Abstract—Traditionally, query execution engines in relational databases have followed a query-centric model: They optimize and execute each incoming query using a separate execution plan, independent of other concurrent queries. For workloads with low contention for resources, or workloads with short-lived queries, this model makes the optimization phase faster and creates efficient execution plans. For workloads with heavy contention, or workloads with long-running analytical queries, this model cannot exploit the sharing opportunities that might exist among concurrent queries in order to save I/O, CPU and RAM resources. We argue that exploiting these sharing opportunities is a crucial step towards handling these increasingly common workloads.

In this paper, we study three research prototype systems that employ various methodologies for sharing data and work: (a) The QPipe query execution engine [1], which employs a circular scan per table and shares work through simultaneous pipelining, (b) the DataPath system [2], which employs an uninterrupted linear scan per disk and shares work through a global query plan, and (c) the SharedDB system [3], which employs a circular scan per table partition, shares work through a global query plan, uses batched execution, and services both OLTP and OLAP workloads under response time guarantees. We classify these methodologies, analyze their commonalities and differences, and identify their strengths and shortcomings.

Index Terms—Database systems, Data warehouses, Query processing, Work sharing, Data sharing

I. INTRODUCTION

The early relational database management systems (DBMS) of the 70’s, and many modern ones following their trail, are architectured mainly for online transaction processing (OLTP) workloads. They service numerous short-lived queries that read and update the database. A representative example are online banking applications. During the 90’s, however, a different form of data analysis emerged. Enterprises started extracting massive data, transforming it and loading into centralized DBMS, namely data warehouses (DW), to be used in conjunction with business intelligence tools. The term online analytical processing (OLAP) was coined for these analytical workloads. In contrast to OLTP, they are characterized by relatively static data (new data is periodically loaded), and consist mostly of ad-hoc, long-running, read-only, scan-heavy queries that may involve aggregations, sorting or joins of multiple tables [4].

Today, in the era of data deluge and multicore processors, both OLTP and OLAP systems need to be radically optimized for the new hardware architectures. The design choices made for one may be inefficient for the other [5]. It is increasingly common for recent analytical engines to depart from design choices that are traditionally linked to OLTP workloads. For example, several analytical engines may prefer column-stores, relax the ACID properties of transactions, or deal with data primarily in main memory.

In that context, we argue that the analytical query execution engine needs to depart from its typical design choices. Traditionally, the execution engine of a DBMS has followed a query-centric model: For each query, it uses a cost-based optimizer, with trails back to System R [6], to select the most efficient alternative execution plan. The optimization and execution phases are independent of other concurrent queries. Concurrent queries, however, often exhibit overlapping data accesses or computations. For example, they might access the same relations or have similar sub-plans.

For workloads without heavy contention for resources, or workloads with short-lived OLTP transactions, the query-centric model makes the optimization phase faster and results in efficient execution plans. For workloads with heavy contention, however, or workloads with long-running OLAP queries, it cannot exploit the sharing opportunities that might exist among concurrent queries. Thus it is not able to save I/O, CPU and RAM resources. We argue that actively exploiting these sharing opportunities is a crucial step towards handling these increasingly common workloads against large data sets.
TABLE I. SHARING METHODOLOGIES EMPLOYED BY TRADITIONAL SYSTEMS AND THE RESEARCH PROTOTYPE SYSTEMS WE STUDY.

<table>
<thead>
<tr>
<th>System</th>
<th>Query Caching, Materialized Views, MQO</th>
<th>Simultaneous Pipelining</th>
<th>Global Query Plan (Star Schemas)</th>
<th>Global Query Plan (with Batched Execution)</th>
<th>Global Query Plan (with Batched Execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing in the execution engine</td>
<td>QPipe</td>
<td>CJOIN</td>
<td>DataPath</td>
<td>SharedDB</td>
<td></td>
</tr>
<tr>
<td>Storage Manager</td>
<td>Any</td>
<td>Any</td>
<td>Any</td>
<td>Any</td>
<td>Special I/O Subsystem (read-only requests)</td>
</tr>
</tbody>
</table>

A. Methodologies for data and work sharing

In Table I, we present the methodologies for sharing data and work, used by traditional query-centric systems and the research prototype systems we will examine.

