MQJoin: Efficient Shared Execution of Main-Memory Joins

Darko Makreshanski[1] Georgios Giannikis[2]* Gustavo Alonso[1] Donald Kossmann[1,3]

[1]ETH Zurich
{darkoma,alonso,donaldk}@inf.ethz.ch

[2]Oracle Labs
georgios.giannikis@oracle.com

[3]Microsoft Research
donaldk@microsoft.com

ABSTRACT

Database architectures typically process queries one-at-a-time, executing concurrent queries in independent execution contexts. Often, such a design leads to unpredictable performance and poor scalability. One approach to circumvent the problem is to take advantage of sharing opportunities across concurrently running queries. In this paper we propose Many-Query Join (MQJoin), a novel method for sharing the execution of a join that can efficiently deal with hundreds of concurrent queries. This is achieved by minimizing redundant work and making efficient use of main-memory bandwidth and multi-core architectures. Compared to existing proposals, MQJoin is able to efficiently handle larger workloads regardless of the schema by exploiting more sharing opportunities. We also compared MQJoin to two commercial main-memory column-store databases. For a TPC-H based workload, we show that MQJoin provides 2-5x higher throughput with significantly more stable response times.

1. INTRODUCTION

In recent years, increased connectivity and availability of information have changed the requirements for databases. Systems catering to large user bases must provide robust performance with strong guarantees. This, together with the trend toward real-time data analytics, has put a strain on database architectures. Under these circumstances, systems must be designed to provide guaranteed response times for complete workloads, rather than the fastest performance for individual queries. For instance, reservation systems used in the airline industry need to execute hundreds of decision support queries per second with tight latency guarantees while sustaining high update rates [27].

An emerging approach to deal with such requirements is to exploit the sharing opportunities available in these workloads. Various techniques for sharing query execution have been explored to date, ranging from exploiting common subexpressions in multi-query optimization [25], simultaneous pipelining in QPipe [15]; sharing of scans in MonetDB [28], Blink [24, 23], and Crescando [27]; to sharing global query plans in CJoin [9], Datapath [3], and SharedDB [12].

*Work done while at ETH Zurich

Proceedings of the VLDB Endowment, Vol. 9, No. 6
Copyright 2016 VLDB Endowment 2150-8097/16/02.

As one of the most expensive relational operations, efficient join processing is crucial for performance. Exploiting sharing opportunities in joins across multiple queries is important to sustain throughput in highly concurrent workloads.

In this paper we present MQJoin, a method for sharing join execution that is able to efficiently exploit sharing opportunities and provide high performance for up to hundreds of concurrent join queries. Similarly to CJoin [9], Datapath [3] and SharedDB [12], MQJoin shares query execution by annotating intermediate results with additional information. What differentiates our approach is the use of several techniques that enables a significantly higher degree of sharing and an efficient use of main-memory bandwidth and CPU resources. This allows MQJoin to outperform state-of-the-art commercial analytical main-memory databases for workloads with high concurrency.

To evaluate MQJoin, we first present a series of microbenchmarks to illustrate the benefits and overhead of the approach with respect to a query-at-a-time counterpart. We analyze how much overlap should intermediate relations of queries have so that sharing pays off. Using an existing shared scan implementation as a storage engine, we then compare MQJoin integrated into a complete system to commercial databases and related work. Performance-wise, we compare to two leading main-memory analytical databases, namely Vectorwise and another popular commercial database which we refer to as System X, on a TPC-H based workload. We show that our system outperforms its commercial counterparts in terms of throughput when the load grows beyond 60 clients. Furthermore, it provides significantly more stable and predictable response times, having a lower 99th percentile even for a handful of clients. In terms of scalability we also compare to CJoin, the closest approach to ours, and show that for the Star Schema Benchmark[21] for which CJoin was designed, MQJoin is able to provide up to an order of magnitude more throughput while maintaining lower response times.

The main contributions of the paper are: 1) we present a method for sharing joins for highly concurrent workloads that supports one order of magnitude more concurrent queries than the best published result to date; 2) we provide an analysis of the impact of sharing on main-memory joins showing how to adapt existing join algorithms to support sharing; and 3) we validate the potential of the idea through a comparison of a shared scan/join system to leading main-memory analytical databases demonstrating 2-5x higher performance.

The rest of the paper is organized as follows: Section 2 discusses related work on join algorithms and shared query execution systems; Section 3 gives a model of the shared join execution approach that we use; Section 4 explains the two-way join algorithm in detail; Section 5 explains how multi-way joins are handled; Section 6 explains the system architecture, including integration with shared scans; Section 7 provides extensive analysis on the effects of sharing and the performance of MQJoin; Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 Main-memory Join Execution

The performance of a join is very important in a relational database. Due to the availability of systems with large main memories, recent research has focused on optimizing in-memory joins. Shatdal et al. [26] proposed partitioning the relations so that they fit in cache to avoid high random access latencies. Manegold et al. [20] partition the relations in two steps to avoid expensive TLB misses during partitioning. Chen et. al [10, 11] propose software prefetching techniques to hide the memory latencies of random accesses. More recently, there has also been discussions on whether sort-merge join or hash-based join are better suited for modern architectures [5, 17], as well as whether it is worth to tune to the underlying architecture [6, 7]. There is also a line of work that optimizes main-memory joins for NUMA architectures [2]. We have carefully evaluated all these results design a join that is as efficient as possible but supports shared execution.

2.2 Shared Query Execution

Several techniques for shared query execution have been developed to date. Sharing execution was initially proposed in the form of multi-query optimization [25]. MQO detects and jointly executes common subexpressions in multiple queries including execution of join operations. StagedDB [14] and QPipe [15] use a simultaneous pipelining technique to share execution of queries that arrive within a certain timeframe. Using a system based on these techniques, Johnson et al. [16] show that there is a trade-off between sharing and parallelism. A limitation in these systems is that they rely on temporal overlap for sharing. Typical results show sharing for a few tens of queries [15].

Sharing data and work for scans has been shown to be effective in various forms and use cases. MonetDB [28] optimizes disk bandwidth utilization by doing cooperative scans where queries are dynamically scheduled according to their data requests and the current status of the disk buffers. Similarly, systems like IBM UDB [18, 19] perform dynamic scan grouping and ungrouping as well as adaptive throttling of scan speeds to increase buffer locality. Blink [24, 23], and Crescando [27] go one step further and answer multiple queries in one table scan, independently of the query predicates, thereby sharing disk bandwidth and memory bandwidth. In those systems, the degree of sharing is between a few hundred to several thousand concurrent queries.

Recently, several systems propose shared execution of complex operations such as joins, for queries without common subexpressions. CJoin [9] achieves high scalability, handling up to 256 concurrent queries, by using a single always-on plan of operators that executes all queries. The approach is tailored to star schemas. Datapath [3] makes the case for a data-centric approach to analytical databases, advocating a push-based model to query processing instead of the traditional pull-based. They work with a more general TPC-H schema and show sharing for up to 7 concurrent queries. A push-based, data-flow model for query processing was also used in the Eddies project [4]. While Eddies are similar to sharing, they were designed to provide runtime-adaptivity of query execution where a static query plan generation is not sufficient. They can not provide high throughput for concurrent workloads.

SharedDB [12, 13] shows that a shared query execution system based on a global query plan and batching can give robust performance for highly concurrent workloads of up to thousands of queries. SharedDB, however, uses single-threaded operators.

Finally, [22] integrates the approaches of CJoin and QPipe. This work shows that a combination of global query plans with shared operators and simultaneous pipelining is better suited for high concurrency, while traditional query execution with simultaneous pipelining is better suited for low concurrency workloads. Similar to CJoin, the authors also focus on star schema workloads.

3. SHARED JOIN MODEL

This section presents a model for the input and output characteristics of the shared join algorithm. The algorithm itself is described in Section 4. For simplicity, this model represents only sharing of two-way inner-joins. Handling of other join types is described in Section 4.7, while Section 5 covers multi-way joins. Before defining a shared join, we will describe a join across two relations. We then formally define a shared scan and then define a shared join as the join between two shared scans.

Let R and S be two relations, and $t_R \in R$ and $t_S \in S$ be tuples of the corresponding relations. A scan and select operation on the relation R is then defined as a function $\sigma^R : R \to \{\top, \bot\}$, and the output of this scan is noted as σ^R for brevity. A join on selections σ^R, σ^S of the two relations is then defined as:

Definition 1: Join

$$\begin{aligned}\sigma^R \bowtie \sigma^S \quad = \{ \quad & (t_R, t_S) \mid \\ & \sigma^R(t_R) \wedge \sigma^S(t_S) \wedge f_{\bowtie}(t_R, t_S) \ \} \qquad \square\end{aligned}$$

Where $f_{\bowtie} : R \times S \to \{\top, \bot\}$ is the join predicate function and (t_R, t_S) is a concatenation of the attributes t_R and t_S.

