QPipe: A Simultaneously Pipelined Relational Query Engine

Stavros Harizopoulos
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
stavros@cs.cmu.edu

Vladislav Shkapenyuk†
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854
vshkap@cs.rutgers.edu

Anastassia Ailamaki
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
natassa@cs.cmu.edu

ABSTRACT

Relational DBMS typically execute concurrent queries independently by invoking a set of operator instances for each query. To exploit common data retrievals and computation in concurrent queries, researchers have proposed a wealth of techniques, ranging from buffering disk pages to constructing materialized views and optimizing multiple queries. The ideas proposed, however, are inherently limited by the query-centric philosophy of modern engine designs. Ideally, the query engine should proactively coordinate same-operator execution among concurrent queries, thereby exploiting common accesses to memory and disks as well as common intermediate result computation.

This paper introduces *on-demand simultaneous pipelining (OSP)*, a novel query evaluation paradigm for maximizing data and work sharing across concurrent queries at execution time. OSP enables proactive, dynamic operator sharing by pipelining the operator's output simultaneously to multiple parent nodes. This paper also introduces *QPipe*, a new operator-centric relational engine that effortlessly supports OSP. Each relational operator is encapsulated in a micro-engine serving query tasks from a queue, naturally exploiting all data and work sharing opportunities. Evaluation of QPipe built on top of BerkeleyDB shows that QPipe achieves a 2x speedup over a commercial DBMS when running a workload consisting of TPC-H queries.

1. INTRODUCTION

Modern decision-support systems (DSS) and scientific database applications operate on massive datasets and are characterized by complex queries accessing large portions of the database. Although high concurrency is predominantly studied in transactional workloads due to intensive updates, decision-support systems often run queries concurrently (hence the throughput metric suggested in the specification of TPC-H, the prevailing DSS benchmark). In a typical data warehousing installation, new data is periodically bulk loaded into the database, followed by a period where multiple users issue read-only (or read-heavy) queries. Concurrent queries often exhibit high data and computation overlap, e.g., they access the same relations on disk, compute similar aggregates, or share intermediate results. Unfortunately, run-time sharing in modern execution engines is limited by the paradigm of invoking an independent set of operator instances per query, potentially missing sharing opportunities if the caches and buffer pool evict data pages early.

†*Work done while the author was at Carnegie Mellon University.*

1.1 Sharing Limitations in Modern DBMS

Modern query execution engines are designed to execute queries following the "one-query, many-operators" model. A query enters the engine as an optimized plan and is executed as if it were alone in the system. The means for sharing common data across concurrent queries is provided by the buffer pool, which keeps information in main memory according to a replacement policy. The degree of sharing the buffer pool provides, however, is extremely sensitive to timing; in order to share data pages the queries must arrive simultaneously to the system and must execute in lockstep, which is highly unlikely. To illustrate the limitations of sharing through the buffer pool, we run TPC-H on X, a major commercial system[1] running on a 4-disk Pentium 4 server (experimental setup details are in Section 5). Although different TPC-H queries do not exhibit overlapping computation by design, all queries operate on the same nine tables, and therefore there often exist data page sharing opportunities. The overlap is visible in Figure 1a which shows a detailed time breakdown for five representative TPC-H queries with respect to the tables they read during execution.

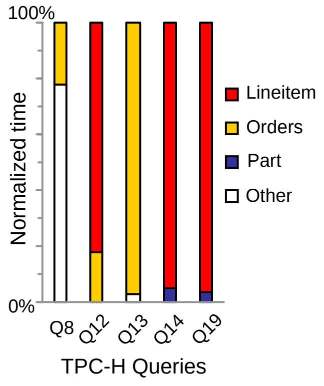

Figure 1a. Time breakdown for five TPC-H queries. Each component shows time spent reading a TPC-H table.

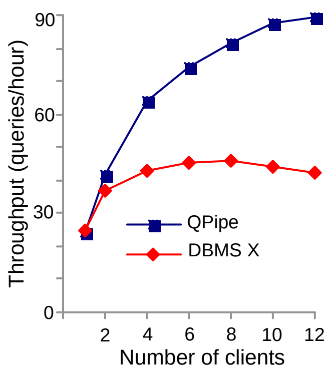

Figure 1b. Throughput for one to twelve concurrent clients running TPC-H queries on DBMS X and QPipe.

1. Licensing restrictions prevent us from revealing the vendor.

Figure 1b shows the throughput achieved for one to twelve concurrent clients submitting requests from a pool of eight representative TPC-H queries, for DBMS X and QPipe, our proposed query engine. QPipe achieves up to 2x speedup over X, with the throughput difference becoming more pronounced as more clients are added. The reason QPipe exhibits higher TPC-H throughput than X is that QPipe proactively shares the disk pages one query brings into memory with all other concurrent queries. Ideally, a query execution engine should be able to always detect such sharing opportunities across concurrent queries at run time, for all operators (not just for table scans) and be able to pipeline data from a single query node to multiple parent nodes at the same time. In this paper, we call this ability *on-demand simultaneous pipelining* (*OSP*). The challenge is to design a query execution engine that supports OSP without incurring additional overhead.

1.2 State-of-the-Art in Data and Work Sharing

Modern database systems employ a multitude of techniques to share data and work across queries. The leftmost part of Figure 2 shows those mechanisms and the center column shows the order in which they are invoked, depending on the high-level phases of query execution. Once a query is submitted to the system, it first performs a lookup to a cache of recently completed queries. On a match, the query returns the stored results and avoids execution altogether. Once inside the execution engine, a query may reuse precomputed intermediate results, if the administrator has created any matching materialized views. To our knowledge, modern engines do not detect and exploit overlapping computation among concurrent queries. When an operator consumes tuples, it first performs a buffer pool lookup, and, on a miss, it fetches the tuples from disk. Buffer pool management techniques only control the eviction policy for data pages; they cannot instruct queries to dynamically alter their access patterns to maximize data sharing in main memory.

The rightmost part of Figure 2 shows that during each of the three basic mechanisms for data and work sharing there is a missed opportunity in not examining concurrent queries for potential overlap. Often, it is the case that a query computes the same intermediate result that another, current query also needs. Or, an in-progress scan may be of use to another query, either by reading the file in a different order or by making a minor change in the query plan. It would be unrealistic, however, to keep all intermediate results around indefinitely, just in case a future query needs it. Instead, what we need is a query engine design philosophy that exploits sharing opportunities naturally, without incurring additional management or performance overhead.

1.3 On-demand Simultaneous Pipelining

To maximize data and work sharing at execution time, we propose to monitor each relational operator for every active query in order to detect overlaps. For example, one query may have already sorted a file that another query is about to start sorting; by monitoring the sort operator we can detect this overlap and reuse the sorted file. Once an overlapping computation is detected, the results are *simultaneously pipelined* to all participating parent nodes, thereby avoiding materialization costs. There are several challenges in embedding such an evaluation model inside a traditional query engine: (a) how to efficiently detect overlapping operators and decide on sharing eligibility, (b) how to cope with different consuming/producing speeds of the participating queries, and, (c) how to overcome the optimizer's restrictions on the query evaluation

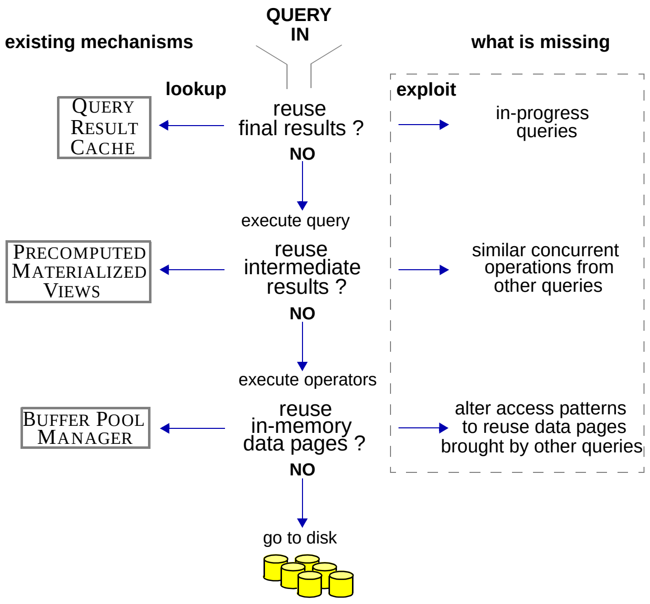
QUERY IN
existing mechanisms
what is missing
lookup
exploit
QUERY RESULT CACHE
reuse final results ?
in-progress queries
NO
execute query
PRECOMPUTED MATERIALIZED VIEWS
reuse intermediate results ?
similar concurrent operations from other queries
NO
execute operators
BUFFER POOL MANAGER
reuse in-memory data pages ?
alter access patterns to reuse data pages brought by other queries
NO
go to disk

Figure 2. Existing and desired mechanisms for sharing data and work across queries.

order to allow for more sharing opportunities. The overhead to meet these challenges using a "one-query, many-operators" query engine would offset any performance benefits.

