Staged Database Systems

Thesis Proposal

Stavros Harizopoulos
Computer Science Department
Carnegie Mellon University

"By organizing and assigning system components into self-contained stages, database systems can exploit instruction and data commonality across concurrent requests thereby increasing throughput. Furthermore, staged database systems are more scalable, easier to extend, and more readily fine-tuned than traditional database systems."

Abstract

Database system architectures face a rapidly evolving operating environment where millions of users store and access terabytes of data. To cope with increasing demands for performance high-end DBMS employ parallel processing techniques coupled with a plethora of sophisticated features. However, the widely adopted work-centric thread-parallel execution model entails several shortcomings that limit server performance, the most important being failure to exploit instruction and data commonality across concurrent requests. Moreover, the monolithic approach in DBMS software has lead to complex designs which are difficult to extend.

This thesis introduces a staged design for high-performance, evolvable DBMS that are easy to fine-tune and maintain. I propose to break the database system into modules and encapsulate them into self-contained stages connected to each other through queues. The staged, data-centric design remedies the weaknesses of modern DBMS by providing solutions at (a) the hardware level: it optimally exploits the underlying memory hierarchy, and (b) at a software engineering level: it is more scalable, easier to extend, and more readily fine-tuned than traditional database systems.

1 Introduction

Database management systems (DBMS) are responsible for executing time-critical operations and supporting an increasing base of millions of users. To cope with high demands for performance and usability modern database systems (a) use a work-centric multi-threaded (or multi-process) execution model, and (b) employ a multitude of sophisticated tools. However, the techniques for boosting performance and functionality also introduce several hurdles. The threaded execution model entails several shortcomings that limit performance under changing workloads. Uncoordinated memory references from concurrent queries may cause poor utilization of the memory hierarchy. In addition, the complexity of modern DBMS poses several software engineering problems such as difficulty in introducing new functionality or in predicting system performance. Furthermore, the monolithic approach in designing and building database software helped cultivate the view that "the database is the center of the world." Additional front/back-ends or *mediators* [Wie92] add to the communication and CPU overhead.

Database researchers indicate the need for a departure from traditional DBMS designs [Be+98][CW00][SZ+96] due to changes in the way people store and access information online. Research [MDO94] has shown that the CPU/memory speed mismatch affects database workloads more than other scientific or desktop applications. Work in *cache conscious* database systems improves the cache performance of query processing algorithms [SKN94]. Subsequent independent studies of DBMS performance on modern processors [AD+99][KP+98][LB+98] narrow the primary memory-related bottlenecks to first-level instruction and second-level data cache misses. Novel data placement schemes [AD+01] reduce level two data cache misses, however, first level instruction cache misses and misses occurring when concurrent threads replace each other's working sets [JK99][RB+95], have yet to be addressed. Larus and Parkes proposed *cohort scheduling*, a grouped request execution discipline, and showed a reduction in L1 instruction cache misses for two simple, custom built-servers [LP02]. Microsoft's SQL Server implements a mechanism to share concurrent file scans across queries [Co01]. Although related work identifies memory-related bottlenecks and proposes techniques to boost performance, current DBMS designs do not have the means to exploit commonality across all levels of the memory hierarchy.

My thesis proposal introduces the *Staged Database System* design for high-performance, evolvable DBMS that are easy to tune and maintain. I propose to break the DBMS software into multiple modules and to encapsulate them into self-contained stages connected to each other through queues. Each stage exclusively owns data structures and sources, independently allocates hardware resources, and makes its own scheduling decisions. This staged, data-centric approach improves current DBMS designs by providing solutions (a) at the hardware level: it optimally exploits the underlying memory hierarchy and takes direct advantage of multi-processor and multi-threaded systems, and (b) at a software engineering level: it aims at a highly flexible, extensible, easy to program, monitor, tune and evolve platform. My thesis is that

by organizing and assigning system components into self-contained stages, database systems can exploit instruction and data commonality across concurrent requests thereby increasing throughput. Furthermore, staged database systems are more scalable, easier to extend, and more readily fine-tuned than traditional database systems.

Upon completion, my thesis work will:

- Provide an analysis of design shortcomings in modern DBMS software.
- Demonstrate a high-performance, scalable DBMS design built on self-contained stages.
- Propose and evaluate query scheduling algorithms for staged database systems.
- Implement efficient fine-grain self-tuning techniques for staged DBMS.
- Demonstrate the extensibility of the staged design by integrating an external application into it.

The proposal is organized as follows. The next section reviews related work. Section 3 presents the preliminary thesis work, structured in three steps: identification of problems in current DBMS designs, description of the proposed staged database system design, and a study of associated scheduling trade-offs. Section 4 contains the ongoing and future work. The ongoing work consists of two steps: exploit instruction commonality across concurrent requests inside a staged database engine (Section 4.1), and share private working data across concurrent queries (Sec. 4.2). Future work is also divided into two steps: build a scalable staged database engine for multi-processor systems (Sec. 4.3), and demonstrate the extensibility and tuning efficiency of the new design (Sec. 4.4). The proposal's plan and goals are summarized in Section 5.

2 Related work

In the past three decades of database research, several new software designs have been proposed. One of the earliest prototype relational database systems, INGRES [SW+76], actually consisted of four "stages" (processes) that enabled pipelining (the reason for breaking up the DBMS software was main memory size limitations). Staging was also known to improve CPU performance in the mid-seventies [AWE]. Parallel database systems [DG92][CHM95] exploit the inherent parallelism in a relational query execution plan and apply a *dataflow* approach for designing high-performance, scalable systems. In the GAMMA database machine project [De+90] each relational operator is assigned to a process, and all processes work in parallel to achieve either *pipelined parallelism* (operators work in series by streaming their output to the input of the next one), or *partitioned parallelism* (input data are partitioned among multiple nodes and operators are split into many independent ones working on a part of data). In *extensible* DBMS [CH90], the goal was to facilitate adding and combining components (e.g., new operator implementations). Both parallel and extensible database systems employ a modular system design with several desirable properties, but there is no notion of cache-related interference across multiple concurrent queries.

Recent database research focuses on a data processing model where input data arrives in multiple, continuous, time-varying streams [BB+02]. The relational operators are treated as parts of a chain where the scheduling objective is to minimize queue memory and response times, while providing results at an acceptable rate or sorted by importance [UF01]. Avnur et al. propose *eddies*, a query processing mechanism that continuously reorders pipelined operators in a query plan, on a tuple-by-tuple basis, allowing the system to adapt to fluctuations in computing resources and data characteristics [AH00]. Operators run as independent threads, using a central queue for scheduling. While the aforementioned architectures optimize the execution engine's throughput by changing the invocation of relational operators, they do not exploit cache-related benefits. Work in "cache-conscious" DBMS optimizes query processing algorithms [SKN94], index manipulation [CGM01][CLH00][GL01], and data placement schemes [AD+01]. Such techniques improve the locality *within* each request, but have limited effects on the locality *across* requests. Context-switching across concurrent queries is likely to destroy data and instruction locality in the caches. For instance, when running transaction processing workloads, most misses occur due to conflicts between threads whose working sets replace each other in the cache [JK99][RB+95].

Recent OS research introduced *cohort scheduling* [LP02], which assembles cohorts of similar tasks and schedules their execution together to reduce memory stalls. Applications are organized into stages using a staged library and a scheduler repeatedly executes requests one stage at a time. The authors built a simple web server and a publish-subscribe server to demonstrate the benefits of this approach. Research on compilers proposes code layout optimizations to reduce instruction cache misses for database workloads [RB+01]. Thread scalability is limited when building highly concurrent applications [Ous96][PDZ99]. Related work suggests inexpensive implementations for context-switching [AB+91][BM98], and also proposes *event-driven* architectures with limited thread usage, mainly for internet services [PDZ99]. Welsh et al. propose a *staged* event-driven

architecture (SEDA) for deploying highly concurrent internet services [WCB01]. SEDA decomposes an event-driven application into stages connected by queues to prevent resource overcommitment when demand exceeds the server's capacity. SEDA does not optimize for memory hierarchy performance, which is the primary bottleneck for data-intensive applications.

