A Case for Staged Database Systems

Stavros Harizopoulos

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
stavros@cs.cmu.edu

Anastassia Ailamaki

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
natassa@cmu.edu

Abstract

Traditional database system architectures face a rapidly evolving operating environment, where millions of users store and access terabytes of data. In order 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 when executing workloads with changing requirements. Moreover, the monolithic approach in DBMS software has lead to complex and difficult to extend designs.

This paper introduces a staged design for high-performance, evolvable DBMS that are easy to tune and maintain. We propose to break the database system into modules and to 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 both a hardware and a software engineering level.

1 Introduction

Advances in processor design, storage architectures and communication networks, and the explosion of the Web, have allowed storing and accessing terabytes of information online. DBMS play a central role in today's volatile IT landscape. They are responsible for executing time-critical operations and supporting an increasing base of millions of users. To cope with these high demands modern database systems (a) use a work-centric multi-threaded (or multi-process) execution model for parallelism, and (b) employ a multitude of sophisticated tools for server performance and usability. 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.

Several database researchers have indicated 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 ever-increasing processor/memory speed gap [HP96] affects commercial database server performance more than other engineering, scientific, or desktop applications. Database workloads exhibit large instruction footprints and tight data dependencies that reduce instruction-level parallelism and incur data and instruction transfer delays [AD+99] [KP+98]. As future systems are expected to have deeper memory hierarchies, a more adaptive programming solution becomes necessary to best utilize available hardware resources and sustain high performance under massive concurrency.

This paper introduces the Staged Database System design for high-performance, evolvable DBMS that are easy to tune and maintain. We 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

Proceedings of the 2003 CIDR Conference

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 SMP systems, and (b) at a software engineering level: it aims at a highly flexible, extensible, easy to program, monitor, tune and evolve platform. The paper's contributions are threefold: (i) it provides an analysis of design shortcomings in modern DBMS software, (ii) it describes a novel database system design along with our initial implementation efforts, and (iii) it presents new research opportunities.

The rest of the paper is organized as follows. The next section reviews related work both in the database and the operating systems community. Section 3 discusses modern commercial DBMS problems that arise from under-exploitation of memory resources and a complex software design. Next, in Section 4 we present the Staged Database System design along with a scheduling trade-off analysis and our initial implementation effort. Section 5 shows how the Staged Database System design overcomes the problems of Section 3, and Section 6 summarizes the paper's contributions.

2 Related research efforts

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]. This section discusses representative pieces of work from a broad scope of research in databases, operating systems, and computer architecture.

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, data characteristics, and user preferences [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. For example, *eddies* may benefit by repeatedly executing different queries at one operator, or by increasing the tuple processing granularity (we discuss similar trade-offs over the next sections).

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 workloads consisting of multiple short transactions, most misses occur due to conflicts between threads whose working sets replace each other in the cache [JK99][RB+95].

Recently, OS research introduced the staged server programming paradigm [LP02], that divides computation into stages and schedules requests within each stage. The CPU processes the entire stage queue while traversing the stages going first forward and then backward. The authors demonstrate that their approach improves the performance of a simple web server and a publish-subscribe server by reducing the frequency of cache misses in both the application and operating system code. While the experiments were successful, significant scheduling trade-offs remain unsolved. For instance, it is not clear under which circumstances a policy should delay a request before the locality benefit disappears.

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, thereby preventing resource overcommitment when demand exceeds service capacity. SEDA does not optimize for memory hierarchy performance, which is the primary bottleneck for data-intensive applications.