To support simultaneous pipelining, we introduce **QPipe**, a new query engine architecture, based on the principles of the *Staged Database System* design [17]. QPipe follows a "*one-operator, many-queries*" design philosophy. Each relational operator is promoted to an independent micro-engine which manages a set of threads and serves queries from a queue. Data flow between micro-engines occurs through dedicated buffers — similar to a parallel database engine [10]. By grouping similar tasks together, QPipe can naturally exploit any type of overlapping operation. We implement QPipe on top of the BerkeleyDB storage manager, using native OS threads. The resulting prototype is a versatile engine, naturally parallel, running on a wide range of multi-processor servers (tested on IA-64, IA-32, Linux and Windows). As Figure 1b shows, QPipe exploits all data sharing opportunities, while executing the same workload as the commercial DBMS.

1.4 Contributions and Paper Organization

This paper (a) describes opportunities for data and computation sharing across concurrent queries, (b) introduces a set of query evaluation techniques to maximize data and work sharing, and, (c) describes the design and implementation of QPipe, a simultaneously-pipelined execution engine. We demonstrate the effectiveness of our techniques through experimentation with microbenchmarks and the TPC-H benchmark. QPipe can efficiently detect and exploit data and work sharing opportunities in any workload and can achieve up to 2x throughput speedup over traditional DBMS when running TPC-H queries.

The paper is organized as follows. Section 2 describes in more detail data and work sharing techniques. Section 3 introduces the techniques and challenges behind simultaneous pipelining. Section 4 describes the design and implementation of QPipe, while Section 5 carries out the experimental results. We conclude with future research opportunities in Section 6.

2. BACKGROUND & RELATED WORK

This section reviews techniques for sharing disk pages in the buffer cache, along with mechanisms to share work across different queries. It also briefly discusses related work in other contexts.

2.1 Buffer Pool Management

In its simplest form, a buffer pool manager keeps track of disk pages brought in main memory, decides when to write updated pages back to disk, and evicts pages (typically using a LRU policy) when new ones are read. The Hot Set [25] and DBMIN [6] algorithms rely on explicit hints from the query optimizer on query access patterns. Since it is infeasible for the optimizer to predict the query patterns in a multi-query environment, several algorithms base replacement decisions on the observed importance of different pages. LRU-K [22] and 2Q [18], for instance, improve the performance of the traditional LRU eviction policy by tracking multiple past-page references, while ARC [21] shows similar performance improvements without relying on tunable parameters.

Since queries interact with the buffer pool manager through a page-level interface, it is difficult to develop generic policies to coordinate current and future accesses from different queries to the same disk pages. The need to efficiently coordinate and share multiple disk scans on the same table has long been recognized [16] and several commercial DBMS incorporate various forms of multi-scan optimizations (Teradata, RedBrick [12], and SQL Server [7]). The challenge is to bypass the restrictions implied by the page-level interface in order to fully exploit the knowledge of query access patterns, even if it requires run-time adjustments to the query evaluation strategy.

2.2 Materialized Views

Materialized view selection [23] is typically applied to workloads known in advance, in order to speed-up queries that contain common subexpressions. The most commonly used technique is to exhaustively search all possible candidate views, while employing various heuristics to prune the search space. It is important to note that materialized views exploit commonality between different queries at the expense of potentially significant view maintenance costs. Modern tools for automatic selection of materialized views [1] take such costs into account when recommending a set of views to create [2]. The usefulness of materialized views is limited when the workload is not always known ahead of time or the workload requirements are likely to change over time.

2.3 Query Caching and Plan Recycling

Caching query results can significantly improve response times in a workload that contains repeating instances of the same query or queries that are subsumed by others. A recently proposed cache manager [29] dynamically decides on which results to cache, based on result computation costs, sizes, reference frequencies, and maintenance costs due to updates. Semantic data caching [9] (as opposed to page or tuple caching) can result in more efficient use of a cache and reduced communication costs in client-server environments. Query plan recycling [26] reduces the query optimization time by exploiting potential similarity in the plans of different queries. The queries are first clustered based on characteristics of their execution plans, and then all queries assigned to a cluster use the same plan generated for the cluster representative query. Both approaches complement any type of run-time optimizations. QPipe improves a query result cache by allowing the run-time detection of exact instances of the same query, thus avoiding extra work when identical queries execute concurrently, with no previous entries in the result cache.

2.4 Multi-Query Optimization

Multiple-query optimization (MQO) [13][27][24] identifies common subexpressions in query execution plans during optimization, and produces globally-optimal plans. Since the detection of common subexpressions is done at optimization time, all queries need to be optimized as a batch. In interactive scenarios where queries may arrive at any time, other queries that share the same computation may be already running (waiting to collect a batch delays the early queries). In addition, to share intermediate results among queries, MQO typically relies on costly materializations. To avoid unnecessary materializations, a recent study [8] introduces a model that decides at the optimization phase which results can be pipelined and which need to be materialized to ensure continuous progress in the system. In contrast, QPipe identifies and exploits common subexpressions at run time without forcing the optimizer to wait for a sufficient number of queries to arrive before optimizing a batch. Moreover, QPipe can efficiently evaluate plans produced by a multi-query optimizer, since it always pipelines shared intermediate results.

2.5 Related Work in Other Contexts

TelegraphCQ (CACQ [20] and PSoup [4]) and NiagaraCQ [5] describe techniques to share work across different queries in stream management systems, by sharing either physical operators or their state. Although the concept of sharing operators is similar to what we propose in this paper, the different context creates an entirely different problem. Queries in stream systems always process the most recently received tuples. In traditional DBMS, queries have specific requirements as to which tuples they need and in what order they need to process them.

Despite a plethora of mechanisms to share data and work across queries, the prevailing relational query execution paradigm is characterized by two key properties that preclude full exploitation of sharing opportunities. First, it deprives individual queries from knowing about the state of other, concurrent queries. In doing so, it prevents the system from taking action at run time, once an overlapping operation across different queries appears. Second, traditional query engines adhere to a static evaluation plan and to a page-level interface to the storage manager. Despite the fact that disk page access patterns are known in advance, sharing opportunities are limited since the system cannot adjust the query evaluation strategy at run time.

3. SIMULTANEOUS PIPELINING

If two or more concurrent queries contain the same relational operator in their plans, and that operator outputs the same tuples on behalf of all queries (or a query can use these tuples with a simple projection), then we can potentially “share” the operator. The operator will execute once, and its output will be pipelined to all consuming nodes simultaneously. In this paper we refer to the ability of a single relational operator to pipeline its output to multiple queries concurrently as simultaneous pipelining. *On-demand simultaneous pipelining* (OSP) is the ability to dynamically exploit overlapping operations at run time. OSP is desirable when there exist opportunities for reusing data pages that the buffer pool manager has evicted early, or intermediate computations across queries that are not covered by pre-computed materialized views.

This section first characterizes what a “missed opportunity” for data and work sharing is (Section 3.1). Then, it classifies all relational operators with respect to their effective “window of opportunity,” i.e., what percentage of the operation’s lifetime is offered for reuse (Section 3.2). Lastly, it describes the challenges in exploiting overlap between relational operators (Section 3.3).

