Sharing Data and Work Across Concurrent Analytical Queries

Iraklis Psaroudakis
École Polytechnique Fédérale
de Lausanne

iraklis.psaroudakis@epfl.ch

Manos Athanassoulis
École Polytechnique Fédérale
de Lausanne

manos.athanassoulis@epfl.ch

Anastasia Ailamaki
École Polytechnique Fédérale
de Lausanne

anastasia.ailamaki@epfl.ch

ABSTRACT

Today's data deluge enables organizations to collect massive data, and analyze it with an ever-increasing number of concurrent queries. Traditional data warehouses (DW) face a challenging problem in executing this task, due to their query-centric model: each query is optimized and executed independently. This model results in high contention for resources. Thus, modern DW depart from the query-centric model to execution models involving sharing of common data and work. Our goal is to show *when* and *how* a DW should employ sharing. We evaluate experimentally two sharing methodologies, based on their original prototype systems, that exploit work sharing opportunities among concurrent queries at run-time: Simultaneous Pipelining (SP), which shares intermediate results of common sub-plans, and Global Query Plans (GQP), which build and evaluate a single query plan with shared operators.

First, after a short review of sharing methodologies, we show that SP and GQP are orthogonal techniques. SP can be applied to shared operators of a GQP, reducing response times by 20%-48% in workloads with numerous common sub-plans. Second, we corroborate previous results on the negative impact of SP on performance for cases of low concurrency. We attribute this behavior to a bottleneck caused by the push-based communication model of SP. We show that pull-based communication for SP eliminates the overhead of sharing altogether for low concurrency, and scales better on multi-core machines than push-based SP, further reducing response times by 82%-86% for high concurrency. Third, we perform an experimental analysis of SP, GQP and their combination, and show when each one is beneficial. We identify a trade-off between low and high concurrency. In the former case, traditional query-centric operators with SP perform better, while in the latter case, GQP with shared operators enhanced by SP give the best results.

1. INTRODUCTION

Data warehouses (DW) are databases specialized for servicing on-line analytical processing (OLAP) workloads. OLAP workloads consist mostly of ad-hoc, long running, scan-heavy queries over relatively static data (new data is periodically loaded). Today, in the era of data deluge, organizations collect massive data for

analysis. The increase of processing power and the growing applications' needs have led to increased requirements for both throughput and latency of analytical queries over ever-growing datasets.

According to a recent study of the DW market [26], more than half of DW have less than 50 concurrent users. Almost 40%, however, of the companies using DW project that in 3 years they will have 200-1000 concurrent users. General-purpose DW, however, cannot easily handle analytical workloads over big data with such concurrency [3]. A limiting factor is their typical **query-centric** model: DW optimize and execute each query independently. Concurrent queries, however, often exhibit overlapping data accesses or computations. The query-centric model misses the opportunities of sharing work and data, and results in performance degradation due to the contention of concurrent queries for I/O, CPU and RAM.

1.1 Methodologies for sharing data and work

A variety of ideas have been proposed to exploit sharing, including buffer pool management techniques, materialized views, caching and multi-query optimization (see Section 2). More recently, DW started sharing data at the I/O layer using **shared scans** (with variants also known as circular scans, cooperative scans or clock scan) [7, 8, 23, 29, 30]. In this paper, we evaluate work sharing techniques at the level of the execution engine. We distinguish two predominant methodologies: (a) Simultaneous pipelining (SP) [13], and (b) Global query plans (GQP) [2, 3, 4, 11].

Simultaneous pipelining (SP) is introduced in QPipe [13], an operator-centric execution engine, where each relational operator is encapsulated into a self-contained module called a **stage**. Each stage detects common sub-plans among concurrent queries, evaluates only one and pipelines the results to the rest when possible (see Sections 2.2 and 2.3). **Global query plans (GQP) with shared operators** are introduced in the CJOIN operator [3, 4]. A single shared operator is able to evaluate multiple concurrent queries. CJOIN uses a GQP, consisting of shared hash-join operators that evaluate the joins of multiple concurrent queries simultaneously. More recent research prototypes extend the logic to additional operators and to more general cases [2, 11] (see Sections 2.4 and 2.5).

Figure 1 illustrates how a query-centric model, shared scans, SP, and a GQP operate through a simple example of three concurrent queries which perform natural joins without selection predicates and are submitted at the same time. The last two queries have a common plan, which subsumes the plan of the first. We note that shared scans are typically used with both SP and GQP.

1.2 Integrating Simultaneous Pipelining and Global Query Plans

In order to perform our analysis and experimental evaluation of SP vs. GQP, we integrate the original research prototypes that in-
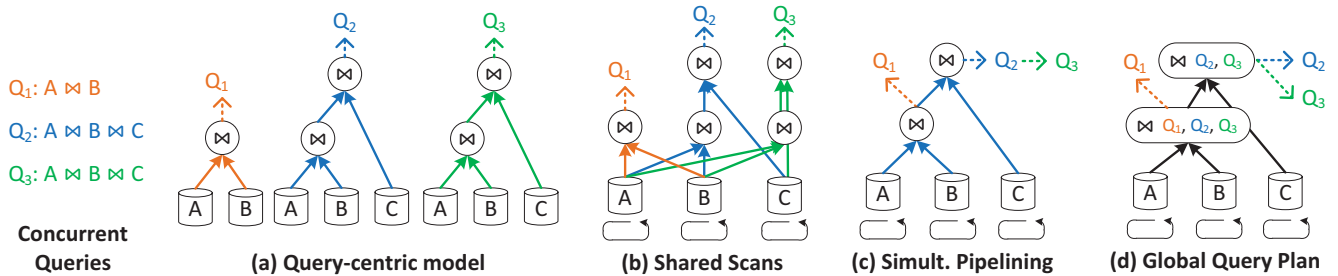
Figure 1: Evaluation of three concurrent queries using (a) a query-centric model, (b) shared scans, (c) SP, and (d) a GQP.

troduced them into one system: we integrate the CJOIN operator as an additional stage of the QPipe execution engine (see Section 3). Thus, we can dynamically decide whether to evaluate multiple concurrent queries with the standard query-centric relational operators of QPipe, with or without SP, or the GQP offered by CJOIN.

Furthermore, this integration allows us to combine the two sharing techniques, showing that they are in fact orthogonal. As shown in Figure 1d, the GQP misses the opportunity of sharing common sub-plans, and redundantly evaluates both $Q2$ and $Q3$. SP can be applied to shared operators to complement a GQP with the additional capability of sharing common sub-plans (see Section 3).

1.3 Optimizing Simultaneous Pipelining

For the specific case of SP, it is shown in the literature [14, 23] that if there is a serialization point, enforcing aggressive sharing does not always improve performance. In cases of low concurrency and sufficient available resources, it is shown that the system should first parallelize with a query-centric model before sharing.

To calculate the turning point where sharing becomes beneficial, a prediction model is proposed [14] for determining at run-time whether SP is beneficial. In this paper, however, we show that the serialization point is due to the push-based communication employed by SP [13, 14]. We show that pull-based communication can drastically minimize the impact of the serialization point, and is better suited for sharing common results on machines with multi-core processors.

We introduce ***Shared Pages Lists*** (SPL), a pull-based sharing approach that eliminates the serialization point caused by push-based sharing during SP. SPL are data structures that store the intermediate results of relational operators, and allow for a single producer and multiple consumers. SPL make sharing with SP always beneficial and reduce response times by 82%-86% in cases of high concurrency, compared to the original push-based SP design and implementation [13, 14] (see Section 4).

1.4 Simultaneous Pipelining vs. Global Query Plans

Having optimized SP, and having integrated the CJOIN operator in the QPipe execution engine, we proceed to perform an extensive analysis and experimental evaluation of SP vs. GQP (see Section 5). Our work answers two fundamental questions: *when* and *how* an execution engine should share in order to improve performance of analytical workloads.

Sharing in the execution engine. We identify a performance trade-off between using a query-centric model and sharing. For a high number of concurrent queries, the execution engine should share, as it reduces contention for resources and improves performance in comparison to a query-centric model. For low concurrency, however, sharing is not always beneficial.

With respect to SP, we corroborate previous related work [14, 23], that if SP entails a serialization point, then enforcing aggressive sharing does not always improve performance in cases of low concurrency. Our newly optimized SP with SPL, however, eliminates the serialization point, making SP beneficial in cases of both low and high concurrency.

With respect to GQP, we corroborate previous work [2, 3, 4, 11] that shared operators are efficient in reducing contention for resources and in improving performance for high concurrency (see Section 5.2.1). The design of a shared operator, however, inherently increases bookkeeping in comparison to the typical operators of a query-centric model. Thus, for low concurrency we show that shared operators result in worse performance than the traditional query-centric operators (see Section 5.2.2).

Moreover, we show that SP can be applied to shared operators of a GQP, in order to get the best out of the two worlds. SP can reduce the response time of a GQP by 20%-48% for workloads with common sub-plans (see Section 5.2.3).

Sharing in the I/O layer. Though our work primarily studies sharing inside the execution engine, our experimental results also corroborate previous work relating to shared scans. The simple case of a circular scan per table is able to improve performance of typical analytical workloads both in cases of low and high concurrency. In highly concurrent cases, response times are reduced by 80%-97% in comparison to independent table scans (see Section 5.2.1).