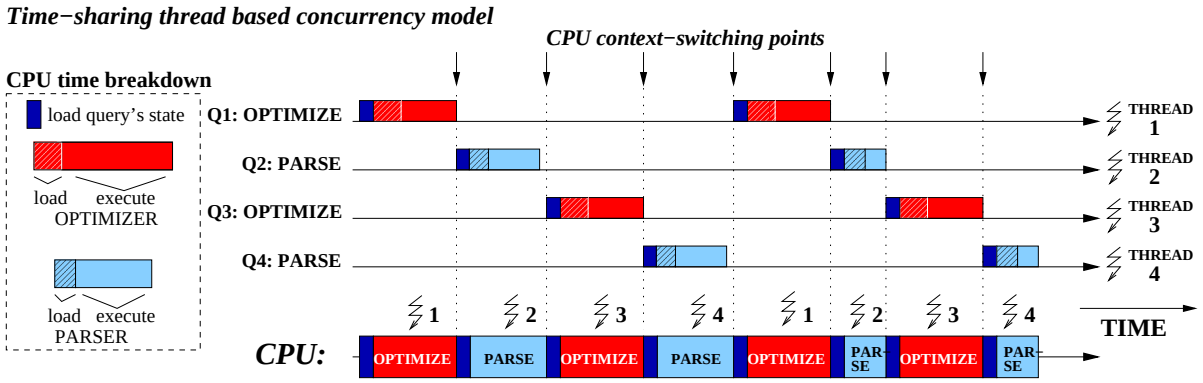

FIGURE 1: *Uncontrolled context-switching can lead to poor performance.*

Finally, computer architecture research addresses the ever-increasing processor-memory speed gap [HP96] by exploiting data and code locality and by minimizing memory stalls. Modern systems employ mechanisms ranging from larger and deeper memory hierarchies to sophisticated branch predictors and software/hardware prefetching techniques. *Affinity scheduling* explicitly routes tasks to processors with relevant data in their caches [SL93][SE94]. However, frequent switching between threads of the same program interleaves unrelated memory accesses, thereby reducing locality. We address memory performance from a single application's point of view, improving locality across its threads.

To summarize, research in databases has proposed (a) cache-conscious schemes to optimize query execution algorithms, and (b) modular or pipelined designs for parallelism, extensibility, or continuous query performance. The OS community has proposed (a) techniques for efficient threading support, (b) event-driven designs for scalability, and (c) locality-aware staged server designs. This paper applies the staged server programming paradigm on the sophisticated DBMS architecture, discusses the design challenges, and highlights the performance, scalability, and software engineering benefits.

3 Problems in current DBMS design

Our work is motivated by two observations, the first of which has received less attention to date. First, 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 increase, memory-related delays and context-switch overheads hurt DBMS performance even more. Second, the monolithic design of today's DBMS software has lead to complex systems that are difficult to maintain and extend. This section discusses problems related to these two observations.

3.1 Pitfalls of thread-based concurrency

Modern database systems adopt a thread-based concurrency model for executing coexisting query streams. To best utilize the available resources, DBMS typically use a pool of threads or processes[1]. Each incoming query is handled by one or more threads, depending on its complexity and the number of available CPUs. Each thread executes until it either blocks on a synchronization condition, or an I/O event, or until a predetermined time quantum has elapsed. Then, the CPU switches context and executes a different thread [IBM01] or the same thread takes on a different task (SQL Server [Lar02]). Context-switching typically relies on generated events instead of program structure or the query's current state. While this model is intuitive, it has several shortcomings:

1. There is no single number of preallocated worker threads that yields optimal performance under changing workloads. Too many threads waste resources and too few threads restrict concurrency.
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 thread scheduling does not exploit cache contents that may be common across a set of threads. When selecting the next thread to run, the scheduler ignores that a different thread might benefit from already fetched data.

These three shortcomings are depicted in Figure 1. In this hypothetical execution sequence, 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

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.

the CPU resumes execution on a query, it first spends some time loading (fetching 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. These shortcomings, along with their trade-offs and challenges are further discussed over the next paragraphs.

3.1.1 Choosing the right thread pool size

Although multithreading is an efficient way to mask I/O and network latencies and fully exploit multiprocessor platforms, many researchers argue against thread scalability [Ous96][PDZ99][WCB01]. Related studies suggest (a) maintaining a thread pool that continuously picks clients from the network queue to avoid the cost of creating a thread per client arrival, and (b) adjusting the pool size to avoid an unnecessarily large number of threads. Modern commercial DBMS typically adopt this approach [IBM01] [Lar02]. The database administrator (DBA) is responsible for statically adjusting the thread pool size. The trade-off the DBA faces is that a large number of threads may lead to performance degradation caused by increased cache and TLB misses, and thread scheduling overhead. On the other hand, too few threads may restrict concurrency, since all threads may block while there is work the system could perform. The optimal number of threads depends on workload characteristics which may change over time. This further complicates tuning[2].

To illustrate the problem, we performed the following experiment using PREDATOR [SLR97], a research prototype DBMS, on a 1GHz Pentium III server with 512MB RAM and Linux 2.4. We created two workloads, A and B, designed after the Wisconsin benchmark

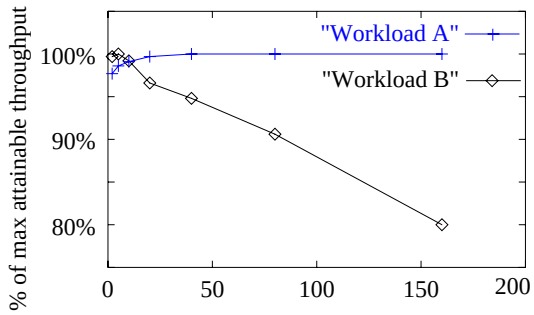

FIGURE 2: *Different workloads perform differently as the number of threads changes.*

[De91]. Workload A consists of short (40-80msec), selection and aggregation queries that almost always incur disk I/O. Workload B consists of longer join queries (up to 2-3 secs) on tables that fit entirely in main memory and the only I/O needed is for logging purposes. We modified the execution engine of PREDATOR and added a queue in front of it. Then we converted the thread-per-client architecture into the following: a pool of threads that picks a client from the queue, works on the client until it exits the execution engine, puts it on an exit queue and picks another client from the input queue[3]. By filling the input queue with already parsed and optimized queries, we could measure the throughput of the execution engine under different thread pool sizes.