3 Preliminary thesis work

This thesis proposal is motivated by two observations. First, the monolithic design of today's DBMS software has lead to complex systems that are difficult to maintain and extend. Second, the prevailing thread-based execution model yields poor cache performance in the presence of multiple clients. As the processor/memory speed-gap and the demand for massive concurrency increases, memory-related delays and context-switch overheads hurt DBMS performance even more. The preliminary thesis work consists of three steps:

- Step 0 (Section 3.1) discusses the problems related to the above mentioned two observations.
- Step 1 (Sec. 3.2) introduces the Staged Database System design which addresses the problems of current designs.
- Step 2 (Sec. 3.3) presents a scheduling analysis needed to achieve basic system operation.

3.1 Step 0: Problems in current DBMS design

3.1.1 Pitfalls of monolithic DBMS design

Extensibility. Modern DBMS are difficult to extend and evolve. While commercial database software offers a sophisticated platform for efficiently managing large amounts of data, it is rarely used as stand-alone service. DBMS require the rest of the applications to communicate with each other and coordinate their accesses through the database. Overall system performance degrades due to unnecessary CPU computation and communication latency on the data path. The alternative, extending the DBMS to handle data conversions and application logic, is a difficult process, since typically there is no well-defined API and security concerns limit the exported functionality.
Tuning. Database software complexity makes it difficult to identify resource bottlenecks and properly tune the DBMS in heavy load conditions. A DBA relies on statistics and system reports to tune the DBMS, but has no clear view of how the different modules and resources are used as current systems can only monitor resource utilization at a coarse granularity. Based on this information it is difficult to build automatic tuning tools to ease DBMS administration. Furthermore, when requests exceed the database server's capacity, new clients are either rejected or experience significant delays. Yet, some of them could still receive fast service (e.g., if they need a cached tuple).

3.1.2 Pitfalls of thread-based concurrency

Modern database systems process concurrent queries by multiplexing their execution using a pool of threads or processes[1]. Each thread executes its assigned task until it blocks on a synchronization condition or an I/O event, or its time quantum has elapsed. The CPU will then switch context and run a different thread. While this widely used model [IBM01][MS00] ensures fairness and low response times, it has several shortcomings:

1. No single number of preallocated worker threads can yield optimal performance under changing workloads. Too many threads waste resources and too few threads restrict concurrency.

1. The choice between threads or processes also depends on the underlying operating system. Since this choice is an implementation detail, it does not affect the generality of our study.

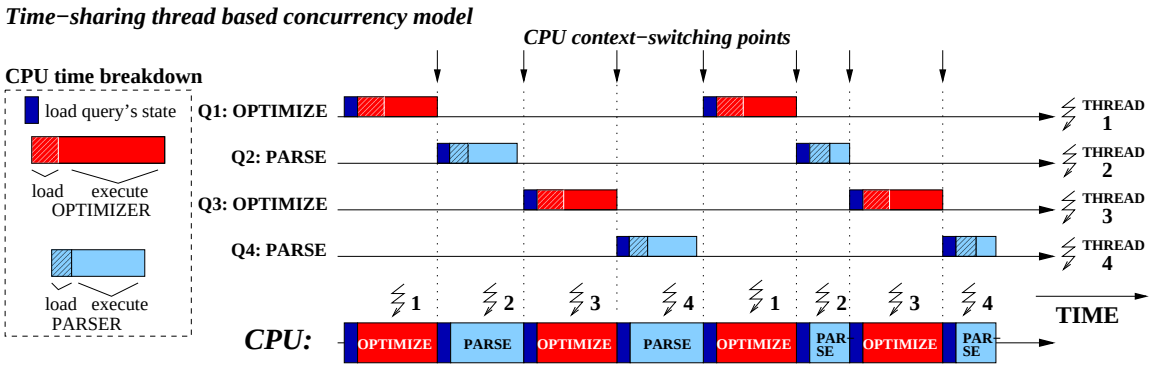

FIGURE 1: *Uncontrolled context-switching can lead to poor performance.*

2. Preemption is oblivious to the thread's current execution state. Context-switches that occur in the middle of a logical operation evict a possibly larger working set from the cache. When the suspended thread resumes execution, it wastes time restoring the evicted working set.
3. Round-robin scheduling does not exploit cache locality across threads at their current state. Current schedulers do not examine cache contents when selecting the next thread to run.
4. Thread-based concurrency gives little or no control to the database programmer of how to detect and exploit commonality in each query's private working data set. Concurrent queries may perform overlapping work yet few techniques to date take advantage of it.

The first three shortcomings are also depicted in Figure 1. Four concurrent queries handled by four worker threads pass through the optimizer or the parser of a single-CPU database server. The example assumes that no I/O takes place. Whenever the CPU resumes execution on a query, it first loads (fetches from main memory) the thread's private state. Then, during each module's execution, the CPU also spends time loading the data and code that are *shared* on average between all queries executing in that module (shown as a separate striped box after the context-switch overhead). A subsequent invocation of a different module will likely evict the data structures and instructions of the previous module, to replace them with its own ones. The performance loss in this example is due to (a) a large number of worker threads: since no I/O takes place, one worker thread would be sufficient, (b) preemptive thread scheduling: optimization and parsing of a single query is interrupted, resulting in unnecessary reloads of its working set, and (c) round-robin scheduling: optimization and parsing of two different queries are not scheduled together and, thus, the two modules keep replacing each other's data and code in the cache.

3.2 Step 1: Staged Database System design [HA03]

A staged database system consists of self-contained modules, each encapsulated into a *stage*. A stage is an independent server with its own queue, thread support, and resource management that communicates and interacts with the other stages through a well-defined interface. Stages accept *packets*, each carrying a query's state and private data (the query's *backpack*), perform work on the packets, and may enqueue the same or newly created packets to other stages. The first-class citizen is the query, which enters stages according to its needs. Each stage is centered around exclusively owned (to the degree possible) server code and data. There are two levels of CPU scheduling: local thread scheduling within a stage and global scheduling across stages. This design promotes stage autonomy, data and instruction locality, and minimizes the usage of global variables.

We divide at the top level the actions the database server performs into five query execution stages (see Figure 2): *connect*, *parse*, *optimize*, *execute*, and *disconnect*. The *execute* stage typically represents the largest part of a query's lifetime and is further decomposed (described in Sec-

