To Share or Not to Share?

Ryan Johnson[*] Nikos Hardavellas[*] Ippokratis Pandis[*] Naju G. Mancheril[*]
Stavros Harizopoulos[†] Kivanc Sabirli[*] Anastasia Ailamaki[*‡] Babak Falsafi[*‡]

[*]Carnegie Mellon University [†]Massachusetts Institute of Technology
[‡]Ecole Polytechnique Fédérale de Lausanne

{ryanjohn, ipandis}@ece.cmu.edu {ngm, ksabirli}@andrew.cmu.edu
stavros@csail.mit.edu {hardavellas, natassa, babak}@cmu.edu

ABSTRACT

Intuitively, aggressive work sharing among concurrent queries in a database system should always improve performance by eliminating redundant computation or data accesses. We show that, contrary to common intuition, this is not always the case in practice, especially in the highly parallel world of chip multiprocessors. As the number of cores in the system increases, a trade-off appears between exploiting work sharing opportunities and the available parallelism. To resolve the trade-off, we develop an analytical approach that predicts the effect of work sharing in multi-core systems. Database systems can use the model to determine, statically or at runtime, whether work sharing is beneficial and apply it only when appropriate.

The contributions of this paper are as follows. First, we introduce and analyze the effects of the trade-off between work sharing and parallelism on database systems running complex decision-support queries. Second, we propose an intuitive and simple model that can evaluate the trade-off using real-world measurement approximations of the query execution processes. Furthermore, we integrate the model into a prototype database execution engine, and demonstrate that selective work sharing according to the model outperforms never-share static schemes by 20% on average and always-share ones by 2.5x.

1. INTRODUCTION

Rather than increasing uniprocessor performance by incorporating aggressive optimizations into single-core micro-architectures, modern hardware improves performance through massive parallelism. Nearly all major hardware vendors already ship chip multiprocessors (CMPs) containing two to eight cores with support for up to eight hardware thread contexts per core. In the next decade we anticipate a computing landscape dominated by multi-core chips featuring hundreds of execution contexts. While many database management systems (DBMS) were designed for one- or few-processor machines, today's capacity for highly parallel execution — even on a single machine — more closely resembles a parallel database environment.

With the advent of highly parallel processors we expect that a single machine could host a significant subset of an enterprise's data warehousing operations. In such a scenario, significant opportunities for work sharing arise. A wealth of techniques, including materialized views [18], cooperative scans [28,14], multi-query optimization [20], and simultaneous pipelining [11], have been proposed to exploit work sharing to the maximum; all are based on the intuitive assumption that reducing the total amount of work to be done in the system should improve performance. Surprisingly, however, this is not always the case.

This paper demonstrates that work sharing does *not* always improve performance, especially for the highly parallel CMP architectures looming on the horizon. This counter-intuitive result arises from an inherent trade-off between work sharing and parallelism. In addition to eliminating redundant work that would otherwise have executed concurrently, work sharing tends to partly serialize execution of shared queries at the point of sharing. If serialization occurs on the critical path of the queries being processed, the reduction in parallelism can hurt performance rather than improving it. In consequence, aggressive work sharing may not improve database system performance as expected, even for systems with relatively few cores. As we demonstrate in this paper, highly parallel environments further accentuate this side-effect. Finally, work sharing that eliminates redundant I/O is especially effective for I/O-bound workloads [11]; the trend of ever-growing memory capacities, however, further increases the importance of understanding the work sharing/parallelism trade-off as more and more workloads fit in main memory.

In order to achieve maximum performance in highly parallel systems the database engine must intelligently exploit work sharing, detecting and avoiding situations where work sharing will be unhelpful.

1.1 To Share, or not to Share?

Intuitively, work sharing should provide performance benefits proportional to the amount of work removed from the system. Unfortunately this is not always the case, especially for in-memory workloads. Prior efforts to incorporate explicit work sharing into

VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

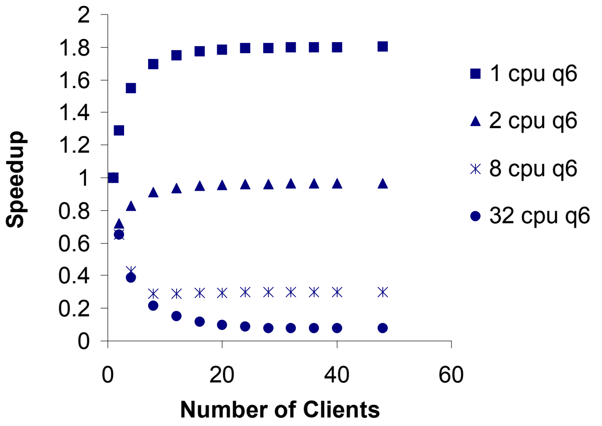

Figure 1. Speedup when sharing part of a data warehousing query (TPC-H query 6) relative to never-share execution.

database systems have encountered unexpected behaviors that discouraged their use [14,28].

To illustrate the fact that work sharing may not always be beneficial, we ran the following experiment. A different number of concurrent clients (from one to 48) submit a simple data warehousing query that is dominated by a scan on a large, in-memory table (query 6 from the TPC-H benchmark, see Section 3 for experimental setup details). Different clients use different predicates, however, all clients share the common task of scanning the same large table before applying their private predicates. In a uniprocessor machine, performing the scan only once for all clients and then applying the predicates for each client separately yields a throughput increase over a non-sharing execution sequence. The increase in throughput is due to the system's ability to eliminate redundant work (in this case, the individual scanning of the large table). The topmost line of Figure 1 illustrates the impact of work sharing in a single-core machine as the number of clients increases from 1 to 48. If, however, we increase the number of available cores, then the system is not penalized as much as before for performing redundant work, since that work can be performed in parallel. In fact, as Figure 1 shows, for more than one core, work sharing is harmful for this specific workload.

1.2 An analytical approach to work sharing

In this paper we propose an analytical model that captures the trade-off between parallelism and work sharing in order to predict the circumstances under which work sharing is helpful. Based on these predictions a DBMS can judiciously apply work sharing, at runtime, to achieve maximum performance.

The model is simple and intuitive. It depends only on the communication patterns and amount of work performed by each node of the query plan. The model uses these parameters to determine the available parallelism and effects of work sharing for the query. Given processor constraints and a number of concurrent clients, the model predicts the speedup of sharing portions of the participating queries vs. executing the queries in parallel.

We assume a closed system where every query that completes is replaced by a new one, as is typical for a system under heavy load. A closed system is a suitable model for data warehousing operations, where data analysts submit long-running queries one after the other, as soon as they get the results of the previous query. In a closed system, arrivals—and therefore peak throughput—are governed by how long it takes to process each query. If X is the average system throughput, N is the number of queries allowed in the system at a time, and R is the average rate at which the system processes a single query, Little's Law concisely captures this relationship as

$$X = NR$$

Little's Law has a startling implication for work sharing in closed systems: *throttling queries lowers throughput even if the amount of work in the system is reduced at the same time*. Intuitively, then, an analytical model of the work sharing parallelism trade-off must be able to determine whether applying work sharing will lower the average query processing rate of the system. On the one hand, work sharing potentially imposes throttling, but on the other hand it also may remove enough redundant work from the system to free needed processing resources.