For sharing data, a DBMS typically incorporates buffer pool management techniques. Most of them, however, cannot maximize data sharing. Queries typically communicate with the buffer pool manager on a per-page basis, thus it is difficult to detect the patterns that might exist among concurrent queries. Scan sharing techniques have been proposed [7], [8], [4] for scan-heavy workloads, that exploit common access patterns. The systems we examine leverage similar techniques. QPipe uses a circular scan per table. DataPath uses an uninterrupted linear scan per disk, skipping unneeded data. SharedDB uses the Crescendo storage manager [9], which performs a circular scan of in-memory table partitions, interleaving read and update requests. Its aim is to service both OLTP and OLAP workloads.

For sharing work, a DBMS typically employs query results caching [10] and materialized views [11]. These techniques, however, are ignorant of the execution state and do not exploit sharing opportunities that might exist among concurrent queries. Multi-Query Optimization (MQO) techniques [12], [13] are an important step towards more sophisticated sharing methodologies. MQO detects common subexpressions among queries in the optimization phase, which are evaluated only once during the execution phase. MQO lengthens the optimization phase and is suitable for long-running analytical queries, as the savings might more than offset this overhead. The disadvantages of classic MQO are that it is applied on batches of queries in the optimization phase, and it depends on costly materializations of the common intermediate results. By using pipelined execution, the costly materialization of intermediate results can be alleviated [14]. Additionally, pipelined execution takes advantage of the aggressive parallelization of work provided by multicore processors. The query plan is divided into sub-plans and relational operators are evaluated in parallel.

The systems we study leverage forms of pipelined execution and sharing methodologies which bear some resemblance to MQO. They use, however, deeper and more dynamic forms of work sharing. QPipe detects identical sub-plans at run-time, at the operator level, even if concurrent queries arrived at different times. It evaluates the sub-plans once and pipelines the results to all interested queries. DataPath uses a single global query plan to share work. When a new query is submitted, an optimizer incorporates it into the global plan in a way that minimizes additional data movement. SharedDB uses a similar global plan. Contrary to DataPath, it pre-computes the global plan, and uses batched execution to facilitate sharing.

B. The global query plan

In the systems we examine, global plans refer to the ability of an operator to share work by evaluating several queries simultaneously. They sacrifice parallelization of similar queries in order to save resources, as such they differ from the ones introduced by classic MQO [12] to share common subexpressions. Though query latencies may increase, they can handle many more queries than query-centric designs, thus increasing overall throughput. They are best suited for workloads with numerous queries having similar plans.

Annotating data with query identifiers is the basic technique used when evaluating a global plan. Every data tuple is correlated with a bitmap, whose bits signify if the tuple is relevant to a specific query. This technique facilitates overall sharing. For data sharing, it avoids copying as interested queries read the same tuple and skip it if it is not relevant to them. For work sharing, suppose that a few queries have the same sub-plan below an operator, with possibly different predicates. The shared operator can evaluate them simultaneously. Input tuples have their bitmaps already modified accordingly by the operators of the sub-plans. The operator examines an input tuple’s relevant queries, and evaluates its unit of work, considering their predicates.

For example, a selection operator can be shared by concurrent queries that select tuples from the same relation. It modifies the bitmap of each tuple according to the selection predicates of the queries. Also, a hash-join can be shared by concurrent queries that have the same join predicate. The shared hash-join only performs an additional bitwise AND operation between the bitmaps of the joined tuples.

In the sections of DataPath and SharedDB, we give more details on how they share work through a global plan. Here, we briefly illustrate how a global plan works in the context of the CJOIN operator [15], that instigated their usage. CJOIN is addressed to relational data warehouses where data is organized in a star schema [16]. A star schema consists of a single large fact table and is linked through foreign-key relationships with numerous smaller dimension tables. CJOIN evaluates the joins of star queries, which join the fact table with dimension tables.