Rules of thumb. Putting all our observations together, we deduce a few rules of thumb for sharing, presented in Table 1. Our rules of thumb apply for the case of typical OLAP workloads involving ad-hoc, long running, scan-heavy queries over relatively static data.

When	*How* to share in the	
	Execution Engine	I/O Layer
Low concurrency	Query-centric operators + SP	Shared Scans
High concurrency	GQP (shared operators) + SP	

Table 1: Rules of thumb for when and how to share data and work across typical concurrent analytical queries in DW.

1.5 Contributions

We perform an experimental analysis of two work sharing methodologies, (a) Simultaneous Pipelining (SP), and (b) Global Query Plans (GQP), based on the original research prototypes that introduce them. Our analysis answers two fundamental questions: *when* and *how* an execution engine should employ sharing in order to improve performance of typical analytical workloads. We categorize different sharing techniques for relational databases, and identify SP and GQP as two state-of-the-art sharing methodologies (Section 2). Next, our work makes the following main contributions:

- **Integration of SP and GQP:** We show that SP and GQP are orthogonal, and can be combined to take the best of the two worlds (Section 3). In our experiments, we show that SP can further improve the performance of a GQP by 20%-48% for workloads that expose common sub-plans.

- **Pull-based SP:** We introduce Shared Pages Lists (SPL), a pull-based approach for SP that eliminates the sharing overhead of push-based SP. Pull-based SP is better suited for multicores than push-based SP, is beneficial for cases of both low and high concurrency, and further reduces response times by 82%-86% for high concurrency (Section 4).

- **Evaluation of SP vs. GQP:** We analyze the trade-offs of SP, GQP and their combination, and we detail through an extensive sensitivity analysis when each one is beneficial (Section 5). We show that query-centric operators combined with SP result in better performance for cases of low concurrency, while GQP with shared operators enhanced by SP are better suited for cases of high concurrency.

Paper Organization. This paper is organized as follows. Section 2 consists of a review of sharing methodologies, SP, and GQP. Section 3 describes our implementation for integrating SP and GQP. Section 4 presents shared pages lists, our pull-based solution for sharing common results during SP. Section 5 includes our experimental evaluation. Section 6 includes a short discussion. We present our conclusions in Section 7.

2. WORK SHARING TECHNIQUES

In this section, we provide the necessary background of sharing techniques in the literature. We start by shortly reviewing related work, and continue to extensively review work related to SP and GQP, which compose our main area of interest. In Table 2, we summarize the sharing methodologies used by traditional query-centric systems and the research prototypes we examine.

2.1 Related Work

Sharing in the I/O layer. By sharing data, we refer to techniques that coordinate and share the accesses of queries in the I/O layer. The typical query-centric database management system (DBMS) incorporates a buffer pool and employs eviction policies [5, 16, 19, 22]. Queries, however, communicate with the buffer pool manager on a per-page basis, thus it is difficult to analyze their access patterns. Additionally, if multiple queries start scanning the same table at different times, scanned pages may not be re-used.

For this reason, shared scans have been proposed. Circular scans [7, 8, 13] are a form of shared scans. They can handle numerous concurrent scan-heavy analytical queries as they reduce buffer pool contention and they avoid unnecessary I/O accesses to the underlying storage devices. Furthermore, more elaborate shared scans can be developed for servicing different fragments of the same table or different groups of queries depending on their speed [18, 30], and for main-memory scans [23].

Shared scans can be used to handle a large number of concurrent updates as well. The Crescando [29] storage manager performs a circular scan over memory-resident table partitions, interleaving the reads and the updates of a batch of queries along the way. The scan first executes the update requests of the batch for a scanned tuple in their arrival order, and then the read requests.

Shared scans, however, are not immediately translated to a fast linear scan of a disk. The DataPath system [2], which uses a disk array as secondary storage, stores relations column-by-column by hashing pages to the disks at random. During execution, it reads pages from the disks asynchronously but sequentially, thus aggregating the throughput of a sequential scan on every disk of the array.

Sharing in the execution engine. By sharing work among queries, we refer to techniques that avoid redundant computations inside the execution engine. A traditional query-centric DBMS typically uses query caching [28] and materialized views [24]. Both, however, do not exploit sharing opportunities among in-progress queries.

Multi-Query Optimization (MQO) techniques [25, 27] are an important step towards more sophisticated sharing methodologies. MQO detects and re-uses common sub-expressions among queries. There are two main disadvantages of classic MQO: (i) it operates on batches of queries *only* during the optimization phase, and (ii) it depends on materializing shared intermediate results, at the expense of memory. This cost can be alleviated by using pipelining [9], which additionally exploits the parallelization provided by multi-core processors. The query plan is divided into sub-plans and operators are evaluated in parallel.

Both SP and GQP leverage forms of pipelined execution and sharing methodologies which bear some superficial similarities with MQO. These techniques, however, provide deeper and more dynamic forms of sharing at run-time. In the rest of this section, we provide an overview of SP and GQP, the systems that introduce them, and more recent research prototypes that advanced GQP.

2.2 Simultaneous Pipelining

SP identifies identical sub-plans among concurrent queries at run-time, evaluates only one and pipelines the results to the rest simultaneously [13]. Figure 2a depicts an example of two queries that share a common sub-plan below the join operator (along with any selection and join predicates), but have a different aggregation operator above the join. SP evaluates only one of them, and pipelines the results to the other aggregation operator.

Fully sharing common sub-plans is possible if the queries arrive at the same time. Else, sharing opportunities may be restricted. The amount of results that a newly submitted $Q2$ can re-use from the ***pivot operator*** (the top operator of the common sub-plan) of the in-progress $Q1$, depends on the type of the pivot operator and the arrival of $Q2$ during $Q1$'s execution. This relation is expressed as a ***Window of Opportunity*** (WoP) for each relational operator (following the original acronym [13]). Figure 2b depicts two common WoP, a step and a linear WoP.

A step WoP expresses that $Q2$ can re-use the full results of $Q1$ if it arrives before the first output tuple of the pivot operator. Joins and aggregations have a step WoP. A linear WoP signifies that $Q2$ can re-use the results of $Q1$ from the moment it arrives up until the pivot operator finishes. Then, $Q2$ needs to re-issue the operation in order to compute the results that it missed before it arrived. Sorts and table scans have a linear WoP. In fact, the linear WoP of the table scan operator is translated into a circular scan of each table.
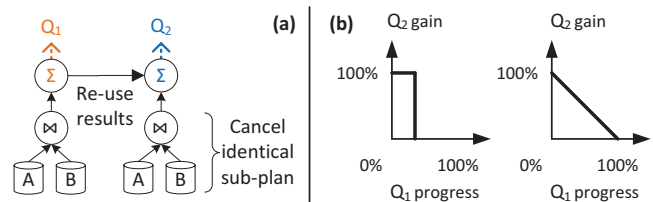

Figure 2: (a) SP example with two queries having a common sub-plan below the join operator. (b) A step and a linear WoP.

System	Traditional query-centric model	QPipe	CJOIN	DataPath	SharedDB
Sharing in the execution engine	Query Caching, Materialized Views, MQO	Simultaneous Pipelining	Global Query Plan (joins of Star Queries)	Global Query Plan	Global Query Plan (with Batched Execution)
Sharing in the I/O layer	Buffer pool management techniques	Circular scan of each table	Circular scan of the fact table	Asynchronous linear scan of each disk	Circular scan of in-memory table partitions
Storage Manager	Any	Any	Any	Special I/O Subsystem (read-only requests)	Crescando (read and update requests)

Table 2: Sharing methodologies employed by a query-centric model and the research prototypes we examine.

2.3 The QPipe execution engine

QPipe [13] is a relational execution engine that supports SP at execution time. QPipe is based on the paradigms of staged databases [12]. Each relational operator is encapsulated into a self-contained module called a *stage*. Each stage has a queue for work requests and employs a local thread pool for processing the requests.

An incoming query execution plan is converted to a series of inter-dependent *packets*. Each packet is dispatched to the relevant stage for evaluation. Data flow between packets is implemented through FIFO (first-in, first-out) buffers and page-based exchange, following a push-only model with pipelined execution. The buffers also regulate differently-paced actors: a parent packet may need to wait for incoming pages of a child and, conversely, a child packet may wait for a parent packet to consume its pages.

This design allows each stage to monitor only its packets for detecting sharing opportunities efficiently. If it finds an identical packet, and their interarrival delay is inside the WoP of the pivot operator, it attaches the new packet (*satellite* packet) to it (*host* packet). While it evaluates the host packet, SP copies the results of the host packet to the output FIFO buffer of the satellite packet.

2.4 Global Query Plans with shared operators

SP is limited to common sub-plans. If two queries have similar sub-plans but with different selection predicates for the involved tables, SP is not able to share them. Nevertheless, the two queries still share a similar plan that exposes sharing opportunities. It is possible to employ *shared operators*, where a single shared operator can evaluate both queries simultaneously. The basic technique for enabling them is sharing tuples among queries and correlating each tuple to the queries, e.g. by annotating tuples with a *bitmap*, whose bits signify if the tuple is relevant to one of the queries.