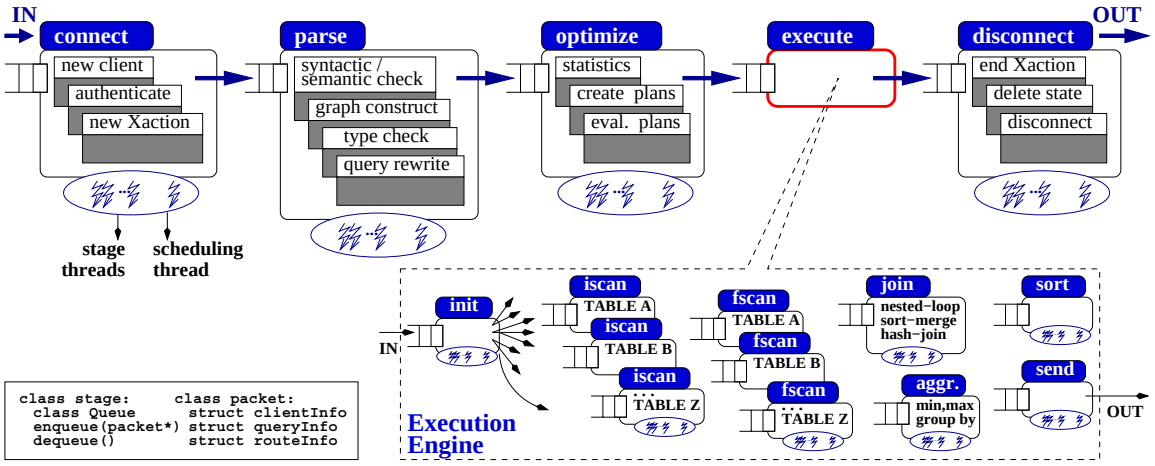

FIGURE 2: ***The Staged Database System design:*** *Each stage has its own queue and thread support. New queries queue up in the first stage, they are encapsulated into a "packet", and pass through the five stages shown on the top of the figure. A packet carries the query's "backpack": its state and private data. Inside QPIPE, the staged execution engine, a query can issue multiple packets to increase parallelism.*

tion 3.2.2). The break-up objective is (a) to keep accesses to the same data structures together, (b) to keep instruction loops within a single stage, and (c) to minimize the query's backpack. For example, *connect* and *disconnect* execute common code related to client-server communication: they update the server's statistics, and create/destroy the client's state and private data. Likewise, while the *parser* operates on a string containing the client's query, it performs frequent lookups into a common symbol table. The design in Figure 2 is general enough to apply to any modern relational DBMS, with minor adjustments. For example, commercial DBMS support precompiled queries that bypass the parser and the optimizer. In our design the query can route itself from the *connect* stage directly to the *execute* stage. Figure 2 also shows certain operations performed inside each stage. Depending on each module's data footprint and code size, a stage may be further divided into smaller stages that encapsulate operation subsets (to better match the cache sizes).

3.2.1 Stage definition

A stage provides two basic operations, *enqueue* and *dequeue*, and a queue for the incoming *packets*. The stage-specific server code is contained within *dequeue*. The proposed system works through the exchange of *packets* between stages. A packet represents work that the server must perform for a specific query at a given stage. It first enters the stage's queue through the enqueue operation and waits until a dequeue operation removes it. Then, once the query's current state is restored, the stage specific code is executed. Depending on the stage and the query, new packets may be created and enqueued at other stages. Eventually, the stage code returns by either (i) destroying the packet (if done with that query at the specific stage), (ii) forwarding the packet to the next stage (i.e. from *parse* to *optimize*), or by (iii) enqueueing the packet back into the stage's queue (if there is more work but the client needs to wait on some condition). Queries use packets to carry their state and private data. Each stage is responsible for assigning memory resources to a query. As an optimization, in a shared-memory system, packets can carry only pointers to the query's state and data structures (which are kept in a single copy).

Each stage employs a pool of worker threads (*stage threads*) that continuously call *dequeue* on the stage's queue, and one thread reserved for scheduling purposes (*scheduling thread*). More than one threads per stage help mask I/O events while still executing in the same stage (when there are more than one packets in the queue). If all threads happen to suspend for I/O, or the stage has used its time quantum, then a stage-level scheduling policy specifies the next stage to execute. When-

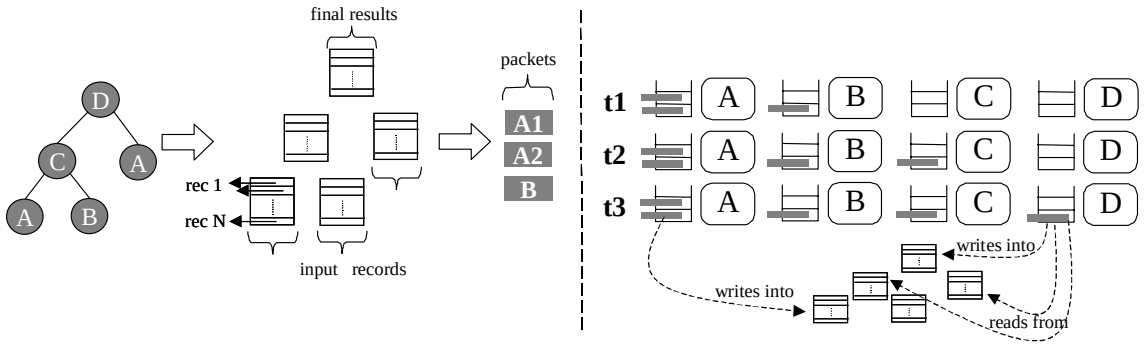

FIGURE 3: *Query execution example in Qpipe. The QEP is a tree containing four operators: A, B, C, and D. Each operator corresponds to a Qop (shown on the bottom part). The query first sets up a similar tree of pages with pointers to records. Then it creates as many packets as the leaves of the tree and enqueues them to the Qops (time t1). Qops A and B work in parallel. When enough records are placed in the record buffers that Qop C will read from, a new packet is enqueued (time t2). Same with Qop D at time t3.*

ever *enqueue* causes the next stage's queue to overflow, a back-pressure flow control mechanism suspends the *enqueue* operation (and subsequently freezes the query's execution thread in that stage). Queries that do not call *enqueue* on the blocked stage will continue to run.

3.2.2 Qpipe: A staged relational execution engine

Relational database engines typically evaluate query execution plans (QEP) which represent a compiled (parsed and optimized) query. A QEP consists of relational operators that form a tree. Data flow from the leaf nodes (stored tuples) through the intermediate nodes (as intermediate results) and to the root of the tree (final query results). Relational operators consume their children's output and produce tuples for their parent node, forming data pipelines. Two methods of evaluating a QEP are the *iterator* (or *pull*) and *push* models [Gra96]. The iterator model recursively invokes the children nodes starting at the root, and produces results in a postfix fashion. In the push model, the leaf nodes keep producing tuples and push them through their parents to the root. A producer-consumer relationship regulates the data flow.

Qpipe is the relational execution engine of the staged database design. Qpipe replaces traditional relational operators with Qpipe Operators, or Qops[2] for short. A Qop contains DBMS-specific code along with a queue for incoming requests, and acts as a scaled-down engine. Queries exist in the form of one or more packets which queue up in front of a Qop. A query issues as many packets as the operators in the QEP (query execution plan). The actual data transfer through Qops happens via dedicated, per-query record buffers that are set up as part of a query's execution initialization. The QEP evaluation follows the "push" model. Before a query enters Qpipe it goes through an initialization phase (illustrated in Figure 3). It traverses the operator tree (left most part of the figure) and sets up a similar tree of private record buffers. Qops will write the data they produce in the corresponding buffer for that query, while the parent Qop will consume data from that buffer. In shared-memory systems Qops process pointers to a single record copy, while in non shared-memory systems Qops ship pages in the CPU the parent Qop resides (same way as in GAMMA [De+90], but in Qpipe there is only one Qop serving multiple concurrent queries). Once the query sets up the record buffers, it enqueues as many packets as the QEP leaves to the corresponding Qops. In the example of Figure 3, the query enqueues two packets in Qop A and one packet in Qop B (time t1). These packets "ask" the Qop to work on behalf of the query and start filling the output record buffer. The QEP leaf nodes work in parallel and activate a parent Qop by

2. pronounced "que-ops"

enqueueing a new packet when enough data accumulate in the parent Qop's input buffers. Eventually, all Qops in the QEP are activated while data keep flowing from the leaves to the root of the QEP (time t2 and t3 in Figure 3).

