Reactive and Proactive Sharing Across Concurrent Analytical Queries

Iraklis Psaroudakis Manos Athanassoulis Matthaios Olma Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne
{iraklis.psaroudakis, manos.athanassoulis, matthaios.olma, anastasia.ailamaki}@epfl.ch

ABSTRACT

Today an ever increasing amount of data is collected and analyzed by researchers, businesses, and scientists in data warehouses (DW). In addition to the data size, the number of users and applications querying data grows exponentially. The increasing concurrency is itself a challenge in query execution, but also introduces an opportunity favoring synergy between concurrent queries. Traditional execution engines of DW follows a *query-centric* approach, where each query is optimized and executed independently. On the other hand, workloads with increased concurrency have several queries with common parts of data and work, creating the opportunity for sharing among concurrent queries. Sharing can be *reactive* to the inherently existing sharing opportunities, or *proactive* by redesigning query operators to maximize the sharing opportunities.

This demonstration showcases the impact of proactive and reactive sharing by comparing and integrating representative state-of-the-art techniques: *Simultaneous Pipelining (SP)*, for reactive sharing, which shares intermediate results of common sub-plans, and *Global Query Plans (GQP)* for proactive sharing, which build and evaluate a single query plan with shared operators. We visually demonstrate, in an interactive interface, the behavior of both sharing approaches on top of a state-of-the-art storage engine using the original prototypes. We show that pull-based sharing for SP eliminates the serialization point imposed by the original push-based approach. Then, we compare, through a sensitivity analysis, the performance of SP and GQP. Finally, we show that SP can improve the performance of GQP for a query mix with common sub-plans.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Design, Experimentation, Performance

Keywords

Data Warehouses, Query Processing, Data Sharing, Work Sharing, Simultaneous Pipelining, Shared Pages List, Global Query Plans, Reactive Sharing, Proactive Sharing, QPipe, CJOIN

1. INTRODUCTION

Today, in the era of big data, organizations are called to process an ever-increasing mass of data to deduce valuable information. To meet their needs, organizations employ data warehouses (DW), which are specialized databases for serving online analytical processing (OLAP) workloads. OLAP queries are mostly ad-hoc, long-running, and scan-heavy queries.

The execution engine of a traditional DW typically evaluates the mix of concurrent analytical queries using a query-centric model: each query is optimized and evaluated independently of the other queries in the system. This approach misses opportunities of sharing common parts of data and work across the query mix. Taking advantage of these opportunities can further enhance the performance of highly-concurrent DW, by reducing contention for I/O, CPU, and RAM resources, and freeing up resources.

Our study shows the performance benefits from state-of-the-art sharing techniques. We explore how and when these techniques should be employed, we make a head-to-head comparison and we combine them. The first study and implementation was presented at VLDB 2013 [8]. In this work, we demonstrate the sharing techniques in a unified system, and provide to the audience the opportunity to visually assess the effect of the sharing techniques on performance, and trigger various workload parameters for an interactive sensitivity analysis.

Overview of the sharing techniques. The demonstration evaluates two fundamental strategies for sharing: (a) *reactive sharing* which acts using the sharing opportunities that inherently exist in the workload (e.g. common sub-plans) and (b) *proactive sharing* which maximizes work and data sharing (e.g, by analyzing the workload and building synergistically shared operators in a global query plan). Throughout our experimentation and analysis we use two representative state-of-the-art techniques. For reactive sharing we use Simultaneous Pipelining (SP) by integrating its original research prototype QPipe [5], and for proactive sharing we use using Global Query Plans (GQP) with shared operators, by integrating the original research prototype CJOIN [2]. Both techniques use shared scans. In the remainder of the paper we refer to the sharing strategies using the respective techniques: SP for reactive sharing and GQP for proactive sharing.

SP identifies common sub-plans among concurrent queries at run-time, evaluates only one and pipelines the results to the rest simultaneously. Figure 1a shows 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.