In Figure 1, we show an example of two star queries. The corresponding global plan is composed of shared selection and hash-join operators that can easily handle more instances of $Q_1$ and $Q_2$ with different selection predicates. Due to the semantics of star schemas, the directed acyclic graph (DAG) of the global plan takes the form of a chain. Furthermore, CJOIN materializes the small dimension tables and stores selected tuples in the hash tables of the corresponding shared hash-join operators (called filters). When a new star query is submitted, CJOIN pauses execution, adds newly referenced filters, and
updates the filters by modifying the bitmaps of their tuples.

Consequently, the global plan is transformed into a single pipeline, depicted in Figure 1. CJOIN uses a circular scan of the fact table, and flows fact tuples through the pipeline. The filters in-between perform the actual join of the fact tuples with the corresponding dimension tuples and additionally perform a bitwise AND between their bitmaps. At the end of the pipeline, CJOIN examines the bitmaps of the fact tuples and forwards them to the relevant queries. For every new query, CJOIN marks its point of entry on the circular scan and signifies its completion when it wraps around to that point.

This paper is organized as follows. We analyze and compare the QPipe execution engine, the DataPath and SharedDB systems in Sections II, III, IV respectively. In Section V we describe our research proposal and our conclusions.

II. THE QPIPE QUERY EXECUTION ENGINE – SHARING THROUGH SIMULTANEOUS PIPELINED

QPipe [1] is an operator-centric execution engine, based on the paradigms of staged databases [17]. Each relational operator is encapsulated into a stage. Query plans are decomposed into packets describing sub-plans and dispatched to the stages accordingly. In order to exploit sharing opportunities, each stage monitors active packets and newly submitted packets. When a sharing opportunity appears, the stage pipelines the operator’s results of one packet to the rest of similar active packets, thus avoiding materialization costs. This ability of a stage is called On-demand Simultaneous Pipelining (OSP).

In the context of QPipe, data sharing and data pipelining are differentiated as follows. Let us assume a query Q begins execution at time $T_s$ and completes at $T_e$. Data sharing misses and work sharing misses are defined as:

Definition 1. At time $T_e$, Q requests page $P$, which was previously referenced at time $T_p$. If the request results in a page fault, and $T_s < T_p$, then it is a data sharing miss.

Definition 2. At time $T_w$, Q initiates a new computation by running operator $W$. If $W$ was also executed between $T_s$ and $T_w$, then there is a work sharing miss $^1$.

$^1$With respect to global plans, we use the term work sharing in a more relaxed manner to signify the savings of I/O, CPU and RAM resources.

A. On-Demand Simultaneous Pipelining

A stage using OSP promotes sharing pro-actively. Assume two or more concurrent query plans contain the same relational operator, and below that operator their sub-plans (along with predicates) are identical. This means that the operator emits the same output tuples for these queries. In this case, the OSP of that stage will evaluate only one of the corresponding packets and pipeline output tuples to all consuming packets simultaneously (performing simple projections if required). OSP works optimally for similar packets that arrive as a batch. If they have an inter-arrival delay, sharing opportunities may be restricted. The authors provide a detailed classification of relational operators with respect to these sharing opportunities.

A Window of Opportunity (WoP) is defined as the time from the invocation of an operator ($Q_1$) up until a newly submitted identical operator ($Q_2$) can take advantage of the one in progress. The WoP signifies what are the expected cost savings of the newly submitted operator as soon as it starts taking advantage of the in-progress operator. Figure 4 shows the different types of WoP that apply for relational operators.

A linear WoP characterizes operations where $Q_2$ can exploit the uncompleted part of $Q_1$. The cost savings vary from 0% to 100% depending on when $Q_2$ starts executing. An unordered table scan (i.e. queries are not interested in ordered disk pages) falls in this category. After $Q_2$ joins $Q_1$, the scan operation will begin pipelining all pages simultaneously to both queries. When it reaches the end of the table, the scan operator will start a new scan for $Q_2$ to read the pages that it missed before it joined. Operations similar to a table scan also have a linear WoP such as a clustered index scan, or the second phase of an unclustered index scan (visiting the sorted matched pages), or the second phase of a sort (outputting the sorted tuples).