The simplest shared operator is a shared selection, that can evaluate multiple queries that select tuples from the same relation. For each received tuple, it toggles the bits of its attached bitmap according to the selection predicates of the queries. A hash-join can also be easily shared by queries that share the same equi-join predicate (more relaxed requirements are also possible [2]). Figure 3 shows a conceptual example of how a single shared hash-join is able to eval-
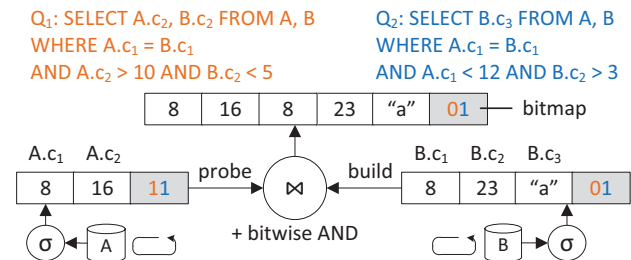

Q_1: SELECT A.c_2, B.c_2 FROM A, B
WHERE A.c_1 = B.c_1
AND A.c_2 > 10 AND B.c_2 < 5

Q_2: SELECT B.c_3 FROM A, B
WHERE A.c_1 = B.c_1
AND A.c_1 < 12 AND B.c_2 > 3

Figure 3: Example of shared selection and hash-join operators.

uate two queries. It starts with the build phase by receiving tuples from the shared selection operator of the inner relation. Then, the probe phase begins by receiving tuples from the shared selection operator of the outer relation. The hash-join proceeds as normal, by additionally performing a bitwise AND operation between the bitmaps of the joined tuples.

The most significant advantage is that a single shared operator can evaluate many similar queries. For example, a shared hash-join can evaluate queries having the same equi-join predicate, and possibly different selection predicates. In the worst case, the union of the selection predicates may force it to join the whole two relations. The disadvantage of a shared operator in comparison to a query-centric one is that it entails increased bookkeeping. For example, a shared hash-join maintains a hash table for the union of the tuples of the inner relation selected by all queries, and performs bitwise operations between the bitmaps of the joined tuples. For low concurrency, as shown by our experiments (see Section 5), query-centric operators outperform shared operators. A similar tradeoff is found for the specific case of shared aggregations on CMP [6].

By using shared scans and shared operators, a GQP can be built for evaluating all concurrent queries. GQP are introduced by CJOIN [3, 4], an operator based on shared selections and shared hash-joins for evaluating the joins of star queries [17]. GQP are advanced by the DataPath system [2] for more general schemas, by tackling the issues of routing and optimizing the GQP for a newly incoming query. DataPath also adds support for a shared aggregate operator, that calculates a running sum for each group and query.

Both CJOIN and DataPath handle new queries immediately when they arrive. This is feasible due to the nature of the supported shared operators: selections, hash-joins and aggregates. Some operators, however, cannot be easily shared. For example, a sort operator cannot easily handle new queries that select more tuples than the ones being sorted [2]. To overcome this limitation, SharedDB [11] batches queries for every shared operator. Batching allows standard algorithms to be easily extended to support shared operators, as they work on a fixed set of tuples and queries. SharedDB supports shared sorts and various shared join algorithms, not being restricted only to equi-joins. Nevertheless, batched execution has drawbacks: a new query may suffer increased latency, and the latency of a batch is dominated by the longest-running query.

2.5 The CJOIN operator

The selection of CJOIN [3, 4] for our analysis, is based on the facts that it introduced GQP, and that it is optimized for the simple case of star queries. Without loss of generality, we restrict our evaluation to star schemas, and correlate our observations to more general schemas (used, e.g., by DataPath [2] or SharedDB [11]).

Star schemas are very common for organizing data in relational DW. They allow for numerous performance enhancements [17]. A star schema consists of a large *fact table*, that stores the measured information, and is linked through foreign-key constraints to
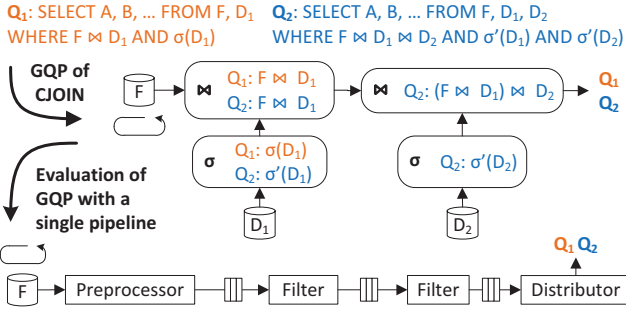
Q₁: SELECT A, B, ... FROM F, D_1 **Q₂**: SELECT A, B, ... FROM F, D_1, D_2
WHERE F ⋈ D_1 AND $\sigma(D_1)$ WHERE F ⋈ D_1 ⋈ D_2 AND $\sigma'(D_1)$ AND $\sigma'(D_2)$

Figure 4: CJOIN evaluates a GQP for star queries.

smaller *dimension tables*. A *star query* is an analytical query over a star schema. It typically joins the fact table with several dimension tables and performs operations such as aggregations or sorts.

CJOIN evaluates the joins of all concurrent star queries, using a GQP with shared scans, shared selections and shared hash-joins. Figure 4 shows the GQP that CJOIN evaluates for two star queries. CJOIN adapts the GQP with every new star query. If a new star query references already existing dimension tables, the existing GQP can evaluate it. If a new star query joins the fact table with a new dimension table, the GQP is extended with a new shared selection and hash-join. Due to the semantics of star schemas, the directed acyclic graph of the GQP takes the form of a chain.

CJOIN exploits this form to facilitate the evaluation of the GQP. It materializes the small dimension tables and stores in-memory the selected dimension tuples in the hash tables of the corresponding shared hash-joins. Practically, for each dimension table, it groups the shared scan, selection and hash-join operators into an entity called *filter*. When a new star query is admitted, CJOIN pauses, adds newly referenced filters, updates already existing filters, augments the bitmaps of dimension tuples according to the selection predicates of the new star query, and then continues. Parts of the admission phase, such as the scan of the involved dimension tables, can be done asynchronously while CJOIN is running [4].

Consequently, CJOIN is able to evaluate the GQP using a single pipeline: the *preprocessor* uses a circular scan of the fact table, and flows fact tuples through the pipeline. The data flow in the pipeline is regulated by intermediate buffers, similar to QPipe. The filters in-between are actually the shared hash-joins that join the fact tuples with the corresponding dimension tuples and additionally perform a bitwise AND between their bitmaps. At the end of the pipeline, the *distributor* examines the bitmaps of the joined tuples and forwards them to the relevant queries. For every new query, the preprocessor admits it, marking its point of entry on the circular scan of the fact table and signifies its completion when it wraps around to its point of entry on the circular scan.

3. INTEGRATING SP AND GQP

By integrating SP and GQP, we can exploit the advantages of both forms of sharing. In Section 3.1, we describe how SP can conceptually improve the performance of shared operators in the presence of common sub-plans, using several examples. These observations apply to general GQP, and are applicable to the research prototypes we mention in Section 2.4. We continue in Sections 3.2 and 3.3 to describe our implementation based on CJOIN and QPipe.

3.1 Benefits of applying SP to GQP

Identical queries. If a new query is completely identical with an ongoing query, SP takes care to re-use the final results of the on-

going query for the new query. If we assume that the top-most operators in a query plan have a full step WoP (e.g. when final results are buffered and given wholly to the client instead of being pipelined), the new query does not need to participate at all in the GQP, independent of its time of arrival during the ongoing query's evaluation. This is the case where the integration of SP and GQP offers the maximum performance benefits. Additionally, admission costs are completely avoided, the tuples' bitmaps do not need to be extended to accommodate the new query (translating to fewer bitwise operations), and the latency of the new query is decreased to the latency of the remaining part of the ongoing query.

Shared selections. If a new query has the same selection predicate as an ongoing query, SP allows to avoid the redundant evaluation of the same selection predicate from the moment the new query arrives until the end of evaluation of the ongoing query (a selection operator has a linear WoP). For each tuple, SP copies the resulting bit of the shared selection operator for the ongoing query, to the position in the tuple's bitmap that corresponds to the new query.

Shared joins. If a new query has a common sub-plan with an ongoing query under a shared join operator, and arrives within the step WoP, SP can avoid extending tuples' bitmaps with one more bit for the new query for the sub-plan. The join still needs to be evaluated, but the number of bitwise operations can be decreased.

Shared aggregations. If a new query has a common sub-plan with an ongoing query under a shared aggregation operator, and arrives within the step WoP, SP avoids calculating a redundant sum. It copies the final result from the ongoing query.

Admission costs. For every new query submitted to a GQP, an admission phase is required that possibly re-adjusts the GQP to accommodate it. In case of common sub-plans, SP can avoid part of the admission costs. The cost depends on the implementation.