A model that captures the essence of Little's Law in the context of work sharing is particularly well-suited to pipelined query processing because the "bottleneck" pipeline stage determines the rate of the entire query's forward progress. For example, [10] uses this reasoning to apply rate-based models to predict the performance in data warehousing workloads. In our case, work sharing may create or accentuate a bottleneck in the queries under consideration and hurt performance instead of increasing it.

Our model provides insight into the source of the unexpected performance loss shown in Figure 1. In this extreme example, work sharing creates a large bottleneck that artificially caps the degree of parallelism and leaves multi-core systems with idle contexts. Under work sharing, the system in Figure 1 utilized only three of 32 available hardware contexts, while independent execution utilized all of them. The resulting 10x performance difference demonstrates that the DBMS must be willing to sacrifice at least a part of the available work sharing opportunity to provide sufficient parallelism.

Incorporating the model directly into the DBMS allows it to make intelligent decisions about when to share work. Section 7 shows how a model-guided work sharing policy outperforms static "always share" and "never share" policies.

1.3 Contributions and Paper Organization

In order to build database systems that benefit from work sharing in the upcoming hardware landscape, designers must understand the underlying relationship between parallelism and work sharing. To our knowledge this paper is the first that discusses the trade-off between parallelism and work sharing in database systems. It provides a simple analytical model that captures the trade-off, allowing database systems to apply work sharing judiciously, at runtime. We show that a system must first strive to utilize all available hardware resources, sacrificing work sharing opportunities if necessary to provide parallelism. Furthermore, we demonstrate a prototype database execution engine that incorporates the model, and show that selective work sharing according to the model outperforms never-share static schemes by 20% on average and always-share ones by 2.5x.

The rest of the paper is organized as follows: Section 2 discusses background and related work. Section 3 experimentally shows that sharing in highly parallel environments is not always beneficial, while Section 4 presents the analytical model that predicts the performance of the database system using pipelined parallelism and work sharing. Section 5 extends the model to include more classes of queries, including those containing stop-and-go operators. Section 6 elaborates on the parameters that significantly affect performance, such as the available processing power and the overhead of work sharing. Section 7 experimentally validates the model, while Section 8 experimentally evaluates the model in a real-world setting and discusses cases where it can be beneficial, followed by our conclusions in Section 9.

2. BACKGROUND AND RELATED WORK

In this paper we consider read-mostly database workloads on a DBMS that employs pipeline parallelism and has some means of exploiting work sharing (such as those discussed over the next paragraphs). The DBMS runs on a highly parallel chip multiprocessor system with abundant main memory.

The remainder of this section discusses each feature in more detail and its implications for the work sharing/parallelism trade-off. In particular, Section 2.1 presents various proposals for work sharing, while Section 2.2 discusses pipelined parallelism, and Section 2.3 elaborates on the emerging hardware landscape.

2.1 Work Sharing

We define work sharing as any operation that reduces the total amount of work in a system by eliminating redundant computation or data accesses.

By far the most common form of work sharing occurs through the buffer pool, which stores both scans and intermediate results. However, buffer pools exploit sharing haphazardly: they do not provide an obvious way to conscientiously exploit sharing opportunities or to arrange for them to occur. The following paragraphs highlight techniques designed to expose and exploit work sharing opportunities.

Materialized Views: Materialized view selection [18] is typically applied to workloads known in advance, in order to speed up queries that contain common sub-expressions. Materialized views exploit commonality between different queries at the expense of potentially significant view maintenance costs. Tools for automatic selection of materialized views take such costs into account when recommending a set of views to create [1]. The usefulness of materialized views is limited when the workload is not always known ahead of time or the workload requirements change frequently.

Multi-Query Optimization: Multiple-query optimization (MQO) [20,19] identifies common sub-expressions in query execution plans during optimization, and produces globally-optimal plans. The detection of common sub-expressions is done at optimization time, thus, all queries need to be optimized as a batch. In addition, to share intermediate results among queries, MQO typically relies on costly materializations. To avoid unnecessary materializations, a recent study [6] introduces a model that decides at the optimization phase which results can be pipelined and which need to be materialized to ensure continuous progress in the system.

Cooperative/Synchronized Scans: Since queries interact with the buffer pool manager through a page-level interface, it is difficult to develop generic policies to coordinate current and future accesses from different queries to the same disk pages. The need to efficiently coordinate and share multiple disk scans on the same table has long been recognized and several commercial DBMSs and research prototypes incorporate various forms of multi-scan optimizations [28,5,7,14]. The challenge is to bypass the restrictions implied by the page-level interface in order to fully exploit the knowledge of query access patterns, even if it requires run-time adjustments to the query evaluation strategy.

Simultaneous Pipelining. Staged database systems [9] implement one or few similar relational operators as independent modules ("stages") that operate in parallel while maintaining private data and control mechanisms. Incoming requests are decomposed into "packets" and routed to the appropriate stages. The packets indicate the work requested on behalf of the incoming query, and form series of producer-consumer pairs. All requests for a particular operation cluster on the same queue. This enables the system to detect work sharing opportunities easily at run-time. Once a sharing opportunity is detected the system executes the corresponding operation only once and simultaneously pipelines the results of the common operation to the interested parties [11].

Sharing in Streaming Engines: In the context of stream management systems there have been proposed techniques to share work across different queries [15,4]. Although the concept of sharing is similar, queries in stream systems always process the most recently received tuples. In traditional query processing there are specific requirements as to which tuples are needed and in what order they need to be processed.

Explicit work sharing generally imposes some amount of overhead, and additionally reduces parallelism by partly serializing operations that otherwise might have occurred in parallel.

2.2 Parallelism

Database workloads exhibit three main types of parallelism:

- **Multiple Requests**. Independent requests can execute simultaneously.
- **Pipelining**. Producer-consumer pairs can execute at least partly in parallel, forming an assembly line of sorts.
- **Partitioning**. Many queries can be divided into independent sub-queries whose results are then combined to answer the original query.

We focus on pipeline parallelism in a system with multiple outstanding requests; partitioning is an orthogonal concern we leave to future work.

Pipeline Parallelism. Queries can be divided into producer-consumer pairs, which can execute in parallel and reduce overall execution time. This type of parallelism is called pipeline parallelism [11,12,22].

The majority of performance analysis on pipelined execution has been conducted in the context of parallel database systems [12,22]. In such environments the major concern is the communication/parallelism trade-off within a single query. In this work we focus on the work sharing/parallelism trade-off when executing multiple queries concurrently. One important characteristic of pipelined execution is that the slowest operation in the pipeline bounds the overall performance. Sharing work between queries can therefore have unexpected effects on performance if it shifts bottlenecks.

Closer to the context of our work, [26] presents a model for the performance of a database system that applies pipelined execution to single multi-operation queries. We use a similar model to study how performance differs when multiple concurrent queries execute in parallel with and without work sharing.

Parallelism becomes increasingly important for database systems in the emerging computing landscape, which consists of highly parallel chip multi-processors and large main memories.