Figure 2 shows the throughput achieved under both workloads, for different thread pool sizes, as a percentage of the maximum throughput possible under each workload. Workload A's throughput reaches a peak and stays constant for a pool of twenty or more threads. When there are fewer than twenty threads, the I/Os do not completely overlap, and thus there is idle CPU time resulting in slightly lower throughput[4]. On the other hand, Workload B's throughput severely degrades with more than 5 threads, as there is no I/O to hide and a higher number of longer queries interfere with each other as the pool size increases. The challenge is to discover an adaptive mechanism with low-overhead thread support that performs consistently well under frequently changing workloads.

3.1.2 Preemptive context-switching

A server's code is typically structured as a series of logical operations (or procedure calls). Each procedure (e.g., query parsing) typically includes one or more sub-procedures (e.g., symbol checking, semantic checking, query rewriting). Furthermore, each logical operation typically

2. Quoting the DB2 performance tuning manual [IBM01] ("agents" are implemented using threads or processes): "If you run a decision-support environment in which few applications connect concurrently, set {num_pool_agents} to a small value to avoid having an agent pool that is full of idle agents. If you run a transaction-processing environment in which many applications are concurrently connected, increase the value of {num_pool_agents} to avoid the costs associated with the frequent creation and termination of agents."

3. We also had to convert the system's non-preemptive threads into "cooperating" ones: an alarm timer was causing context-switches roughly every 10msec.

4. We used user-level threads which have a low context-switch cost. In systems that use processes or kernel threads (such as DB2), increased context-switch costs have a greater impact on the throughput.

consists of loops (e.g., iterate over every single token in the SQL query and symbol table look-ups). When the client thread executes, all global procedure and loop variables along with the data structures that are frequently accessed (e.g., symbol table) consist the thread's working set. Context-switches that occur in the middle of an operation evict its working set from the higher levels of the cache hierarchy. As a result, each resumed thread often suffers additional delays while re-populating the caches with its evicted working set.

However, replacing preemption with cooperative scheduling where the CPU yields at the boundaries of logical operations may lead to unfairness and hurt average response time. The challenge is to (a) find the points at which a thread should yield the CPU, (b) build a mechanism that will take advantage of that information, and (c) make sure that no execution path holds the CPU too long, leading to unfairness.

3.1.3 Round-robin thread scheduling

The thread scheduling policy is another factor that affects memory affinity. Currently, selection of the next thread to run is typically done in a round-robin fashion among equal-priority threads. The scheduler considers thread statistics or properties unrelated to its memory access patterns and needs. There is no way of coordinating accesses to common data structures among different threads in order to increase memory locality.

Table 1 shows an intuitive classification of commonality in data and code references in a database server. *Private* references are those exclusive to a specific instance of a query. *Shared* references are to data and code accessible by any query, although different queries may access different parts. Lastly, *common* references are those accessed by the majority of queries. Current schedulers miss an opportunity to exploit *shared* and *common* references and increase performance by choosing a thread that will find the largest amount of data and code already fetched in the higher levels of the memory hierarchy.

In order to quantify the performance penalty, we performed the following experiment. We measured the time it takes for two similar, simple selection queries to pass through the parser of PREDATOR under two scenarios: (a) after the first query finishes parsing, the CPU works on different, unrelated operations (i.e. optimize, scan a table) before it switches into parsing the second query, and, (b) the second query starts parsing immediately after the first query is parsed (the first query suspends its execution after exiting the parser). Using the same setup as in 3.1.1, we found that Query 2 improves its parsing time by 7% in the second scenario, since it finds part of the parser's code and data structures already in the server's cache. As shown in simulation results in Section 4.2, even such a modest average improvement across all server modules results into more than 40% overall response time improvement when running multiple concurrent queries at high system load (the full simulation results are described elsewhere [HA02]).