The step WoP applies to identical operations that can fully share work, as long as the first output tuple has not yet been produced. Operations that have a step WoP include a nested-loop join, the merging phase of a sort-merge join, the probe phase of a hash-join and a group-by aggregate. The full WoP applies to identical operations that can fully share work, no matter when $Q_2$ joins $Q_1$ (before it finishes). Operations having a full WoP include the first phase of an unclustered index scan (sorting matched pages), the first phase of a sort (the actual sorting), a single-result aggregate and the building phase of a hash-join. The spike WoP is a specialization of the step WoP, and applies to operations that produce the first output tuple instantaneously. Operations are converted to a spike WoP when the query requires that output tuples be ordered. For instance, an ordered table scan has a spike WoP.

Fig. 4. Different types of Windows of Opportunity. From left-most to right-most: (a) Linear, (b) Step, (c) Full, (d) Spike.
show four relational operators (Table scan, Join, Aggregation and Sorting).

Fig. 2. The QPipe Engine. As an example, we show how a query’s execution plan is decomposed into packets that are forwarded to the relevant stages in order to be evaluated. For simplicity we show four relational operators (Table scan, Join, Aggregation and Sorting).

B. The query execution engine

In Figure 2, we depict QPipe’s design. The Packet Dispatcher converts a query plan to a series of inter-dependent packets. Data pages are pipelined through dedicated FIFO buffers between the packets. The buffers regulate the flow among differently-paced actors. Each packet is dispatched to the relevant relational operator stage. A stage dequeues a packet from its incoming queue, processes it (evaluates the operator with any arguments contained in the packet), writes the resulting pages to the packet’s output buffer and destroys it. Each stage has its own local pool of worker threads.

When a new packet queues up in a stage, OSP scans the existing packets for overlapping work, following the semantics of the stage’s WoP. If a sharing opportunity is detected, OSP attaches the new packet (satellite packet) to the in-progress packet (host packet). While the operator evaluates the host packet, OSP pipeliners the results to all satellite packets simultaneously. Additional actions may be taken depending on the stage. For example, if the host operator has a linear WoP, OSP will need to issue an additional packet to complete the non-overlapping part of the operation that the new packet missed.

In Figure 3, we depict an example of two identical hash-join packets (also having identical sub-plans). If their inter-arrival delay is outside the full-step WoP, they are evaluated separately. If it is inside the WoP, OSP can share the full work of the operator: It attaches Q2’s packet as a satellite of Q1’s packet. Then, it cancels all children packets of Q2 while Q1’s packet is being evaluated, it copies the output pages to the output buffers of both Q1 and Q2 simultaneously.

C. I/O Subsystem

QPipe maintains a dedicated scanner thread for each table, which performs a circular scan. When an unordered table scan packet queues up, it is attached to the scanner thread with linear WoP semantics. When the scanner thread reaches the end of the table, it continues from the beginning to serve the pages that the packet missed.

The main advantage of a circular scan is that it reduces contention for I/O resources. Without a circular scan, if many queries started scanning a table, they would compete with each other for bringing pages into the buffer pool and would result in random accesses on the disk.

III. The DataPath System – Sharing through a Global Plan

QPipe’s simultaneous pipelining is limited to identical sub-plans. Queries, however, often exhibit similar sub-plans with different predicates. A global query plan can save resources by employing a shared operator to evaluate similar queries simultaneously against the same data. The DataPath system [2] uses a single global query plan to evaluate all concurrent queries. Compared to the global plan of the CJOIN operator (§I-B), DataPath adds support for general schemas and for a shared single-result aggregation operator.

In the context of DataPath, the global plan is named path network. It is adjusted with every newly submitted query. Data is automatically pushed through the paths of the path network, and is processed by any interested shared operators (called waypoints). DataPath optimizes data sharing by sharing the same data with all interested computations.

A. Sharing through the global path network

In Figure 5, we illustrate the path network constructed after the depicted queries are submitted. When Q1 is submitted, the global plan is initially the query plan of Q1. When Q2 is submitted, it shares the scan of the lineitem table with Q1, and adds a scan of the orders table, a join and an aggregation. When Q3 is submitted, it adapts existing waypoints with its predicates, incurring a minimal marginal cost.