A shared join for a set of queries $Q = \{q_1, q_2, \ldots q_n\}$, where $q_i = \sigma_i^R \bowtie \sigma_i^S$ for $i \in \{1, 2, \ldots n\}$, is defined as the join between the result of the shared scans σ_Q^R, σ_Q^S. The result of a shared scan σ_Q^R can be defined as:

Definition 2: Shared Scan

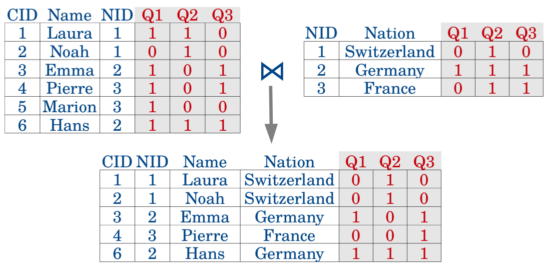
$$\begin{aligned}\sigma_Q^R \quad = \{ \quad & (t_R, (b_{q_1}^R, b_{q_2}^R, \ldots b_{q_n}^R)) \mid \\ & b_{q_i}^R = \top \iff \sigma_i^R(t_R) \wedge \\ & \exists i . b_{q_i}^R = \top \ \} \qquad \square\end{aligned}$$

Thus, a shared scan outputs intermediate relations with an extended schema that has one extra Boolean attribute $b_{q_i}^R$ for every query q_i. The attribute $b_{q_i}^R$ for a tuple t_R holds a value of true if and only if the query q_i is interested in that tuple, i.e. $\sigma_{q_i}^R(t_R) = \top$. Furthermore, a tuple t_R is output by the shared scan if at least one query is interested in t_R. The set of the attributes $b_{q_i}^R$ for all queries $q_i \in Q$ is denoted as b_Q^R and a set of values of these attributes for a particular tuple t_R is called the *set of query IDs* for t_R. Having the output of a shared scan defined, we define a shared join as the join of the output of two shared scans or:

Definition 3: Shared Join

$$\begin{aligned}\sigma_Q^R \bowtie \sigma_Q^S \quad = \{ \quad & (t_R, t_S, (b_{q_1}^{R \bowtie S}, b_{q_2}^{R \bowtie S}, \ldots b_{q_n}^{R \bowtie S})) \mid \\ & b_{q_i}^{R \bowtie S} = \top \iff \\ & (b_{q_i}^R = \top \wedge b_{q_i}^S = \top) \wedge \\ & \exists i . b_{q_i}^{R \bowtie S} = \top \wedge f_{\bowtie}(t_R, t_S) \} \qquad \square\end{aligned}$$

In other words, a shared join outputs a relation with extended schema that also contains one extra attribute $b_{q_i}^{R \bowtie S}$ for each query q_i. This attribute is the result of the conjunction of the corresponding attributes of the input relations: $b_{q_i}^R \wedge b_{q_i}^S$. Similarly



CID	Name	NID	Q1	Q2	Q3
1	Laura	1	1	1	0
2	Noah	1	0	1	0
3	Emma	2	1	0	1
4	Pierre	3	1	0	1
5	Marion	3	1	0	0
6	Hans	2	1	1	1

$\bowtie$

NID	Nation	Q1	Q2	Q3
1	Switzerland	0	1	0
2	Germany	1	1	1
3	France	0	1	1

CID	NID	Name	Nation	Q1	Q2	Q3
1	1	Laura	Switzerland	0	1	0
2	1	Noah	Switzerland	0	1	0
3	2	Emma	Germany	1	0	1
4	3	Pierre	France	0	0	1
6	2	Hans	Germany	1	1	1

Figure 1: Sample Shared Join On Attribute NID

to the shared scan, the shared join outputs only tuples for which at least one query is interested. One thing to note is that, for this inner-join based model, queries need to share a common join predicate function $f_{\bowtie}$ so that the join can be shared. For other join types, Section 4.7 shows examples of queries whose join can be shared even if they do not share any predicate.

An example for the input and output relations of a shared join is shown in Figure 1. Here we show a shared join for three queries: $Q1$, $Q2$, and $Q3$, on two relations Customers $(CID, Name, NID)$ and Nations $(NID, Nation)$ with each query having different predicates on each relation. The upper part of Figure 1 shows the two input relations of the join or, in other words, the output relations of the shared scan. As explained previously, intermediate relations have an additional Boolean attribute for each query, which has a value of 1 if the corresponding tuple belongs to the query or 0 otherwise. The set of query IDs in this case is the set of values of all Boolean attributes for a particular tuple. The bottom part of Figure 1 shows the output of the shared join, where the set of query IDs of an output tuple is simply an intersection of the sets of query IDs of the matching pair of input tuples.

4. TWO-WAY JOIN ALGORITHM

We faced two key challenges when designing MQJoin: minimize time spent per tuple and minimize the number of tuples processed for a set of queries. To address the first challenge we combine approaches from related work on optimizing joins with techniques to efficiently reuse data-structures over multiple join sessions and to minimize the overhead imposed by sharing, such as handling query IDs. To address the second challenge we use techniques that schedule queries in a way that minimizes redundant work and develop ways to share execution of queries that require different types of joins.

4.1 Algorithm Overview

From a high level perspective, the algorithm is a parallel hash join running on a single multi-core machine similar to those available in the literature [6, 7, 10]. During the build step, multiple threads consume the build relation to populate the hash table. In the next phase, the threads consume the probe relation and probe the hash table to find matches of tuples. Unlike a traditional hash join algorithm, the threads do an additional step of computing the intersection of the query ID sets of all matching pairs of tuples and filtering out tuples with an empty intersection.

The algorithm inherits several features from recent work on main-memory hash joins. Similarly to Blanas et al. [7], threads during the build step synchronize using spin-locks, where there is one lock per hash entry in the table. Similarly to Balkesen et al. [6] we optimize the algorithm by minimizing the number of random accesses per tuple. Although it was shown to be more effective [6], we do not partition the relations to cache sizes, mostly because we have a row-store based system and larger tuple sizes. In particular, due to the meta-data per tuple introduced by sharing, the partitioning step would be more expensive. Instead, we reduce the latency of random accesses by applying a grouped software prefetching as proposed by Chen et al. [10, 11]. One novelty in our approach is the introduction of a *sessionID* attribute to each hash entry to provide an efficient reset operation of the hash table.

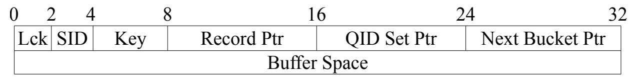


Figure 2: Structure of a Hash Bucket

4.2 Hash Table Structure

The hash table is structured in a way that each bucket is aligned at the $64B$ cacheline boundary, guaranteeing that each hash bucket lookup will access a single cacheline. The structure of a bucket is shown in Figure 2. A hash bucket consists of the following fields: **Lck: Lock** is used to synchronize between threads during building of the hash table; **SID: Session Id** is used to identify the last session when this hash bucket was updated. This is needed in order to reuse the memory of a hash table for multiple join cycles without the need of an expensive `memzero` operation; **Record Ptr** points to the address in memory where the record is located. This can be either in the buffer space of the hash bucket or somewhere else; **Query ID Set Ptr** points to the address in memory where the set of query IDs for the tuple are located; **Next Bucket Ptr** points to the next hash bucket in cases of overflow. Each thread has its own dedicated pool of overflow buckets; **Key: Join Key** is typically a 4 byte integer cached in the hash bucket for quick access in case the record is stored somewhere outside; **Buffer space** is the extra memory located on the cacheline that is used to store the record and/or the query ID set in cases when they are small enough.

Algorithm 1 Build Phase

```
for $group \in relation$ do
       for $tuple \in group$ do
S1 {       $bucket \leftarrow$ COMPUTEBUCKETADDRESS($tuple$)
           PREFETCH(bucket)
       end for
       for $tuple \in group$ do
           LOCKBUCKET
           if $bucket.SID \; != currentSID$ then
               POPULATEBUCKET($tuple, bucket$)
               $bucket.SID = currentSID$
           else
S2 {           $ofbucket \leftarrow$ GETOVERFLOWBUCKET
               SWAPNEXTBUCKETPTRS($bucket, ofbucket$)
               POPULATEBUCKET($tuple, ofbucket$)
           end if
           UNLOCKBUCKET
       end for
end for
```

4.3 Join Procedure

Next we explain the build and probe phases of the hash join algorithm in more detail. For clarification purposes we also provide an example shown in Figure 3 and work through the example as we explain the algorithms. The figure shows a simple setup with a building relation of 5 tuples, probing relation with 10 tuples and the populated hash table.