3.3 Step 2: Preliminary study of scheduling trade-offs [HA02]

There is a fundamental scheduling trade-off in any staged execution scheme that tries to exploit locality by queueing and executing requests on a per module basis. The trade-off is between decreasing cache misses for a group of requests in the same software module while increasing response time of other requests that need to access different modules. The challenge is to find scheduling policies that exploit a module's affinity to memory resources while improving throughput and response time. This step evaluates database systems as a candidate for a staged execution scheme by studying this trade-off. In order to compare alternative strategies for forming and scheduling batches of queries at various degrees of inter-query locality, we develop a simulated database execution environment that is also analytically tractable.

3.3.1 Staged model

Each submitted query passes through several stages of execution that contain a server module (see Figure 4). For instance, such a module is the parser or the optimizer of the database. Once a module's data structures and instructions, that are *shared* (on average) by all queries, are accessed and loaded in the cache, subsequent executions of different requests within the same module will significantly reduce memory delays. To model this behavior, we charge the first query in a batch with an additional CPU demand (quantity l_i in Figure 4). The model assumes, without loss of generality, that the entire set of a module's data structures that are shared on average by all requests can fit in the cache, and that a total eviction of that set takes place when the CPU switches to a different module. The prevailing scheduling policy processor-sharing (PS) fails to reuse cache contents, since it switches from query to query in a random way with respect to the query's current execution module. The model described here can also apply to a wide class of servers that follow a "production-line" model of operation (see [HA02] for an example, along with a Markov chain based analysis of the proposed algorithms).

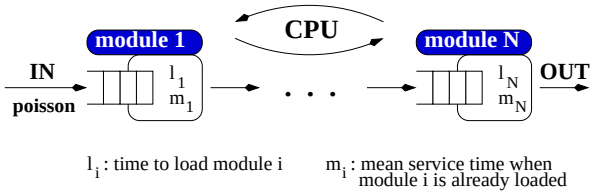

FIGURE 4: *A production-line model for staged servers.*

3.3.2 Scheduling algorithms

The execution flow in the model is purely sequential, thereby reducing the search space for scheduling policies into combinations of the following parameters: the number of queries that form a batch at a given module (one, several, all), the time they receive service (until completion or up to a cutoff value), and the module visiting order. We consider two basic policies, **d-gated** and **t-gated(N)**, and a variation of those, **c-gated**.

- D-gated (dynamic-gated) dynamically imposes a gate on the incoming queries, and executes the admitted group of queries as a batch at each module, up to their completion. Execution takes place in a first-come first-served basis at the queue of each module. When the admitted batch finishes execution, the CPU shifts to the first module, admits all accumulated queries, and imposes a gate to the rest of incoming queries. Note that d-gated reduces to FCFS whenever there is at most one query queued up at the first module.
- T-gated(N) (threshold-gated) explicitly defines an upper threshold N for the number of queries that will pass through the first module and form a batch of size N. Whenever there are more

than N queries queued up, the CPU admits the first N of them and considers the rest only after the completion of the selected batch. A possible issue with D- and T-gated is that a very large query can block the way to other, smaller ones, thereby causing higher response times.

- C-gated (cutoff-gated) complements the previous policies by applying an additional cutoff value to the time the CPU spends on a given query at a given module. Whenever the CPU exceeds that cutoff value, the current query is left behind and rejoins the next batch. This way, both small and large queries make progress while still benefiting from increased data locality.

In order to compare the different scheduling policies, the incoming queries are modeled on a Poisson arrival stream with sizes (CPU demand) drawn from an exponential distribution. Whenever a query starts execution at a module and the common data structures do not already lie in the cache (this happens when the CPU was previously working on a different module), then the query is charged with an additional fixed CPU demand, for that module. This extra CPU demand represents the time spent in memory stalls due to common data structures fetching from main memory to cache, for a given query. Figure 5 shows the relative speedup of T-gated(2) over PS, for a wide range of different locality scenarios. The x-axis is the percentage of execution time of a query spent servicing cache misses that are attributed (on average) to common data structures of a module. A query that finds all modules "warmed up" will execute faster by that percentage. In Figure 3 we vary this value from 1% to 70%. The y-axis is the server load; we varied the arrival rate to achieve server loads between 1% and 98%. On the right of the y-axis we denote the areas where the relative speedup of T-gated(2) over PS is within a certain range. Areas with darker color correspond to higher speedup, while the white area corresponds to those combinations that both policies perform almost the same. PS is only able to perform better in a small area on the left of the graph; the relative speedup of PS in that area does not exceed 1.1. The full experimentation and simulation results for all policies considered can be found in [HA02].

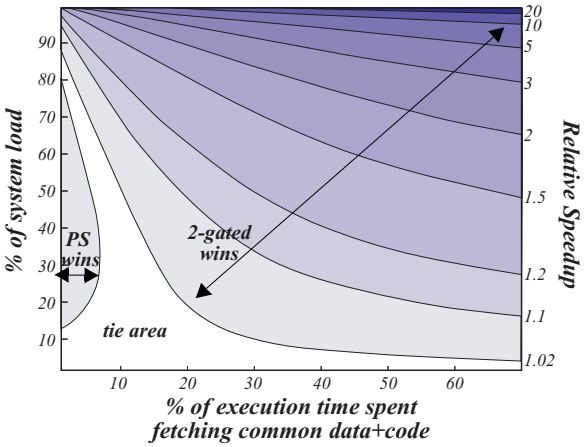

FIGURE 5: *Direct comparison of PS and T-gated(2).*

4 Ongoing and future work

We are currently implementing Qpipe on top of the Shore [Ca+94] storage manager. The reasons for choosing this particular system are: (a) it has a modular code design, (b) it is well-documented, and (c) its behavior closely matches modern commercial DBMS [AD+01]. We have also modified Predator [SLR97], an object-relational DBMS also built on top of Shore, for experiments that require full staged DBMS functionality. The rest of this section describes ongoing and future thesis work. It is organized in four steps (step 3 through step 6):

- Ongoing work focuses on Qpipe on a single-CPU system. We target performance benefits by sharing instructions (step 3, Section 4.1) and data (step 4, Sec. 4.2) across queries at each Qop.
- Future work on Qpipe (step 5, Sec. 4.3) will focus on Qop scheduling, and port the implementation to a new platform to support multi-processor and multi-threaded systems. We will study scheduling algorithms for SMP/SMT systems, and demonstrate a scalable version of Qpipe.
- Future work on the Staged Database design (step 6, Sec. 4.4) will focus on self-tuning and extensibility of a staged DBMS built around Qpipe.

4.1 Step 3: Vertical query execution (ongoing work)

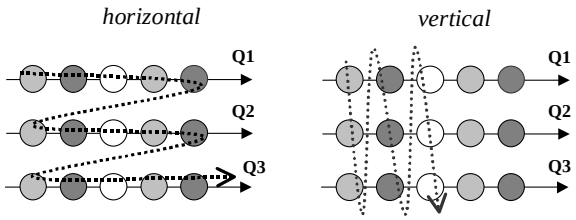

FIGURE 6: *Horizontal vs. vertical execution.*

Commercial database engines typically use a pool of threads (or processes) to evaluate concurrent queries. Each thread is assigned to a query and is responsible for executing part or all of the QEP. When a thread runs, it may invoke multiple operators, depending on the query, before giving up the CPU for a different query (performing a context-switch). We call this type of execution a "horizontal" traverse. Instead, a "vertical" traverse would keep executing the same operator for a group of queries before switching to a different operator. Figure 6 shows how vertical execution works across queries. Same color circles correspond to the same operators. The dotted line shows the order in which the CPU visits each operator for the three concurrent queries. Qpipe applies vertical execution by repeatedly processing the queue of a Qop before the CPU switches to a different Qop.