TABLE 1: Data and code references across all queries

classification	data	code
PRIVATE	Query Execution Plan, client state, intermediate results	NO
SHARED	tables, indices	operator specific code (i.e. nested-loop vs. sort-merge join)
COMMON	catalog, symbol table	rest of DBMS code

However, a thread scheduling policy that suspends execution of certain queries in order to make the best use of the memory resources may actually hurt average response times. The trade-off is between decreasing cache misses by scheduling all threads executing the same software module while increasing response time of other queries 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.

3.2 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. Typically, it is deployed in conjunction with other applications and services. Two common usage scenarios are the following: (a) Data streams from different sources and in different form (e.g., XML or web data) pass through "translators" (middleware) which act as an interface to the DBMS. (b) Different applications require different logic which is built by the system programmer on top of the DBMS. These scenarios may deter administrators from using a DBMS as it may not be necessary for simple purposes, or it may not be worth the time spent in configurations. A compromise is to use plain file servers that will cover most needs but will lack in features. DBMS require the rest of the services and applications to communicate with each other and coordinate their accesses through the database. The overall system performance degrades since there is unnecessary CPU computation and communication latency on the data path. The alternative, extending the DBMS to handle all data conversions and application logic, is a difficult process, since typically there is no well-defined API and the exported functionality is limited due to security concerns.

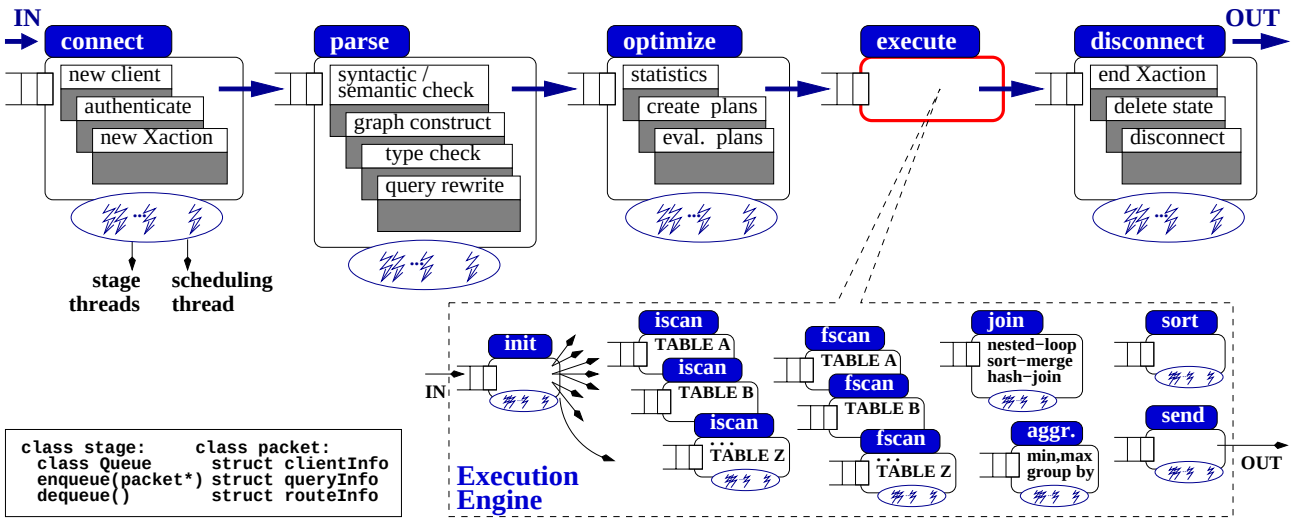

FIGURE 3: ***The Staged Database System design:*** *Each stage has its own queue and thread support. New queries queue up in the first stage, they are encapsulated into a "packet", and pass through the five stages shown on the top of the figure. A packet carries the query's "backpack": its state and private data. Inside the execution engine a query can issue multiple packets to increase parallelism.*

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. For example, the optimizer may need separate tuning (e.g., to reduce search space), or the disk read-ahead mechanism may need adjustment. Current database software can only monitor resource or component utilization at a coarse granularity (e.g., total disk traffic or table accesses, but not concurrent demand to the lock table). Based solely on this information it is difficult to build automatic tuning tools to ease DBMS administration. Furthermore, when client requests exceed the database server's capacity (overload conditions) then new clients are either rejected or they experience significant delays. Yet, some of them could still receive fast service (e.g., if they only need a cached tuple).

Maintainability. An often desirable property of a software system is the ability to improve its performance or extend its functionality by releasing software updates. New versions of the software may include, for example, faster implementations of some algorithms. For a complex piece of software, such as a DBMS, it is a challenging process to isolate and replace an entire software module. This becomes more difficult when the programmer has no previous knowledge of the specific module implementation. The difficulty may also rise from a non-modular coding style, extended use of global variables, and module interdependencies.