Though SP is limited to common sub-plans with identical predicates, a shared operator in a GQP can handle multiple concurrent

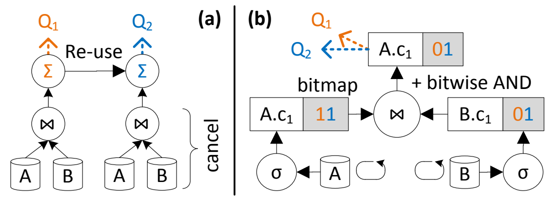


Figure 1: (a) SP on two queries having a common sub-plan below a join. (b) Example of shared scans and a shared join operator in a GQP. The GQP evaluates two queries with the same join predicate but possibly different selection predicates.

queries with similar plans but potentially different predicates. 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, the bits of which signify whether the tuple is relevant to one of the queries. Figure 1b gives an example of two queries that join two relations using the same join predicate, but have different selection predicates for both relations. The shared scans attach a bitmap to each scanned tuple, calculated after evaluating the selection predicates of the queries, and the shared join operator performs a bitwise AND operation between the bitmaps of the joined tuples to preserve their correlation to the queries.

Improving SP. In order to overcome the negative effects of aggressive sharing with SP [6] we study the mechanism of copying common results among operators. We show that the push-based model for SP creates a serialization point during sharing, and we address this bottleneck by introducing the novel Shared Pages List (SPL) data structure which implements a pull-based model of SP [8], allowing a single producer and multiple consumers accessing the data. The demonstration allows the audience to verify the serialization point of the push-based model, and dynamically trigger the usage of SPL to evaluate its impact (see Section 4.3).

Comparing SP vs. GQP. We integrate the original research prototypes that introduced SP and GQP into one system: we integrate the CJOIN operator as an additional stage of the QPipe execution engine (see Section 3) on top of the Shore-MT storage manager [7]. Thus, we can dynamically decide whether to evaluate multiple concurrent queries with the standard query-centric operators of QPipe (with or without SP) or the shared operators of the GQP of CJOIN. The demonstration allows the audience to dynamically choose which technique to use when evaluating an analytical workload, under a combination of adjustable parameters, and visually assess the differences between the two sharing techniques (see Section 4.4). The live comparison gives the audience the opportunity to validate the rules of thumb presented in the experimental comparison of SP vs. GQP [8]. Shared operators enhance performance for workloads with high concurrency, but their high book-keeping overhead in comparison with query-centric operators, overwhelms performance for workloads with low concurrency. On the other hand, SP enhanced by SPL is beneficial for both low and high concurrency.

Combining SP and GQP. SP and GQP are orthogonal sharing techniques that can be combined to get the best of the two worlds. SP can be applied on the shared operators of a GQP (e.g., aggregations), further improving the performance of a GQP in a query mix with common sub-plans, as it avoids reevaluating them through the GQP, and instead it reuses the common intermediate results. The demonstration showcases an analytical workload where the audience can dynamically trigger the number of common sub-plans in the mix and assess the effect of SP on GQP.

Visual experience. This demonstration includes a graphical interface that allows the audience to change system and workload characteristics in order to assess the effect of the different techniques of sharing on the execution performance of concurrent analytical queries. Workload parameters, such as the number of concurrent queries or their similarity, affect performance, and the audience can explore different combinations of parameters. The output is shown with a set of dynamically generated graphs, accompanied by system measurements (e.g. CPU times).

2. RELATED WORK

Sharing in the I/O layer. By sharing data, the accesses of queries in the I/O layer are coordinated. The typical DW uses buffer pool management techniques and eviction policies. More recently, shared scans [8] have been proposed to better co-ordinate multiple queries that scan the same relation, reducing buffer pool contention and avoiding unnecessary I/O. Both QPipe and CJOIN use a simple form of shared scans, circular scans [5].

Sharing in the execution engine. By sharing work among queries, we refer to techniques that avoid redundant computations inside the execution engine. A typical DW uses query caching, materialized views, or Multi-Query Optimization (MQO) techniques. These techniques, however, do not exploit sharing opportunities among in-progress queries. Both SP and GQP provide deeper and more dynamic forms of sharing at run-time. The main aim of our demonstration is to showcase these sharing techniques.

For both techniques, we use their original research prototype systems: QPipe for SP, and the CJOIN operator for GQP. CJOIN is restricted to the evaluation of star queries, with a GQP composed of shared scans and shared hash-joins. After CJOIN, more recent research prototypes (DataPath [1] and SharedDB [4]) have advanced GQP to more general schemas and operators. Without loss of generality, we restrict our evaluation of GQP with CJOIN to star schemas, which are very common in relational DW, and correlate our observations to more general schemas and operators [8].

3. DEMONSTRATED SYSTEMS

In this section, we give quick overview of the original systems that implemented SP and GQP: the QPipe execution engine and the CJOIN operator respectively. We also explain how we improve QPipe with a pull-based model of SP, and how we integrate the CJOIN operator into the QPipe execution engine to take advantage of both sharing techniques.