Algorithm 2 Probe Phase

```
  for $group \in relation$ do
S1 { for $tuple \in group$ do
        $bucket \leftarrow$ COMPUTEBUCKETADDRESS($tuple$)
        PREFETCH(bucket)
     end for
     while $group$ != [ ] do
        $otherGroup \leftarrow$ [ ]
        $keyMatchGroup \leftarrow$ [ ]
S2 {    for $tuple \in group$ do
           if $bucket.SID == currentSID$ then
              if $bucket.key == tuple.key$ then
                 ADDTOKEYMATCHGROUP($tuple$)
                 PREFETCH(bucket.queryIDs)
                 PREFETCH(bucket.record)
              end if
              if HASNEXTBUCKET($bucket$) then
                 ADDTOOTHERGROUP($tuple$)
                 PREFETCH($bucket.nextBucket$)
              end if
           end if
        end for
S3 {    for $tuple \in keyMatchGroup$ do
           $resQIDs \leftarrow$ INTERSECT($tuple.QIDs, bucket.QIDs$)
           if !EMPTYSET($resQIDs$) then
              OUTPUTTUPLE($tuple, bucket, resQIDs$)
           end if
        end for
        SWAP($group, otherGroup$)
     end while
  end for
```

Both the build and probe algorithms divide their input intermediate relations in small groups and iterate over these groups multiple times before proceeding to the next group. The reason for this is to avoid high memory access latencies to memory locations not present in the cache. In each iteration, addresses to non-cache-resident memory to be accessed in the next iteration are calculated and prefetch instructions are issued. We experimented with different group sizes and found that groups of about 50 tuples are enough to hide random main memory access latencies. For simplicity, in the example in Figure 3 we use group sizes of 5 tuples, so there is only one group for the building relation and two groups for the probing relation.

The **build phase** (Algorithm 1) consists of two iterations over the small groups of tuples. In the first iteration (S1), the hashes of the join keys are precomputed and prefetch statements to the addresses of the corresponding hash buckets are issued. In the second iteration (S2), the hash buckets are populated with the input tuples. If a hash bucket is already populated, the input tuple is populated in an overflow bucket and the pointer of the hash bucket is updated to point to the overflow bucket.

For the example join, in the first step key hashes of tuples BT{1,2,3,4,5} are computed and the prefetch instructions are issued on the hash buckets HB{1,4,2,1,4} in the corresponding order. In the second step, building tuples BT{1,2,3} are populated in hash buckets HB{1,2,4} respectively, while tuples BT{4,5} are populated in new overflow tuples OB{1,2} that are linked to hash buckets HB{1,4}. In this example, the buffer space in the buckets is enough to hold the QIDs bitset, but not enough to store the record as well. Therefore, only the bitsets are copied into the buffer space of buckets and QID pointers are set to point to the buffer space, while the record pointers are set to point to the original memory of the input relation.

The **probe phase** (Algorithm 2) is a bit more involved. Step 1 (S1) computes hashes and prefetches the hash buckets. Step 2 (S2)

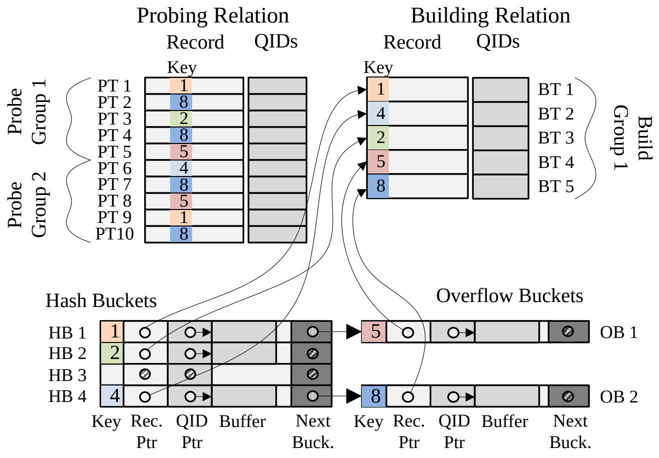


Figure 3: Two-way Join Algorithm Example

evaluates join predicates and prefetches the set of query IDs and the records. If the set of query IDs and the record span multiple cachelines, it issues a prefetch statement for each cacheline. In case of overflow, it also issues a prefetch statement to the overflow buckets. Step 3 (S3) computes the set intersection of the query ID sets and materializes the output tuple if the set intersection is not empty. In cases of overflows, the procedure from step 2 to step 3 is repeated as many times as the length of the longest bucket chain.

In the example in Figure 3, the probing procedure for the first probing tuple group will be the following. In the first iteration (S1), key hashes of probing tuples PT{1,2,3,4,5} are computed and prefetch instructions are issued on the memory addresses of hash buckets HB{1,4,2,4,1} in the corresponding order. In the next iteration (S2), key comparison of corresponding tuples and hash buckets are performed and the record and QID pointers are prefetched for the buckets with matching key comparison (HB{1,2}). In the same iteration (S2), prefetch instructions are also issued for the overflow buckets OB{1,2}. In the next iteration (S3), QID set intersection is performed and output tuples are materialized for tuple pairs: {(BT1,PT1),(BT3,PT3)}. Since there overflow buckets, steps S2 and S3 are repeated for the overflow buckets. In the next iteration (S2) key comparison is performed for tuple-bucket pairs: {(OB2,PT2),(OB2,PT4), (OB1,PT5)}, and since all keys match, record and QID pointers are prefetched for both overflow buckets OB{1,2}. No overflow buckets are prefetched in this S2 iteration since all next bucket pointers are null. Finally, the algorithm proceeds with the last iteration of the first probing group of tuples (S3) with performing set intersection and output tuple materialization of tuple pairs: {(BT5,PT2),(BT5,PT4), (BT4,PT5)}. As the algorithm processed all overflow buckets, it exits the while loop and continues on with the next group of probing tuples in a similar manner.

4.4 Query Scheduling

Our model for sharing joins assumes a fixed set of queries during the join operation. To satisfy this property, the join runs in cycles for batches of queries where arriving queries wait in a queue if the system is busy executing a join, similarly to SharedDB. One might argue that this waiting causes response times to increase. The waiting, however, is more than compensated by allowing the system to organize the join execution in a way that both the build and probe phases are shared for all queries currently being executed. The result is, as we will show, higher throughput and more predictable performance.

The other alternative is to schedule queries immediately as they arrive. This is common for systems that share scans such as IBM DB2 [18, 19], MonetDB [28], and is also used by CJoin [9] and Datapath [3]. The effect on join execution is that the build and probe phases need to be executed concurrently in a pipeline fashion. The problem is in the redundant work to be done for a set of concurrently running queries, which is the first stage of the pipeline, i.e. the build phase. The effect on performance depends on the relative cost between the build and probe phases and the amount of sharing missed. The effect is further aggravated in cases of multi-way joins where there are multiple stages in the pipeline.

4.5 Query ID Set Representation

As mentioned previously, shared query execution introduces an additional attribute for each intermediate tuple in the system. This attribute keeps information on which queries are interested in each tuple. There are several ways to store and handle this attribute, each with advantages and disadvantages. One way to represent this attribute is as an array of integers each of which represents the ID of the query that is interested in the tuple. The impact of this is that the size of the attribute is $N_i \cdot s_{int}$ bytes where N_i is the number of queries interested in the tuple and s_{int} is the size of the integer.

Another way of representing the set of query IDs is to use a bitset where each query in the system has a dedicated bit position in the bitset. For a certain tuple with a bitset B, and a query Q whose bit position is i if the i^{th} bit in B is 1 then the query is interested in the tuple, and vice-versa if the bit is 0. The size of the attribute is then the size of the bitset which is $\frac{N_s}{8}$ bytes where N_s is the total number of concurrently running queries.

One factor affecting the performance trade-off between the two methods is the ratio $\frac{Avg(N_i)}{N_s}$ of average number of queries per tuple $Avg(N_i)$ to the number of concurrent queries N_s. Bitsets work well when this ratio is high or N_s is sufficiently low. Arrays work well in cases where this ratio is low and N_s is very high. In practice, we found that, for analytical workloads, bitsets perform better.

4.6 Discussion

MQJoin is similar to CJoin and Datapath in using bitsets to handle the query ID set for each tuple, as well as the use of a hash based algorithm.

An important difference to CJoin and Datapath is that in those systems the build and probe phases are executed concurrently. As a result, the hash table is constantly updated for all incoming and outgoing queries, which introduces extra work per query. In CJoin, for instance, the hash table is updated for each query individually, both when the query enters and exits the system. This works for star schemas and low concurrency cases where the effort required in the build phase is significantly lower than during the probe phase. In Datapath, colliding hash table entries from newly arrived queries will be placed in the next available entry in the hash table. This causes extra work to be done during probing and it is unclear how the hash table is purged when queries finish. To minimize the work required for building, updating and clearing the hash table, our join algorithm runs in cycles, performing build and probe one after the other in each cycle. This allows us to share the build operation for all queries that are being executed in the current cycle. At the end of each cycle we clear all data in the hash table by simply incrementing the session ID number. This avoids replaying build subqueries to clear data from the hash table.