3.1 Data and Work Sharing Misses

Whenever two or more concurrent queries read from the same table, or compute the same (or subset of the same) intermediate result, there is potentially an opportunity to exploit overlapping work and reduce I/O traffic, RAM usage, and CPU processing time. A *sharing miss* in a workload is defined in terms of memory page faults and computation as follows:

> *A query Q begins execution at time T_s and completes at time T_c.*
>
> ***Definition 1**. At time T_r, Q requests page P, which was previously referenced at time T_p. If the request results in a page fault, and $T_s < T_p$, the page fault is a **data sharing miss**.*
>
> ***Definition 2**. At time T_w, Q initiates new computation by running operator W. If W was also executed between T_s and T_w, then there is a **work sharing miss**.*

Sharing misses can be minimized by proactively sharing the overlapping operator across multiple queries. To clarify this procedure, consider the scenario illustrated in Figure 3 in which two queries use the same scan operators. For simplicity, we assume that the main memory holds only two disk pages while the file is $M \gg 2$ pages long. Query 1 starts a file scan at time T_{t-1}. As the scan progresses, pages are evicted to make room for the incoming data. At time T_t, Query 2 arrives and starts a scan on the same table. At this point, pages P_{n-2} and P_{n-1} are in main memory. These will be replaced by the new pages read by the two scans: P_n for Q1 and P_0 for Q2. At time T_{t+1}, Q1 has finished and Q2 is about to read page P_n. The main memory now contains P_{n-2} and P_{n-1} that Q2 just read. Page P_n, however, was in main memory when Q2 arrived in the system. This page (and all subsequent ones) represent *data sharing misses* by Q2.

With simultaneous pipelining in place, Query 2 can potentially avoid all data sharing misses in this scenario. Assuming that Q2 is not interested in which order disk pages are read — as long as the entire table is read — then, at time T_t, Q2 can “piggyback” on Q1’s scan operator. The scan operator will then pipeline all pages read simultaneously to both queries, and, on reaching the end of file, a new scan operator, just for Q2, will read the skipped pages. What happens, however, if Q2 expects all disk pages to be read in the order stored in file? To help understand the challenges involved in trying to minimize sharing misses, the next subsection classifies relational operators with respect to their sharing opportunities.

3.2 Window of Opportunity (WoP)

Given that query Q1 executes a relational operator and query Q2 arrives with a similar operator in its plan, we need to know whether we can apply simultaneous pipelining or not, and what are the expected cost savings for Q2 (i.e., how many sharing misses will be eliminated). We call the time from the invocation of an operator up until a newly submitted identical operator can take

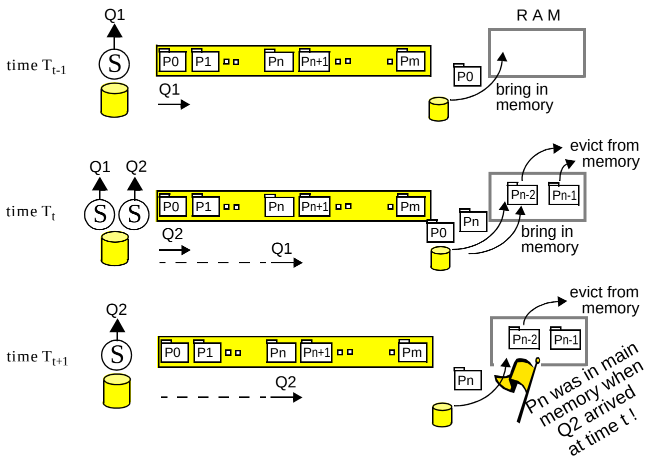

Figure 3. Two queries independently start a file scan on the same table. Query 2 is missing the opportunity to reuse all pages, after Pn, that Query 1 brings momentarily in RAM.

advantage of the one in progress, *window of opportunity* or *WoP*. Once the new operator starts taking advantage of the in-progress operator, the cost savings apply to the entire cumulative cost of all the children operators in the query’s plan.

Figure 4a shows a classification of all basic operations in a relational engine with respect to the WoP and the associated cost savings for a simultaneously pipelined second query. We identify four different types of overlap between the various relational operations (shown on the top of the figure). ***Linear*** overlap characterizes operations that can always take advantage of the uncompleted part of an in-progress identical operation, with cost savings varying from 100% to 0%, depending how late in the process Q2 joins Q1. For example, *unordered* table scans (which do not care about the order in which the tuples are received) fall in this category. ***Step*** overlap applies to concurrent operations that can exploit each other completely (100% cost savings), as long as the first output tuple has not been produced yet. For example, in the probing phase of hash-join, it may take some time before the first match is found; during that time, Q2 can join Q1. ***Full*** overlap is the ideal case: 100% cost savings for the entire lifetime of the in-progress operation (for example, computing a single aggregate). The last category, ***spike*** overlap, is all operations that cannot be overlapped, unless they start at the exact same time; for example, a table scan that must output tuples in table order can only piggyback on any other scan if the first output page is still in memory. A *spike* overlap is the same as a *step* overlap when the latter produces its first output tuple instantaneously.

Figure 4b shows two “enhancement” functions that can apply to the aforementioned categories in order to increase both the WoP and the cost savings. The ***buffering*** function refers to the ability of an operator to buffer a number of output tuples. Since output is not discarded immediately after it is consumed, an incoming request has a wider window of opportunity for exploiting the precomputed output tuples. For example, an ordered table scan that buffers N tuples can be converted from *spike* to *step*. The ***materialization*** function stores the results of an operator to be used later on. For example, consider an ordered table scan. If a new, highly selective (few qualifying tuples) query needs to scan the same table in stored tuple order, then we can potentially exploit the scan in progress by storing the qualifying tuples for the new query. This way we trade

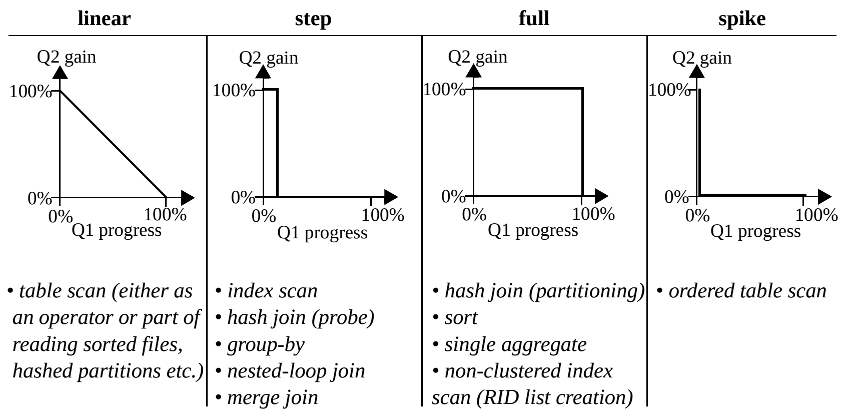

Figure 4a. Windows of Opportunity for the four basic operator overlap types.

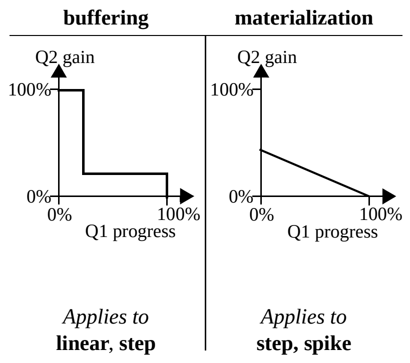

Figure 4b. WoP enhancement functions.

reading part of the table with storing and then reading a potentially significantly smaller number of tuples. This function can apply to *spike* to convert it to *linear*, albeit with a smaller effective slope for the cost savings.

Next, we break down each operator to its basic overlap types.

File scans. File scans have only one phase. If there is no restriction on the order the tuples are produced, or the parent operator needs the tuples in order but can output them in any order, then file scans have a *linear* overlap. If tuple ordering is strict then file scans have a *spike* overlap.

Index scans. Clustered index scans are similar to file scans and therefore exhibit either *linear* or *spike* overlap depending on the tuple ordering requirements. Unclustered index scans are implemented in two phases. The first phase probes the index for all matches and constructs a list with all the matching record IDs (RID). The list is then sorted on ascending page number to avoid multiple visits on the same page. This phase corresponds to a *full* overlap as a newly arrived operator can exploit work in progress at any point of time. The second phase is similar to file scan and so is either *linear* or *spike* overlap.

Sort. Sorting consists of multiple phases, though, in our context, we treat it as a two-phase operator. In the first phase the input is sorted on the sorting attribute (either in memory or disk, depending on the size of the file). During this phase any new arrival can share the ongoing operation, and therefore it is a *full* overlap. The second phase is pipelining the sorted tuples to the parent operator and it is similar to a file scan (either *linear* or *spike*).

Aggregates. All aggregate operators producing a single result (min, max, count, avg) exhibit a *full* overlap. Group-by belongs to *step* overlap, since it produces multiple results. *Buffering* can potentially provide a significant increase in the WoP, especially if the provided buffer size is comparable to the output size.