2.3 Computing Landscape

Chip Multi-processors. Computer systems are currently undergoing a major shift toward improving performance through parallel execution rather than relying on advances in clock speed and aggressive micro-architectures. Clock speed increases have recently slowed and even reversed due to overriding power concerns in some cases[1]. Aggressive micro-architectural techniques suffer from scalability and complexity problems while yielding diminishing returns, especially for database workloads [17]. At the same time, transistor budgets continue to increase relentlessly per Moore's Law.

Stamping out multiple cores per die and increasing on-chip cache capacities provide two power- and complexity-friendly uses for those extra transistors. Nearly all major hardware vendors already sell CMP configurations featuring two to eight cores with support for up to four hardware thread contexts per core. L2 caches approaching 20MB or more are also available, but oversized caches actually reduce database performance [8]. Therefore, in the near future we anticipate a computing landscape dominated by multi-core chips featuring modest caches and dozens or even hundreds of execution contexts. The DBMS must provide scalable thread-level parallelism to achieve peak performance in this new landscape.

Main Memory Capacity. Available system memory continues to increase exponentially. Five years ago 20-40GB hard drives were common; reasonable server configurations today sport 32-64GB of main memory. Versioned buffer pools [3] reflect this trend: they reduce lock contention by accumulating multiple versions of database pages in memory as they are modified, trading extra memory for improved performance. Large memories mean the working set of many databases fits entirely in main memory, resulting in compute- or memory-bound query processing. Even when workloads do not fit entirely in main memory, a large fraction does fit.

Work sharing consistently and significantly improves performance when it eliminates redundant I/O; the benefit of work sharing in memory resident workloads is much less predictable and depends on factors such as the nature of the queries involved, available processing power, and even scheduling decisions.

3. SHARING IS NOT ALWAYS GOOD

In this paper, we present a counter-intuitive result regarding work sharing: sharing may be detrimental to performance, even if the system has multiple instances of sharable queries awaiting execution. The explanation is rooted on the delay imposed on the sharers at the *pivot* operator (the root of the query sub-plan being shared). The pivot operator incurs additional work for each sharer, because it has to send results to each one of them. This additional work imposes delays that affect all sharers and may offset the performance advantage of work sharing. Once the delays become too large reducing the degree of sharing and performing some redundant work in parallel can result in higher performance because it shortens the critical path of the query while using otherwise idle processors.

We show experimentally the existence of the sharing trade-off and investigate the sharing trends as a function of the number of sharers and the number of processors. We perform experiments on a selection of representative scan-heavy and join-heavy TPC-H queries [23] running on a Sun UltraSparc T1 server.

3.1 Experimental Setup

The UltraSparc T1 server features 8 cores on a single chip, each with a private first level cache, while communication between the cores is facilitated through a shared L2 cache. Each core supports 4 hardware contexts, or as many 32 active threads on a single chip. These threads appear as individual processors to the operating system. Each core executes instructions from available threads in a round-robin fashion, guaranteeing fairness of execution.

We execute a selection of TPC-H queries on a 1GB database (memory-resident) and measure the speedup of executing the query with work sharing over executing the query without work sharing. To validate our model, we compare the measured speedup against the model's prediction. Our selection of TPC-H queries includes scan-heavy queries (q1, q6) and join-heavy queries (q4, q13). The selection of the queries is based on the characterization in [21]. To avoid random perturbations that may skew our results, we fix the query predicates to constant values.

Under normal circumstances the system exploits work sharing opportunities wherever they might occur in the plan, coalescing even entire queries when possible. To keep the database engine from eliminating the identical queries used in these experiments, we therefore allow sharing only at one selected node of each query plan. Queries q1 and q6 potentially share the scan operator, while q4 and q13 share at the join operator. To measure the effect of sharing under varying conditions, in our experiments we vary the number of clients, where all clients submit identical queries.

The model takes as input the amount of work performed by each node in the query plan. We build a model for each query type by profiling the system during a few test query invocations, both with and without work sharing. We then solve a system of linear

[1] Consider Intel's 3.8GHz Pentium 4 "Prescott" (2004) and 2.66 GHz Core 2 (2007) processors, for example.

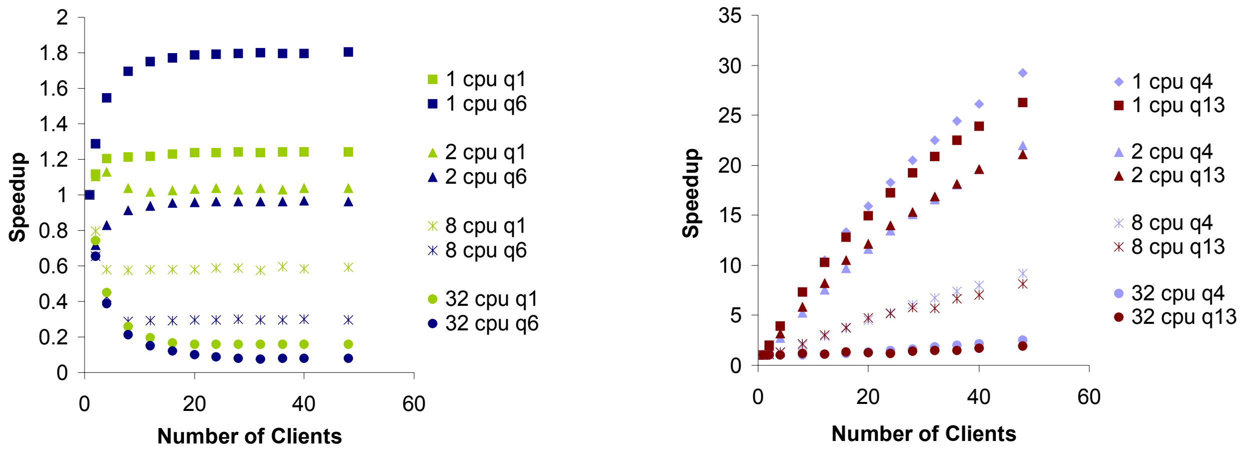

Figure 2. Speedup of work sharing for scan-heavy (left) and join-heavy (right) queries.

equations to divide up the active time for of each operator among the different nodes of the query plan. Once the amount of work performed by each operator is known, the values can be plugged into the model to predict the effect of work sharing.

Ideally, the model's parameters would be estimated online, allowing the system to respond effectively to changing workloads. In this paper we employ off-line parameter estimation and concentrate on evaluating the model's effectiveness. However, because parameter estimation is straightforward we anticipate no significant barriers to online estimation.

3.2 Database System Framework

We run our selection of TPC-H queries on a prototype database system called Cordoba. Our results are applicable to any database system capable of sharing work between pipelined query plans, so we choose to utilize Cordoba because of its flexibility in sharing at arbitrary operators of a query plan, as well as source code availability.

Cordoba is a staged database system [9] that focuses on in-memory databases running on highly parallel processors (e.g., CMPs). A staged server processes work in "stages" rather than as monolithic requests. Each stage implements one relational operator and maintains private data and control mechanisms. Incoming requests are decomposed into "packets" and routed to the appropriate stages. Each packet indicates the work requested by this operator on behalf of the incoming query, and can be scheduled individually, as its execution is decoupled from the others. Thus, staging naturally enhances workload parallelism and can utilize otherwise idling computational resources.