Another difference is in the micro-architectural properties of the algorithm. SharedDB uses single-threaded join operators which is inefficient for analytical workloads on multi-core systems. Similarly, CJoin builds and updates the hash table in a single thread, which is inefficient if the build relations are not insignificantly small. This is another reason why CJoin is restricted to star schemas. Datapath uses a single hash table for all joins which is divided in 64 regions with exclusive locks. To avoid contention on these locks, it needs to update the hash table in two phases. Our algorithm parallelizes both build and probe phases and synchronizes build operations on a per bucket basis. We optimize for both CPU and bandwidth efficiency by hiding random access latencies through software prefetching and minimizing the number of random cachelines accessed per tuple. This makes the performance of our algorithm comparable to that of the fastest published query-at-a-time join algorithms, something that none of the competing versions can do.

```
1. SELECT * FROM R, S WHERE R.A = S.A
2. SELECT * FROM R LEFT OUTER JOIN S ON R.A = S.A
3. SELECT * FROM R RIGHT OUTER JOIN S ON R.A = S.A
4. SELECT * FROM R FULL OUTER JOIN S ON R.A = S.A
5. SELECT * FROM R WHERE R.A IN (SELECT S.A FROM S)
6. SELECT * FROM R WHERE R.A NOT IN (SELECT S.A FROM S)
7. SELECT * FROM S WHERE S.A IN (SELECT R.A FROM R)
8. SELECT * FROM S WHERE S.A NOT IN (SELECT R.A FROM R)
```

Figure 4: List of Queries Whose Join Can be Shared

4.7 Sharing Execution for Other Join Types

The algorithm we just described works for queries that require inner joins. It can be extended to share execution for queries that also require (left, right, full) outer-joins and (anti) semi-joins, provided that they are all on the same equality predicate. Consider a simple schema of relations $R(A\ int, B\ int)$ and $S(A\ int, C\ int)$, and a shared join operator which builds the hash table with the relation S and probes with the relation R. Then consider the set of queries shown in Figure 4. To handle these queries, the algorithm is extended to perform additional operations on the bitsets during probing. These additional operations depend on the join type and can be setting bits of individual queries to 1, conditional set to zero, or bitwise OR. This extension allows to answer queries 2, 4, 5, 6 and 7. Another extension is to modify tuples' bitsets in the hash table during probing, and iterate over the build relation again after probing to output tuples which did not have a match in the probing relation. This extension allows to answer queries 3, 4 and 8.

5. MULTI-WAY JOINS

In this section we describe how multiple join operations are handled. As with the query-at-a-time approach, the shared join approach also requires an optimization decision on how to create query plans involving multiple joins. The shared join optimizer not only needs to decide the order of a multi-way join but also which queries' execution should be shared. It can be undesirable to share the execution of some queries either for performance isolation reasons or if sharing would hurt overall performance.

Building such an optimizer is actually a non-trivial task and is out-of-scope of this paper. Recent work by Giannikis et al. [13] has addressed this problem and these techniques can also be applied to our approach. In this paper we focus on one end of the spectrum, that is to maximize sharing across all concurrent queries. While this might not be always optimal or desirable, it provides a lower bound for the performance of our approach. The ordering of the joins is currently done by hand. Next we explain how we maximize sharing for queries with multi-way joins. The key challenge to address is handling queries without common subplans. In this regard, we use two techniques: query plan equalization and global query batching.

Relation Orders:

OID	CID	Date
1	2	2012.JUL.01
2	3	2013.JAN.01
3	1	2011.NOV.31
4	3	2013.FEB.01
5	2	2012.NOV.15
5	2	2012.DEC.14

Relation Customers:

CID	Name	NID
1	John	1
2	Maria	1
3	Dieter	2

Relation Nations:

NID	Name
1	USA
2	Germany

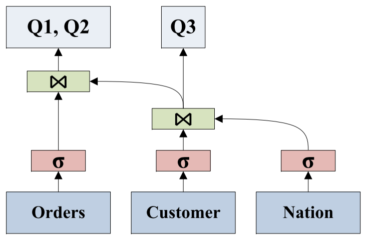
(a) Sample Database

Q1	SELECT * FROM Orders O, Customers C
	WHERE O.CID = C.CID AND C.Name = 'John'

Q2	SELECT * FROM Orders O, Customers C, Nations N
	WHERE O.CID = C.CID AND C.NID = N.NID
	AND O.Date ¿ 2012.DEC.21 AND N.Name = 'USA'

Q3	SELECT * FROM Customers C, Nations N
	WHERE C.CID = N.NID AND N.Name = 'Germany'

(b) Sample Queries



(c) Shared Join Plan for Q1, Q2, Q3

Figure 5: Sample Database with Queries

5.1 Query Plan Equalization

A key property of our shared join approach that minimizes the number of tuples processed in the join operators is that a join operation is shared even for queries without common subplans. We illustrate this with an example. Figure 5 shows a sample database with three relations: Orders, Customers and Nations together with 3 queries with various joins and predicates on the three relations. The three queries share almost no common subexpressions: Q1 asks for all the orders from John, Q2 asks for all orders after 2012.DEC.21 from USA, and Q3 asks for all customers from Germany. The query execution plan that we create to execute all 3 queries together is shown in Figure 5c.

One can notice that the organization of join operations in the plan does not directly correspond to the join operations required by all queries. In particular, Query 1 does not require the join of Customers $\bowtie$ Nations, however it is still included in its plan. The reason is that it allows for the Orders $\bowtie$ Customers join to be shared for Query 1 and 2.

We considered the two following techniques to exploit this type of sharing opportunity: (i) modify the query plan of Query 1 to include an unnecessary full table scan of Nations followed by a join with Customers (ii) have the Customer $\bowtie$ Nations join always forward tuples for which Query 1 is interested. Both methods have advantages and disadvantages, and could be used interchangeably depending on the situation. The advantage of the first method is that it requires less computation per tuple, however it might process more tuples. This would not be a problem, if for instance a full table join is already required by the union of all joins that are processed. The advantage of the second method is that it might process less tuples, but would require more computation per tuple. Currently we modify the query plans to include extra full table scans and joins. This technique can be referred to as query plan equalization where query plans are modified to increase sharing opportunities. The approach differs the ones in Datapath and SharedDB where, for join queries with different subplans where the join operations are either replicated or process intermediate relations from separate query plans, thereby creating redundant work.

5.2 Global Query Batching

As explained earlier, to exploit more sharing opportunities, our algorithm requires a fixed set of queries during its execution. This means that queries or subqueries need to be coscheduled (or batched), so that they are executed together. To facilitate this, we use query queues where there is one queue for each class of queries or subqueries that need to be coscheduled. If the system is busy executing a certain class of queries, arriving queries of that class will wait in the queue. The decision of how many queues to have involves making a trade-off between the amount of sharing exploited and isolation of query performance. To show the effects of sharing compared to query-at-a-time execution, we focus on one end of this spectrum which maximizes sharing. Therefore, we use a single query queue which gives the system maximum flexibility in how to schedule execution of concurrent queries. We refer to the use of a single query queue as global query batching. The effect of this can also be shown with the example in Figure 5. Assume that there are many clients connected to the system where each client repeatedly executes one of the three queries at random. Although the instances of Query 3 do not require the Orders$\bowtie$Customers join like Query 1 and 2, all queries will be coscheduled together. This means that query execution can be organized in a way that the Customers$\bowtie$Nations join is executed only once for all queries that are being executed at the same time, minimizing redundant work.

6. SYSTEM ARCHITECTURE

For an efficient end-to-end query execution, a shared join must run on top of a storage engine that supports shared scans. Sharing computation and bandwidth in scans is a common technique used in many systems. Some examples include: Blink [23], MonetDB [28] and Crescando [12]. To provide a clearer picture of the performance of MQJoin when it runs as part of a complete system, we integrated it with Crescando. Crescando is a row-oriented storage engine where relations are partitioned across cores and fully reside in main-memory. Therefore, neither the scan nor the join need to fetch data from disk to execute queries.

An example of how this integration works is shown in Figure 6. This example depicts two shared scan operations, one join operator divided into build and probe parts, and an output operator. The output operator is shown just to illustrate an operator on top of the shared join. It can be the build or probe part of an additional join operator, an aggregate operator, or the end point of query processing that communicates with the clients.