For CJOIN [3, 4], the admission cost of a new query includes (a) scanning all involved dimension tables, (b) evaluating its selection predicates, (c) extending the bitmaps attached to tuples, (d) increasing the size of hash tables of the shared hash-joins to accommodate newly selected dimension tuples (if needed), and (e) stalling the pipeline to re-adjust filters [3, 4]. For identical queries, SP can avoid these costs completely. For queries with common sub-plans, SP can avoid parts of these costs, such as avoiding scanning dimension tables for which selection predicates are identical.

For DataPath [2], SP can decrease the optimization time of the GQP if it assumes that the common sub-plan of a new query can use the same part of the current GQP as the ongoing query. For SharedDB [11], SP can help start a new query before the next batch at any operator, if it has a common sub-plan with an ongoing query and has arrived within the corresponding WoP of the operator.

3.2 CJOIN as a QPipe stage

We integrate the original CJOIN operator into the QPipe execution engine as a new stage, using Shore-MT [15] as the underlying storage manager. In Figure 5, we depict the new stage that encapsulates the CJOIN pipeline.

The CJOIN stage accepts incoming QPipe packets that contain the necessary information to formulate a star query: (a) the projections for the fact table and the dimension tables to be joined, and (b) the selection predicates. The CJOIN operator does not support selection predicates for the fact table [3], as these would slow the preprocessor significantly. We have ran experiments with the preprocessor evaluating fact table selection predicates, but in most cases the cost of a slower pipeline defeated the purpose of potentially flowing fewer fact tuples in the pipeline. To respect space
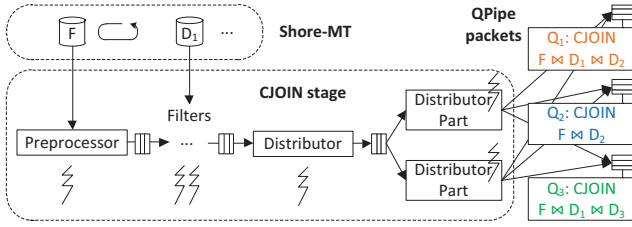
Figure 5: Integration scheme of CJOIN as a QPipe stage.

Figure 7: Sharing identical results during SP with: (a) push-only model and (b) a SPL.

limitations, we do not include these experiments. Fact table predicates are evaluated on the output tuples of CJOIN.

To improve admission costs we use batching, following the original CJOIN proposal [4]. In one pause of the pipeline, the admission phase adapts the filters for all queries in the batch. During the execution of each batch, additional new queries form a new batch to be subsequently admitted.

With respect to threads, there is a number of threads assigned to filters (we assume the horizontal configuration of CJOIN [3, 4]), each one taking a fact tuple from the preprocessor, passing it through the filters up to the distributor. The original CJOIN uses a single-threaded distributor which slows the pipeline significantly. To address this bottleneck, we augment the distributor with several *distributor parts*. Every distributor part takes a tuple from the distributor, examines its bitmap, and determines relevant CJOIN packets. For each relevant packet, it performs the projection of the star query and forwards the tuple to the output buffer of the packet.

CJOIN supports only shared hash-joins. Subsequent operators in a query plan, e.g. aggregations or sorts, are query-centric. Nevertheless, our evaluation gives us insight on the general behavior of shared operators in a GQP, as joins typically comprise the most expensive part of a star query.

3.3 SP for the CJOIN stage

We enable SP for the CJOIN stage with a step WoP. Evaluating the identical queries Q_2 and Q_3 of Figure 1d employing SP, requires only one packet entering the CJOIN stage. The second satellite packet re-uses the results.

CJOIN is itself an operator, and we integrate it as a new stage in QPipe. As with any other QPipe stage, SP is applied on the overall CJOIN stage. Conceptually, our implementation applies SP for the whole series of shared hash-joins in the GQP. Our analysis, however, gives insight on the benefits of applying SP to fine-grained shared hash-joins as well. This is due to the fact that a redundant CJOIN packet involves all redundant costs we mentioned in Section 3.1 for admission, shared selections operators and shared hash-joins. Our experiments show that the cost of a redundant CJOIN packet is significant, and SP decreases it considerably.

4. SHARED PAGES LISTS FOR SP

In this section, we present design and implementation issues of sharing using SP, and how to address them. Contrary to intuition, it is shown in the literature that work sharing is not always beneficial: if there is a serialization point during SP, then sharing common results aggressively can lead to worse performance, compared to a query-centric model that implicitly exploits parallelism [14, 23]. When the producer (host packet) forwards results to consumers (satellite packets), it is in the critical path of the evaluation of the remaining nodes of the query plans of all involved queries. Forwarding results can cause a significant serialization point. In this case, the DBMS should first attempt to exploit available resources
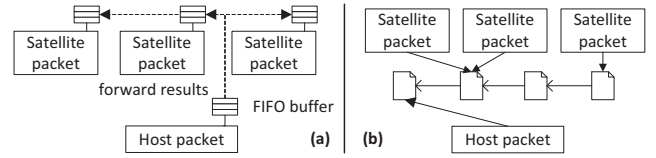
and parallelize as much as possible with a query-centric model, before sharing. A prediction model is proposed [14] for determining at run-time whether sharing is beneficial. In this section, however, we show that SP is possible without a serialization point, thus rendering SP always beneficial.

The serialization point is caused by strictly employing push-only communication. Pipelined execution with push-only communication typically uses FIFO buffers to exchange results between operators [13]. This allows to decouple query plans and have a distinct separation between queries, similar to a query-centric design. During SP, this forces the single thread of the pivot operator of the host packet to forward results to all satellite packets sequentially (see Figure 7a), which creates a serialization point.

This serialization point is reflected in the prediction model [14], where the total work of the pivot operator includes a cost for forwarding results to all satellite packets. By using copying to forward results [14], the serialization point becomes significant and delays subsequent operators in the plans of the host and satellite packets. This creates a trade-off between sharing and parallelism, where in the latter case a query-centric model without sharing is used.

Sharing vs. Parallelism. We demonstrate this trade-off with the following experiment, similar to the experiment of [14], which evaluates SP for the table scan stage with a memory-resident database. Though the trade-off applies for disk-resident databases and other stages as well, it is more pronounced in this case. Our experimental configuration can be found in Section 5. We evaluate two configurations of the QPipe execution engine: (a) No SP (FIFO), which evaluates query plans independently without any sharing, and (b) CS (FIFO), with SP enabled only for the table scan stage, thus supporting circular scans (CS). FIFO buffers are used for pipelined execution and copying is used to forward pages during SP, following the original push-only design [13, 14]. We evaluate identical TPC-H [1] Q1 queries, submitted at the same time, with a database of scale factor 1. Figure 6a shows the response times of the configurations, while varying the number of concurrent queries.

For low concurrency, No SP (FIFO) efficiently uses available CPU resources. Starting with as few as 32 queries, however, there is contention for CPU resources and response times grow quickly, due to the fact that our server has 24 available cores and the query-centric model evaluates queries independently. For 64 queries, it uses all cores at their maximum, resulting in excessive and unpredictable response times, with a standard deviation up to 30%.

CS (FIFO) suffers from low utilization of CPU resources, due to the aforementioned serialization point of SP. The critical path increases with the number of concurrent queries. For 64 queries, it uses an average of 3.1 of available cores. In this experiment, the proposed prediction model [14] would not share in cases of low concurrency, essentially falling back to the line of No SP (FIFO), and would share in cases of high concurrency.

Nevertheless, the impact of the serialization point of SP can be minimized. Simply copying tuples in a multi-threaded way would not solve the problem, due to synchronization overhead and increased required CPU resources. A solution would be to forward
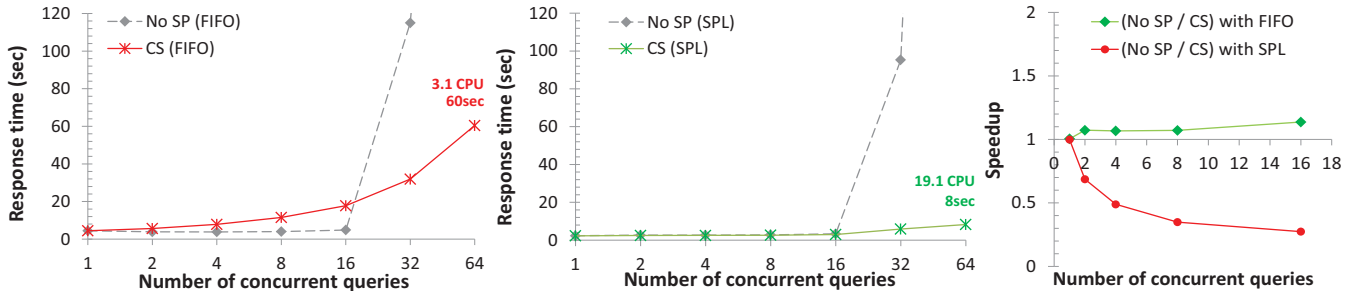
Figure 6: Evaluating multiple identical TPC-H Q1 queries (a) with a push-only model during SP (FIFO), and (b) with a pull-based model during SP (SPL). In (c), we show the corresponding speedups of the two methods of SP, over not sharing, for low concurrency.

tuples via pointers, a possibility not considered by the original system. We can, however, avoid unnecessary pointer chasing; by employing pull-based communication, we can share the results and eliminate forwarding altogether. In essence, we transfer the responsibility of sharing the results from the producer to the consumers. Thus, the total work of the producer does not include any forwarding cost. Our pull-based communication model is adapted for SP for any stage with a step or linear WoP.