All data is shared in the system in order to avoid copying. Data is organized into light-weight structures, called chunks, which do not own data but are linked with the shared data, and pushed through the path network. The layout of a chunk is depicted in Figure 6. Shared data is stored column-by-column. A chunk links to all columns referenced by the queries that will use it. To facilitate queries in finding linked columns of different chunks, each chunk has a slot array, whose each slot corresponds to an active column. Furthermore, the technique of annotating data with query identifiers (§I-B) is used to support shared operators in the path network. Each tuple is associated with a bitmap that signifies relevant queries.

B. I/O Subsystem

Relations are stored on disk column-by-column. Tuples are read in batches of two millions, which corresponds to the

---

Fig. 2. The QPipe Engine.

Fig. 3. Simultaneous pipelining on two identical hash-join packets.
maximum size of a chunk. Each batch is assigned a chunk identifier and is decomposed into columns, which are further decomposed into pages. The chunk identifier, the column identifier and the page sequence number are used to hash (randomly) each page to a disk, where the page is written.

The I/O subsystem is always informed of which relations and columns are required by active queries. It knows which chunks are needed by the execution engine, and which chunks have been sent. In order to serve the next needed chunks, it allocates a staging area, as shown in Figure 7. It generates read requests for all of the pages needed by the linked columns of these chunks. Disks read requested pages asynchronously. When all required pages of a chunk have been constructed, the chunk is sent off to the execution engine, and its place in the staging area is filled by the next chunk needed to be read. Page requests in disk queues are ordered. In fact, disks perform a fast linear scan, skipping unneeded parts. The size of the staging area is configured so that all disks are kept busy.

C. The query execution engine

Chunks are not exchanged between waypoints in a pipelined fashion as in QPipe. An explicit scheduler routes them. It maintains a work queue where the I/O subsystem and waypoints place their produced chunks, and a pool of worker threads for their processing. When a chunk is extracted from the queue, the scheduler examines if there is a spare worker. If there is not, it drops the chunk and notifies its producer to reproduce it later. If the producer is the I/O subsystem, the chunk is reproduced the next time it is read in sequence. If a worker is available, the scheduler gives the chunk to the waypoint that is supposed to process it. The waypoint adds metadata and sends it to the worker. The worker processes the tuple-processing loop of the waypoint for the chunk.

DataPath differentiates between non-terminal and terminal paths in the global path network. Terminal paths provide the final destination for chunks and include the final output of a query, or the smaller relation of an in-memory hash join. In Figure 5, terminal paths are depicted with a dot. Terminal paths are used by the scheduler to route chunks correctly, and determine if a chunk is destined for multiple directions and needs to be copied. This is achieved by stamping chunks with query-exit pairs which signify the terminal paths where the chunk needs to be routed to for each relevant query.

Query-exit pairs are also used to notify the I/O subsystem when it should start producing chunks for a particular query-exit. The scheduler periodically polls the waypoints between the table scan and the exit whether they are ready to evaluate the query. A hash-join operator, for example, will signify its left-hand side (LHS) as ready only after the in-memory hash table for the smaller right-hand side (RHS) has been built.

DataPath supports shared hash-joins where the RHS relation fits in memory. All joins share a single fixed hash table which occupies around 80% of the memory. The LHS of a join waypoint can be shared by any query that shares the same path in the global plan, as long as they share an equality check with the same attribute(s) on the LHS tuples. The RHS of the shared queries need not match, as they are all hashed into the same hash table. A join waypoint starts by indexing its RHS to the hash table. Afterwards, chunks streamed into the LHS are looked up into the hash table where the RHS was indexed. If the LHS is shared by many queries, the lookup is also shared.

Disadvantages of the single hash table include a complicated concurrency control for different joins and translating tuples from a column format to a row format. For the former, a CPU core hashes small portions of the RHS independently and then merges them into the big hash table using locking. For the latter, a tuple is stored into many hash table entries.