The QPipe execution engine. In QPipe, each relational operator is encapsulated into a self-contained module called a *stage*, which has a queue for work requests and employs a local thread pool for processing them. A query plan is converted to a series of interdependent *packets*, dispatched to the relevant stages. Data flow between packets is implemented through FIFO buffers and page-based exchange, following a push-only model with pipelined execution (as shown in [3]). SP is implemented at execution time at any stage, when an incoming packet has a common sub-plan with an ongoing packet, by copying the results of the ongoing packet to the FIFO buffer of the incoming packet. SP may not be activated for certain cases of relational operators and query inter-arrival delays [5].

Shared Pages List. The push-based model using intermediate FIFO buffers involves a serialization point during SP when the single producer forwards the common intermediate results to multiple consumers. This bottleneck makes SP beneficial only for cases of high concurrency, where sharing proves better than not sharing by avoiding contention for resources [6]. A pull-based model for SP

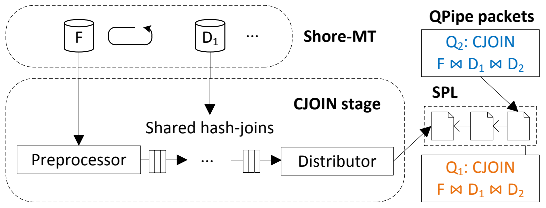


Figure 2: Integration of CJOIN into QPipe for the evaluation of star queries joining the fact table with dimension tables.

virtually eliminates this serialization point, by sharing intermediate results instead of forwarding them [8]. To achieve this, an intermediate data structure, the *Shared Pages List* (SPL), replaces FIFO buffers, allowing a single producer and multiple consumers.

The CJOIN operator. CJOIN evaluates the joins of concurrent star queries, using a GQP with shared scans, shared selections and shared hash-joins. Due to the semantics of star schemas, the directed acyclic graph of the GQP takes the form of a chain [8]. CJOIN exploits this form and evaluates a single pipeline: the *preprocessor* uses a circular scan of the fact table, and flows fact tuples through the pipeline. The shared hash-joins in-between 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.

Integration. We integrate the CJOIN operator into the QPipe execution engine as a new stage, using Shore-MT [7] as the underlying storage manager. CJOIN supports only shared hash-joins, hence, any subsequent operators are query-centric. Having CJOIN integrated into QPipe, we can enable SP for the CJOIN stage, combining the two sharing techniques. Figure 2 shows the new stage that encapsulates the CJOIN pipeline, and how two star queries, having a common sub-plan under their CJOIN packets, share the intermediate results with a SPL. Only Q_1 is evaluated by CJOIN, saving admission costs and unnecessary book-keeping costs for Q_2.

4. DEMONSTRATION WALKTHROUGH

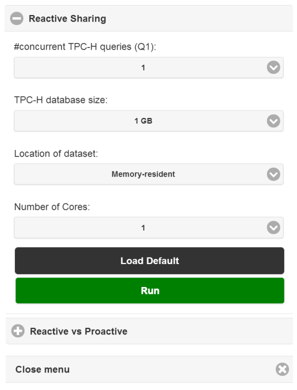
The demonstration will be executed using a poster, and an interactive graphical application. The application will showcase a number of predefined scenarios, that are executed on the spot, to demonstrate the advantages of the sharing techniques, their various aspects and trade-offs. Apart from the predefined scenarios, the audience will be able to modify the parameters to create custom scenarios, and get an insight on the effect of the parameters on the overall performance of analytical workloads.

4.1 Poster

We use a poster to introduce the audience to the traditional query-centric model, SP, and GQP. We explain how the query-centric model misses sharing opportunities across concurrent analytical queries, and how SP and GQP can take advantage of sharing opportunities. We also introduce the user to the general design of the original prototype systems QPipe and CJOIN, how we integrate the CJOIN operator as an additional stage in QPipe, and how the SPL data structure supports a pull-based version of SP. Finally, we include key plots of the predefined scenarios (detailed in the following sections) and explain their implications.

4.2 System setup

The graphical web interface runs on a dedicated virtual machine in our lab and can be accessed by a laptop at the conference, where the audience can define the parameters for the scenarios, and visualize the results. The experiments run remotely on multiple machines hosting multiple instance of our database server at EPFL. We use three machines, in parallel, to measure the response time or throughput of the different configurations required for the following scenarios. Each machine has two 8-core Intel Xeon E5-2660 processors (with hyper-threading enabled), 128GB RAM, and seven 300GB 15kRPM SAS 3.5" hard disks. The O/S is a 64-bit SMP Linux (Red Hat), with a 2.6.32 kernel.