The serialization point of push-based SP was not a visible bottleneck in the original implementation of QPipe, due to experiments being ran on a uni-processor [13]. On machines with multi-core processors its impact grows as the available parallelism increases, and the aforementioned prediction model [14] was proposed to decide at run-time whether (push-based) sharing should be employed. Our pull-based communication model for SP, however, eliminates the serialization point, leading to better scaling on machines with modern multi-core processors with virtually no sharing overhead.

To achieve this, we create an intermediate data structure, the **Shared Pages Lists** (SPL). SPL have the same usage as the FIFO buffers of the push-only model. A SPL, however, allows a single producer and multiple consumers. A SPL is a linked list of pages, depicted in Figure 7b. The producer adds pages at the head, and the consumers read the list from the tail up to the head independently.

In order to show the benefits of SPL, we run the experiment of Figure 6a, by employing SPL instead of FIFO buffers. When SP does not take place, a SPL has the same role as a FIFO buffer, used by one producer and one consumer. Thus, the No SP (SPL) line has similar behavior with the No SP (FIFO) line. During SP, however, a single SPL is used to share the results of one producer with all consumers. Figure 6b shows the response times of the configurations, while varying the number of concurrent queries.

With SPL, sharing has the same or better performance than not sharing, for all cases of concurrency. We avoid using a prediction model altogether, for deciding whether to share or not. Parallelism is achieved due to the minimization of the serialization point. For high concurrency, CS (SPL) uses more CPU resources and reduces response times by 82%-86% in comparison to CS (FIFO).

Additionally, Figure 6c shows the speedup of sharing over not

sharing, for both models. We depict only values for low concurrency, as sharing is beneficial for both models in cases of high concurrency. We corroborate previous results on the negative impact of sharing with push-only communication [14] for low concurrency, and show that pull-based sharing is always beneficial.

4.1 Design of a Shared Pages List

Figure 8 depicts a SPL. It points to the head and tail of the linked list. The host packet adds pages at the head. Satellite packets read pages from the SPL independently. Due to different concurrent actors accessing the SPL, we associate a lock with it. Contention for locking is minimal in all our experiments, mainly due to the granularity of pages we use (32KB). A lock-free linked list, however, can also be used to address any scalability problems.

Theoretically, if we allow the SPL to be unbounded, we can achieve the maximum parallelism possible, even if the producer and the consumers move at different speeds. There are practical reasons, however, why the SPL should not be unbounded, similar to the reasons why a FIFO buffer should not be unbounded, including: saving RAM, and regulating differently paced actors.

To investigate the effect of the maximum size, we ran the experiment of Figure 6b, for the case of 8 concurrent queries, varying the maximum size of SPL up to 512MB. We observed that changing the maximum size of the SPL does not heavily affect performance. Due to space limitations, we do not present the relevant graph. Hence, we chose a maximum size of 256KB for our experiments in Section 5 in order to minimize the memory footprint of SPL.

In order to decrease the size of the SPL, the last consumer is responsible for deleting the last page. Each page has an atomic counter with the number of consumers that will read this page. When a consumer finishes processing a page, he decrements its counter, deleting the page if he sees a zero counter. In order to know how many consumers will read a page, the SPL stores a list of active satellite packets. The producer assigns their number as the initial value of the atomic counter of each emitted page.

4.2 Linear Window of Opportunity

In order to handle a linear WoP, such as circular scans, the SPL stores the point of entry of every consumer. When the host packet finishes processing, the SPL is passed to the next host packet that handles the processing for re-producing missed results.

When the host packet emits a page, it checks for consumers whose point of entry is this page, and will need to finish when they reach it. The emitted page has attached to it a list of these finishing packets, which are removed from the active packets of the SPL (they do not participate in the atomic counter of subsequently emitted pages). When a consumer (packet) reads a page, it checks whether it is a finishing packet, in which case, it exits the SPL.
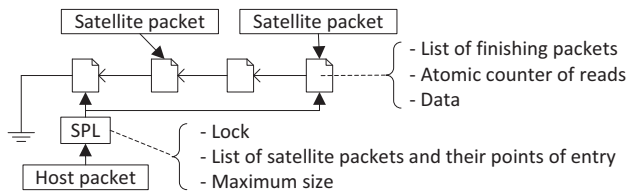

Figure 8: Design of a shared pages list.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Methodology

We compare five configurations of the QPipe execution engine:

- **QPipe**, without SP, which is similar to a typical query-centric model that evaluates query plans separately with pipelining, without any sharing. This serves as our baseline.

- **QPipe-CS**, supporting SP only for the table scan stage, i.e. circular scans (CS). It improves performance over QPipe by reducing contention for CPU resources, the buffer pool and the underlying storage device.

- **QPipe-SP**, supporting SP additionally for the join stage. It improves performance over QPipe-CS, in cases of high similarity, i.e. common sub-plans. In cases of low similarity, it behaves similar to QPipe-CS.

- **CJOIN**, without SP, which is the result of our integration of CJOIN into QPipe, hence the joins in star queries are evaluated with a GQP of shared hash-joins. We remind that CJOIN only supports shared hash-joins, thus subsequent operators are query-centric. Nevertheless, this configuration allows us to compare shared hash-joins with the query-centric ones used by the previous configurations, giving us insight on the performance characteristics of general shared operators.

- **CJOIN-SP**, which additionally supports SP for the CJOIN stage (see Section 3.3). We use this configuration to evaluate the benefits of combining SP with a GQP. It behaves similar to CJOIN in cases of low similarity in the query mix.

In all our experiments, SP for the aggregation and sorting stages is off. This is done on purpose to isolate the benefits of SP for joins only, so as to better compare QPipe-SP and CJOIN-SP.

We use the Star Schema Benchmark [21] and Shore-MT [15] as the storage manager. SSB is a simplified version of TPC-H [1] where the tables lineitem and order have been merged into lineorder and there are four dimension tables: date, supplier, customer and part. Shore-MT is an open-source multi-threaded storage manager developed to achieve scalability on multi-core platforms.

Our server is a Sun Fire X4470 server with four hexa-core processors Intel Xeon E7530 at 1.86 Ghz, with hyper-threading disabled and 64 GB of RAM. Each core has a 32KB L1 instructions cache, a 32KB L1 data cache, and a 256KB L2 cache. Each processor has a 12MB L3 cache, shared by all its cores. For storage, we use two 146 GB 10kRPM SAS 2.5" disks, configured as a RAID-0. The O/S is a 64-bit SMP Linux (Red Hat), with a 2.6.32 kernel.

We clear the file system caches before every measurement. All configurations use a large buffer pool that fits datasets of scale factors up to 30 (scanning all tables reads 21GB of data from disk). SPL are used for exchanging results among packets. We use 32KB pages and a maximum size of 256KB for a SPL (see Section 4).

Unless stated otherwise, every data point is the average of multiple iterations with standard deviation less or equal to 10%. In some cases, contention for resources results in higher deviations. Furthermore, we mention the average CPU usage and I/O throughput of representative iterations (averaged only over their activity period), to gain insight on the performance of the configurations.

Our sensitivity analysis is presented in Section 5.2. We vary (a) the number of concurrent queries, (b) whether the database is memory-resident or disk-resident, (c) the selectivity of fact tuples, (e) the scale factor, and (d) the query similarity which is modeled in our experiments by the number of possible different submitted
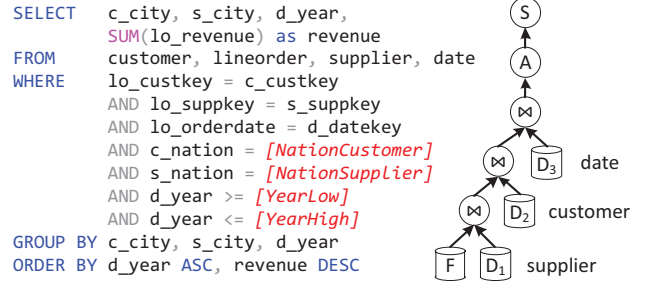
```
SELECT   c_city, s_city, d_year,
         SUM(lo_revenue) as revenue
FROM     customer, lineorder, supplier, date
WHERE    lo_custkey = c_custkey
         AND lo_suppkey = s_suppkey
         AND lo_orderdate = d_datekey
         AND c_nation = [NationCustomer]
         AND s_nation = [NationSupplier]
         AND d_year >= [YearLow]
         AND d_year <= [YearHigh]
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC
```


Figure 9: The SSB Q3.2 SQL template and the query plan.

query plans. Queries are submitted at the same time, and are all evaluated concurrently. This single batch for all queries allows us to minimize query admission overheads for CJOIN, and additionally allows us to show the effects of SP, as all queries with common sub-plans arrive surely inside the WoP of their pivot operators. We note that variable interarrival delays can decrease sharing opportunities for SP, and refer the interested reader to the original QPipe paper [13] to review the effects of interarrival delays for different cases of pivot operators and WoP.