At the same time, Cordoba's highly modular design easily incorporates dynamic work sharing policies. When a new packet arrives at a stage's queue, the stage thread searches the queue for other packets that request the same operation. If it finds work sharing opportunities, it "merges" the packets. That is, it executes the operation only once and outputs the intermediate results to all interested parties.

The main difference between Cordoba and its predecessor, QPipe [11], with respect to our discussion in this paper is that the execution model does not follow the traditional demand-driven, pipelined model. Instead, the intermediate results between operators are packed into pages (of typical size of 4K). Such an execution model has been shown to improve both instruction and data locality, as well as reducing the synchronization cost between the producer-consumer pairs [16,2,27].

3.3 Results

We experimentally measure the speedup of work sharing for our selection of TPC-H queries running on 1, 2, 8, and 32 processors. Figure 2 (left) shows the speedup of work sharing in scan-heavy TPC-H queries q1 and q6 as the number of clients increases. Work sharing attains speedups up to 1.8x when the queries execute on a uniprocessor. However, as the number of participating processors increases, sharing quickly hampers performance.

These queries incur a significant amount of per-sharer work at the pivot operator which slows down each sharer. Thus, as the number of potential sharers increases, this slowdown quickly overwhelms the performance benefit of sharing work and causes speedup to level off. If only few processors participate (e.g., 1 processor) the slowdown imposed on each sharer has no detrimental effect as compared to unshared execution, because the sharer will have to wait for a processor quantum to execute anyway. Thus, any amount of saved work improves performance, as shown by the 1 processor speedups. However, if many processors are available for execution (e.g., 32 processors) then work sharing slows down each sharer enough to result in overall performance degradation, as shown by the 32 processor speedups.

Figure 2 (right) shows the experimentally measured speedups of work sharing in join-heavy TPC-H queries q4 and q13 for 1, 2, 8, and 32 processors. Our results indicate that work sharing is always beneficial for the join-heavy queries in our benchmark suite. These queries perform most of the work at the scan and join operators. Under work sharing, both operators are shared, and the per-sharer work at the pivot operator (join) is insignificant compared to the work performed by the scan and the rest of the join. Thus, work sharing always attains high speedup compared to unshared execution. Also, the fewer the processors participating, the larger the effect of saving work. Thus, speedups are higher for small number of processors (e.g., 1 processor).

4. ANALYTICAL MODEL

In order to predict the benefit of work sharing in a given system we desire a model with the following properties. It should be as simple as possible, while handling a large enough class of queries to be generally useful. It can be a binary variable; it need not give perfectly accurate speedup predictions. Ideally, the parameters should also be easy to obtain automatically, perhaps even at runtime.

Our goal is to predict the performance of the queries when they execute either independently or with work sharing, given some number of processors with which to execute them. The system can then use this comparison to apply work sharing intelligently.

Because we consider pipelined execution of queries, performance is proportional to the rate (x) at which tuples flow through the queries. Given m potentially shared queries and n processors to execute them on, the benefit from work sharing is simply the ratio of the rates with and without work sharing:

$$Z(m, n) = \frac{x_{\text{shared}}(m, n)}{x_{\text{unshared}}(m, n)}$$

If $Z(m, n) > 1$ work sharing is net win; otherwise unshared execution is better.

The extensive literature on parallel databases provides a good starting point. The model we develop here is similar to the one described in [26]. In particular, we assume that the workload is memory-resident, that all operators are fully pipelinable, and that cardinalities are large enough that the flow of tuples between operators can be treated as continuous rather than discrete. We also assume that buffering between operators is sufficient to smooth out any burstiness in the tuple streams. Significant differences with the model in [26] include

- Processing occurs in a single shared-everything CMP with many cores rather than a networked set of shared-nothing processing nodes.
- The system must process multiple queries in parallel.
- We are interested only in relative performance, so we ignore tuple counts and time units whenever possible and focus on average rates instead.
- All operator rates are relative to the forward progress of one reference tuple stream for the query (full explanation follows).
- Finite buffering means that slow consumers throttle producers.
- We assume that operators pass results on to their consumers as soon as possible and at a constant rate. We therefore neglect pipeline filling times, which will be short compared to total execution time.

Table 1 summarizes the terms used in the following sections.

4.1 Components of the Model

We begin with a simplistic model which assumes that all queries consist of fully pipelinable operators and have the same peak rate of forward progress, and that none of the queries has begun execution yet[1]. We defer for Section 5 the discussion on how to deal with mismatched rates and stop-&-go operators such as sorts.

Table 1. Definitions of terms used in the model

Term	Definition
p	Total work per unit of forward progress in an operator
w	Work an operator performs per unit of forward progress
s	Work required to output a unit of forward progress to each consumer of an operator
i	An input stream for the current operator
j	An output stream for the current operator
$k \in K$	One of the operators from query plan K
r	The peak rate of forward progress for a query
u	The maximum processor utilization per query
$x(m,n)$	The rate of forward progress given m queries and n available processors
ϕ	The "pivot" operator, the highest point in each query plan where work sharing is possible
$Z(m,n)$	The benefit of work sharing given m queries and n available processors

4.1.1 Operators

Each operator in a query plan features both input streams (usually one or two) and output streams (usually one). We treat all streams as carrying units of forward progress, rather than tuples, so that all operators in a plan are comparable. Suppose we choose some node in the query plan as a reference. The query will not complete until the reference node has finished processing every tuple it is destined to receive. Each operator can therefore define "forward progress" as how much each tuple it handles will contribute toward the reference operator's total progress. This notion of progress implicitly captures the selectivity of operations in the query plan.

Each unit of overall forward progress delivered by input stream i requires some amount of work from the operator, denoted by w_i. In addition the operator must output units of forward progress to its consumers; consumer j requires s_j work per unit of forward progress. The total work per unit of forward progress, p, is therefore

$$p = \sum_{i \in inputs} w_i + \sum_{j \in outputs} s_j$$

4.1.2 Queries

A query plan consists of a set of operators, K, linked to form a pipeline. Due to the tight coupling of operators in the pipeline all operators proceed at the same rate of forward progress, as determined by the slowest (bottleneck) operator in the query. The peak rate of forward progress for the query is therefore given by

$$r = \frac{1}{\max\{ p_k : k \in K\}} = \frac{1}{p_{max}}$$

[1] [11] discusses situations when this need not be the case.

All operators other than the bottleneck spend at least some of their time waiting for input or output to become ready. We can therefore calculate the maximum processor utilization of a query as

$$u = \sum_{k \in K} \frac{p_k}{p_{max}} = \frac{u'}{p_{max}}$$

Note that u reflects the level of pipeline parallelism available in the query and can easily be greater than 1. In a moment, u', the total work per unit of forward progress will also become useful.

4.1.3 Limited Hardware Resources

A query's peak throughput, r, assumes that the system makes at least u processors available to the query. However, a loaded system may not have enough spare processing power to achieve the peak rate.