Reactive Sharing

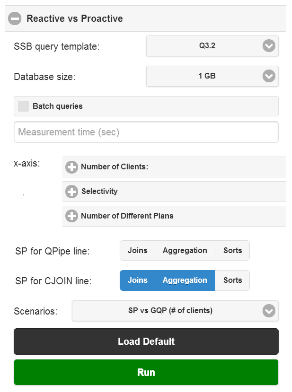
#concurrent TPC-H queries (Q1): 1

TPC-H database size: 1 GB

Location of dataset: Memory-resident

Number of Cores: 1

(a) Scenario I


Reactive vs Proactive

SSB query template: Q3.2

Database size: 1 GB

Batch queries

Measurement time (sec)

x-axis: Number of Clients:

Selectivity

Number of Different Plans

SP for QPipe line: Joins | Aggregation | Sorts

SP for CJOIN line: [✓ Joins] [✓ Aggregation] Sorts

Scenarios: SP vs GQP (# of clients)

(b) Scenario II-IV

Figure 3: Execution options

4.3 Scenario I: Push-based vs. Pull-based SP

The aim of this scenario is to show the advantages of a pull-based model of SP, using our new SPL data structure, versus the original push-based model of SP. Sharing intermediate results with push-based SP proves to be beneficial when executing concurrent queries, only when the system does not have available parallelism to execute each query independently because the overhead to copy the results dominates the response time. Pull-based sharing with SPL, however, minimizes the copying overhead and is beneficial even when the system has available parallelism. The audience can see this behavior by plotting the response time of concurrent queries and binding the database server process to a minimum of 1 and a maximum of 32 cores. In addition, the interactive interface allows to experiment with different database sizes, with memory-resident and disk-resident databases, with different number of concurrent queries, and with different buffer-pool sizes for the case of the disk-resident database.

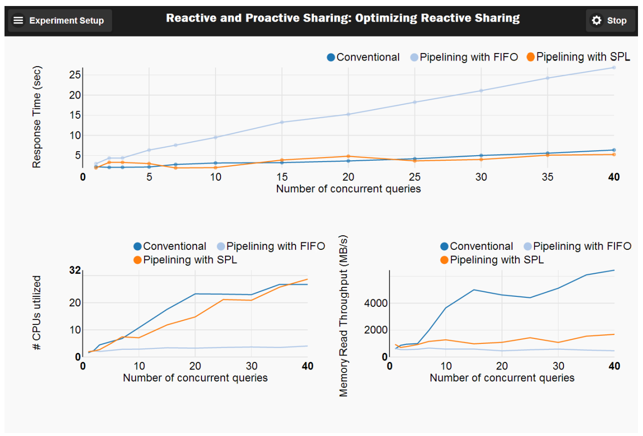


Figure 4: GUI for Scenario I

The default scenario demonstrates the performance of the two models using an experiment which evaluates SP for the table scan stage with a memory-resident database [8]. Specifically, we use identical TPC-H Q1 instances, submitted at the same time. Figure 3(a) shows an example of the GUI for the default scenario where the user can alter the parameters of the experiment, and Figure 4 shows how the experiment execution will be monitored by showing for each level of concurrency (i) the response time of the workload, (ii) the CPU utilization, and (iii) the I/O (memory bus) throughput for the disk-resident (memory-resident) dataset.

The serialization point of the push-based SP (FIFO) increases the response time with additional concurrent queries, while CPU load does not increase significantly. The pull-based SP (SPL) overcomes the serialization point, and is able to fully use CPU resources. The query-centric execution has marginally lower response time than the pull-based approach when the available parallelism is lower than the concurrency of the workload, but pull-based sharing improves performance for higher concurrency.

4.4 Scenarios II-IV: SP vs GQP

The rest of the scenarios explore the trade-offs of the two sharing techniques, SP and GQP. The workload now consists of queries issued concurrently by a number of clients following instantiations of a Star Schema Benchmark (SSB) template. Each client submits queries iteratively. We present performance in terms of throughput. Figure 3(b) shows the GUI that the user can alter the different starting parameters building different execution scenarios. Figure 5 shows the output of a sample execution for these scenarios.