The potential benefits of vertically executing multiple queries are in sharing instructions and data across different queries at all levels of the memory hierarchy. Table 1 lists the different types of data and instructions in a database execution engine implementation. Instructions are always common across queries and can be shared. *Common* data, such as the database catalog, is accessed by all queries. Although this type of data could be shared at the higher levels of the memory hierarchy, its size is much smaller than a query's private data, and thus, the overall savings are not significant. *Private* data that include the stack of each query cannot be shared. Private data that include intermediate results could be shared when two or more queries compute the same or almost the same intermediate result. *Shared* data, such as database tables and indices, can also be shared when two or more queries access overlapping regions. The last two cases for data sharing are covered in the next step (Section 4.2). The rest of this section examines the potential of instruction sharing across queries in Qpipe.

TABLE 1: Potential for I+D sharing across queries

Instructions	Data			
common	**private-** stack	**private-** intermediate results	**shared** (tables, indices)	**common** (catalog)
	✖			✖

4.1.1 Making Qops I-cache resident

Recent studies [AD+99][LB+98] of database workloads on modern processors have shown that in most cases first-level instruction cache misses are an important stall factor, while L2 instruction misses have a limited effect on overall performance. To examine the potential of instruction sharing at the L1 cache across different queries in Qpipe, we classify Qops into three types: (i) *sQops* which contain relatively sequential code with no loops (typically operating on a single tuple, such as index probe), (ii) *lQops* which contain a code loop (e.g., nested loop join), and (iii) *s+lQops* which have both sequential code and loops (such as reading a page of tuples). lQops and s+lQops with the loop component taking significantly more time than the sequential component, present the least opportunity for sharing instructions across queries, since a loop already reuses instructions to a certain degree. If a s+lQop has equally large sequential and loop components then we may consider breaking it into separate sQops and lQops. If a sQop's code fits in the L1 cache then its instructions can be shared across queries. Otherwise, we will need to (a) identify components of the sQop that fit in the L1 cache and then, (b) apply vertical execution within the sQop.

(a) The first step is to divide the sQop code into L1 cache-contained components. Code segments that fit entirely in the cache can be shared across multiple queries, reducing significantly the number of L1 instruction misses. To effectively apply our methodology the following two conditions must hold: (i) L1 conflict misses are significantly fewer than the cold misses, and (ii) all queries follow almost the same code path, or a subset of the component's instructions. Conflict misses cannot be dealt with at the software level, but we can accommodate different code paths with different classes of Qops. In our experiments so far on a Pentium III we did not encounter any problems with conflict misses. The rule for deciding the limits of each component is that the number of L1 instruction misses must be close but smaller than the number of L1 instruction cache lines. Since each miss causes at least one cache line to be fetched from L2 cache (more if there is instruction prefetching), the total number of misses gives a rough estimate of the component's footprint (assuming few conflict misses, and that instruction prefetching is not affected by a heavy branching pattern). To measure the number of misses for each candidate component we use PAPI [MB+99], a library that provides an API to a processor's hardware performance counters.

(b) Once the code components within a Qop are identified, we consider the following two ways of applying vertical execution within the Qop: (i) We assign each query to a thread and we insert *yield()* operations in the boundaries of each component. By carefully setting the priorities of each thread we can force each code component to execute subsequently for all the related queries. The trade-off in this approach is that the context-switch cost expressed both in cycles required and the code footprint, may eliminate the savings from vertical execution. (ii) The second approach for applying vertical execution is to modify the Qop code to operate on an array of packets (each representing a different query). Code segments are wrapped in a loop and execute once for each query. This approach requires many code modifications but does not have the performance trade-offs of the previous one. Next, we describe a case study of making a Qop I-cache resident.

4.1.2 Case study: Index probe Qop on Pentium III

We examined the index probe (single tuple retrieval) operation of the Shore Storage manager [Ca+94]. This operation can be treated as a sQop, since the code is relatively sequential. Our target platform is the Pentium III processor. Pentium III has a 16KB, 4-way set associative L1 instruction cache with hardware prefetching, consisting of 500 32-byte lines (blocks). It uses the IA-32 instruction set which specifies multiple length instructions, ranging from 1 to 17 bytes. We opted for the second approach in applying vertical execution, changing the code to operate on an array of packets. On the specific platform, a context switch has a 200 lines code footprint and takes 2000-3000 cycles to complete, which negates the savings from vertical execution.

Index probe in Shore consists of the following high-level operations: finding the index structure, perform a B-tree lookup operation, pin and load the located record. These operations may include requests to the lock manager or the I/O module. By inserting in various points of the code calls to the performance hardware counters and measuring the number of L1 instruction misses, we identified 15-20 code segments that could potentially become cache-resident for multiple queries. So far, we have modified parts of the code that account for 35% of the total number of L1-I misses, to operate on an array of packets. To measure the performance improvement in the presence of multiple queries we performed the following experiment. We set up a database consisting of 10 tables of 100,000 tuples each, with each tuple consisting of 25 integers. Initially we run 10 different index probes on the 10 different tables, one after another, and we measured the number of L1-I misses and the total cycles. Then, we repeated the experiment this time using our modified index probe Qop. In this experiment there was no I/O involved and there were no other requests in the system. Even with only about a third of the code modified, the total number of L1-I misses dropped from 26,709 to 19,962 (25.3% reduction), while the total processor cycles dropped from 2,703,377 to 2,273,654 (speedup of 1.19). We are currently optimizing the rest of this Qop.

4.2 Step 4: Run-time query merging (ongoing work)

Vertical execution across multiple queries per Qop also has the benefit of exploiting commonality of intermediate results. Whenever two or more queries have overlapping computations, Qops can naturally detect the overlap and perform the work only once. Consider for example a file scan in progress on a given relation. If a second query arrives at the fscan Qop and wants to start a scan on the same relation, it can take advantage of the existing file scan. It can attach to the incoming tuple stream and read the "missed" tuples later (assuming that the scan order is not important, e.g., it is not a sorted file). This specific technique is actually used by commercial systems [Coo01][Fer94].

Qpipe can detect operator commonality across queries at each Qop. This type of commonality is workload-dependent and has been studied to a certain degree in the past. Work done in multi-query optimization detects common subexpressions at the query optimizing phase [Sel88]. The restriction is that all queries that are probed for common parts must be optimized as a batch before entering the execution engine. Qpipe's query merging techniques are complementary and independent of the optimizer. Furthermore, they dynamically apply during all of a query's lifetime. To illustrate how query merging techniques work, consider the two queries in Figure 7 (note that in this example we consider two sort-based group-by's instead of hash-based). Both queries share an operator subtree and differ only in the root. If Q2 arrives after Q1 is inserted in the execution engine, a traditional optimizer has no means of identifying and taking advantage of the common subexpressions. In Qpipe, Q1 will enqueue packets in all participating Qops. If Q1's packet in sort stays enough time for Q2 to queue up at that Qop, then Q2 can take advantage of all the work that Q1 has been doing.

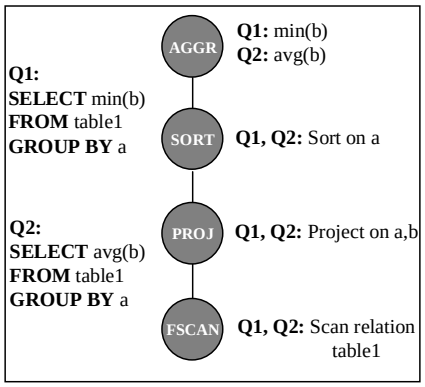

FIGURE 7: *Two "mergeable" query plans*

Query merging is not limited to identical subtrees. If one subtree's results can inexpensively transform into the output of another subtree (using for example projection or additional filtering), then we can still compute the subtree only once. Consider the following two queries:

Q3: SELECT CustID, Amnt, Date FROM Orders ORDER BY CustID

Q4: SELECT CustID, SUM (Amnt) FROM Orders GROUP BY CustID

The sort result in Q3 differs from Q4, but can be easily reused through an additional projection (removing the Date attribute).