Let n be the number of processors the system makes available to a query or group of queries. If $u > n$ the system will be forced to time-share operators, uniformly reducing the rate of the query by a factor of n / u. We can therefore define the true rate of forward progress, given n available processors, as

$$x(n) = r \cdot \min\left(1, \frac{n}{u}\right) = \min\left(\frac{1}{p_{max}}, \frac{n}{u'}\right)$$

where the term u' appears in the last, simplified version.

4.1.4 Contention for Shared Hardware Resources

CMP architectures contain many shared resources such as caches, memory bandwidth, and functional units. As more processors share a given resource, growing contention will reduce the effective processing power through increased stalls. However, the model we have described so far optimistically assumes that queries—and threads within queries—interact only through the buffers that connect pipeline stages. In order to account for contention in shared resources, we assume that there are effectively only n^k processors available, with $0 < k < 1$. The value of k depends on both the hardware and workload, and whether the system applies work sharing. We choose this simple function because it works well in practice and k is easy to measure empirically; a more accurate contention model could be substituted instead if need be.

4.2 Unshared Execution

If the system executes a set (M) of queries independently we simply increase the rate by a factor of $|M|$ and sum up the peak utilizations of the queries:

$$r_{\text{unshared}}(M) = |M| \cdot r$$

$$u'_{\text{unshared}}(M) = \sum_{m \in M} u'_m$$

The value of $r_{unshared}$ reflects the fact that all queries have the same rate and finish at the same time. Note, however, that $r_1 = r_2$ does *not* imply that two queries perform the same amount of work (i.e., $u'_1 = u'_2$) unless the queries also have the same plan. The value of $u'_{unshared}$ reflects this fact.

4.3 Shared Execution

Let ϕ be the "pivot" operator, that is, the point in each query plan where work sharing will occur. Three major changes occur if the system chooses to apply work sharing to the query group:

1. All replicated work below the pivot is eliminated.
2. The pivot must multiplex output to multiple consumers.
3. The slowest operator in M throttles all queries.

Let the phrase "k is below ϕ" indicate that operator k is a member of the query plan sub-tree rooted at ϕ and therefore provides input to the pivot either directly or indirectly. Similarly, "k is above ϕ" indicates that k is *not* part of the sub-tree rooted at ϕ. In a shared plan there is only one instance of each k below ϕ; all shared queries use the output of ϕ. To reflect this we drop the query identifier from subscripts for k below ϕ.

Because ϕ must send output to all consumers in M, p_ϕ becomes

$$p_\phi(M) = w_\phi + \sum_{m \in M} s_{m\phi}$$

and may become the new bottleneck (p_{max}) if the sum of the s_m values grows too large.

If operator ϕ did not become a new/worse bottleneck then $r_{shared} = r_{unshared}$; otherwise $r_{shared} = |M|/p_\phi < r_{unshared}$. In contrast, $u'_{shared} \neq u'_{unshared}$; we compute it as follows:

$$u'_{\text{shared}}(M) = \sum_{k \text{ below } \phi} p_k + p_\phi(M) + \sum_{\substack{m \in M \\ k_m \in K_m \text{ above } \phi}} p_{k_m}$$

4.4 Example Revisited

Using the profiling procedure given in Section 3 we extract the parameters for TPC-H Q6 running on our test machine. The query consists of two pipeline stages—table scan and aggregation—with work sharing allowed at the table scan stage in our experiments. The query is captured by three parameters: $w = 9.66$ and $s = 10.34$ describe the scan, and $p = 0.97$ captures the aggregate. As we will see, these parameters accurately capture the query's performance even without accounting for hardware contention, so we fix k = 1.

Applying the parameters to the model for unshared execution first, we get

$$p_{max} = p_\phi = 20$$

$$u'_{\text{unshared}}(M) = \sum_{m \in M} u_m = 21 \cdot |M|$$

$$x_{unshared}(M, n) = |M| \cdot min\left(\frac{1}{20}, \frac{n}{21 \cdot |M|}\right)$$

$$= \min\left(\frac{|M|}{20}, \frac{n}{21}\right)$$

Similarly, applying the model for shared execution yields

$$\begin{aligned} p_{max}(M) &= 9.66 + 10.34 \cdot |M| \\ u'_{shared}(M) &= 9.66 + 11.31 \cdot |M| \\ x_{\text{shared}}(M, n) &= |M| \cdot min\left(\frac{1}{p_{max}(M)}, \frac{n}{u'_{shared}(M)}\right) \\ &= min\left(\frac{1}{\frac{9.66}{|M|} + 10.34}, \frac{n}{\frac{9.66}{|M|} + 11.31}\right) \end{aligned}$$

A close examination of the equations reveals several interesting points. First, s_{ϕ} is quite large, and ϕ is a worsening bottleneck as more queries share its output. This, in turn reduces peak utilization to the point that shared execution only utilizes slightly more than one processor no matter how many sharers are added to the mix. Unshared execution, on the other hand, does not suffer from rate throttling or bounded utilization. Its performance scales linearly until all available processors are utilized. In this particular case we see that work sharing is only attractive when one processor is available.

Section 6 explores how parameters such as N and s_{ϕ} impact the usefulness of work sharing, while Section 7 evaluates the accuracy of the model, including the parameters for Q6 discussed here.

5. EXPANDED COST MODEL

The model presented in the previous section makes several simplifying assumptions that limit its applicability. This section expands the model by removing some of those assumptions

5.1 Mismatched Rates

Up until now we have assumed that all queries in a group proceed at the same rate. This will not always be the case in practice. The model for shared query execution already handles differing peak rates because all queries will get throttled to match the slowest consumer. However, it is more difficult to model unshared execution because faster queries will complete and exit the system before slower ones. As a result the behavior of the system is no longer uniform over time, breaking a key assumption of the model. In order to address this difficulty we must first distinguish between *open* and *closed* systems in the queueing theory sense.

Query arrivals in an open system are independent of each other; as long as the system can process queries faster than they arrive, on average, changing the response time of a request has no effect on overall throughput. The arrival rate controls peak throughput. Because arrivals in an open system are independent of response time, throttling a group of queries to the same rate is equivalent to letting the faster queries complete early and leave their resources idle. Therefore, in an open system, we can simply model all unshared queries as if they were throttled to the rate of the slowest one. The equations all remain unchanged.

A closed system, on the other hand, has a fixed number of requests in the system. Completed requests are immediately replaced by new ones, so delays imposed by work sharing directly affect system throughput by preventing future requests from arriving. In addition, we can no longer assume that resources remain idle once a query completes.

Modeling unshared query execution in a closed system remains an open problem. For now we suggest a crude approximation that assumes a similar query will replace the current one as soon as it completes. We therefore leave the equations for shared execution unchanged, but modify $r_{unshared}$ to reflect the harmonic mean of peak throughputs, and $u_{unshared}$ so that each query is throttled only by its own p_{max} rather than that of the group:

$$r_{\text{unshared}} = \frac{|M|}{\sum_{m \in M} p_{\text{max}}(m)}$$

$$u_{\text{unshared}} = \sum_{m=1}^{M} \sum_{k=1}^{K(m)} \frac{p_k}{p_{\text{max}}(m)}$$

The model then allows faster queries to raise $r_{unshared}$, while assuming that they use their full allotment of resources until the last query completes. While certainly not optimal, this approximation is arguably better than simply assuming an open system, and in our experience it leads to better binary decisions.