Query execution in our architecture is performed in one or more steps. In each step, a set of threads work on separate partitions, and execution progresses to the next step only when all threads are done processing their partition. In the example in Figure 6 there are two steps, where in step 1, n threads scan the R relation and build a hash table, and in step 2 they scan the S relation, probe the hash table, and output the results.

For optimizing data and instruction cache locality, tuples are processed in vectors similarly to MonetDB/X100 [8]. Within a

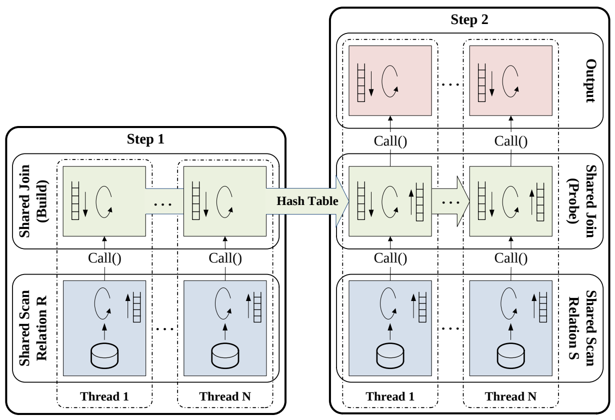


Figure 6: Integration with Shared Scans

step, threads switch contexts between operations when they fill up a buffer of 1000 tuples. For instance in step 1, thread 1 will scan its partition of relation R until it fills up the buffer with 1000 tuples. Consequently, it will push the buffer of tuples to the operator on top, which in this case is a function call to the build operator.

6.1 Interpreting the Results of the Shared Join

The shared join outputs intermediate results in a similar way as it gets the input, by including a bitset to every tuple representing the query ID sets. These intermediate relations can be used by any other operator or can be sent directly to the clients. In the absence of mechanisms to support the execution of more complex queries, we only add a simple aggregate operator on top of the join operator, which can evaluate arbitrarily complex expressions for each query individually. This aggregate operator iterates over the bits in the bitset for each query, evaluates the expressions of the corresponding queries and updates their states. This way we avoid sending large materialized relations over the network.

6.2 Tuple Format

Since we built MQJoin on top of a row-oriented storage engine, the system uses a row-wise format (NSM) throughout the whole query execution process. Nevertheless, MQJoin could also be integrated with column-oriented storage engines that use a column-wise format (DSM) to store relations. In this case we would need to employ an on-the-fly conversion between DSM and NSM to be able to benefit from both formats. Zukowski et al. [29] showed that this kind of conversion can be efficiently implemented and that it enables significant performance improvement for in-memory analytical query processing.

6.3 Main-memory Footprint

One may argue that sharing the execution of multiple join operations increases the main-memory requirements of a database. The reason behind is based on a traditional trade-off between the number of concurrent queries and the available memory. While our shared join does take more memory than a single join, we run many queries through it and exploit the sharing opportunities that arise, thereby reducing the overall demand for memory when considering how many queries are being executed concurrently.

6.4 Scaling Out

As main-memory gets cheaper and larger, an increasing number of datasets can fit in the memory of a single machine. For this reason, our algorithm assumes that relations and intermediate data structures are memory resident. Nevertheless, although our system is designed for single-node join processing, the techniques we use can also be applied in a distributed setting. As a memory intensive operation, network bandwidth is a limiting factor when running a join across multiple machines. Therefore, our approach of sharing the bandwidth for multiple queries will also be beneficial in this case. Similar rationale can also be applied to disk-based joins that spill intermediate relations to disk.

7. EVALUATION

To evaluate the performance of MQJoin we first run a series of microbenchmarks where we investigate the micro-architectural effects of sharing the join. We then evaluate MQJoin running on top of a shared scan with a TPC-H based scan and join workload and compare the performance to main-memory, column-store databases optimized for analytical workloads such as TPC-H, namely Vectorwise [30] and System X. We also compare our approach with CJoin using the star schema benchmark and based on the code provided by the authors of CJoin. All experiments were done on a machine with 4 $\times$ twelve-core AMD Opteron 6174 “Magny-cours” processors clocked at 2,200 MHz. The machine has 8 NUMA nodes each with 16 GB of memory, for a total of 128 GB of RAM.

7.1 Microbenchmarks

The purpose of these microbenchmarks is to show how sharing affects the performance of a join between two relations. We evaluate performance and compare it to a query-at-a-time join for various relevant factors, such as number of concurrent queries, hash table size, tuple size, and workload type. All experiments refer to an equi-join between relations $R(int\ A, int\ B)$ and $S(int\ A, int\ C)$. Unless otherwise noted both relations have 100 million tuples, each of which is 8 bytes, where the first 4 bytes contain the join key. The join key ranges from 1 to 100 million and is randomly distributed across tuples in both relations. The relationship between R and S is a primary-key foreign-key relationship, thus the join key in R is always unique. There is no skew in the workload, so keys in S are evenly distributed. The reason this workload is chosen is that it resembles workloads used to evaluate query-at-a-time join algorithms in related work on joins [6, 7, 17], giving us a fair base for comparison.

7.1.1 Scaling with the Number of Concurrent Queries and Record Size

In the first experiment (Figure 7) we measure absolute performance of our join algorithm in isolation and investigate how performance is affected by the size of the bitset, i.e., the number of concurrently running queries, as well as, the record size. We measure the time it takes to execute all join queries while varying the number of queries. To keep the number of tuples constant and to minimize the effect of the scan, all queries ask for a full table join of the two relations. We show performance for two cases, one where the join runs on 48 cores and one where it runs on a single core. This indicates how the algorithm performs both in memory-bound and compute-bound scenarios.

The first thing to note is the absolute performance of the algorithm compared to state-of-the-art join algorithms. From the 48-thread graph we see that the maximum performance obtained for small-sized tuples and few queries is a little bit less than 0.5 seconds, which for our 100M $\times$ 100M join corresponds to around 200 million tuples per second. This is in the same ballpark with highest performing joins that is around several hundred million tuples per second [5]. Next, we analyze the performance effect of

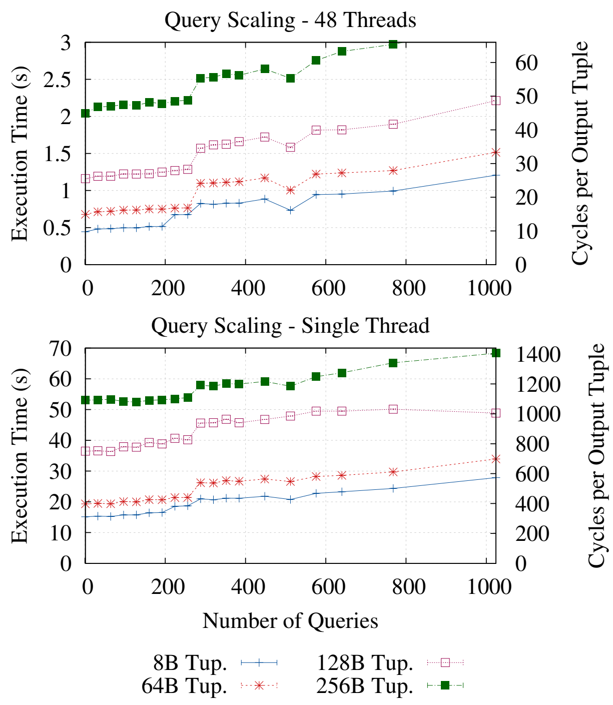


Figure 7: Performance of a 100M×100M MQJoin for Various Number of Queries and Record Size

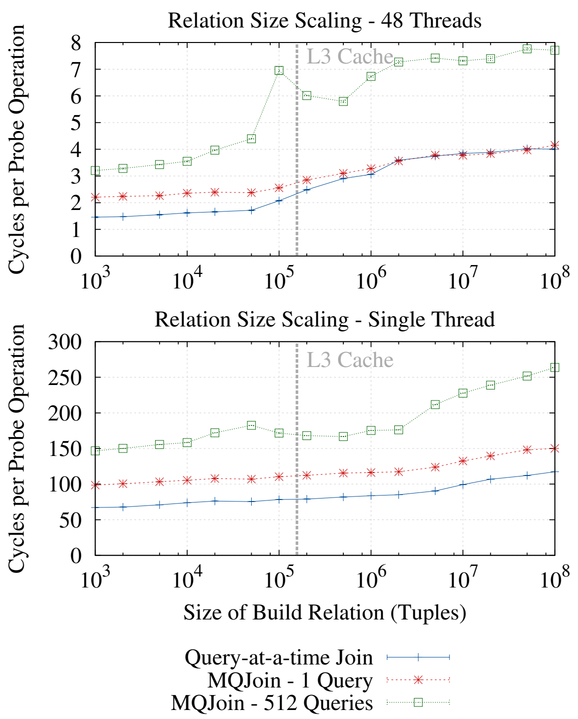


Figure 8: Probe Performance versus Build Relation Size

bitset and record size. Dealing with bitsets is an important source of overhead as it is not present in query-at-at-time join algorithms. We note that the difference in performance of sharing join execution for around 500 concurrent queries instead of 1 is in general small and at most a factor of 2, the main reason being that 500 bits can still be accommodated in a single cacheline of 64 bytes.