4.2.1 Proposed techniques

Recall that a Qpipe packet represents work a query needs to perform at a given Qop. Whenever two or more queries share the same subexpression (same operator subtree), one packet is sufficient to satisfy all queries. This "merged" packet can then copy the result tuples (or pointers to them in our implementation) into the parent buffers of all participating queries. We detect merging opportunities when a new packet is enqueued. During a query's initialization phase we traverse the QEP in prefix order and encode all parameters, operations and attributes pertaining to that query at each operator. When a packet is enqueued in a Qop we compare the encoded expression with the ones of the existing packets and merge accordingly. Note that different consumption rates among the parents of a merged operation may dictate a packet split. Furthermore, file scans on the same relation but with different selectivities can still merge, albeit with different producing rates. Different producing rates may negate the effect of different parent consuming rates but they can also amplify it. Next, we outline packet merging/splitting techniques in fscan, iscan, sort, aggregate, and join operators. The full spectrum of the techniques considered and implemented is in [HAS03].

File scans. Two or more queries that scan the same relation can potentially merge into a single packet serving all common queries. Newly arrived queries can attach to an existing, in-progress scan, and retrieve the "missed" tuples using partial scans of the relation (this technique is also used in [Coo01] and [Fer94]). To avoid a large number of partial scans covering different overlapping and disjoint regions of the relations, Qpipe implements a new, circular form of synchronized file scans that simplifies the management of multiple partial scans. We maintain a separate scan thread that repeatedly scans a relation until no more queries are interested in the results. Tuples are passed to every interested query which checks whether the tuple satisfies their selection predicates. Each query is responsible to track down which tuples have been processed at any time. Instead of splitting on a slow parent, queries can skip any number of tuples and process them during the next round. After receiving every tuple from the relation a query will detach itself from the scan thread.

Index scans. We distinguish between clustered and unclustered index scans. The most straightforward merging technique that works for both types is to merge packets with exactly the same predicates, that have not activated their parent yet. For clustered indices we can still exploit partially overlapping predicates, but the effectiveness heavily depends on the query arrival order and the nature of the overlapping predicates. Non-clustered index scans are typically performed in two phases to avoid incurring an I/O for every tuple that matches the selection predicate. The first step is to build a list of record IDs (RID) of the tuples that match the predicate and sort the list by page ID. Next, the tuples are retrieved by their RIDs in sorted order that guarantees that no page is retrieved multiple times. A new query can arrive arbitrarily late, and still be able to reuse the sorted list of RID that is kept in memory.

Sorting operators. "Stop-and-go" operators [Gra96], such as sort, allow for a wide time window during which newly arrived queries at the same Qop and with the same subexpression can merge with existing ones. Merging is always possible while the existing packet still scans or sorts the file, or has started filling the parent buffer, but the parent operator is not yet activated (in the latter case we first copy the existing packet's parent record buffer into the new packet's buffer). If the existing packet is well ahead into reading the already sorted file, instead of merging, we reuse the already sorted file. For merged packets that have parents with different consuming rates, we first try to double the parent buffer size to avoid splitting. Otherwise, we split the packets by reusing the sorted file.

Aggregating and join operators. For join and aggregate operators, new packets can always merge with existing ones as long as the parent operator hasn't consumed any tuples. Note that a simple aggregate expression (i.e., without a GROUP BY clause) produces a single tuple, so merging is always possible. Splitting is actually more complicated than in the previous operators. We are currently considering the following two alternatives: (a) Materialize the output of the active packet and point the detached packets to that output (this approach is also used in [DS+01]). (b) Traverse the main packet's tree and copy all of its state into the detached packets. The trade-off in these two approaches is additional storage vs. the overhead for creating and activating all the children of the detached packets.

4.2.2 Experimentation

We sent two similar 3-way join queries from the Wisconsin Benchmark [De91] to both our system (Qpipe modified to include the proposed packet merging techniques) and the original execution engine (Predator), running on a 4-way Pentium-III 700MHz SMP server, with 2MB of L2 cache per CPU, 4GB of shared RAM, and Linux 2.4.18. The left part of Figure 8 shows the SQL statement and the execution plan for those queries. We used the integer attributes from the Wisconsin Benchmark for all joining and group-by attributes. We used 100,000 200-byte tuple tables for the big tables (*big1* and *big2*) and 10,000 tuples for the *small* table. The only difference in the two queries is the range of index scanning for table big2 (i.e., the values of X and Y differ in the two que-

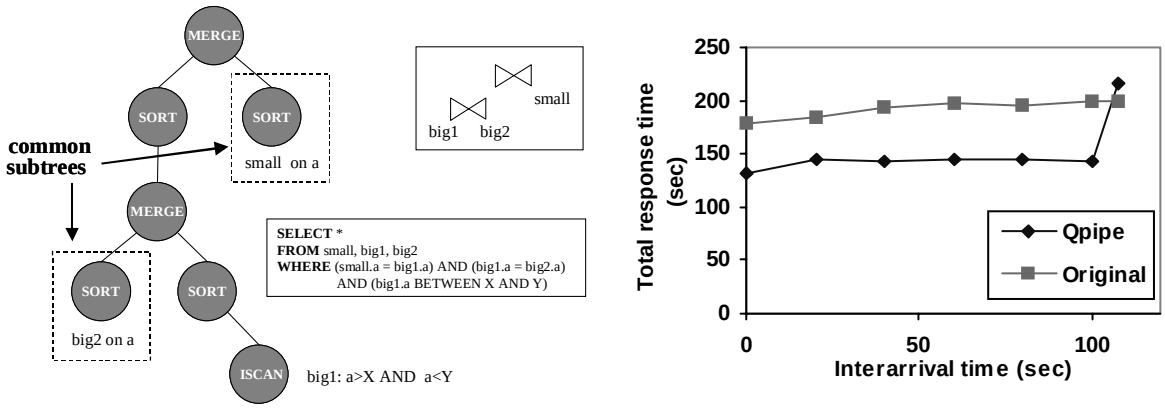

FIGURE 8: *SORT reusing in 2 join queries*

ries). While both joins in the plan are not common across the two queries (because of the different index scan predicates) the actual join algorithm which is sort-merge in this case exhibits commonality in the sort operations. Both queries need to sort tables big2 and small on attribute *a*. As long as the temporary sorted files are used (when created, sorted, or read), Qpipe can avoid duplicating the ongoing work for any newly arrived queries.

The graph in the right part of Figure 8 shows the total elapsed time from the moment the first query arrived until the system is idle again. We vary the interarrival time for the two queries from 0 secs (i.e.,the two queries arrive together) up to the time it takes the first query to finish when alone in the system (i.e., the second query arrives immediately after the first has finished). The graph shows that Qpipe is able to perform packet merging at the sort Qop, thereby exploiting commonality for most of a query's lifetime (that's why the line for Qpipe remains flat most of the time) and provide up to 25% reduction in the total elapsed time. Note that both systems perform better when the queries arrive close to each other, since the system can overlap some of the I/Os. Also, note that when both queries execute with no overlap between them (right most data point) Qpipe results into a slightly higher total elapsed time because of the Qop queue overhead (this overhead actually pays off when there are multiple queries inside Qpipe).

4.3 Step 5: Extensions to Qpipe (future work)

4.3.1 Scheduling techniques for Qpipe and optimizations for storage architectures

The departure from a time-sharing thread-based execution model to a stage-based engine introduces new scheduling problems. The CPU is no longer assigned directly to queries, rather it processes stage queues which include different queries at different phases of their execution. This step will propose and evaluate operator and query scheduling techniques for Qpipe that yield low response time and high throughput, and ensure fairness while at the same time take advantage of code and data commonality. In Qpipe, database tables are accessed through specialized Qops. Because of the Qop queues, I/O requests are issued in groups and not at random points in time. This fact allows the storage manager and the disks to perform deeper and more effective scheduling. We plan to leverage this effect and also further expose the I/O Qops to the underlying storage architecture. Ideally, I/O Qops should be pushed as close to the disks as possible.