D. The global path network optimizer

The path network optimizer takes as input the existing path network and attempts to integrate a new query in a way that minimizes the additional data movement. The cost function is equal to the total estimated tuples moved through all waypoints. Estimations differ from standard formulas because an exchanged tuple does not necessarily need to be double-counted. The search process uses an A*-style algorithm to integrate the joins of a new query. In Figure 8a, we depict an exemplary path network and the graph of a new query which represents its join predicates between the tables. The optimizer attempts to integrate one more join by enumerating all possible ways to collapse all nodes in the query graph. Collapsing two adjacent nodes results in either (a) the join being integrated into an existing waypoint, or (b) creating a new join waypoint. The former is possible if the table scans descending from the LHR of the existing waypoint match the tables of one of the adjacent nodes to be collapsed, and if the join attributes originating from this set of tables match the ones of the adjacent nodes. After enumerating all ways to collapse all nodes, the optimizer chooses the one which minimizes the cost function.

In Figure 8b, we depict an example of the first enumeration step of the optimizer. It chose to collapse \( \{S\} \) and \( \{PS\} \) into...
waypoint \( w_1 \). The enumeration can continue with collapsing \( \{L\} \) and \( \{O\} \) into \( w_2 \), and finally collapsing \( \{L, O\} \) and \( \{S, PS\} \) into a new join waypoint. Of course, this is just one possible enumeration considered by the search process.

As the authors note, the prototype path network optimizer is relatively simple. It does not take into account other factors that might influence performance considerably, such as the complexity and latency of an operation. For example, writing a big RHS of a join to the hash table takes significant time.

The consequent probing suffers a degradation of performance.

**IV. THE SHAREDDB SYSTEM – SHARING THROUGH A GLOBAL PLAN AND BATCHED EXECUTION**

QPipe, CJOIN and DataPath handle new queries the moment they arrive. Executing queries having different arrival times imposes book-keeping and might restrict sharing opportunities. QPipe needs to examine the WoP of a stage before sharing work. CJOIN needs to pause execution upon a new query, to update its filters. DataPath needs to run the optimizer for every new query, notify the I/O subsystem, and update column slots in chunks. Additionally, sharing some operators like sorting is difficult in a global plan. After the operator starts sorting, a new query might be added which selects more tuples than the ones being sorted.

In order to decrease book-keeping and facilitate sharing, SharedDB [3] has elected **batched execution**. It uses a pre-computed global plan and each shared operator evaluates one batch of queries at a time. By paying the optimization of the global plan beforehand, it can evaluate efficiently a large number of short-lived queries. The underlying storage manager, Crescando [9], adds support for serving a large number of reads and updates. Thus, SharedDB is able to support OLTP and OLAP workloads.

Nevertheless, SharedDB has limitations. Firstly, ad-hoc queries cannot re-adapt the global plan. Secondly, a new query suffers the latency of the ongoing batch, plus the new batch. Additionally, the latency of a batch is dominated by the longest-running query. Finally, a lot of resources are needed to keep the latency of a batch low. For example, Crescando uses main-memory storage. For these reasons, we argue that SharedDB favors mostly memory-resident databases and workloads without ad-hoc long-running queries.

**A. The Crescando storage manager**

Crescando [9] is designed to tackle a large number of short-running queries and updates with latency guarantees. It uses main-memory storage and data partitioning to scale up on multi-core machines. Crescando is based on a collaborative scan called **ClockScan**. In this aspect, typical trade-offs of shared scans apply: For few scan-light or update-light queries, traditional databases may prove more efficient.

ClockScan also uses batched execution. It performs a circular scan of a partition, executing firstly the update requests of the batch, strictly in arrival order, and then the read requests. Batched execution guarantees that the reads see a consistent snapshot of the writes, regardless of the order in which reads are served. Furthermore, ClockScan uses **query/update-data joins**. At the beginning of a batch, it builds an index of the predicates of read and update requests. During scanning, the index enables to quickly find requests for a scanned tuple.

Crescando takes care to support the durability and isolation transactional properties. For the former, it supports full recovery by checkpointing and logging all data to disk. For the latter, Crescando’s design favors snapshot isolation.