The overhead of dealing with larger record sizes is also important, since queries might be interested in different attributes requiring for larger records to be projected and processed by the join operators. Similarly to the bitset size, the results show that increasing the record sizes from 8 bytes to a cacheline size of 64 bytes has a marginal overhead. Enlargening the records to sizes bigger than a cacheline of up to 256 bytes, however, adds a significant overhead and performance starts degrading linearly with the record size as more non-cache resident memory has to be accessed per probe operation. To avoid this type of overhead several techniques can be used. One way is to employ standard techniques used in current databases to avoid processing large records in the join such as data compression and late materialization. Another way is to compress individual records and have record-specific projection using only the attributes that are of interest to the queries the record belongs to. This technique prevents the increase in record size at the expense of having more complex data dependent code. In our case, we found that such techniques are not necessary, since for the workloads we used the tuples did not exceed 64 bytes. And as mentioned before, the impact on performance in this case is negligibly small.

7.1.2 *Effect of Hash Table Size*

The join is an operation which scales supralinear with relation size. Typical breaking points are when the hash table no longer fits in cache or no longer fits in main memory. When sharing a join the input relations are a union of all relations required by the queries. Thus, knowing how exactly does a join scale with the size of a hash table is important to understand the effect of sharing the join.

In this paper we focus on main-memory databases. Thus, we consider only the cases when a hash table fits in main-memory. We vary the size of relation R from 1,000 tuples to 100 million tuples which covers the cases from when the hash table fits in `L1` cache until it is much larger than `L3` cache. As before, we measure performance of full table joins for a join on 1 core and 48 cores. Since the size of the build relation is not constant, we only measure the performance of the probe operation. We take measurements for 3 different join cases. The first one is a query-at-a-time join for which we used our join algorithm without sharing support. The second one is a shared join with only few queries (< 64). Finally, the third one is a shared join with 512 queries which is already enough to feel the impact of the bitset size.

The most important thing to get from these graphs is the ratio between the lowest performance of the shared join and the highest performance of a query-at-a-time join. The reason this is important is that it depicts a worst-case scenario where the hash tables of each individual query fits in cache, but the union of all hash tables does not fit in cache. The highest ratio in this case is around 6, which means in the worst case a probe operation will cost 6 times more for a shared join. However, it is important to note that since this corresponds to shared execution of 512 concurrent queries, the extra cost is compensated by the sharing.

Another observation to make is that the single-threaded case is less sensitive to hash table size, and does not experience a performance drop as the hash-table grows larger than the `L3` cache. This means that the software prefetching technique we use is able to successfully hide the large random main memory access latencies which occur when each hash table access is a cache miss.

7.1.3 *Effects of Sharing the Join*

In the following experiment we illustrate the effect of sharing the join for queries with predicates. We vary the selectivity as well as the location of the predicates and we measure how much time it would take to execute a set of queries if they were to be

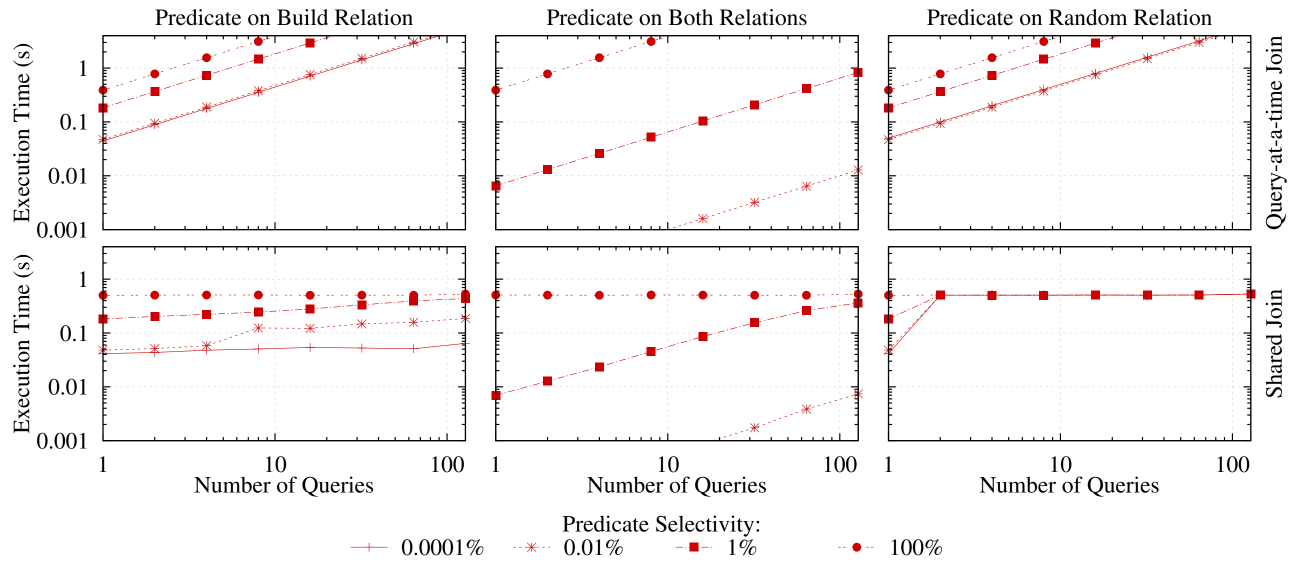


Figure 9: Performance of MQJoin versus Query-at-a-time Join

executed with a shared join or one by one with a query-at-a-time join. We consider three types of queries with predicates: one where all queries have a predicate on one of the relations; one where all queries have predicates on both relations; and one where half of the queries have a predicate on one relation and the other half have a predicate on the other relation. We use the default relations R and S both with 100 million tuples. A selectivity of a predicate of, for instance 0.001% on relation R, means that that query selects randomly around 1,000 tuples from R. To avoid measuring the effects from scanning and evaluation of predicates, the input relations are precomputed both for the shared join and the query-at-at-time join. Results are shown in Figure 9.

One important thing to emphasize from these results is that the execution time of the shared join has a ceiling. This represents the point where the union of all tuples is the whole relation and thus shared join does a full table join. The performance then is constant until the size of the bitset gets high enough to make an impact.

For the first type of queries where all predicates are on one relation a shared join almost always performs better than a query-at-a-time approach. This is true even in the cases when the predicates are mutually exclusive for all queries, making the output relations mutually exclusive as well. The benefit in this case comes from sharing the probe relation, where every probing tuple is shared for all queries.

For the second type of queries where each query has a predicate on both relations we can see that a shared join is only beneficial if there are some common tuples between the queries. Due to the randomness of the predicates in this setup, only the queries with lower selectivity predicates share tuples as the number of queries increase. As the bitsets increase with more queries, the performance of the shared join will suffer. However, if there are no common tuples between queries then the bitset will contain mostly zeros so it will be easily compressible.

While the previous two cases were interesting to point out, we expect that a realistic workload will consist of more diverse sets of queries. The worst case scenario for shared join is when there are two queries one with a predicate on one relation, while the other has a predicate on the other relation. The shared join in this case will do a full table join, and the number of queries in this case required for the shared join to do better than the query-at-at-time join depends the impact of the size of the hash table on the join, which is what we saw in Figure 8.

7.2 TPC-H Benchmark

The TPC-H benchmark suite [1] consists of 22 analytical queries most of which require heavy scans and joins on large portions of data. As we focus only on the scan and join operations of the queries we took the TPC-H queries and extracted their scan and join subqueries. In order to avoid sending large materialized relations over the network we included simple `SUM` aggregate on top of every query. To ensure that all attributes are projected during the joins as required in the original queries, we added all necessary attributes to the `SUM` aggregate expression. For instance the transformed version of Query 9 is shown in Listing 1

Listing 1: Transformed SQL version of Query 9

```
SELECT SUM(
    ps_supplycost + l_extendedprice + l_discount
    l_quantity + s_nationkey + o_totalprice)
FROM lineitem, part, supplier, partsup, orders
WHERE l_orderkey = o_orderkey
AND l_partkey = p_partkey
AND l_suppkey = s_suppkey
AND l_partkey = ps_partkey
AND l_suppkey = ps_suppkey
AND p_name LIKE '%[COLOR]%';
```

Furthermore, we removed any queries that required no joins, queries that contain more complex predicates which our shared scan implementation does not yet support, and queries that required joins other than equi-joins, which are not currently supported by our system. The final set of queries include 13 query templates that contained the scan and join subqueries of the following TPC-H queries: 2, 3, 5, 7, 8, 9, 10, 11, 14, 16, 17, 19, 20. For comparison, related work uses a smaller subset of TPC-H. Both QPipe and Datapath work with only 8 queries. The global operator plan that is used to process these queries is shown in Figure 10. This plan was created as described in Section 5 to maximize sharing for a batch of queries. The scale of the TPC-H data used was 10.