Fortunately, mismatched rates are often not an issue. First, we are only concerned with modeling queries that could potentially be shared; queries with mismatched rates are not a problem if they cannot be shared anyway. Second, many queries are bottom-heavy, meaning that potentially shared queries (which have the same "bottom" to their query plans) will often have the same peak rates. Third, results bound for extremely slow consumers can be materialized, rather than pipelined, to prevent the latter from slowing down the entire pipeline.

If all else fails, the system can simply avoid applying work sharing among queries with mismatched rates. Because the queries have identical sub-plans, the mismatched rate indicates a top-heavy query that will benefit less from work sharing anyway (see Section 6.3). In addition our results in Section 8.1 show that not sharing at all is generally better than blindly sharing work.

5.2 Modeling Stop-&-Go Operators

The key observation when considering stop-&-go operators, such as sorting (used in aggregations, merge joins, etc.) or the hash building phase of the mainstream hash join, is that the production/consumption rates of the operators below the stop-&-go operator are decoupled from the those of the operators above it.

For the purposes of work sharing, each phase of the query plan can be modeled separately. For example, consider a query containing a sort stage. From a modeling perspective the query looks like three queries: the first sub-query has a moderately slow root node (sorting runs), and is followed by a sub-query that does not interact with the system (merging runs), followed by a final sub-query with an extremely fast scan at its leaf node (outputting the sorted result).

During the sort phase the inputs to the sort can be shared with other queries; once the sort phase completes the inputs have been completely consumed and work sharing is no longer possible. Similarly, during the output phase queries requesting similar sort operations can share the sort's output values, once they become available.

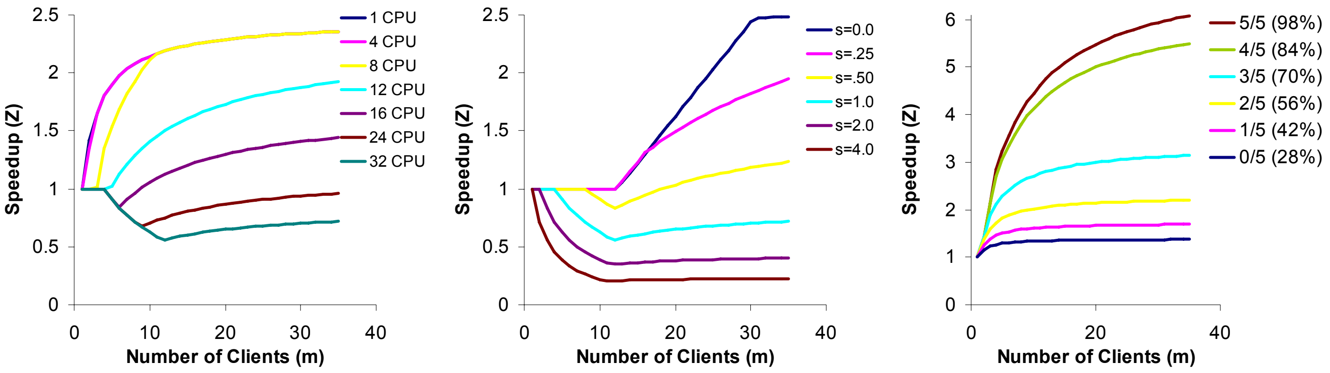

Figure 4. Predicted speedup of work sharing over independent execution as the number of processors varies (left), as the amount of serial work varies (center), and as the amount of work below the pivot varies (right).

5.3 Modeling Joins

The model presented in the previous sections naturally captures pipelined join operators, but not all join algorithms are fully pipelinable. The following subsections discuss the three basic join types, namely (block) nested-loop join (NLJ), merge join (MJ), and hash join (HJ).

5.3.1 Nested-Loop Join (NLJ)

The nested-loop join is a fully pipelinable join operator. It is the simplest case and does not require any change in the model. We simply model an operator with two input streams, with one stream potentially much more expensive than the other.

5.3.2 Merge Join (MJ)

We treat merge joins as three different operations. The first two being the two sort (stop-&-go) operations, one for each input, and one being their merging. In this case, the query is divided into three different sub-queries. If, however, any input is already sorted then the corresponding sort operation is unnecessary and the merge join can be pipelined.

5.3.3 Hash Join (HJ)

Similarly with merge joins the mainstream version of the hash join is represented as two separate operations, the hash building and the hash probing phases. The hash building is a stop-&-go operation, while the hash probing is fully pipelineable. Therefore, the query is divided into two sub-queries, one consisting of all the operations below (and including) the hash building operation, and the second with all the operations above it.

Starting with the symmetric hash join [25], several proposals for pipelinable hash joins can be found in the literature [13,24]. If such an implementation of the hash join is employed by the system, then this operation is pipelineable and the simple model again suffices.

6. SENSITIVITY ANALYSIS

As can be seen in the example from Section 3, aspects of work sharing that reduce available parallelism can lessen or even eliminate the advantages that come from reduced work in the system. This section explores how various parameters from the model impact parallelism and performance in a system. This section explores how the effect of work sharing changes with

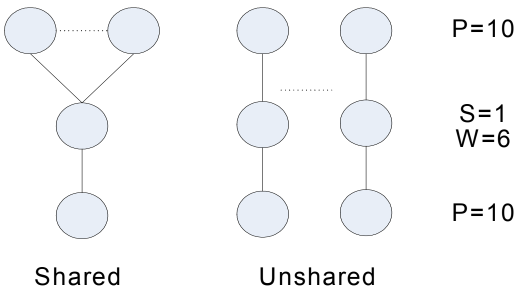

Figure 3. Executing M queries with and without work sharing

- Available processing power (n)
- Cost to output shared data to each consumer (s_i)
- Fraction of work eliminated by sharing,

In all the cases the lesson is that work sharing only helps if the unshared execution would saturate the available hardware.

Unless otherwise stated, for the analysis we use a set of M identical queries, each with three stages. Sharing can potentially occur at the middle stage. Figure 3 depicts the execution plans if work sharing is employed (left) or not (right). The bottom operator has $p = 10$, the second (pivot) operator has $w = 6$ and $s = 1$, and the top operator again $p = 10$. Work sharing therefore eliminates nearly 60% of the work for this query.

6.1 Available Processing Power (*n*)

The number of available processors has perhaps the most direct effect on the performance of work sharing. Figure 4 (left) plots the speedup over independent execution achieved by work sharing *m* queries at a time along the x-axis. The figure has plots for available processor counts ranging between 1 and 32. As the figure indicates, systems with very few processors available benefit the most from work sharing, while those with an abundance of processing power must seek parallelism as a first priority.