4.3.2 Port Qpipe to multi-processor and simultaneous multi-threaded systems

High-end DBMS typically run on clusters of PCs or multi-processor systems. The database software runs either as a different process on each CPU, or as a single process with multiple threads assigned to the different processors. In either case, a single CPU handles a whole query or a random part of it. Qpipe instead naturally maps one or more Qops to a dedicated CPU. Qops may also

migrate to different processors to match the workload requirements. A single query in Qpipe will visit several CPUs during the different phases of its execution. In shared memory systems the query's state and private data remain in one copy as the packets are routed through different processors. In non-shared memory systems, Qop mapping incorporates the overhead of copying packets (and not pointers to them) along with each client's private data. This scheme resembles the architecture of parallel shared-nothing architectures (such as GAMMA [De+90]), where each operator is assigned to a processor and parallel processing techniques are employed in order to minimize the overhead of shipping data between the different CPUs.

A recently introduced processor feature is *simultaneous multithreading* (SMT) [Eg+97] [LB+98], where the processor issues instructions from multiple threads in a single cycle. Widely available processors have already started adopting this feature (Intel's Pentium 4, introduced in 2002, implements a 2-way SMT technique, marketed as *hyper-threading technology*). Since the simultaneously active threads inside the processor core share the same cache hierarchy, they can benefit from increased instruction locality if they execute the same piece of code. Whereas traditional DBMS cannot control which exact piece of code each thread executes at any time, Qpipe can schedule multiple threads belonging to the same Qop and thus exploit instruction locality. This step will port Qpipe to SMP and SMT systems, and will demonstrated Qpipe's scalability by proposing and evaluating techniques for Qop placement, Qop replication, and query scheduling.

4.4 Step 6: Extensions to Staged Database System design (future work)

4.4.1 Self-tuning

We plan to implement a mechanism that will continuously monitor and automatically tune the following four parameters of a staged DBMS:

- *The number of threads at each stage*. This choice entails the same trade-off as the one discussed in Section 3.1.1 but at a much smaller scale. For example, it is easier and more effective for the stage responsible for logging to monitor the I/Os and adjust accordingly the number of threads, rather than doing this for the whole DBMS.
- *The stage size in terms of server code and functionality.* Assuming that the staged DBMS is broken up into many fine-grain self-contained modules, the tuning mechanism will dynamically merge or split stages by reassigning the various server tasks. Different hardware and system load configurations may lead to different module assignments.
- *The page size for exchanging intermediate results among Qops.* This parameter affects the time a Qop spends working on a query before it switches to a different one.
- *The choice of a thread scheduling policy.* We have found that different scheduling policies prevail for different system loads [HA02].

4.4.2 Extensibility: Integrate external wrappers into a staged DBMS

We plan to demonstrate the extensibility of the staged database design by integrating external applications into the staged DBMS code base. Current integrated solutions that use the DBMS as a back-end pay the performance penalty of a longer communication path, processing overhead, and inefficiency in maintaining different caches for essentially the same data. A stage-integrated solution can offer fewer connections, better code and data locality, and the use of a unified cache. This step proposes a staged, high performance integrated Web and Database server, tailored to support cost-efficiently e-commerce applications.

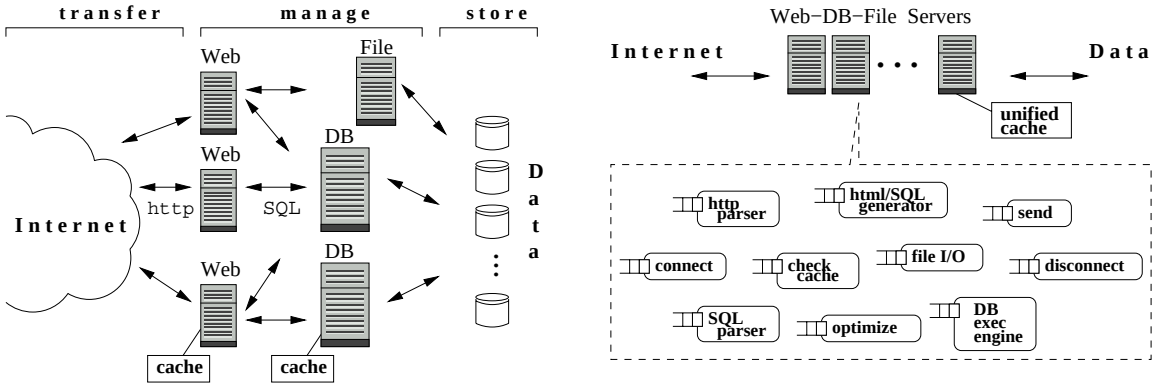

FIGURE 9: *Current Web/DB architectures (left)* and the *proposed integrated Web-DB design (right)*

Figure 9 shows the typical infrastructure of an e-commerce site. Client requests arrive over the internet and communication takes place through the http protocol with the Web servers of the site. The Web servers first try to satisfy the request from their local caches, otherwise they either issue file I/Os (which may be handled by a separate File server) or they form SQL statements and send queries to the Database servers. The DB servers eventually send the result back to the Web server using either their caches or the underlying data repository. There are three distinct phases in this model: transfer - manage - store (data). By encapsulating the Web server functionality into stages and integrate them into a staged database system, we can optimize for the “manage” phase of the model. Similar integration solutions are discussed in [FLM98].

5 Conclusion and summary of goals

Modern database servers suffer from high processor-memory data and instruction transfer delays. Despite the ongoing effort to create locality-aware algorithms, DBMS fail to exploit instruction and data commonality across multiple concurrent queries. Furthermore, the current threaded execution model used in most commercial systems is susceptible to suboptimal performance caused by an inefficient thread allocation mechanism. Looking from a software engineering point of view, years of DBMS software development have lead to monolithic, complicated implementations that are difficult to extend, tune and evolve.

Based on fresh ideas from the OS community [LP02][WCB01] and applying them in the complex context of a DBMS server, this thesis proposal suggests a departure in the way database software is built. The proposal for a staged, data-centric DBMS design remedies the weaknesses of modern commercial database systems by providing solutions at both a hardware and a software engineering level. The goals of the proposed design along with the steps that will satisfy them are:

- Increase DBMS throughput by optimizing accesses to the memory hierarchy in the presence of multiple requests. Steps 3 and 4 will demonstrate techniques to exploit instruction and data commonality across concurrent requests.
- Improve DBMS scalability and memory behavior on multi-processor and simultaneous multi-threaded systems. Step 5 will port Qpipe to SMP/SMT systems, it will evaluate Qop scheduling techniques, and will demonstrate Qpipe’s improved scalability properties.
- Make database systems easier to extend and perform fine-grain tuning. Step 6 will implement efficient self-tuning techniques for staged DBMS, and will also demonstrate an integrated DB/Web server.

The time line is the following:

TABLE 2: Time line

Date	Step
Aug. 2003	3 : instruction commonality for Qpipe
	4 : query merging techniques for Qpipe
Nov. 2003	5a : scheduling techniques for Qpipe
Mar. 2004	5b : port Qpipe to SMP/SMT systems
Aug. 2004	6a : self-tuning for staged DBMS
Jan. 2005	6b : integrated DB/Web server
Feb. 2005	Thesis Defense

References

[AD+01] A. Ailamaki, D. DeWitt, *et al*. "Weaving Relations for Cache Performance." In *Proc. VLDB*, 2001.

[AD+99] A. Ailamaki, D. DeWitt, *et al*. "DBMSs on a modern processor: Where does time go?" In *Proc. VLDB*, 1999.