**B. Sharing through the global plan with batched execution**

SharedDB assumes that the types of possible queries are given in advance in the form of prepared statements. It pre-computes a global plan that is used for all batches. Currently, it is constructed manually. In Figure 9, we illustrate an exemplary global plan corresponding to the depicted prepared statements. It can evaluate numerous instances of the prepared statements with different values. Ad-hoc queries are processed individually, but can take advantage of the current global plan.

In order to share data among queries, SharedDB employs the technique of annotating tuples with query identifiers (§I-B). It extends the model of a relation \( R \) by adding an additional set-valued attribute \( \text{query}_id \).

Batching of queries is **fine-grained** per input relation and query operator. In Figure 9, queries of type \( Q_1, Q_2, Q_3 \) would queue for reading the users relation. User tuples would queue for the \( \Gamma \) and \( \triangleright \) operators which in turn would process these tuples in batches so that all user tuples belonging to a specific query are processed within a single batch.

Annotating with query identifiers and batched execution allow SharedDB to exploit shared computations in a generic way. A shared operator does not need to tackle queries with different arrival times, and receives all required data for a query within the batch. Shared operators such as sorting, group-by aggregations, nested-loops and sort-merge join are now easier to support. Also, SharedDB is able to reason about shared operators using standard relational algebra.

For instance, SharedDB executes a shared join operator as shown in Figure 10. One big join is executed instead of two smaller joins. The input consists of the union of the tuples selected by the two queries. The join predicate takes into account the \( \text{query}_id \) attribute in order to join tuples relevant to each query. Finally, the joined tuples are grouped by \( \text{query}_id \) in order to route them to each query. The inner join operator can be implemented with any traditional or specialized join algorithm.
C. The query execution engine

Shared operators of the global plan evaluate concurrent queries in batches. At the beginning of a batch, the operator activates queries by issuing their sub-queries to the respective operators, recursively. Then, it receives input tuples, processes them and forwards output tuples to the issuers of the queries. Blocking operators, such as sort, buffer input tuples and process them once they are all received.

If there are enough CPU resources, each operator (except for simple operators like selections) is assigned to a different core for optimal instruction cache locality. If an operator becomes a bottleneck, SharedDB can partition its load across a few replicas of the same operator. Replication, along with binding operators to CPU cores, are key factors for SharedDB’s response time guarantees. A mix of them can guarantee that the worst-case scenario for the pre-computed global plan, like sorting and joining whole relations, does not exceed a specified amount of time.

V. Preliminary results and research proposal

In this section, we present (a) our research plan in §V-A, (b) our preliminary results on sharing methodologies and our current work in §V-B, and (c) our conclusions in §V-C.

A. Future directions

We have identified several restrictions of the examined systems that hinder the full and dynamic exploitation of shared opportunities. Our research plan includes advancing the aforementioned sharing methodologies and combining them under an appropriate optimizer and execution engine, exploiting the cache hierarchy and NUMA processor topologies. We intend to combine simultaneous pipelining with a global query plan.

With respect to simultaneous pipelining, we plan to examine the benefits of batched execution, caching results and canceling queries. When the WoP of ongoing query has passed, a new similar query can join if cached results exist [1], or cancel the ongoing query in order to re-evaluate both simultaneously and save resources. A cost model and a prediction model can be used for determining the waiting time for forming a batch, and whether caching or canceling an ongoing query is beneficial.

With respect to global plans, the optimizer should be able to estimate the trade-off between using a global plan and exploiting parallelization, depending on the current workload and resources. Also, it should combine dynamic execution with batched execution to support more shared operators. Furthermore, parts of the global plan can be pre-computed to service a large number of similar short-lived queries. Moreover, the optimizer of a global plan needs to be studied under a theoretical and practical perspective for producing optimal global plans.

With respect to the I/O layer, the system should use an array of disks and aggregate the throughput of their linear scans. We intend to examine whether update requests can be interleaved along a linear scan, to support OLTP workloads as well.

Finally, we intend to investigate sharing in parallel DBMS [18]. Each node can employ a separate global query plan, and a coordinator should send a new query to the node which will incur the minimum estimated marginal cost for its evaluation.

