi-core SIMD CPU architecture*, Proc. VLDB Endow. **1** (2008), no. 2, 1313–1324.

[7] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann, *SharedDB: killing one thousand queries with one stone*, Proc. VLDB Endow. **5** (2012), no. 6, 526–537.

[8] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki, *QPipe: a simultaneously pipelined relational query engine*, Proceedings of the 2005 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD '05, ACM, 2005, pp. 383–394.

[9] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey, *Sort vs. Hash revisited: fast*

join implementation on modern multi-core CPUs, Proc. VLDB Endow. **2** (2009), no. 2, 1378–1389.

[10] Stefan Manegold, Peter Boncz, and Martin Kersten, *Optimizing Main-Memory Join on Modern Hardware*, IEEE Trans. on Knowl. and Data Eng. **14** (2002), no. 4, 709–730.

[11] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten, *Cache-conscious radix-decluster projections*, Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, VLDB '04, VLDB Endowment, 2004, pp. 684–695.

[12] Timos K. Sellis, *Multiple-query optimization*, ACM Trans. Database Syst. **13** (1988), no. 1, 23–52.

[13] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton, *Cache Conscious Algorithms for Relational Query Processing*, Proceedings of the 20th International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '94, Morgan Kaufmann Publishers Inc., 1994, pp. 510–521.

[14] Transaction Processing Performance Council, *TPC-H Benchmark*.

[15] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, *Predictable performance for unpredictable workloads*, Proc. VLDB Endow. **2** (2009), no. 1, 706–717.

Appendix A

Modified TPC-H Queries

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;
```