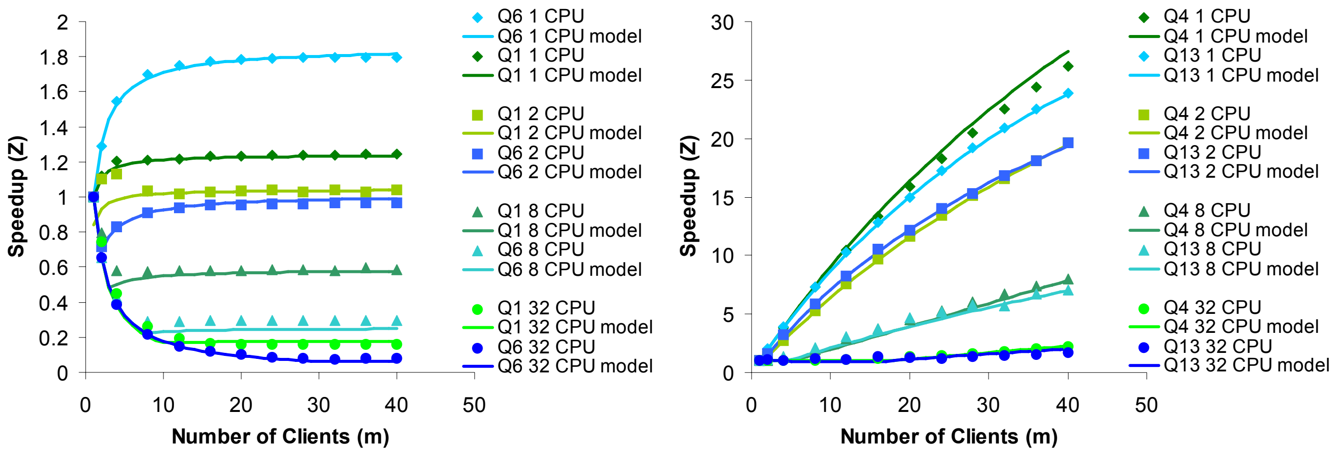

Figure 5. Model validation using scan-heavy queries (left) and join-heavy queries (right)

From the model we learn that, individually, each query requires 2.7 processors for peak throughput, while work sharing provides limited parallelism and utilizes only 10 cores even for large numbers of shared queries. For a given number of available processors there are (up to) three phases of behavior for work sharing. At first there is not enough work to saturate the machine even without work sharing; so the latter cannot improve performance. As the load increases the limited parallelism available through work sharing actually hurts performance. Finally, as load increases still further, the elimination of extra work due to work sharing achieves a net speedup for some values of n (e.g.,16 CPU). As we can see, the model can help predict whether work sharing is always (4 CPU), never (32 CPU), or sometimes (16 CPU) worthwhile, depending on the current query and situation.

6.2 Serialization due to Work Sharing

One of the primary drawbacks of work sharing is its serializing effect on the shared queries: the pivot (which executes as a single thread like all operators) must sequentially output results to all M consumers. The pivot can quickly become a bottleneck if s, the cost of outputting to each consumer, is significant compared to p_{max}. Figure 4 (center) plots the performance of a 32-core system as s varies. It demonstrates how a high s quickly saps all benefits from work sharing if abundant processing resources are available. For s = 0, work sharing imposes no serialization and saturates the machine by 30 shared queries. As serialization increases it becomes difficult (s = .25) and finally impossible (s = 1.0) to utilize the hardware fully, resulting in a net loss of performance. In order to achieve scalable performance through work sharing, systems should therefore strive to keep s as low as possible.

6.3 Fraction of Work Eliminated

Intuitively, the most attractive queries for sharing would be those that allow the most work to be eliminated. The model confirms that this is indeed the case. We modified the baseline query by splitting the top operator into five balanced pipeline stages, each with $p = 8$; each of the five represents 14% of the total work in the query. We then vary the fraction of work eliminated by moving operators below the pivot one by one. Figure 4 (right) shows how, for an 8-processor system, work sharing improves performance by a larger fraction each time another operator moves below the pivot, except the last operator, which gives an unexpected diminishing return. Examining u_{shared} (the peak processor utilization) for this case shows that work sharing only exploits two of the eight processors available, while a single unshared query utilizes more than four. Though work sharing is beneficial because it eliminates so much work, its tendency to reduce parallelism bounds the maximum achievable speedup to roughly one eighth of the 50x we might expect from eliminating 98% of the work in the system.

7. EXPERIMENTAL VALIDATION

We experimentally validate our analytic model on a selection of scan-heavy and join-heavy TPC-H queries running on an UltraSparc T1 server. The experimental setup is described in Section 3.1 and Section 3.2.

Figure 5 (left) shows the predicted and experimentally measured speedups of work sharing for scan-heavy TPC-H queries Q1 and Q6 for 1, 2, 8, and 32 processors. The experimental measurements are depicted using points, while the lines represent the corresponding model predictions. The maximum error of the model's predictions is 22%, while the average error is only 5.7%. Thus, the analytic model provides an accurate first-order approximation of the expected speedup, especially in light of the fact that the model, in the interest of simplicity, ignores important hardware effects like cache misses (Q1 2 CPU) and contention for shared resources like caches, memory, on-chip network (Q6 1 CPU).

Figure 5 (right) shows the predicted and experimentally measured speedups of work sharing for join-heavy TPC-H queries Q4 and Q13 for 1, 2, 8, and 32 processors. Our results indicate that work sharing is always beneficial for these join-heavy queries, a behavior accurately captured by the corresponding models. The model's maximum error for join-heavy queries is 30%, while the average error is 5.9%.

The scan-heavy and join-heavy queries show radically different behaviors, with the scan-heavy speedup curves flattening out quickly and the join-heavy queries providing ever-increasing speedups. Despite these differences, we have shown how a simple

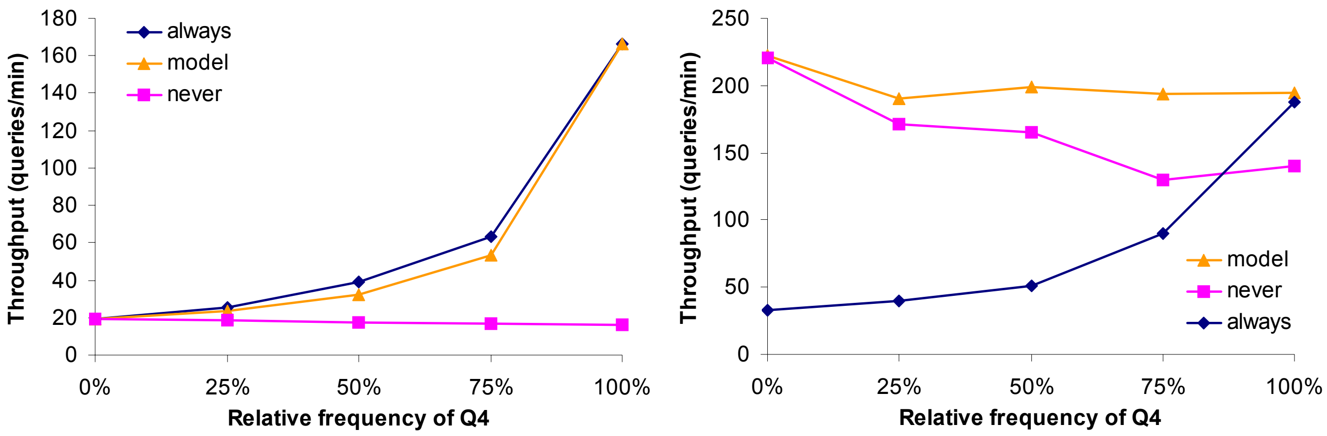

Figure 6. Comparisons of the always-share, never-share, and model-guided policies in Cordoba supporting 20 clients on 2 processors (left) and 32 processors (right), as the relative frequency of Q1 and Q4 varies

model captures the first order performance effects and achieves predictions with reasonable accuracy. It is also important to note that despite the model's error rate, the model's recommendations on the benefits of sharing are nearly always correct: it accurately distinguishes cases where sharing improves performance from those where unshared execution should be preferred. Thus, when used in a binary decision-making process, our analytic model will provide a good basis for deciding whether to share or not to share.