The dark blue line shows the performance of the QPipe execution engine and query-centric relational operators. On the configuration pane, the user can enable SP for the different stages of QPipe. Without SP for any stage, the QPipe engine is similar to a query-centric execution engine with shared scans. The light blue line shows the performance of the QPipe execution engine with the CJOIN operator evaluating the hash-joins of all concurrent star queries in the workload. By enabling SP for the CJOIN stage, the user can assess the performance of the combination of the two sharing techniques.

The rest of the configuration parameters allow the audience to differentiate from the default scenarios II-IV, and create custom ones. The x-axis can be selected among the number of concurrent clients, the selectivity of the queries, and the number of possible different plans for the selected query template (affecting the number of common sub-plans in the query mix for SP). The user can also modify which SSB query template to be used, the scale factor of the dataset, whether the database is memory-resident or disk-resident, whether clients should co-ordinate to submit their queries in batches (ensures maximal SP sharing and decreases admission costs for GQP), and the throughput measurement duration for each point of the selected x-axis.

Scenario II: Impact of concurrency. This scenario fixes selectivity to 1%, randomizes the template's parameters to decrease the efficiency of SP, and modifies the number of concurrent clients (x-axis). The database is disk-resident, and SP is enabled for all stages for both lines. Here we show that shared operators in a GQP are more efficient in evaluating a high number of concurrent queries in comparison to standard query-centric operators.

Scenario III: Impact of selectivity. This scenario randomizes the template's parameters to decrease the efficiency of SP, fixes the number of concurrent clients to a low concurrency value, and modifies the parameters' selectivity (x-axis). The database is memory-resident, and SP is enabled for all stages for both lines. The aim of this scenario is to show that shared operators in a GQP have a high bookkeeping overhead in comparison to query-centric operators.

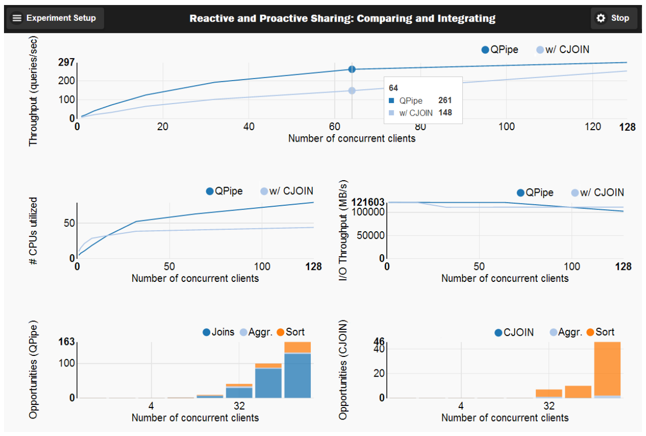


Figure 5: GUI for Scenarios II-IV

Scenario IV: Impact of similarity. This scenario fixes the number of concurrent clients to a high concurrency value, fixes the selectivity, and modifies the number of possible different plans (x-axis) to restrict their randomness. Fewer available plans translates to more common sub-plans for the same number of clients, while a higher number of available plans translates to fewer common sub-plans and fewer SP opportunities. The database is disk-resident, and SP is enabled for all stages for both lines. Batching is used to maximize SP opportunities. The aim of this scenario is to show that we can combine SP with a GQP to improve the performance of shared operators for a query mix with common sub-plans. The measurements with regards to the number of SP opportunities exploited for each stage comprise the most significant metric for this scenario.

5. CONCLUSIONS

In this demonstration, we show how and when reactive and proactive work sharing outperforms query-centric execution in the context of analytical workloads. The demonstration is composed of a poster and an interactive graphical user interface that allows the audience to assess the performance benefits of the sharing techniques, how they compare against each other, and how they can be combined to exploit both techniques.

Acknowledgments. This work was supported by the FP7 project BIGFOOT (grant n. 317858).

6. REFERENCES

[1] 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. *SIGMOD*, 2010.

[2] G. Candea, N. Polyzotis, and R. Vingralek. Predictable performance and high query concurrency for data analytics. *VLDBJ*, 2011.

[3] K. Gao, S. Harizopoulos, I. Pandis, V. Shkapenyuk, and A. Ailamaki. Simultaneous Pipelining in QPipe: Exploiting Work Sharing Opportunities Across Queries. *ICDE*, 2006.

[4] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries with One Stone. *VLDB*, 2012.

[5] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: a simultaneously pipelined relational query engine. *SIGMOD*, 2005.

[6] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To share or not to share? *VLDB*, 2007.

[7] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. *EDBT*, 2009.

[8] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing data and work across concurrent analytical queries. *VLDB*, 2013.