Finally, in Section 5.3, we evaluate QPipe-SP, CJOIN-SP, and Postgres with a mix of SSB queries and a scale factor 30. We use PostgreSQL 9.1.4 as another example of a query-centric execution engine that does not share among concurrent queries. We configure PostgreSQL to make the comparison with QPipe as fair as possible. We use 32KB pages, large shared buffers that fit the database, ensure that it never spills to the disk and that the query execution plans are the same. We disable query caching, which does not execute a previously seen query at all. We do not want to compare caching of previously executed queries, but the efficiency of sharing among in-progress queries.

5.2 Sensitivity Analysis

In this section, we measure performance by evaluating multiple concurrent instances of SSB Q3.2. It is a typical star query that joins three of the four dimension tables with the fact table. The SQL template and the execution plan are shown in Figure 9. We select a single query template for our sensitivity analysis because we can adjust the similarity of the query mix to gain insight on the benefits of SP, and also, the GQP of CJOIN is the same for all experiments, with the same 3 shared hash-joins for all star queries.

5.2.1 Impact of concurrency

We start with an experiment that does not involve I/O accesses to study the computational behavior of the configurations. We store our database in a RAM drive. We evaluate multiple concurrent SSB Q3.2 instances for a scale factor 1. The predicates of the queries are chosen randomly, keeping a low similarity factor among queries and the selectivity of fact tuples varies from 0.02% to 0.16% per query. Figure 10 (left) shows the response times of the configurations, while varying the number of concurrent queries.

For low concurrency, QPipe successfully uses available CPU resources. Starting with as few as 32 concurrent queries, there is contention for CPU resources, due to the fact that our server has 24 cores and QPipe evaluates queries separately. Response times grow quickly and unpredictability results in standard deviations up to 50%. For 256 queries it uses all cores at their maximum.

The circular scans of QPipe-CS reduce contention for CPU resources and the buffer pool, improving performance. For high concurrency, however, there are more threads than available hardware contexts, thus increasing response time.
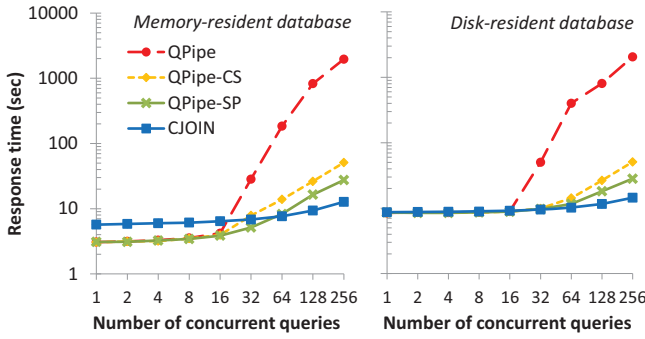
Measurement \Configur.	QPipe	QPipe-CS	QPipe-SP	CJOIN
Experiment with memory-resident database				
Avg. # Cores Used	23.91	19.72	18.75	3.47
Experiment with disk-resident database				
Avg. # Cores Used	23.86	19.84	17.06	3.49
Avg. Read Rate (MB/s)	1.88	74.47	97.67	156.11

Figure 10: Experiment with memory-resident (left) and disk-resident (right) database of SF=1. The table includes measurements for the case of 256 concurrent queries.

QPipe-CS misses several sharing opportunities at higher operators in the query plans. QPipe-SP can exploit them. Even though we use random predicates, the ranges of variables of the SSB Q3.2 template allows QPipe-SP to share the first hash-join 126 times, the second hash-join 17 times, and the third hash-join 1 time, on average for 256 queries. Thus, it saves more CPU resources, and results in lower response time than the circular scans alone.

The shared operators of CJOIN offer the best performance, as they are the most efficient in saving resources. CJOIN has an initialization overhead in comparison to the other configurations, attributed to its admission phase, which has a large part that pauses the pipeline (see Section 3.1). The shared hash-joins in the GQP can effortlessly evaluate many instances of SSB Q3.2. Nevertheless, admission and evaluation costs accumulate for an increasing number of queries, thus the CJOIN line also starts to degrade.

We do not depict CJOIN-SP, as it has the same behavior as CJOIN. As we noted in Section 3.3, our implementation of CJOIN-SP supports sharing CJOIN packets with all predicates identical. This is rare due to this experiment's random selection predicates.

Our observations apply also for the same experiment with the database on disk, shown in Figure 10 (right). QPipe suffers from CPU contention, which de-schedules scanner threads regularly resulting in low I/O throughput. Additionally, scanner threads compete for bringing pages into the buffer pool. Response times for low concurrency have increased, but not significantly for high concurrency because the workload becomes CPU-bound. QPipe-CS improves performance (by 80%-97% for high concurrency) by reducing contention for resources and the buffer pool. QPipe-SP further improves performance by eliminating common sub-plans. The shared operators of CJOIN still prevail for high concurrency. Furthermore, the overhead of the admission phase of CJOIN, that we observed for a memory-resident database, is masked by file system caches for disk-resident databases. We explore this effect in a next experiment, where we vary the scale factor.

Implications. Shared scans improve performance by reducing contention for resources, the buffer pool and the underlying storage devices. SP is able to eliminate common sub-plans. Shared operators in a GQP are more efficient in evaluating a high number of queries, in comparison to standard query-centric operators.
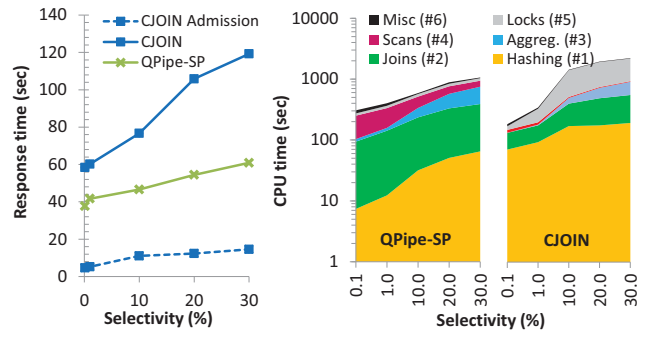
Measurement \Configuration	QPipe-SP	CJOIN
Avg. # Cores Used	17.79	18.86

Figure 11: 8 queries with a memory-resident database of SF=10. The table includes measurements for 30% selectivity.

5.2.2 Impact of data size

In this section, we study the behavior of the configurations by varying the amount of data they handle. We perform two experiments: In the first, we vary the selectivity of fact tuples of queries, and in the second, the scale factor.

Impact of selectivity. We use a memory-resident database with scale factor 10. The query mix consists of 8 concurrent queries which are different instances of a modified SSB Q3.2 template. For the modified template, we select the maximum possible range for the year. Moreover, we extend the WHERE clause of the query template by adding more options for both customer and supplier nation attributes. For example, if we use a disjunction of 2 nations for customers and 3 nations for suppliers, we achieve a selectivity of $\frac{2}{25}\frac{3}{25} \approx 1\%$ of fact tuples. Nations are selected randomly over all 25 possible values and are unique in every disjunction, keeping a minimal similarity factor. The results are shown in Figure 11. In this experiment, there is no contention for resources and no common sub-plans. Thus, we do not depict QPipe and QPipe-CS, as they have the same behavior as QPipe-SP, and we do not depict CJOIN-SP, as it has the same behavior as CJOIN.

Our selectivity experiments provide more insight on the general behavior of the configurations. For this reason, we also include the time of the admission phase of CJOIN, and performance breakdown graphs. The latter show the CPU time of all cores, as measured with Intel VTune Amplifier 2011, for different parts of the query evaluation. We compare the effect of sharing on the CPU time of hash-joins rather than analyze the bottlenecks of QPipe and CJOIN, which are largely dependent on implementation details. We further break down the CPU time of hash-joins to two categories. The first, shown as "Hashing", includes the total CPU time of the `hash()` and `equal()` functions, which are the heart of the building and probing phases, and allow us to compare the effect of sharing between the configurations, without strong side-effects from implementation details. The remaining CPU time of the hash-joins is shown as "Joins".

Both QPipe-SP and CJOIN show a degradation in performance as selectivity increases, due to the increasing amount of data they need to handle. CJOIN, however, is always worse than QPipe-SP. This is due to three reasons mainly. Firstly, the cost of the admission phase of CJOIN is increased, as more tuples are selected for referencing in the hash tables of the filters.