Joins. The most widely used join operators are hash-join, sort-merge join, and nested-loop join. Nested-loop join has a *step* overlap (it can be shared while the first match is not found yet). The sorting phase of sort-merge join is typically a separate sort operator. The merging phase is similar to nested-loop join (*step*). Hash-join first hashes and partitions the input relations. This phase is a *full* overlap. The joining phase is again *step* overlap. Both *buffering* and *materialization* can further increase the WoP.

Updates. By their nature, update statements cannot be shared since that would violate the transactional semantics.

3.3 Challenges in Simultaneous Pipelining

A prerequisite to simultaneous pipelining is decoupling operator invocation and query scope. Such a decoupling is necessary to allow an operator to copy its output tuples to multiple queries-consumers. In commercial DBMS this decoupling is visible only at the storage layer. Whenever a query needs tuples from the disk it waits for them to be placed at a specified buffer. From the query's point of view, it does not make a difference whether there is a single or multiple I/O processes delivering the same tuples to multiple queries. A similar decoupling should apply to all relational operators to implement simultaneous pipelining techniques. Following, we outline the remaining challenges.

Run-time detection of overlapping operations. To make the most out of simultaneous pipelining, the query engine must track the progress of all operators for all queries at all times. Whenever a query is submitted, the operators in its plan must be compared with all the operators from all active queries. The output of each comparison should specify whether there is an overlapping computation in-progress and whether the window of opportunity (WoP) has expired. This run-time detection should be as efficient as possible and scale well with the number of active queries.

Multiple-scan consumers. When new scan requests for the same table arrive repeatedly and dynamically share a single scan, a large number of partial scans will then be active on the same relation. Ideally, these partial scans should again synchronize the retrieval of common tuples, which requires additional bookkeeping. File scans with different selectivities and different parent consumption rates can make the synchronization difficult. If one file scan blocks trying to provide more tuples than its parent node can consume, it will need to detach from the rest of the scans. This might create a large number of partial scans covering different overlapping and disjoint regions of the relations, further complicating synchronization efforts.

Order-sensitive operators. Query optimizers often create plans that exploit "interesting" table orders by assuming that the scanned tuples will be read in table order. For example, if a table is already sorted on a join attribute, the optimizer is likely to suggest a merge-join and avoid sorting the relation. Such scans have a *spike*

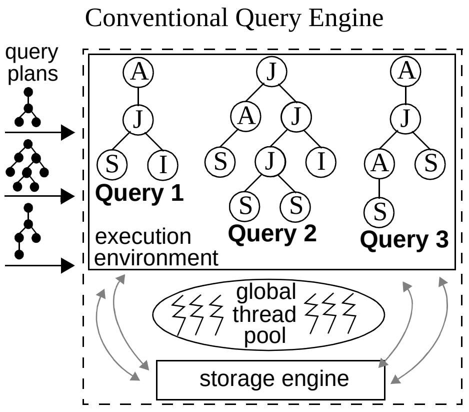

Figure 5a. Conventional engines evaluate queries independently of each other. Disk requests are passed to the storage engine.

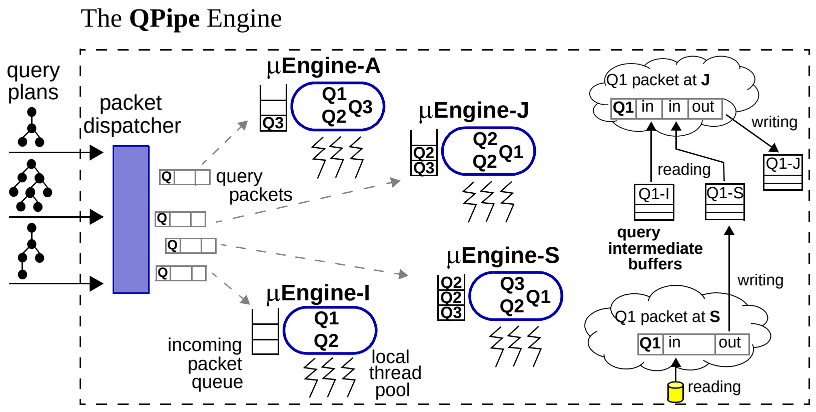

Figure 5b. In QPipe every relational operator is a *micro*-engine. For simplicity, only four operators are shown (Scan, Index-scan, Join, Aggregation). Queries are broken into *packets* and queue up in the μEngines.

WoP and therefore cannot take advantage of an ongoing scan. In case the ordered scan is highly selective (few qualifying tuples), a materialization function could help by storing the qualifying tuples, and reusing them later, in order. The challenge, however, is to exploit the scan in progress even if the new, order-sensitive scan does not perform any filtering.

Deadlocks in pipelining. The simultaneous evaluation of multiple query plans may lead to deadlocks. Consider for example two queries that share the results of two different scans (table A and B). If one query needs to advance scan A to be able to process the last value read from B, while the other query has the opposite need, advancing B to process A's last read value, then the two queries become deadlocked. The existence of a buffer can only delay the appearance of a deadlock in this case. The challenge is to efficiently detect potential deadlock situations and avoid them while still making the most out of overlapping computations.

4. QPIPE: DESIGN & IMPLEMENTATION

This section describes QPipe, a new architecture that efficiently applies simultaneous pipelining, and the techniques QPipe employs to address the above-mentioned challenges. First, we briefly describe the design philosophy behind conventional query engines (Section 4.1), before introducing the QPipe engine design (Section 4.2). We then describe how OSP is implemented in QPipe (Section 4.3) along with the details of the QPipe/BerkeleyDB prototype (Section 4.4).

4.1 Conventional Engine Design

Traditional relational query engine designs follow the "one-query, many-operators" model, and therefore are *query-centric*. Query plans generated by the optimizer drive the query evaluation process. A query plan is a tree with each node being a relational operator and each leaf an input point (either file scan or index scan) [14]. The execution engine evaluates queries independently of each other, by assigning one or more threads to each query. The high-level picture of the query engine consists of two components — the execution environment, where each query performs all of its intermediate computations, and the storage manager which handles all requests for disk pages (see also Figure 5a). Queries dispatch requests to the disk subsystem (storage engine) and a notification mechanism informs the query when the data is placed in a pre-specified memory location. The storage engine optimizes resource management by deciding which pages will be cached or evicted. Since all actions are performed without having cumulative knowledge of the exact state of all current queries, conventional engines cannot fully exploit data and work sharing opportunities across queries.

4.2 The QPipe Engine

QPipe implements a new, alternative execution model that we call "one-operator, many-queries," and therefore is an *operator-centric* architecture (Figure 5b). We first introduced this execution model in the *Staged Database System* design [17], which assigns DBMS components into independent stages, allowing for database installations that are easier to scale and maintain. In QPipe, each operator is promoted to an independent *micro*-engine (μEngine). μEngines accept requests (in the form of *packets*) and serve them from a queue. For example, the Sort μEngine only accepts requests for sorting a relation. The request itself must specify what needs to be sorted and which tuple buffer the result needs to be placed into. The way a query combines the independent work of all μEngines is by linking the output of one μEngine to the input of another, therefore establishing producer-consumer relationships between participating μEngines. In the current prototype, tuple buffers are implemented in shared-memory, however, this communication module can easily be replaced with a message passing mechanism, to deploy QPipe in distributed environments.

The input to QPipe is precompiled query plans (we use plans derived from a commercial system's optimizer). Query plans pass through the packet dispatcher which creates as many packets as the nodes in the query tree and dispatches them to the corresponding μEngines. Each μEngine has a queue of incoming requests. A worker thread that belongs to that μEngine removes the packet from the queue and processes it. Packets mainly specify the input and output tuple buffers and the arguments for the relational operator (e.g., sorting attributes, predicates etc.). μEngines work in parallel to evaluate the query. The evaluation model resembles a *push-based* execution design [15], where each operator independently produces tuples until it fills the parent's input buffer. If the output is consumed by a slower operator, then the intermediate buffers regulate the data flow.

Since QPipe involves multiple local thread pools (one for each μEngine), efficient scheduling policies are important to ensure low query response times. We follow a two-level scheduling approach. At the higher level, the scheduler chooses which μEngine runs next and on which CPU(s). Within each μEngine, a local scheduler decides how the worker threads are scheduled. In this paper, we use a round-robin schedule for the μEngines, with a fixed number of CPUs per μEngine, and the default, preemptive processor-sharing (PS) that the OS provides for the worker threads. Since this simple policy guarantees that the system always makes progress, response times for all queries were held low. As part of future work, we plan to experiment with different, self-adjustable scheduling policies.