Testing and debugging. Large software systems are inherently difficult to test and debug. The test case combinations of all different software components and all possible inputs are practically countless. Once errors are detected, it is difficult to trace bugs through millions of lines of code. Furthermore, multithreaded programs may exhibit race conditions (when there is need for concurrent access to the same resource) that may lead to deadlocks. Although there are tools that automatically search program structure in run-time to expose possible race conditions [SB+97] they may slow down the executable or increase the time to software release. A monolithic software design makes it even more difficult to develop code that is deadlock-free since accesses to shared resources may not be contained within a single module.

4 A staged approach for DBMS software

This section describes a staged design for high-performance, scalable DBMS that are easy to tune and extend. Section 4.1 presents the design overview and Section 4.2 describes performance improvement opportunities and scheduling trade-offs. The results are drawn from experiments on a simulated staged database server. Finally, Section 4.3 discusses the current status of our ongoing implementation on top of an existing prototype DBMS, while Section 4.4 outlines additional design issues.

4.1 Staged database system design

A staged database system consists of a number 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 3): *connect*, *parse*, *optimize*, *execute*, and *disconnect*. The *execute* stage typically represents the largest part of a query's lifetime and is further decomposed into several stages (as described in Section 4.1.2). 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 3 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 3 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).

There are two key elements in the proposed system: (a) the stage definition along with the capabilities of stage communication and data exchange, and (b) the redesign of the relational execution engine to incorporate a staged execution scheme. These are discussed next.

4.1.1 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 (the *stage threads*) that continuously call *dequeue* on the stage's queue, and one thread reserved for scheduling purposes (the *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. Whenever *enqueue* causes the next stage's queue to overflow we apply back-pressure flow control by suspending the *enqueue* operation (and subsequently freeze the query's execution thread in that stage). The rest of the queries that do not output to the blocked stage will continue to run.

4.1.2 A staged relational execution engine

In our design, each relational operator is assigned to a stage. This assignment is based on the operator's physical implementation and functionality. We group together operators which use a small portion of the common or shared data and code (to avoid stage and scheduling overhead), and separate operators that access a large common code base or common data (to take advantage of a stage's affinity to the processor caches). The dashed box in Figure 3 shows the execution engine stages we are currently considering (these are further discussed in Section 4.3).

Although control flow amongst operators/stages still uses packets as in the top-level DBMS stages, data exchange within the execution unit exhibits significant peculiarities. Firstly, stages do not execute sequentially anymore. Secondly, multiple packets (as many as the different operators involved) are issued per each active query. Finally, control flow through packet enqueueing happens only once per query per stage, when the operators/stages are activated. This activation occurs in a bottom-up fashion with respect to the operator tree, after the *init* stage enqueues packets to the leaf node stages (similarly to the "push-based" model [Gra96] that aims at avoiding early thread invocations). Dataflow takes place through the use of intermediate result buffers and page-based data exchange using a producer-consumer type of operator/stage communication.

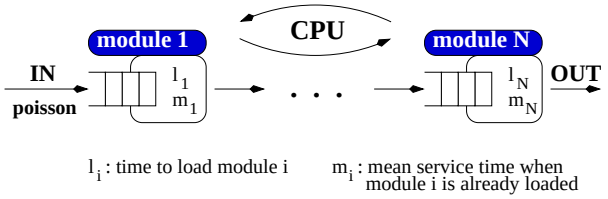

FIGURE 4: *A production-line model for staged servers.*

4.2 The scheduling trade-off

Staged architectures exhibit a fundamental scheduling trade-off (mentioned in Section 3.1.3): On one hand, all requests executing as a batch in the same module benefit from fewer cache misses. On the other hand, each completed request suspends its progress until the rest of the batch finishes execution, thereby increasing response time. This section demonstrates that an appropriate solution to the scheduling trade-off translates into a significant performance advantage for the staged DBMS design. We summarize our previous work [HA02] in the generalized context of a simulated single-CPU database server that follows a "production-line" operation model (i.e. requests go through a series of stages only once and always in the same order).

To compare alternative strategies for forming and scheduling query batches at various degrees of inter-query locality, we developed a simple simulated execution environment that is also analytically tractable. Each submitted query passes through several stages of execution that contain a server module (see Figure 4). 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 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 (the scheduling alternatives are described and evaluated elsewhere [HA02]).