Secondly, the shared operators inherently entail a bookkeeping overhead, in comparison to standard query-centric operators. In our case, the additional cost of shared hash-joins includes the mainte-
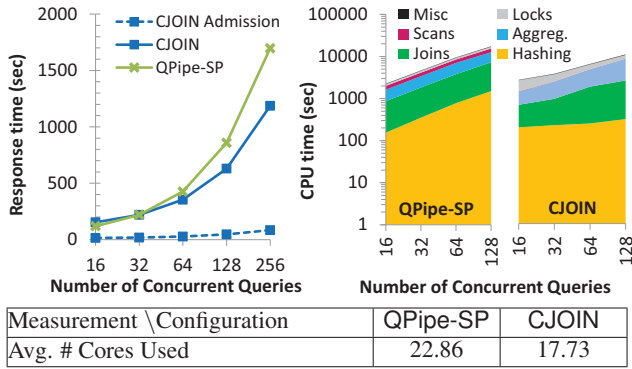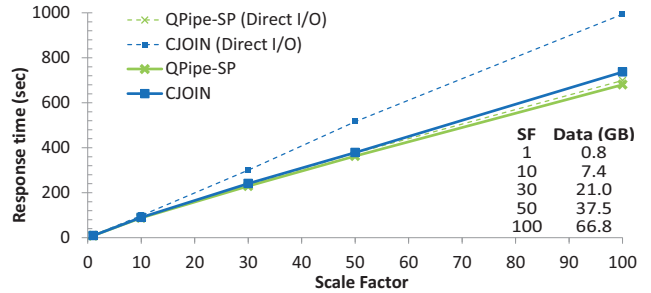
Measurement \ Configuration	QPipe-SP	CJOIN
Avg. # Cores Used	22.86	17.73

Figure 12: Memory-resident database of SF=10 and 30% selectivity. The table includes measurements for 256 queries.

nance of larger hash tables for the union of the selected dimension tuples of all concurrent queries, and bitwise AND operations between the bitmaps of tuples. Query-centric operators do not entail these costs, and maintain a hash table for one query. The increased bookkeeping costs are reflected in the CPU time of the area under Joins of CJOIN, which is more expensive than QPipe-SP for all cases of selectivity. As the selectivity increases, the hashing CPU time of QPipe-SP increases faster than CJOIN, as it does not share parts of the hash-joins of the concurrent queries. We note that the bookkeeping overhead can be decreased significantly with careful implementation choices. DataPath [2] uses a single large hash table for all shared hash-joins, and techniques to decrease the maintenance and access costs for the hash table.

Thirdly, the horizontal configuration of CJOIN (all threads in one "stage" [3]) results in synchronization costs, as threads contend while passing tuples through the pipeline. The synchronization costs are a significant reason for the worse trend of CJOIN. Nevertheless, synchronization costs are highly dependent on implementation. For example, in CJOIN, the synchronization costs can be minimized with the vertical (one thread per filter) or hybrid configuration of CJOIN. These configurations, however, do not necessarily provide better performance [3]. In DataPath or SharedDB, a shared operator in a GQP does not necessarily require multiple threads. Nevertheless, for low concurrency, the synchronization costs for a query are higher in a GQP than in the query-centric model, as a GQP tends to be much larger than the constituent query plans. For one query in a GQP, tuples not selected by it, but selected by other queries, need to pass through shared operators, and tuples selected by the query may need to pass by additional shared operators to accommodate other concurrent queries. This is also a reason why GQP achieve better throughput for high concurrency, but may hurt the latency of queries, especially for low concurrency.

We used 8 queries to avoid CPU contention. For higher concurrency, shared operators still prevail, due to their efficiency in saving resources. Figure 12 shows the response times for the case of 30% selectivity. For high concurrency, the query-centric operators of QPipe-SP contend for resources. This is also shown in the CPU times of the break-down graph, which all scale (superlinearly) with the number of queries. CJOIN is able to save more resources and outperform the query-centric operators. This is best reflected by the hashing CPU time, which stays at the same level, irrespective of the number of queries, as the hashing is shared.

Impact of scale factor. The same trade-off between shared operators and query-centric operators is observed by varying the scale factor between 1 to 100. We use disk-resident databases and 8 concurrent queries with randomly varied predicates and selectivity be-
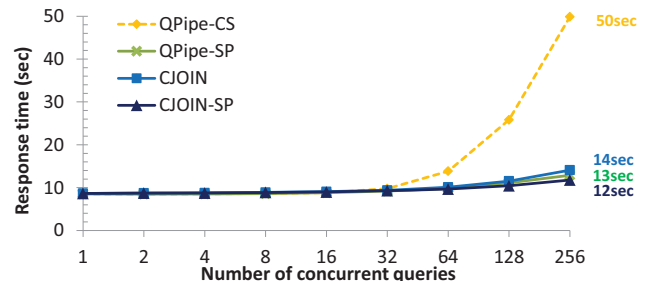
tween 0.02% and 0.16%. The results are shown in Figure 13. The response times of QPipe-SP and CJOIN increase linearly. Their slopes, however, are different. The reasons are the same as in our selectivity experiment.

We also show the response time of the two configurations by using direct I/O for accessing the database on disk, to bypass file system caches. This allows us to isolate the overhead of CJOIN's preprocessor. As we have mentioned, the preprocessor is in charge of the circular scan of the fact table, the admission phase of new queries, and finalizing queries when they wrap around to their point of entry. These responsibilities slow down the circular scan significantly. Without direct I/O, file system caches coalesce contiguous I/O accesses and read-ahead, achieving high I/O read throughput in sequential scans, masking the preprocessor's overhead.

Implications. For low concurrency, a GQP with shared operators entails a bookkeeping overhead in comparison to query-centric operators. For high concurrency, however, the overhead of shared operators is amortized.

5.2.3 Impact of Similarity

In this experiment we use a disk-resident database of scale factor 1. We limit the randomness of the predicates of queries to a small set of values. Specifically, there are 16 possible query plans for instances of Q3.2. The selectivity of fact tuples ranges from 0.02% to 0.05%. In Figure 14, we show the response times of the configurations, varying the number of concurrent queries. We do not depict

Measur. \ Configur.	QPipe-SP (Direct I/O)	CJOIN (Direct I/O)	QPipe-SP	CJOIN
# Cores Used	5.96	1.68	5.38	2.47
Read Rate (MB/s)	97.16	70.01	215.58	204.71

Figure 13: 8 concurrent queries with disk-resident databases. The table includes measurements for the case of SF=100.

Measur. \ Configur.	QPipe-CS	QPipe-SP	CJOIN	CJOIN-SP
Avg. # Cores Used	20.32	9.60	2.50	2.34
Avg. Read Rate (MB/s)	74.37	120.20	130.18	130.15

Figure 14: Disk-resident database of SF=1 and 16 possible plans. The table includes measurements for 256 queries.
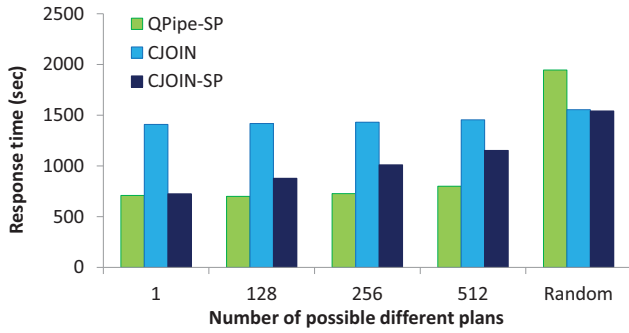
Differ. plans	1	128	256	512	Random
QPipe-SP	1/0/510	18/94/381	49/156/287	106/196/188	362/82/5
CJOIN-SP	510	384	287	190	12

Figure 15: Evaluating 512 concurrent queries with a varying similarity factor, for a SF=100. The table includes the SP sharing opportunities (average of all iterations), in the format 1st/2nd/3rd hash-join for QPipe-SP.

QPipe, as it does not exploit any sharing and results in increased contention and high response times for high concurrency.

QPipe-SP evaluates a maximum of 16 different plans and re-uses results for the rest of similar queries. It shares the second hash-join 1 time, and the third hash-join 238 times, on average, for 256 queries. This leads to high sharing and minimal contention for computations. On the other hand, QPipe-CS does not share operators other than the table scan, resulting in high contention.

Similarly, CJOIN misses exploiting these sharing opportunities and evaluates identical queries redundantly. In fact, QPipe-SP outperforms CJOIN. CJOIN-SP, however, is able to exploit them. For a group of identical star queries, only one is evaluated by the GQP. CJOIN-SP shares CJOIN packets 239 times on average for 256 queries. Thus, CJOIN-SP outperforms all configurations.

To further magnify the impact of SP, we perform another experiment for 512 concurrent queries, a scale factor of 100 (with a buffer pool fitting 10% of the database), and varying the number of possible different query plans. Figure 15 shows the results. CJOIN is not heavily affected by the number of different plans. For the extreme cases of high similarity, QPipe-SP prevails. For lower similarity, the number of different plans it needs to evaluate is larger and performance is deteriorated due to contention. CJOIN-SP is able to exploit identical CJOIN packets and improve performance of CJOIN by 20%-48% for cases with common sub-plans.

Implications. We can combine SP with a GQP to eliminate redundant computations and improve performance of shared operators for a query mix that exposes common sub-plans.

5.3 SSB query mix evaluation

In this section we evaluate QPipe-SP, CJOIN-SP, and Postgres using a mix of three SSB queries (namely Q1.1, Q2.1 and Q3.2), with a disk-resident database and a scale factor of 30. The predicates for the queries are selected randomly and the selectivity of fact tuples is less than 1%. Each query is instantiated from the three query templates in a round-robin fashion, so all configurations contain the same number of instances for each query type.