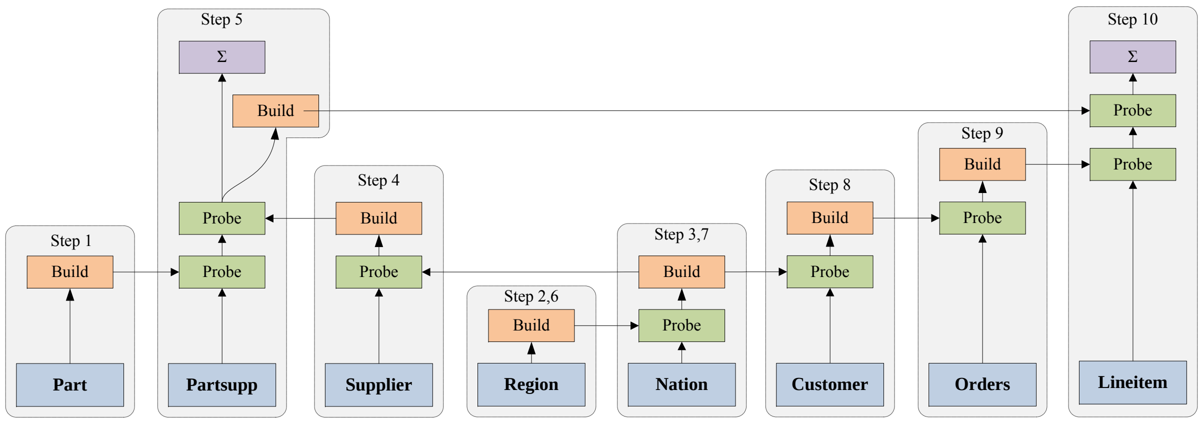


Figure 10: Global Join Plan for the TPC-H Workload

7.2.1 *TPC-H Execution*

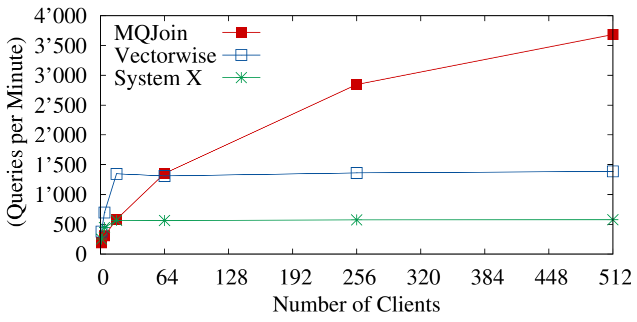
To compare our approach to a query-at-a-time system we run an experiment with multiple clients where each client executes the TPC-H based queries one by one, in randomized order, and with randomized parameters as per the TPC-H specification. The clients execute the queries without think-time for a particular period of time, while we vary the number of clients and measure throughput and response time.

Our shared join system in this case is running on all 48 cores, with every TPC-H relation partitioned across the cores. The memory of each partition is bound to the NUMA node of the core, while the memory of the hash tables is interleaved across all NUMA nodes. As mentioned before, the system executes queries in batches where queries are queued up in a batch while the current batch is being executed. For Vectorwise, we have one connection per client for up to 64 clients. With more than 64 clients the clients start sharing 64 connections in a FCFS fashion.

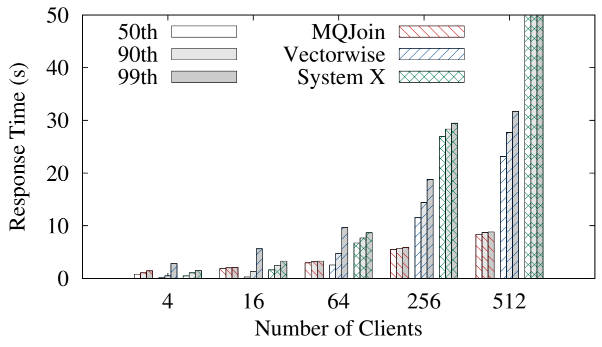
In Figure 11a we show the throughput comparison of our system, Vectorwise and System X as we increase the number of clients. The results indicate that our system outperforms both commercial counterparts for more than 60 clients and gives 2-5x higher throughput for 256 clients. Although this might not seem to be a large improvement, it should be taken into account that we are comparing to systems that are leading TPC-H benchmark performers for single node main-memory processing. There are many optimizations in Vectorwise and System X that are orthogonal to shared query processing, in particular column-wise processing. The reason why performance increase slows down from 256 to 512 queries is explained in the next section where we profile the performance of our system.

In Figure 11b we show response time percentiles for the same experiment. This graph shows that our system has significantly more stable and consistent response time. Both Vectorwise and System X are slower for the most expensive queries for as low as 4 clients. This graph also shows the equalization effect of the single query queue on response time. This might not always be a desired property for all queries in the system, however it provides predictable performance that is important for systems which must provide response time guarantees.

Figure 12 illustrates the advantage of MQJoin over its query-at-a-time counterparts. For this experiment we measured the total number of input and output tuples processed by hash join operators for different sized sets of concurrently running queries. Due to technical limitations we only collected this data from MQJoin and System X. Nevertheless, the results clearly show the difference between MQJoin and a traditional query-at-a-time approach. For small number of concurrent queries, System X processes significantly less join tuples than MQJoin since it can optimize the join order of each query individually. Furthermore, System X makes use of bitmaps to prefilter join tuples, further reducing them at additional cost to the scan operation. On the other side, the number of processed join tuples in MQJoin, grows significantly slower reaching a plateau due to the sharing effect as shown in Figure 9. This enables MQJoin to process larger workloads more efficiently reaching higher performance.



(a) TPC-H Throughput



(b) TPC-H Response Time Percentiles

Figure 11: Throughput and Response Time for TPC-H Based Workload on a Database of Scale 10

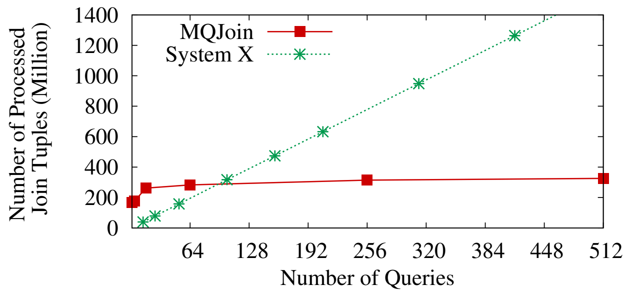


Figure 12: Number of Tuples Processed in Join Operations in MQJoin vs System X

7.2.2 *Performance Profiling*

In the performance results in the previous experiment the throughput no longer increased for MQJoin after a certain point. In the following experiment we show the reason behind this. Figure 13 shows the breakdown of CPU time spent in our system per operator class while varying the number of concurrently running queries. The concurrent queries are a multiple of the set of 13 TPC-H based queries with randomized parameter values. The results shows that, as the number of queries in the batch increase, the scan operators take most of the CPU time. The reason for this is that, as shown previously, a shared join scales almost constantly with the number of queries as soon as the point of doing full table joins is reached. On the other hand, scaling the evaluation of predicates is more difficult and depends on workload parameters such as complexity and selectivity of the predicates.

7.2.3 *Workload Properties*

Since we are running a workload with hundreds of concurrent queries, it is important to understand the amount of overlap in the queries and its effect on performance. For this reason we performed both a static analysis of each of the 13 query templates and a dynamic analysis on the workload as it is being executed in the system. The results show little overlap in the amount of data queries are interested in and demonstrates the reason why MQJoin is able to benefit from sharing opportunities in this case.

Table 1 shows the summary from the static workload analysis with two key properties for each of the 13 query templates. The number of possible predicate parameters indicate how many unique queries there are in a certain workload. For the largest workload of 512 concurrent clients, this corresponds to around 40 query instantiations per template. As the table shows, only 3 of the 13 templates have less than 40 possible parameter values, the smallest one having 24. The rest have many more possible parameter values, meaning that even in a set of 512 concurrent queries, the expected amount of identical queries will be marginally small.