Figure 5 compares the query mean response time for a server consisting of 5 modules with an equal service

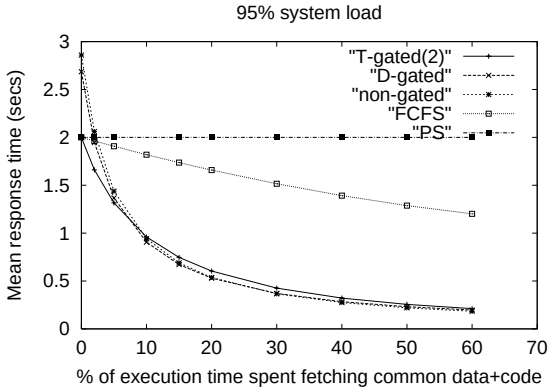

FIGURE 5: *Mean response times for 95% system load.*

time breakdown (other configurations did not alter the results). The graph shows the performance of PS, First Come First Serve, and three of the proposed policies for a system load of 95% and for various module loading times (the time it takes all modules to fetch the common data structures and code in the cache, quantity *l*). This quantity varies as a percentage of the mean query CPU demand, from 0% to 60% (the mean query service time that corresponds to private data and instructions, *m*, is adjusted accordingly so that *m*+*l*=100ms). This value (*l*) can also be viewed as the percentage of execution time spent servicing cache misses, attributed to common instructions and data, under the default server configuration (e.g. using PS). The results of Figure 5 show that the proposed algorithms outperform PS for module loading times that account for more than 2% of the query execution time. Response times are up to twice as fast and improve as module load time becomes more significant. Referring to the experiment of Section 3.1.3, the 7% improvement in the execution time of the second query corresponds to the time spent by the first query fetching the parser's common data and code. Figure 5 shows that a system consisting of modules with similar code and data overlap can improve the average query response time by 40%.

4.3 Implementing a staged database system

We are currently building a staged mechanism on top of PREDATOR [SLR97], a single-CPU[5], multi-user, client-server, object-relational database system that uses 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) it is currently actively maintained. Our approach includes three steps: (1) identifying stage boundaries in the base system and modifying the code to follow the staged paradigm (these were relatively straightforward changes that transformed the base code into a series of procedure calls and are not further discussed here), (2) adding support for stage-

5. Section 5.3 discusses how our design applies to SMPs.

aware thread scheduling techniques, and (3) implementing page-based dataflow and queue-based control-flow schemes inside the execution engine.

Thread scheduling. SHORE provides non-preemptive user-level threads that typically restrict the degree of concurrency in the system since a client thread yields only upon I/O events. We use this behavior to explicitly control the points at which the CPU switches thread execution. We incorporated the proposed affinity scheduling schemes (from Section 4.2) into the system's thread scheduling mechanism by rotating the thread group priorities among the stages. For example, whenever the CPU shifts to the *parse* stage, the stage threads receive higher priority and keep executing *dequeue* until either (a) the queue is empty, or (b) the global scheduler imposes a gate on the queue, or (c) all working threads are blocked on a I/O event. The thread scheduler tries to overlap I/O events as much as possible within the same stage.

Execution engine. Our implementation currently maps the different operators into five distinct stages (see also Figure 3): *file scan (fscan)* and *index scan (iscan)*, for accessing stored data sequentially or with an index, respectively, *sort*, *join* which includes three join algorithms, and a fifth stage that includes the aggregate operators (min-max, average, etc.). The *fscan* and *iscan* stages are replicated and are separately attached to the database tables. This way, queries that access the same tables can take advantage of relations that lie already in the higher levels of the memory hierarchy. We implemented a producer-consumer type of operator/stage communication, through the use of intermediate result buffers and page-based data exchange. Activation of operators/stages in the operator tree of a query starts from the leaves and continues in a bottom-up fashion, to further increase code locality (this is essentially a "page push" model). Whenever an operator fills a page with result tuples (i.e. a join's output or the tuples read by a scan) it checks for parent activation and then places that page in the buffer of the parent node/stage. The parent stage is responsible for consuming the input while the children keep producing more pages. A stage thread that cannot momentarily continue execution (either because the output page buffer is full or the input is empty) enqueues the current packet in the same stage's queue for later processing.

Current status. Our system in its current state and with only 2,500 new lines of code (PREDATOR is 60,000 lines of C++) supports most of the primary database functionality of the original system. We are currently integrating the following into our implementation: updates, inserts, deletes, transactions, data definition language support, and persistent client sessions. We are also considering a finer granularity in the stage definition in order to find an optimal setting for different workloads and system configurations. We expect the high-level design of Figure 3 to remain mostly the same since: (a) the existing stages can add new routing information (i.e. bypassing the optimizer on a DDL statement), and (b) additional stages will include the new functionality (i.e. a "create/delete table" stage inside the execution engine). A similar approach can apply to the process of "staging" a commercial DBMS which is far more complicated than our prototype. The wizards, tools and statistic collection mechanisms may be assigned to new stages, while complicated stages (such as the optimizer) may break down into several smaller stages.