QPipe can achieve better resource utilization than conventional engines by grouping requests of the same nature together, and by having dedicated μEngines to process each group of similar requests. In the same way a disk drive performs better when it is presented with a large group of requests (because of better disk head scheduling), each μEngine can better optimize resource usage by processing a group of similar requests. Although the focus of this paper is reusing data pages and similar computation between different queries at the same μEngine, we built QPipe to achieve better utilization of all resources in the system (extra CPUs, RAM, CPU caches etc.). We discuss other benefits of this architecture along with future work in the last section. Next, we describe the implementation of OSP techniques.

4.3 Support for Simultaneous Pipelining

In QPipe, a query packet represents work a query needs to perform at a given μEngine. Every time a new packet queues up in a μEngine, we scan the queue with the existing packets to check for overlapping work. This is a quick check of the encoded argument list for each packet (that was produced when the query passed through the packet dispatcher). The outcome of the comparison is whether there is a match and which phase of the current operation can be reused (i.e., a sorted file, and/or reading the sorted file). Each μEngine employs a different sharing mechanism, depending on the encapsulated relational operation (the sharing opportunities are described in Section 3.2).

There are two elements that are common to all μEngines: the **OSP Coordinator** and the **Deadlock Detector** (Figure 6a). The OSP Coordinator lays the ground for the new packet (the “satellite” packet) to attach to the in-progress query’s packet (the “host” packet), and have the operator’s output simultaneously pipelined to all participating queries. The OSP Coordinator handles the additional requirements and necessary adjustments to the evaluation strategy of the satellite’s packet original query. For example, it may create an additional packet to complete the non-overlapping part of an operation (this scenario is described in Section 4.3.2). The Deadlock Detector ensures a deadlock-free execution of simultaneously pipelined schedules. The pipelining deadlock problem is explained in Section 4.3.3 whereas the details of the algorithms employed are discussed elsewhere [30].

Figure 6b illustrates the actions the OSP coordinator takes when two queries have an overlapping operation. In this scenario, we assume Query 1 has already initiated a join of *step* overlap (e.g., merge-join), and a few tuples have already been produced, but are still stored in Q1’s output buffer. Without OSP (left part of Figure 6b), when Q2 arrives, it will repeat the same join operation as Q1, receiving input and placing output to buffers dedicated to Q2.

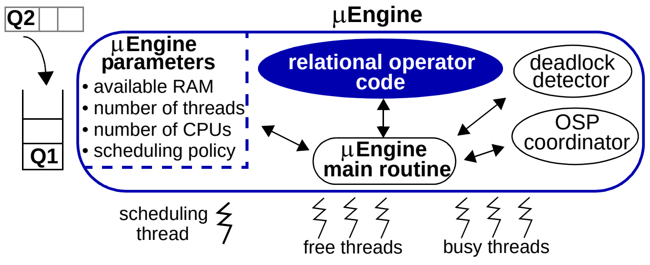

Figure 6a. A μEngine in detail.

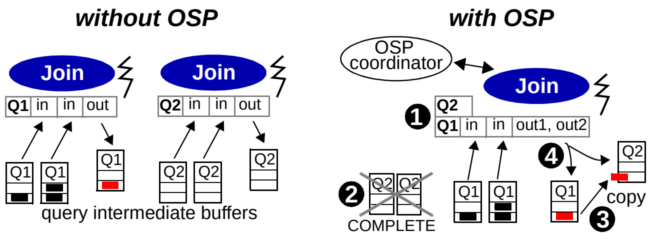

Figure 6b. Simultaneous pipelining on two join operations.

When the OSP Coordinator is active, it performs the following actions:

1. It attaches Q2’s packet (satellite) to Q1 (host).

2. It notifies Q2’s children operators to terminate (recursively, for the entire subtree underneath the join node).

3. It copies the output tuples of the join that are still in Q1’s buffer, to Q2’s output buffer.

4. While Q1 proceeds with the join operation, the output is copied simultaneously to both Q1’s and the satellite’s output.

The above steps are illustrated in Figure 6b (right part).

Once the OSP Coordinator attaches one or more satellite packets to a host packet, a “1-producer, N-consumers” relationship is formed between the participating queries. QPipe’s intermediate buffers regulate the dataflow. If any of the consumers is slower than the producer, all queries will eventually adjust their consuming speed to the speed of the slowest consumer. Next, we describe (a) how QPipe deals with the burden of frequently arriving/departing satellite scans (4.3.1), (b) the actions the OSP Coordinator takes to exploit order-sensitive overlapping scans (4.3.2), (c) how QPipe prevents deadlocks (4.3.3), and, (d) how QPipe handles lock requests and update statements.

4.3.1 Synchronizing Multiple Scan Consumers

Scan sharing of base relations is a frequently anticipated operation in QPipe. A large number of different scan requests with different requirements can easily put pressure on any storage manager and make the bookkeeping in a design that shares disk pages difficult. Sharing of multiple scans to the same table was first described in the RedBrick Data Warehouse implementation [12], and several other commercial systems such as Teradata and SQL Server [7] mention a similar functionality in their implementation. Details of the mechanisms employed, such as what kind of bookkeeping the storage manager performs and how the technique scales to multiple concurrent scans with different arrival times, are not publicly

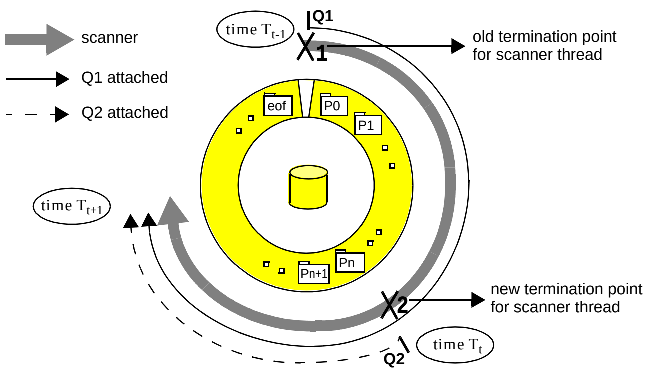

Figure 7. Circular scan operation. Q1 initiates the scanner thread (T_{t-1}). Q2 attaches immediately when it arrives (T_t) and sets the new termination point for the circular scan at page P_n.

disclosed. Moreover, the existing literature describes only scenarios where queries do not depend on the table scan order.

To simplify the management of multiple overlapping scans in QPipe, we maintain a dedicated scan thread that is responsible for scanning a particular relation. Once a new request for scanning a relation arrives, a *scanner thread* is initiated and reads the file (Figure 7). The scanner thread essentially plays the role of the host packet and the newly arrived packet becomes a satellite (time T_{t-1} in Figure 7). Since the satellite packet is the only one scanning the file, it also sets the termination point for the scanner thread at the end of the file. When later on, (time T_t), a new packet for scanning the same relation arrives, the packet immediately becomes a satellite one and sets the new termination point for the scanner thread at the current position of the file. When the scanner thread reaches the end-of-file for the first time, it will keep scanning the relation from the beginning, to serve the unread pages to Query 2.

This circular scan implementation simplifies the bookkeeping needed to track which queries are attached at any time. Moreover, it is the job of the OSP Coordinator to allow a packet to attach to the scanner thread or not, depending on the query requirements. For example, the query may need to start consuming pages only after another operator in the plan has started producing output. In our implementation, the OSP coordinator applies a late activation policy, where no scan packet is initiated until its output buffer is flagged as ready to receive tuples. Late activation prevents queries from delaying each other.

4.3.2 Order-Sensitive Scans

Consider a join operation where the base relations are already sorted on a joining key. In this case, the query plans may use a merge operator directly on the sorted files. If a scan is already in progress and a second query arrives, it encounters a *spike* overlap, and thus, it will not be able to attach. There are two cases, however, that the scan in progress can still be exploited.

First, if the parent operator of the merge-join does not depend on the order in which its input tuples are received, then the OSP Coordinator creates two merge-join packets for the same query. The first packet joins the remaining portion of the shared relation with the non-shared relation, providing output tuples to the order-insensitive parent. Afterwards, the second packet processes the unread part of the shared relation and joins it again with the non-shared relation. To avoid increasing the total cost, the OSP Coordinator always assumes the worst case scenario of reading the non-shared relation twice in order to merge the two disjoint parts of the shared relation. If the total cost does not justify sharing the operation, the OSP Coordinator does not attach the packets.