[AB+91] T. E. Anderson, B. N. Bershad, *et al.* "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism." In *Proc. SOSP-13*, pp 95-109, 1991.

[AH00] R. Avnur and J. M. Hellerstein. "Eddies: Continuously Adaptive Query Processing." In *Proc. SIGMOD*, 2000.

[AWE] Asynchronous Work Elements, IBM/IMS team. Comments from anonymous reviewer, Oct. 2002.

[BB+02] B. Babcock, S. Babu, *et al*. "Models and Issues in Data Stream Systems." Invited paper. In *Proc. PODS*, 2002.

[BM98] G. Banga and J. C. Mogul. "Scalable kernel performance for Internet servers under realistic loads." In *Proc. USENIX*, 1998.

[Be+98] P. Bernstein et al. "The Asilomar Report on Database Research." In *SIGMOD Rec.*, Vol. 27, No. 4, Dec. 1998.

[CH90] M. Carey and L. Haas. "Extensible Database Management Systems." In *SIGMOD Rec.*, 19(4), Dec. 1990.

[Ca+94] M. Carey et al. "Shoring Up Persistent Applications." In *Proc. SIGMOD*, 1994.

[CW00] S. Chaudhuri and G. Weikum. "Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System." In *Proc. VLDB*, 2000.

[CHM95] C. Chekuri, W. Hasan, and R. Motwani. "Scheduling Problems in Parallel Query Optimization." In *Proc. PODS*, pp. 255-265, 1995.

[CGM01] S. Chen, P. Gibbons, and T. Mowry. "Improving Index Performance through Prefetching." In *SIGMOD*, 2001.

[CLH00] T. M. Chilimbi, J. R. Larus, and M. D. Hill. "Making Pointer-Based Data Structures Cache Conscious." In *IEEE Computer*, Dec. 2000.

[Coo01] C. Cook. "Database Architecture: The Storage Engine". Miscrosoft SQL Server 2000 Technical Article, July 2001. Available online at: http://msdn.microsoft.com/library/

[De+90] D. DeWitt et al. "The Gamma Database Machine Project." In *IEEE TKDE*, 2(1), pp. 44-63, Mar. 1990.

[De91] D. DeWitt. "The Wisconsin Benchmark: Past, Present, and Future." *The Benchmark Handbook*, J. Gray, ed., Morgan Kaufmann Pub., San Mateo, CA (1991).

[DG92] D. DeWitt and J. Gray. "Parallel Database Systems: The Future of High Performance Database Systems." In *Communications of the ACM*, Vol. 35, No. 6, June 1992.

[DS+01] N. N. Dalvi, S. K. Sanghai, *et al*. "Pipelining in Multi-Query Optimization." In *Proc. PODS*, 2001.

[Eg+97] S. Eggers, *et al.* “Simultaneous multithreading: A platform for next-generation processors.” In *IEEE Micro* pp. 12-19, Oct. 1997.

[Fer94] P. Fernandez. “Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms.” In *SIGMOD*, 1994.

[FLM98] D. Florescu, A. Y. Levy, and A. O. Mendelzon. “Database Techniques for the World-Wide Web: A Survey”. ACM SIGMOD Record Vol. 27, No. 3, Sep. 1998: 59-74.

[GL01] G. Graefe and P. Larson. “B-Tree Indexes and CPU Caches.” In *Proc. ICDE*, pp. 349-38, 2001.

[Gra96] G. Graefe. “Iterators, Schedulers, and Distributed-memory Parallelism.” In *Software-practice and experience*, Vol. 26 (4), pp. 427-452, Apr. 1996.

[HA03] S. Harizopoulos and A. Ailamaki. “A Case for Staged Database Systems.” In *Proc. CIDR*, 2003.

[HAS03] S. Harizopoulos, A. Ailamaki, and V. Shkapenyuk. “Vertically Pipelined Query Execution.” *In preparation.*

[HA02] S. Harizopoulos and A. Ailamaki. “Affinity scheduling in staged server architectures.” *TR CMU-CS-02-113.*

[HP96] J. Hennessy and D. Patterson. “Computer Architecture: A Quantitative Approach.” Morgan Kaufmann, 1996.

[JK99] J. Jayasimha and A. Kumar. “Thread-based Cache Analysis of a Modified TPC-C Workload.” In *Proc. 2nd CAECW Workshop*, 1999.

[IBM01] IBM DB2 Universal Database V7 Manuals. “Administration Guide V7.2, Volume 3: Performance,” ftp://ftp.software.ibm.com/ps/products/db2/info/vr7/pdf/letter/db2d3e71.pdf

[KP+98] K. Keeton, D. A. Patterson, *et al*. “Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads.” In *Proc. ISCA-25*, pp. 15-26, 1998.

[Lar02] Paul Larson. Personal communication, June 2002.

[LP02] J. R. Larus and M. Parkes. “Using Cohort Scheduling to Enhance Server Performance.” In *Proc. USENIX*, 2002.

[LB+98] J. L. Lo, L. A. Barroso, *et al.* “An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors.” In *Proc. ISCA*, 1998.

[MB+99] P. J. Mucci, S. Browne, *et al*. “PAPI: A Portable Interface to Hardware Performance Counters.” In Proc. Department of Defense HPCMP Users Group Conference, Monterey, CA, June 7-10, 1999.

[MDO94] A. M. G. Maynard, C. M. Donelly, and B. R. Olszewski. “Contrasting Characteristics and Cache Performance of Technical and Multi-user Commercial Workloads.” In *Proc. ASPLOS-6*, 1994.

[MS00] Microsoft SQL Server 2000. Documentation available online at: *http://msdn.microsoft.com/library*

[Ous96] J. K. Ousterhout. “Why threads are a bad idea (for most purposes).” Invited talk at 1996 *USENIX* Tech. Conf. (slides available at http://home.pacbell.net/ouster/), Jan. 1996.

[PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. “Flash: An Efficient and Portable Web Server.” In *USENIX*, 1999.

[RB+01] A. Ramirez, *et al.* “Code layout optimizations for transaction processing workloads.” In *Proc.* ISCA 2001.

[RB+95] M. Rosenblum, E. Bugnion, *et al.* “The Impact of Architectural Trends on Operating System Performance.” In *Proc. SOSP-15*, 1995.

[RS+00] P. Roy, *et al*. “Efficient and Extensible Algorithms for Multi Query Optimization.” In *Proc. SIGMOD*, 2000.

[Sel88] T. Sellis. “Multiple Query Optimization.” In *ACM Transactions on Database Systems*, 13(1):23-52, Mar. 1988.

[SLR97] P. Seshadri, M. Livny, and R. Ramakrishnan. “The Case for Enhanced Abstract Data Types.” In *VLDB*, 1997.

[SKN94] A. Shatdal, C. Kant, and J. Naughton. “Cache Conscious Algorithms for Relational Query Processing.” In *Proc. VLDB*, pp. 510-521, 1994.

[SZ+96] A. Silberschatz, S. Zdonik *et al*. “Strategic Directions in Database Systems - Breaking Out of the Box.” *ACM Computing Surveys* Vol.28, No.4, pp. 764-778, Dec. 1996.

[SW+76] M. Stonebraker, E. Wong, *et al*. “The Design and Implementation of INGRES.” In *ACM TODS*, 1(3), 1976.

[UF01] T. Urhan and M. J. Franklin. “Dynamic Pipeline Scheduling for Improving Interactive Query Performance.” In *Proc. VLDB*, 2001.

[WCB01] M. Welsh, D. Culler, and E. Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services.” In *Proc. SOSP-18*, 2001.

[Wie92] G. Wiederhold. “Mediators in the Architecture of Future Information Systems.” In *IEEE Computer*, 25:3, pp. 38-49, Mar. 1992.