4.4 Additional design issues

Stage granularity. There are several trade-offs involved in the stage size (amount of server code and data structures attached). On the one end the system may consist of just five high-level stages, same as those on the top part of Figure 3. While this break-up requires a minimum redesign for an existing DBMS, it may fail to fully exploit the underlying memory hierarchy, since the large stage code base and data structures will not entirely fit in the cache. Furthermore, a scheme like that still resembles the original monolithic design. On the other end, and depending also on the characteristics of the server's caches, the system may consist of many, fine-granularity modules (i.e. a log manager, concurrency control, or a B-tree module). In that case a stage is attached to every data structure, making the query execution totally data-driven. This scheme requires a total redesign of a DBMS. Furthermore, the large number of self-executing stages may cause additional overheads related to queueing delays and resource overcommitment. Our initial approach towards addressing this trade-off is to create many self-contained modules and decide their assignment into stages during the tuning process (which is discussed next).

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

(a) *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.

(b) *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.

(c) *The page size for exchanging intermediate results among the execution engine stages.* This parameter affects the time a stage spends working on a query before it switches to a different one.
(d) *The choice of a thread scheduling policy.* We have found that different scheduling policies prevail for different system loads [HA02].

5 Benefits of staged DBMS design

This section discusses the benefits and advantages of the staged DBMS design over traditional database architectures. Section 5.1 shows how the staged execution framework can solve the thread-based concurrency problems discussed in 3.1. Next, in Section 5.2 we describe the software engineering benefits of the design stemming from its modular, self-containing nature. Section 5.3 and 5.4 discuss additional opportunities for future research.

5.1 Solutions to thread-based problems

The Staged DBMS design avoids the pitfalls of the traditional threaded execution model as those were described in Section 3.1 through the following mechanisms:

1. Each stage allocates worker threads based on its functionality and the I/O frequency, and not on the number of concurrent clients. This way there is a well-targeted thread assignment to the various database execution tasks at a much finer granularity than just choosing a thread pool size for the whole system.
2. A stage contains DBMS code with one or more logical operations. Instead of preempting the current execution thread at a random point of the code (whenever its time quantum elapses), a stage thread voluntarily yields the CPU at the end of the stage code execution. This way the thread's working set is evicted from the cache at its shrinking phase and the time to restore it is greatly reduced. This technique can also apply to existing database architectures.
3. The thread scheduler repeatedly executes tasks queued up in the same stage, thereby exploiting stage affinity to the processor caches. The first task's execution fetches the common data structures and code into the higher levels of the memory hierarchy while subsequent task executions experience fewer cache misses. This type of scheduling cannot easily apply to existing systems since it would require annotating threads with detailed application logic.

5.2 Solutions to software-complexity problems

Flexible, extensible and evolvable design. Stages provide a well-defined API and thus make it easy to:

1. Replace a module with a new one (e.g., a faster algorithm), or develop and plug modules with new functionality. The programmer needs to know only the stage API and the limited list of global variables.
2. Route packets through modules with the same basic functionality but different complexity. This facilitates run-time functionality decisions. For example, important transactions may pass through a module with a sophisticated recovery mechanism.
3. Debug the code and build robust software. Independent teams can test and correct the code of a single stage without looking at the rest of the code. While existing systems offer sophisticated development facilities, a staged system allows building more intuitive and easier to use development tools.
4. Encapsulate external wrappers or "translators" into stages and integrate them into the DBMS. This way we can avoid the communication latency and exploit commonality in the software architecture of the external components. For example, a unified buffer manager can avoid the cost of subsequent look-ups into each component's cache. A well-defined stage interface enables the DBMS to control distribution of security privileges.

Easy to tune. Each stage provides its own monitoring and self-tuning mechanism. The utilization of both the system's hardware resources and software components (at a stage granularity) can be exploited during the self-tuning process. Each stage is responsible for adjusting all stage-related parameters: the number of threads, buffer space, slice of CPU time, scheduling policies, and routing rules. Although there are more parameters to tune than in a traditional DBMS, it is easier to build auto-tuning tools. Stage autonomy eliminates interdependencies and facilitates performance prediction. Furthermore, under overload conditions, the staged design can (a) quickly push through the system requests easily served (i.e. request for a cached tuple) and (b) respond by routing packets through alternative, faster stages (i.e. trading off accuracy for speed) and thus momentarily increase server capacity. Back-pressure packet flow techniques ensure that all modules can reach near-maximum utilization.