Table 1: Workload Properties: Selectivity and Number of Possible Parameters per Query Template

	Select.	#Param.		Select.	#Param.
Q2	0.08%	1250	Q11	4%	25
Q3	0.465%	155	Q14	0.277%	60
Q5	4%	25	Q16	2%	3750
Q7	0.064%	625	Q17	0.1%	1000
Q8	0.0053%	18750	Q19	0.0028%	250
Q9	5.26%	92	Q20	0.043%	2300
Q10	4.16%	24			

The selectivity values show the combined selectivity of the predicates for each query template. The results show that the majority of the queries have a selectivity of less than 1 percent, which indicates possibly little overlap in the data of interest even for several hundred queries. This is confirmed by the results of our dynamic workload analysis shown in Figure 14. In this experiment we measured the average number of queries per tuple for different types of intermediate results. The solid red line corresponds to the intermediate results, which are the output of join operators and input to aggregate operators. The very small amount of queries per tuple of around 1.4 for 100 queries and 2.6 for 400 queries confirms the small overlap in data mentioned before.

Unlike the output, the input to the join operators contains a larger overlap in data with an average of 120 queries per tuple for 400 concurrent queries. For this case we measured the average number of queries per tuple in the intermediate results that are the output of scan operators and input to join operators. As is also shown in Figure 14, the majority of this overlap comes from full table scans. This experiment demonstrates the benefits of sharing join execution even for queries with a disjoint set of predicates, since there is still a large overlap in the data that needs to be processed.

7.3 Comparison to CJoin

As the closest related work, we also compare our approach to CJoin [9]. We use the same Star Schema Benchmark [21] workload used to test CJoin. The data set has a scale of 100 and we use three workload types. The first two come from the same workload used in the CJoin paper with the predicates on the dimension relations set to 1% and 10% respectively. The third one uses the queries and selectivity as defined by the Star Schema Benchmark specification. We do not use queries 1.1, 1.2 and 1.3 as they contain predicates on the fact relation which is not supported by CJoin. Since we do not support a `group by` operation, we used a corresponding

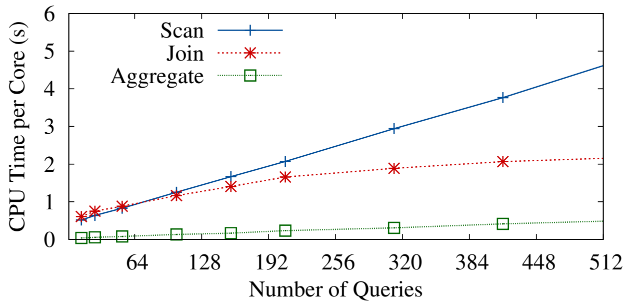


Figure 13: CPU Time spent per Operator Class for TPC-H Based Workload

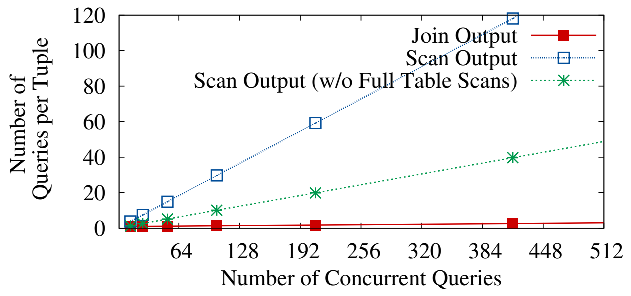


Figure 14: Amount of Data Overlap in Intermediate Results

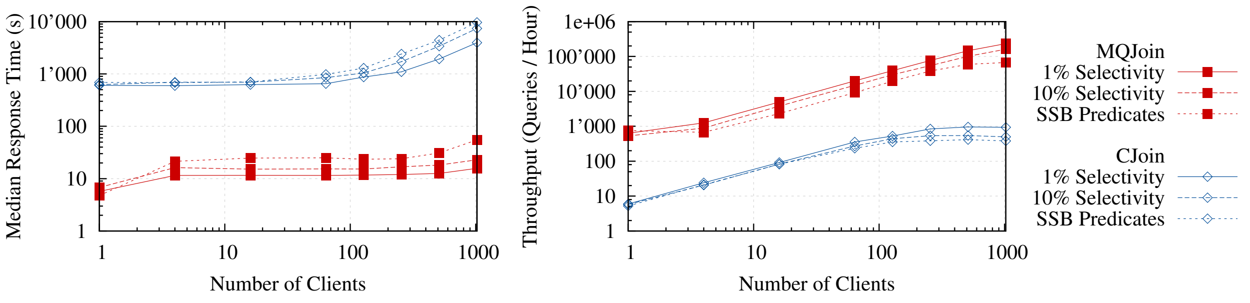


Figure 15: MQJoin and CJoin Performance for Star Schema Benchmark Dataset of Scale 100

`sum` operation for CJoin as well. To avoid any disk accesses for CJoin, we placed the underlying Postgres instance in a temporary in-memory file system. For both systems, we varied the number of clients and measured response time and throughput. Clients issue queries one after another without think-time.

The results are shown in Figure 15. The first thing to note is the large performance difference between the two systems. One reason is that CJoin was designed with a disk-resident fact relation in mind, and was run on a smaller machine with 8 cores. Although Postgres resides fully in main-memory, the streaming of the fact relation from Postgres to CJoin becomes a bottleneck and is not able to supply the CJoin operator running on 40+ cores. Nevertheless, the main conclusion to draw from these results comes from the relative performance of the two systems as the number of clients increases. CJoin's performance starts degrading significantly sooner as a result of missed sharing opportunities. Since CJoin updates the hash tables for each query as the query arrives to the system, it misses out on sharing the build operation for concurrent queries. As the number of clients increase, updating the hash tables becomes a bottleneck. The per query cost of building and updating the hash table is also a relevant factor. As shown in the results, workloads with less selective or more complex predicates on the dimension relations aggravate the problem. It is also for this reason why CJoin is suitable only in a star schema scenario.

8. CONCLUSIONS

This paper presented an algorithm that exploits the sharing potential of join execution up to a very high level to meet the demands of such workloads. The goal is achieved by using techniques that minimize redundant work across concurrent queries and efficiently use the hardware resources such as CPU and memory bandwidth. The resulting method handles significantly larger workloads than the state-of-the-art and outperforms leading main-memory analytical databases by providing higher throughput and more stable and predictable response times.

9. REFERENCES

[1] TPC-H Benchmark. `http://www.tpc.org/tpch/spec/tpch2.17.0.pdf`.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *PVLDB*, 5(10):1064–1075, June 2012.

[3] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath System: a Data-centric Analytic Processing Engine for Large Data Warehouses. In *Proc. SIGMOD 2010*, pages 519–530, 2010.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. SIGMOD 2000*, pages 261–272, 2000.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[6] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proc. ICDE 2013*, pages 362–373, 2013.

[7] S. Blanas, Y. Li, and J. M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proc. SIGMOD 2011*, pages 37–48, 2011.

[8] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR 2005*, pages 225–237, 2005.

[9] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, Aug. 2009.

[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance Through Prefetching. In *Proc. ICDE 2004*, pages 116–, 2004.

[11] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst.*, 32(3), Aug. 2007.

[12] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one Thousand Queries with One Stone. *PVLDB*, 5(6):526–537, Feb. 2012.

[13] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared Workload Optimization. *PVLDB*, 7(6):429–440, Feb. 2014.

[14] S. Harizopoulos and A. Ailamaki. StagedDB: Designing Database Servers for Modern Hardware. In *In IEEE Data*, pages 11–16, 2005.

[15] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: a Simultaneously Pipelined Relational Query Engine. In *Proc. SIGMOD 2005*, pages 383–394, 2005.

[16] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To Share or Not to Share? In *Proc. VLDB 2007*, pages 351–362, 2007.

[17] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *PVLDB*, 2(2):1378–1389, Aug. 2009.

[18] C. A. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling. In *Proc. ICDE 2007*, pages 1136–1145, 2007.

[19] C. A. Lang, B. Bhattacharjee, T. Malkemus, and K. Wong. Increasing Buffer-locality for Multiple Index Based Scans through Intelligent Placement and Index Scan Speed Control. In *Proc. VLDB 2007*, pages 1298–1309.

[20] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):709–730, July 2002.

[21] P. O'Neil, B. O'Neal, and X. Chen. Star Schema Benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[22] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing Data and Work across Concurrent Analytical Queries. *PVLDB*, 6(9):637–648, July 2013.

[23] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory Scan Sharing for Multi-core CPUs. *PVLDB*, 1(1):610–621, Aug. 2008.

[24] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *Proc. ICDE 2008*, pages 60–69, 2008.

[25] T. K. Sellis. Multiple-query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[26] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. VLDB 1994*, pages 510–521, 1994.

[27] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1):706–717, Aug. 2009.

[28] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proc. VLDB 2007*, pages 723–734, 2007.

[29] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-oriented Query Processing. In *Proc. DaMoN 2008*, pages 47–54, 2008.

[30] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A Vectorized Analytical DBMS. In *Proc. ICDE 2012*, pages 1349–1350, 2012.