Second, if the selectivity of the scan is high (few qualifying tuples) or the selectivity of the merge operation is high, then the OSP Coordinator may choose to use the *materialization* function to save out-of-order results that are cheap to produce. Once the scan reaches the beginning of the relation, the query resumes regular execution, passing the result tuples to the parent of the merge. Once the scan reaches the page it first attached to, the saved results are used to compute the rest of the merge.

4.3.3 Deadlocks in Simultaneous Pipelining

Whenever the execution engine pipelines tuples produced by a query node to multiple consumers, it introduces the possibility of deadlock. Since nodes can only produce tuples as fast as the slowest consumer allows them to, loops in the combined query plans can lead to deadlocks. One such scenario was described in Section 3.3. This problem is not specific to QPipe; it has been also identified and studied in the context of multi-query optimization [8], where materialization of intermediate results is used as a deadlock prevention mechanism. The proposed algorithm makes conservative decisions since it relies on static analysis of the query plans.

We developed a dynamic model that uses the well-understood concept of Waits-For graphs to define deadlocks in pipelined execution engines. Our model uses information about the state of QPipe buffers (full, empty, or non-empty) without making any assumptions about operator consumer/producer rates. This allows us to pipeline the results of every query node, only materializing the tuples in the event of a real deadlock. Based on our model, we propose an efficient algorithm to detect deadlocks at run time and choose an optimal set of nodes to materialize that minimizes the total cost of executing all concurrent queries. Having run-time information available enables us to select a provably optimal set of nodes to materialize. This work, along with experimentation with MQO plans using commercial DBMS optimizers is described elsewhere [30].

4.3.4 Locks and Updates

Data and work sharing techniques are best exploited in read-mostly environments, such as concurrent long-running queries in data warehouses, where there is high probability of performing overlapping work. Workloads with frequent concurrent updates to the database limit the percentage of time that scans can be performed (due to locking), and therefore restrict the overall impact of data sharing techniques. QPipe runs any type of workload, as it charges the underlying storage manager (BerkeleyDB in the current implementation) with lock and update management by routing update requests to a dedicated μEngine with no OSP functionality. As long as a sharing opportunity appears, even in the presence of concurrent updates, QPipe will take advantage of it. If a table is locked for writing, the scan packet will simply wait (and with it, all satellite ones), until the lock is released.

4.4 The QPipe/BerkeleyDB Prototype

The current QPipe prototype is a multi-threaded, parallel application that runs on shared-memory multiprocessor systems. Each μEngine is a different C++ class with separate classes for the

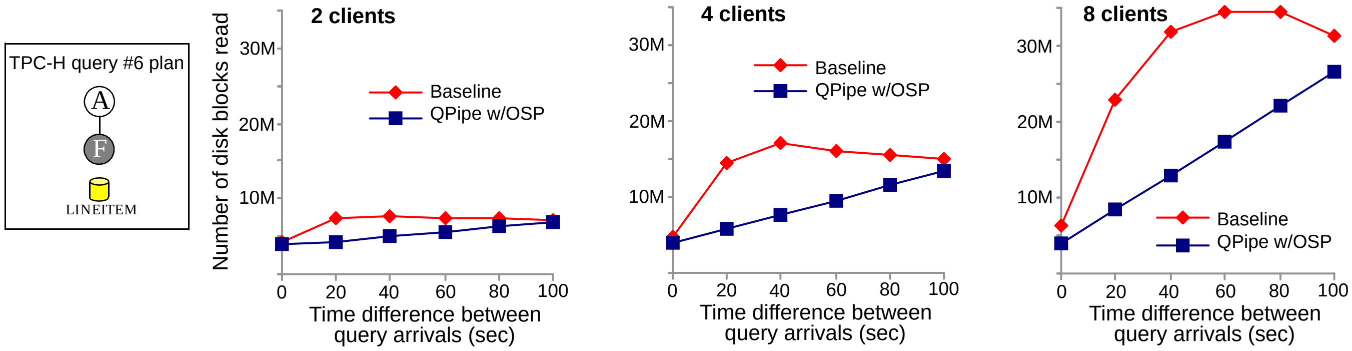

Figure 8. Total number of disk blocks read for three different configurations (2, 4, and 8 concurrent users sending TPC-H Query 6) with varying user interarrival times (0-100 sec). The number of blocks read remains flat for longer than 120 sec interarrival times.

thread-pool support, the shared-memory implementation of queues and buffers (including query packets), the packet dispatcher, and the OSP Coordinator. Calls to data access methods are wrappers for the underlying storage manager. The bare system is a runtime consisting of a number of idle threads, as many as the specified µEngines times the number of threads per µEngine. The OS schedules the threads on any of the available CPUs. Client processes can either submit packets directly to the µEngines or send a query plan to the packet dispatcher which creates and routes the packets accordingly. The basic functionality of each µEngine is to dequeue the packet, process it, optionally read input or write output to a buffer, and destroy the packet. The client process reads the final results from a shared-memory buffer.

We implement relational-engine functionality by inserting relational processing code to each µEngine, and providing the packet dispatcher code to transform precompiled query plans into packets. The database storage manager adds the necessary transactional support, a buffer-pool manager, and table access methods. In the current prototype we use the BerkeleyDB database storage manager and have implemented the following relational operators: table scan (indexed and non-indexed), nested-loop join, sort, merge-join, hybrid hash join, aggregate (both simple and hash-based). The current implementation is about 7,000 lines of C++ code (BerkeleyDB itself is around 210,000 lines).

In addition to BerkeleyDB, we have successfully applied the QPipe runtime with OSP support to two other open source DBMS, MySQL and Predator[28]. Since there is typically a clear division between the storage manager and the rest of the DBMS, it was straightforward to transfer all function calls to the storage manager inside the µEngines. Taking the optimizer's output and redirecting it to the packet dispatcher was also straightforward. The time consuming part of the conversion is to isolate the code for each relational operator. Fortunately, each relational operator uses a limited set of global variables which makes it easy to turn the operator into a self-contained module with parameters being passed as an encoded structure.

5. EXPERIMENTAL RESULTS

This section presents our experimentation with the QPipe prototype. We experiment using two datasets. The first dataset is based on the Wisconsin Benchmark [11] which specifies a simple schema with two large tables and a smaller one. We use 8 million 200-byte tuple tables for the big tables (BIG1 and BIG2 in the experiments) and 800,000 200-byte tuples for the small table (SMALL). The total size of the tables on disk is 4.5GB. The second dataset is a 4GB TPC-H database generated by the standard dbgen utility. The total size of the dataset on disk (including indices and storage engine overhead) is 5GB. All experiments are run on a 2.6 GHz P4 machine, with 2GB of RAM and four 10K RPM SCSI drives (organized as software RAID-0 array), running Linux 2.4.18. We discard all result tuples to avoid introducing additional client-server communication overhead. In all of the graphs, “Baseline” is the BerkeleyDB-based QPipe implementation with OSP disabled, “QPipe w/OSP” is the same system with OSP enabled, and “DBMS X” is a major commercial database system. When running QPipe with queries that present no sharing opportunities, we found that the overhead of the OSP coordinator is negligible.

5.1 Sharing Data Pages

5.1.1 Exploiting Overlapping Unordered Scans

In this experiment we examine how well QPipe with OSP performs when exploiting the *linear* overlap of table scans. We evaluate three different workloads with 2, 4, and 8 concurrent clients running TPC-H Query 6. The 99% of execution time is spent performing an unordered table scan of the LINEITEM relation. We evaluate the performance of circular scans in QPipe as a function of different query interarrival times. We vary the interarrival time for a set of queries from 0 sec (highest overlap) to 100 sec (relatively little overlap). The goal of the experiment is to investigate the amount of redundant I/O that we can save by employing OSP.

The results are shown in Figure 8. The vertical axis is the total number of disk blocks read during the workload execution time. For workloads where queries arrive simultaneously, traditional disk page sharing through the buffer pool manager performs well. However, as the query interarrival time grows to 20 sec, the data brought in by the running query are completely evicted from the buffer pool by the time the next query arrives. On workloads with a high degree of overlap (20 sec interarrival time) QPipe with OSP can save up to 63% of the total I/O cost. As the interarrival time grows and the overlap between queries shrinks the two curves approach each other (and remain flat at the same point for 120 sec or more when there is no overlap). QPipe w/OSP always exploits all data sharing opportunities, whereas the baseline system depends on the timing of different arrivals to share data.