B. Evaluating star queries via simultaneous pipelining and a global query plan

The work sharing methodologies employed by the systems we examined in this paper can be organized into two main classes: Simultaneous pipelining (QPipe) and global query plans (CJOIN, DataPath and SharedDB). We plan to better understand the intricate details of these two classes and identify any trade-offs. We test them using star schemas, which are widely used in data warehouses to organize data [16]. For our evaluation, we compare QPipe and CJOIN. CJOIN is a representative example of the second class, as it uses an optimized global plan targeted for star queries (§I-B). We present the summary of our most significant findings.

We compare four variations: (a) QPipe, without OSP, which is similar to evaluating query plans separately, (b) QPipe-OSP, with OSP enabled, (c) CJOIN where the joins in the star queries are evaluated with the CJOIN operator (integrated as a new stage in the QPipe execution engine) instead of hash-joins and (d) CJOIN-OSP where we enable OSP for the CJOIN stage with a step WoP. We use Shore-MT [19] as the storage manager and the Star Schema Benchmark [20] for our evaluation. Queries are submitted in batches, using the template of SSB Q3.2, a typical star query that joins three of the four dimension tables of SSB with the fact table.

Our hardware configuration consists of a Sun Fire X4470 server with four 6-core processors Intel Xeon E7530 at 1.86 GHz, no hyper-threading and 64 GB RAM. We use two 146 GB 10kRPM SAS 2.5” disks under RAID 0. We use direct I/O and clear file system caches before every measurement.

In our first experiment (Figure 11), we vary the number of concurrent queries. The predicates are chosen randomly,
with a selectivity of fact tuples < 0.2%. With this experiment, we identify that a query-centric engine (QPipe) suffers from resources contention, and global plans (CJOIN) can handle more queries than simultaneous pipelining (QPipe-OSP).

QPipe’s OSP suffers from a serialization point. The host packet needs to copy its results to all satellite packets. This may result in worse performance than the query-centric model, especially for memory-resident databases. This has been confirmed in the literature [21], where a prediction model is proposed for determining whether sharing is beneficial. We argue, however, that this serialization point is as an implementation artifact of the push-only model. It is possible to remove copying by sharing data. We are currently working on a shared pages list with one producer and multiple consumers. The host packet enters pages into the list, while satellite packets walk the list without copying.

In our second experiment (Figure 12), we vary the similarity of 128 queries. For the extreme cases with high similarity, QPipe-OSP outperforms CJOIN. By enabling OSP for CJOIN, we can take advantage of both forms of sharing, thus CJOIN-OSP proves to be the best. With this experiment, we identify that simultaneous pipelining is orthogonal to sharing through a global plan.

In our third experiment (Figure 13), we vary the selectivity of fact tuples of 8 queries. The higher selectivity has an adverse effect on CJOIN. There is an increased overhead due to maintaining larger dimension hash tables, also prolonging the pause of CJOIN for every newly submitted query.

C. Conclusions

The traditional query-centric model cannot exploit sharing opportunities that might exist among concurrently running queries. For high concurrency, this model typically results in contention for resources. By actively exploiting these opportunities, the execution engine can save I/O, CPU and RAM resources, reduce contention and improve overall performance. In this paper, we studied several research prototype systems that diverge from the query-centric model and employ various forms of data and work sharing methodologies: QPipe, CJOIN, DataPath and SharedDB.

For data sharing, all systems use forms of shared scans. QPipe, CJOIN and Crescendo use circular scans. Crescendo employs batched execution to service read and update requests of short-lived queries. Additionally, DataPath aggregates the throughput of the asynchronous linear scans of several disks.

For work sharing, we identified two methodologies, simultaneous pipelining (QPipe) and global query plans (CJOIN, DataPath and SharedDB). CJOIN evaluates a global plan for star schemas. DataPath evaluates a global plan for general schemas. SharedDB pre-computes the global plan for known query types, and uses batched execution to facilitate sharing.

Our research plan includes advancing the aforementioned sharing methodologies, and combining them under an adaptive DBMS that dynamically decides which sharing methodologies are beneficial, and uses them transparently.

REFERENCES