5.3 Multi-processor 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. A staged system instead naturally maps one or more stages to a dedicated CPU. Stages may also migrate to different processors to match the workload requirements. A single query visits several CPUs during the different phases of its execution. The data and code locality benefits are even higher than in the

single-CPU server, since fewer stages are exclusively using a single processor's cache. 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, stage 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.

5.4 Multiple query optimization

Multiple query optimization [Sel88][RS+00] has been extensively studied over the past fifteen years. The objective is to exploit subexpression commonality across a set of concurrently executing queries and reduce execution time by reusing already fetched or computed input tuples. Our design complements past approaches by providing a staged infrastructure that naturally groups accesses to common data sources at every phase of a query's lifetime (and not only to input tuples or intermediate results). Since all queries are forced to repeatedly execute at well defined stages (for example perform a join operation or scan a specific table), new scheduling algorithms can take advantage of this infrastructure. Information collected at the optimizer combined with run-time queue information of each execution engine stage, can be used to further increase the data overlap among queries at any phase of their execution. A query that arrives at a stage and finds an ongoing computation of a common subexpression, can reuse those results. This way, the burden of multiple query optimization can move from the optimizer to the run-time execution engine stages. Furthermore, the optimizer can spend less time waiting for a sufficient number of incoming queries with common subexpressions, since newly arrived queries can still exploit common data from other queries already inside the execution engine.

6 Conclusions

In this paper we discussed several issues of modern database architectures and introduced a new staged DBMS design with many desirable properties. Modern database servers suffer from high processor-memory data and instruction transfer delays. Despite the ongoing effort to create locality-aware algorithms, the interference caused by context-switching during the execution of multiple concurrent queries results in high penalties due to additional conflict and compulsory cache misses. 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, we suggested a departure in the way database software is built. Our proposal for a staged, data-centric DBMS design remedies the weaknesses of modern commercial database systems by providing solutions (a) at the hardware level: it optimally exploits the underlying memory hierarchy and takes direct advantage of SMP systems, and (b) at a software engineering level: it aims at a highly flexible, extensible, easy to program, monitor, tune, maintain, and evolve platform.

The paper's contributions are threefold: (i) it provides an analysis of design shortcomings in modern DBMS software, (ii) it describes a novel database system design along with our initial implementation efforts, and (iii) it presents new research opportunities.

Acknowledgements

We thank David DeWitt, Goetz Graefe, Jim Gray, Paul Larson, and the anonymous CIDR'03 reviewers for their valuable comments. We also thank Philippe Bonnet and Josef Burger (Bolo) for their technical support, and Stavros Christodoulakis, Manolis Koubarakis, and Euripides Petrakis for providing equipment while the authors were at the Technical University of Crete. This work is supported in part by the National Science Foundation under grants IIS-0133686 and CCR-0205544, and by IBM and the Lilian Voudouri Foundation through graduate student fellowships. We are grateful for their support.

References

[AD+01] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. "Weaving Relations for Cache Performance." In *Proc. VLDB*, 2001.

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

[AB+91] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "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, M. Datar, R. Motwani, and J. Widom. "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 Record*, Vol. 27, No. 4, Dec. 1998.

[CH90] M. Carey and L. Haas. “Extensible Database Management Systems.” In *SIGMOD Record*, Vol. 19, No. 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. B. Gibbons, and T. C. Mowry. “Improving Index Performance through Prefetching.” In *Proc. 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.

[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.

[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.

[HA02] S. Harizopoulos and A. Ailamaki. “Affinity scheduling in staged server architectures.” *TR CMU-CS-02-113*, Carnegie Mellon University, Mar. 2002.

[HP96] J. L. Hennessy and D. A. Patterson. “Computer Architecture: A Quantitative Approach.” 2nd ed., 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, Y. Q. He, R. C. Raphael, and W. E. Baker. “Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads.” In *Proc. ISCA-25*, pp. 15-26, 1998.

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

[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.

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

[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 *Proc. USENIX*, 1999.

[RB+95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. “The Impact of Architectural Trends on Operating System Performance.” In *Proc. SOSP-15*, pp.285-298, 1995.

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

[SB+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. “Eraser: A Dynamic Race Detector for Multi-Threaded Programs.” In *Proc. SOSP-16*, pp. 27-37, 1997.

[Sel88] T. K. 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 *Proc. 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.

[SL93] M. S. Squillante and E. D. Lazowska. “Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling.” In *IEEE TPDS*, Vol. 4, No. 2, Feb. 1993.

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

[SE94] S. Subramaniam and D. L. Eager. “Affinity Scheduling of Unbalanced Workloads.” In *Proc. Supercomputing*, 1994.

[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.