5.1.2 Exploiting Overlapping Clustered Index-Scans

In this experiment we evaluate our technique for exploiting overlaps between ordered scans, essentially converting a *spike* overlap

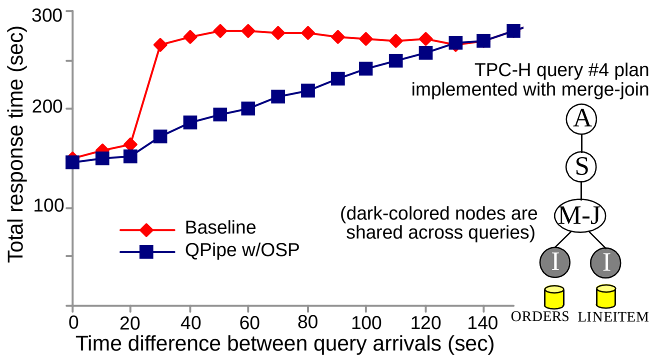

Figure 9. Sharing order-sensitive clustered index scans (I) on ORDERS and LINEITEM between two queries starting at different time intervals. Merge-join (M-J) expects tuples in key order. Since sort (S) does not assume a specific ordering, QPipe w/OSP performs 2 separate joins to share the in-progress scan.

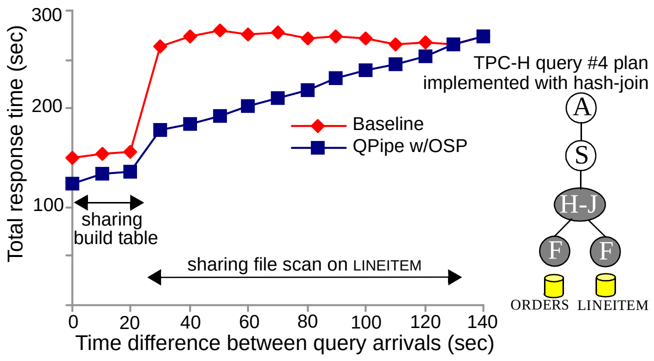

Figure 11. Changing the plan of TPC-H query #4 to use hash-join allows for a window of opportunity on sharing the build phase of the hash-join (first 20 secs). If the second query arrives later than that, it still can share the scan on LINEITEM.

to *linear*. We submit to QPipe two instances of TPC-H Query #4 which includes a merge-join at different time intervals. The full plan of Query #4 is shown in Figure 9. Even though the merge join relies on the input tuples being ordered, there is no need for the output of the join to be properly ordered. QPipe with OSP takes advantage of this property of the query plan, and allows ordered scans to attach to the existing one even though it is already in progress. Once the merge join consumes the input produced by the overlapping scans it initiates a new partial scan to retrieve the records it missed due to the late arrival. Figure 9 shows that QPipe with OSP significantly outperforms the baseline system with OSP disabled.

5.2 Reusing Computation in Aggregates/Joins

5.2.1 Sort-merge Join

The sort-merge join operator consists of a sort which is a *full* + *linear* overlap, followed by a merge which is a *step* overlap. In the next experiment, we use two similar 3-way join queries from the Wisconsin Benchmark. The graph in Figure 10 shows the total elapsed time from the moment the first query arrives until the system is idle again. We vary the interarrival time for the two queries from 0 sec up to the when there is no overlap between the queries. The graph shows that QPipe with OSP can exploit commonality for most of the query's lifetime (that's why the line for QPipe w/

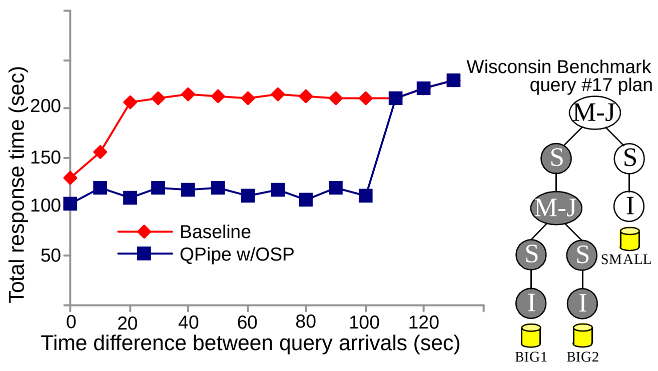

Figure 10. Sharing multiple operators, with sort (S) at the highest level. The two queries have the same predicates for scanning BIG1 and BIG2, but different ones for SMALL.

OSP remains flat most of the time) resulting in a 2x speedup. In this case, QPipe w/OSP is able to merge the packets from the two different queries during the merge phase of the sort-merge join. The baseline system performs better when the queries arrive close to each other (point zero on the horizontal axis), as it can share data pages in the buffer pool.

5.2.2 Hash join

In this experiment we evaluate a *full* + *step* overlap operator, hash join. We submit to QPipe two instances of TPC-H query #4 which uses a hash join between the LINEITEM and ORDERS relations varying interarrival time. We expect that QPipe with OSP will be able to reuse the building phase of hash join. The graph axes are the same as in the previous figures. Figure 11 shows that QPipe with OSP can reuse the entire results of the build phase of the hash-join (20 seconds mark). After the hash join starts producing the first output and it is no longer possible to reuse the results of the build phase, QPipe still is able to significantly reduce the I/O costs by sharing the results of the scan in progress on LINEITEM.

5.3 Running Full Workloads

In the next experiment we compare the performance of QPipe with OSP against the baseline system and the commercial DBMS X, using a set of clients executing a random mix of queries from the TPC-H benchmark. The query mix is based on TPC-H queries #1, #4, #6, #8, #12, #13, #14, and #19. To make sure that multiple clients do not run identical queries at the same time, the selection predicates for base table scans were generated randomly using the standard *qgen* utility. Even though all the queries had different selection predicates for table scans, QPipe's circular scans are able to take advantage of the common accesses to LINEITEM, ORDERS and PART. We use hybrid hash joins exclusively for all the join parts of the query plans. Since hash joins do not rely on the ordering properties of the input streams, we are able to use unordered scans for all the access paths, which have large windows of opportunity. We vary the number of clients from 1 to 12 and measure the overall system throughput. Each client is given 128MB of memory to use for the sort heap and the hash tables. When running a single client we observe that the workload is disk-bound.

Figure 12 shows that QPipe w/OSP outperforms both the baseline system and X. For a single client, the throughput of QPipe and X is almost identical since the disk bandwidth is the limiting factor. As

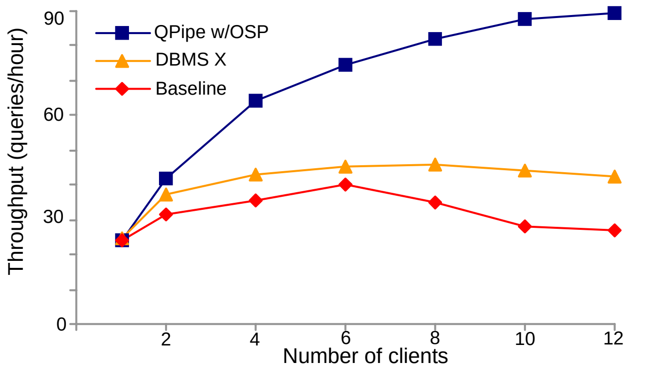

Figure 12. TPC-H throughput for the three systems increasing the number of concurrent users from 1 to 12, and keeping think time to zero.

the number of clients increases beyond 6, DBMS X is not able to significantly increase the throughput. On other hand, QPipe with OSP takes full advantage of overlapping work and achieves a 2x speedup over DBMS X. The difference in the throughput between the baseline system and DBMS X shows that X's buffer pool manager achieves better sharing than the one BerkeleyDB employs. In Figure 13 we show the average response time for the same mix of TPC-H queries, for QPipe w/OSP and the baseline system, using 10 concurrent users and changing the think time of each user. As this experiment shows, QPipe w/OSP achieves high throughput without sacrificing query response times.