Figure 16 shows the response times of the configurations, while varying the number of concurrent queries. As Postgres is a more mature system than the two research prototypes, it attains a better performance for low concurrency. Our aim, however, is not to compare the per-query performance of the configurations, but their
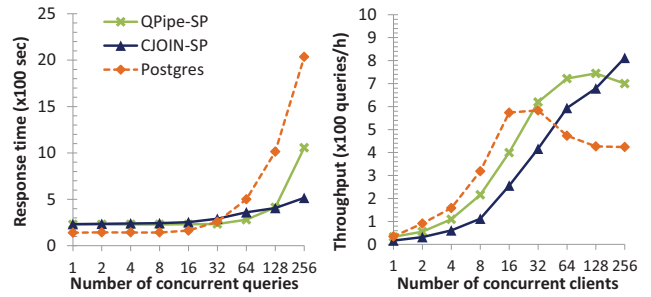
Measur. \ Configuration	Postgres	QPipe-SP	CJOIN-SP
Response time experiment (256 concurrent queries)			
Avg. # Cores Used	18.56	19.07	19.11
Avg. Read Rate (MB/s)	15.93	84.98	110.03
Throughput experiment (256 concurrent clients)			
Avg. # Cores Used	18.29	19.59	13.70
Avg. Read Rate (MB/s)	15.94	67.42	79.98

Figure 16: Disk-resident database of SF=30. Response time (left) and throughput experiment (right), varying the number of concurrent queries and clients respectively. The table include measurements for both experiments.

efficiency in sharing among a high number of concurrent queries. Postgres follows a traditional query-centric model of execution, and does not share among in-progress queries. For this reason, it results in contention for resources. QPipe-SP results in a better performance due to circular scans and the elimination of common sub-plans. CJOIN-SP attains the best performance, as shared operators are the most efficient in sharing among concurrent queries.

Figure 16 also shows the throughput of the three configurations, by varying the number of concurrent clients. Each client initially submits a query, and when it finishes, the next one is submitted. The shared operators of a GQP are able to handle new queries with minimal additional resources. Thus, the throughput of CJOIN-SP continues to increase. The throughput of the query-centric operators of Postgres and QPipe-SP, however, ultimately degrades with an increasing number of clients, due to resources contention.

6. DISCUSSION

Shared scans and SPL. Pull-based models, similar to SPL, have been proposed for shared scans that are specialized for efficient buffer pool management and are based on the fact that all data is available for accessing (see Section 2.1). SPL differ because they are generic and can be used during SP at any operator which may be producing results at run-time. It is possible, as well, to use shared scans for table scans, and use SPL during SP for other operators.

Prediction model for sharing with a GQP. Shared operators of a GQP are not beneficial for low concurrency, in comparison to the query-centric model, because they entail an increased bookkeeping overhead (see Section 5.2.2). The turning point, however, when shared operators become beneficial needs to be pinpointed. A simple heuristic is the point when resources become saturated (see Table 1). An exact solution would be a prediction model, similar to [14]. This model, however, targets only sharing identical results during SP (see Section 4). Shared operators in a GQP do not share identical results, but part of their evaluation among possibly different queries. A potential prediction model for a GQP needs to consider the bookkeeping overhead, and the cost of optimizing the GQP, for the current query mix and resources.

Distributed environments. This work focuses on scaling up rather than out. Following prior work [2, 3, 11, 13], we consider scaling up as a base case, because it is a standard means of increasing throughput in DBMS. Further research in parallel DBMS [20] and other distributed data systems [10] will have interesting implications. For example, we can improve global scheduling in parallel DBMS by considering sharing: each replica node can employ a separate GQP, and a new query should be dispatched to the node which incurs the minimum estimated marginal cost for evaluation.

7. CONCLUSIONS

In this paper we perform an experimental study to answer *when* and *how* an execution engine should share data and work across concurrent analytical queries. We review work sharing methodologies and we study Simultaneous Pipelining (SP) and Global Query Plans with shared operators (GQP) as two state-of-the-art sharing techniques. We perform an extensive evaluation of SP and GQP, based on their original research prototype systems.

Work sharing is typically beneficial for high concurrency because the opportunities for common work increase, and it reduces contention for resources. For low concurrency, however, there is a trade-off between sharing and parallelism, particularly when the sharing overhead is significant. We show that GQP are not beneficial for low concurrency as shared operators inherently involve a bookkeeping overhead compared to query-centric ones. For SP, however, we show that it can be beneficial for low concurrency as well, if the appropriate communication model is employed: we introduce SPL, a pull-based approach that scales better on machines with modern multi-core processors than push-based SP. SPL is a data structure that promotes parallelism by shifting the responsibility of sharing common results from the producer to the consumers.

Furthermore, we show that SP and GQP are two orthogonal sharing techniques and their integration allows to share operators and handle a high number of concurrent queries, while also sharing any common sub-plans presented in the query mix. In conclusion, analytical query engines should dynamically choose between query-centric operators with SP for low concurrency and GQP with shared operators enhanced by SP for high concurrency.

8. REFERENCES

[1] TPC-H Benchmark: Standard Specification, Revision 2.14.3.

[2] S. Arumugam et al. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proc. of the 2010 ACM SIGMOD Int'l Conf. on Management of Data*, pages 519–530, 2010.

[3] G. Candea et al. A scalable, predictable join operator for highly concurrent data warehouses. *Proc. of the VLDB Endowment*, 2(1):277–288, 2009.

[4] G. Candea et al. Predictable performance and high query concurrency for data analytics. *The Int'l Journal on Very Large Data Bases*, 20(2):227–248, 2011.

[5] H.-T. Chou et al. An evaluation of buffer management strategies for relational database systems. In *Proc. of the 11th Int'l Conf. on Very Large Data Bases*, pages 127–141, 1985.

[6] J. Cieslewicz et al. Adaptive aggregation on chip multiprocessors. In *Proc. of the 33rd Int'l Conf. on Very Large Data Bases*, pages 339–350, 2007.

[7] L. Colby et al. Red brick vista^{TM}: aggregate computation and management. In *Proc. of the 14th Int'l Conf. on Data Engineering*, pages 174–177, 1998.

[8] C. Cook. Database Architecture: The Storage Engine, 2001. http://msdn.microsoft.com/library/aa902689(v=sql.80).aspx.

[9] N. N. Dalvi et al. Pipelining in multi-query optimization. In *Proc. of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Databases*, pages 59–70, 2001.

[10] J. Dean et al. MapReduce: Simplified data processing on large clusters. *Communications ACM*, 51(1):107–113, 2008.

[11] G. Giannikis et al. SharedDB: killing one thousand queries with one stone. *Proc. of the VLDB Endowment*, 5(6):526–537, 2012.

[12] S. Harizopoulos et al. A case for staged database systems. In *Proc. of the 2003 Conf. on Innovative Data Systems Research*, 2003.

[13] S. Harizopoulos et al. QPipe: a simultaneously pipelined relational query engine. In *Proc. of the 2005 ACM SIGMOD Int'l Conf. on Management of Data*, pages 383–394, 2005.

[14] R. Johnson et al. To share or not to share? In *Proc. of the 33rd Int'l Conf. on Very Large Data Bases*, pages 351–362, 2007.

[15] R. Johnson et al. Shore-MT: a scalable storage manager for the multicore era. In *Proc. of the 12th Int'l Conf. on Extending Database Technology: Advances in Database Technology*, pages 24–35, 2009.

[16] T. Johnson et al. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of the 20th Int'l Conf. on Very Large Data Bases*, pages 439–450, 1994.

[17] R. Kimball et al. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2nd edition, 2002.

[18] C. Lang et al. Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling. In *Proc. of the 23rd Int'l Conf. on Data Engineering*, pages 1136–1145, 2007.

[19] N. Megiddo et al. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of the 2nd USENIX Conf. on File and Storage Technologies*, pages 115–130, 2003.

[20] M. Mehta et al. Batch Scheduling in Parallel Database Systems. In *Proc. of the 9th Int'l Conf. on Data Engineering*, pages 400–410, 1993.

[21] P. O. Neil et al. Star Schema Benchmark. 2009.

[22] E. J. O'Neil et al. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the 1993 ACM SIGMOD Int'l Conf. on Management of Data*, pages 297–306, 1993.

[23] L. Qiao et al. Main-memory scan sharing for multi-core cpus. *Proc. of the VLDB Endowment*, 1(1):610–621, 2008.

[24] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

[25] P. Roy et al. Efficient and extensible algorithms for multi query optimization. In *Proc. of the 2000 ACM SIGMOD Int'l Conf. on Management of Data*, pages 249–260, 2000.

[26] P. Russom. High-Performance Data Warehousing. *TDWI*, 2012. http://tdwi.org/research/2012/10/tdwi-best-practices-report-high-performance-data-warehousing.aspx.

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

[28] J. Shim et al. Dynamic Caching of Query Results for Decision Support Systems. In *Proc. of the 11th Int'l Conf. on Scientific and Statistical Database Management*, pages 254–263, 1999.

[29] P. Unterbrunner et al. Predictable performance for unpredictable workloads. *Proc. of the VLDB Endowment*, 2(1):706–717, 2009.

[30] M. Zukowski et al. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proc. of the 33rd Int'l Conf. on Very Large Data Bases*, pages 723–734, 2007.