6. CONCLUSIONS AND FUTURE WORK

Multiple concurrent queries often operate on the same set of tables, using the same set of basic operators in their query plans. Modern DBMS can therefore execute concurrent queries faster by aggressively exploiting commonalities in the data or computation involved in the query plans. Current query engines execute queries independently and rely on the buffer pool to exploit common data, which may miss sharing opportunities due to unfortunate timing. Previous efforts to explore overlapping work include multiple query optimization, materialized views, and exploitation of previous results; all these approaches, however, involve caveats and overhead that makes them impractical in several cases. If, however, we change the query engine philosophy from query-centric (one-query, many-operators) to operator-centric (one-operator, many-queries) we can proactively detect and exploit common data and computation at execution time with no additional effort or overhead. In this paper we first propose a set of techniques and policies to exploit overlapping work between concurrent queries at run time. Then, we present QPipe, a new architecture that naturally supports the proposed techniques and offers several advantages:

- By applying on-demand simultaneous pipelining of common intermediate results across queries, QPipe avoids costly materializations. In fact, Qpipe can efficiently evaluate plans produced by a multi-query optimizer.
- QPipe improves performance (throughput and response time) when compared to tuple-by-tuple evaluation engines (iterator model) by saving extraneous procedure calls and by improving temporal locality. As an example, recent work [31] introduces a buffer operator to increase the number of tuples processed at one time at each operator. QPipe reaps such memory-hierarchy benefits for free, because it is designed to proactively exploit locality in memory hierarchy.
- QPipe provides full intra-query parallelism, taking advantage of all available CPUs in a multi-processor server for evaluating a single query, regardless of the plan's complexity.

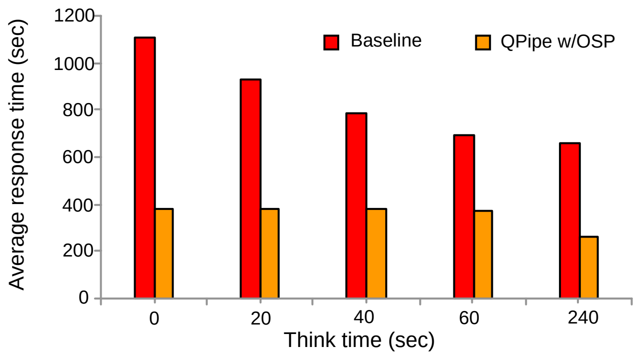

Figure 13. Average response time for QPipe w/OSP and the baseline system for a mix of TPC-H queries, varying the think time, for 10 concurrent users (low think times correspond to high system load).

In the near future we plan to use QPipe as a platform to conduct research in the following areas:

- Since QPipe naturally resembles parallel database designs, we plan to deploy it in distributed environments (grid computing) and study work allocation (an orthogonal problem to data placement) along with load balancing and query scheduling algorithms.
- QPipe's ability to group similar tasks together gives us the opportunity to further optimize RAM and disk usage by leveraging the characteristics of each operation in the system independently.
- As an extension to the techniques presented in this paper we plan to study dynamic, transparent plan alteration techniques (similar to [19]) to create more opportunities for reusing overlapping work.

7. EPILOGUE

Database computing arguably represents the most challenging server computing environment, whereas decision support (DSS) installations of massive data sets and multiple concurrent users are typical in today's enterprises. Locality and predictability of different tasks running in a system has long been the key property that computer and storage architects, along with software designers have exploited to build high-performance computing systems. Different implementation iterations of caching and prefetching techniques both in hardware and software already span more than three decades. This paper shows that the key to optimal performance is exposing all potential locality both in data and in computation, by grouping similar tasks together. QPipe, our proposed query execution engine architecture, leverages the fact that a query plan offers an exact description of all task items needed by a query, and employs techniques to expose and exploit locality in both data and computation across different queries. Most importantly, we have successfully applied the QPipe architecture in two open-source DBMS and two storage managers, making the case that QPipe design can apply with relatively few changes to any DBMS.

8. ACKNOWLEDGEMENTS

We cordially thank Kun Gao and Steve Schlosser for technical support, and the SIGMOD reviewers for their comments. We are also grateful to IBM for supporting this work through an IBM faculty partnership award and a graduate student fellowship, and to NSF for supporting grants CCR-0113660, IIS-0133686, and CCR-0205544.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. "Automated selection of materialized views and indexes in SQL databases." In *Proc. VLDB,* 2000.

[2] J. A. Blakeley, P. Larson, and F. W. Tompa. "Efficiently updating materialized views." In *Proc SIGMOD,* 1986.

[3] M. Carey et al. "Shoring Up Persistent Applications." In *Proc. SIGMOD*, 1994.

[4] S. Chandrasekaran and M. J. Franklin. "Streaming Queries over Streaming Data." In *Proc. VLDB*, 2002.

[5] J. Chen, D. DeWitt, F. Tian, and Y. Wang. "NiagaraCQ: A scalable continuous query system for internet databases." In *Proc. SIGMOD,* 2000.

[6] H.T. Chou and D. J. DeWitt. "An evaluation of buffer management strategies for relational database systems." In *Proc. SIGMOD,* 1985.

[7] C. Cook. "Database Architecture: The Storage Engine." Miscrosoft SQL Server 2000 Technical Article, July 2001. Available at: `http://msdn.microsoft.com/library`

[8] N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. "Pipelining in Multi-Query Optimization." In *PODS*, 2001.

[9] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. "Semantic Data Caching and Replacement." In *Proc. VLDB,* 1996.

[10] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. "The Gamma Database Machine Project." In *IEEE TKDE*, 2(1), pp. 44-63, Mar. 1990.

[11] D. J. DeWitt. "The Wisconsin Benchmark: Past, Present, and Future." *The Benchmark Handbook,* J. Gray, ed., Morgan Kaufmann Pub., San Mateo, CA (1991).

[12] P. M. Fernandez. "Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms." In *SIGMOD,* 1994.

[13] S. Finkelstein. "Common expression analysis in database applications." In *Proc. SIGMOD,* 1982.

[14] G. Graefe. "Iterators, Schedulers, and Distributed-memory Parallelism." In *Software-practice and experience,* Vol. 26 (4), pp. 427-452, Apr. 1996.

[15] G. Graefe. "Volcano - An Extensible and Parallel Query Evaluation System." In *TKDE 6(1)*: 120-135, 1994.

[16] J. Gray. "The Next Database Revolution." Keynote, *SIGMOD,* 2004.

[17] S. Harizopoulos and A. Ailamaki. "A Case for Staged Database Systems." In *Proc. CIDR,* 2003.

[18] T. Johnson and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm." In *Proc. VLDB,* 1994.

[19] N. Kabra and D. J. DeWitt. "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans." In *Proc. SIGMOD,* 1998.

[20] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. "Continuously Adaptive Continuous Queries over Streams." In *Proc. SIGMOD*, 2002.

[21] N. Megiddo and D. S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." In *Proc. FAST,* 2003.

[22] E. J. O'Neil, P. E. O'Neil, and G. Weikum. "The LRU-K page replacement algorithm for database disk buffering." In *Proc. SIGMOD,* 1993.

[23] N. Roussopoulos. "View indexing in relational databases." In *ACM Trans. on Database System*s *7(2)*:258-290,1982.

[24] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. "Efficient and Extensible Algorithms for Multi Query Optimization." In *Proc. SIGMOD*, 2000.

[25] G. M. Sacco and M. Schkolnick. "Buffer management in relational database systems." In *ACM TODS,* 11(4):473-498, Dec. 1986.

[26] P. Sarda, J. R. Haritsa. "Green Query Optimization: Taming Query Optimization Overheads through Plan Recycling," In *Proc. VLDB,* 2004.

[27] T. K. Sellis. "Multiple Query Optimization." In *ACM TODS*, 13(1):23-52, Mar. 1988.

[28] P. Seshadri, M. Livny, and R. Ramakrishnan. "The Case for Enhanced Abstract Data Types." In *Proc. VLDB,* 1997.

[29] J. Shim, P. Scheuermann, and R. Vingralek. "Dynamic caching of query results for decision support systems." In *Proc. SSDBM*, 1999.

[30] V. Shkapenyuk, R. Williams, S. Harizopoulos, and A. Ailamaki. "Deadlock Resolution in Pipelined Query Graphs." *Carnegie Mellon University Technical Report,* CMU-CS-05-122, 2005.

[31] J. Zhou and K. A. Ross. "Buffering Database Operations for Enhanced Instruction Cache Performance." In *Proc. SIGMOD*, 2004.