8. EXPLOITING THE MODEL

The model is simple enough to directly incorporate into a DBMS. For example, systems that detect and exploit work sharing at run-time, like [11,8], can integrate the model to their decision procedure, while systems that are based on a multi-query optimizer [20,19] can evaluate the proposed execution plans using the model. In this paper we evaluate the effectiveness of the model by implementing it in Cordoba.

8.1 Multiple Groups of Shared Queries

Sharing fewer queries at a time is one potential way to exploit work sharing while reducing the serialization penalty that results. For example, consider the query from Section 6.1. Work sharing hurts performance in many-CPU cases because it allows the serialization penalty to grow too large. If the system instead limits the number of queries allowed to join any one work sharing group, and partitions the available processors among multiple groups of shared queries, the system could reap the benefits of both work sharing and parallelism.

Ideally the system would add shared queries to a group only until the pivot operator started to become a bottleneck, thus maximizing the benefits of work sharing without reducing available parallelism. Our implementation in Cordoba naturally captures this optimization because it only allows queries to join a sharing group if the model predicts a benefit. If not, the system checks the remaining groups in turn. If none of the groups permit sharing the query will begin executing independently, though it may be joined later on by other queries.

8.2 Experimental Evaluation

Our implementation of the model in Cordoba makes educated recommendations to the execution system at run time about whether to allow work sharing or execute the participating queries independently. We evaluate the effectiveness of our model by running a mix of scan- and join-heavy TPC-H queries (Q1 and Q4, respectively) using the experimental setup described in Section 3. We vary the fraction of the workload consisting of Q4, from 0% to 100% of the queries submitted, then measure the throughput achieved by each of three policies. An *always-share* policy applies work sharing whenever possible, while a *never-share* policy conservatively executes all queries independently. The model-guided policy dynamically evaluates conditions at runtime to determine whether to share a particular query or not.

Figure 6 compares the throughput achieved by Cordoba using each of the three policies, running 5 clients on 2 processors (left) and 20 clients on 32 processors (right). Work sharing is always beneficial in the two-processor system, so the always-share policy achieves the highest performance of the three, while the never-share policy leads to poor (and worsening) results by comparison. The model-guided policy closely tracks the always-share policy. The performance gap occurs because Cordoba makes sharing decisions at runtime as queries arrive in the system, and has no way to know how many queries might eventually come. By contrast, the always-share policy makes the offline optimal decision. Approaches that work with batches of queries (offline), such as multiple query optimization, would not suffer this shortcoming.

The 32-core system (Figure 6 right) presents a completely different hardware landscape to the query engine. Surprisingly, the never-share policy achieves an average of 165 queries/min, more than double the 80 queries/min achieved by always-share. This occurs because the penalty for sharing the wrong queries outweighs the benefit of sharing the right ones. The model-based policy outperforms both of the other two by considering each query individually in the context of the conditions at runtime. It achieves 200 queries/min— speedups of 20% and 2.5x, on average, over the static policies. The maximum speedup is even larger— 40% and 6.7x, respectively— indicating that intelligent work sharing is vital to achieve maximum performance.

9. CONCLUSIONS

This paper presents and evaluates the trade-off between work sharing and parallelism, a phenomenon that cannot be ignored by database systems that aim for high performance in the looming multi-core computing landscape. We show that indiscriminate work sharing actually hurts performance by serializing work that otherwise could have gone in parallel. On the other hand, policies that completely avoid work sharing give up the opportunity to eliminate redundant work. We provide a simple analytical model that captures the trade-off, allowing database systems to apply work sharing judiciously, at runtime. We show that a system must first strive to utilize all available hardware resources, sacrificing work sharing opportunities if necessary to provide parallelism. We integrate the model into a staged database system, capable of exploiting work sharing opportunities at run-time, and we show that basing the decision for work sharing on the model outperforms static schemes that always or never apply work sharing.

10. ACKNOWLEDGEMENTS

We would like to thank Brian Gold and Jared Smolens for their help and feedback during the research process. We also thank the reviewers for their insightful comments. This work was partially supported by grants and equipment from Intel, two Sloan research fellowships, an IBM faculty partnership award, and NSF grants CCR-0205544, CCR-0509356, and IIS-0133686.

11. REFERENCES

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

[2] P. A. Boncz, S. Manegold, and M. L. Kersten. "Database Architecture Optimized for the New Bottleneck: Memory Access." In Proc. VLDB, 1999.

[3] W. Brigde, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza and N. MacNaughton. "The Oracle Universal Server Buffer." In Proc. VLDB, 1997.

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

[5] C. Cook. "Database Architecture: The Storage Engine." Microsoft SQL Server 2000 Technical Article, 2001.

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

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

[8] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. "Database Servers on Chip Multiprocessors: Limitations and Opportunities." In Proc. CIDR, 2007.

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

[10] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. "Performance Tradeoffs in Read-Optimized Databases." In Proc. VLDB, 2006.

[11] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. "QPipe: A Simultaneously Pipelined Relational Query Engine." In Proc. SIGMOD, 2005.

[12] W. Hasan, and R. Motwani. "Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism." In Proc. VLDB, 1994.

[13] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. "An Adaptive Query Execution System for Data Integration." In Proc. SIGMOD, 1999.

[14] C. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. "Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling." In Proc. ICDE, 2007.

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

[16] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures." In Proc. ICDE, 2001.

[17] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors." In Proc. ASPLOS, 1998.

[18] N. Roussopoulos. "View Indexing in Relational databases." In ACM TODS, 7(2):258-290, 1982.

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

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

[21] M. Shao, A. Ailamaki and B. Falsafi. "DBmbench: Fast and Accurate Database Workload Representation on Modern Microarchitecture." In Proc. of CASCON, 2005.

[22] M. Spiliopoulou, M. Hatzopoulos, and C. Vassilakis. "A Cost Model for the Estimation of Query Execution Time in a Parallel Environment Supporting Pipeline." In Computers & Artificial Intelligence, 14(1), 1996.

[23] TPC (Transaction Processing Performance Council). TPC Benchmark H (Decision Support) Standard Specification, Revision 2.6.0, 2006.

[24] T. Urhan, and M. J. Franklin. "XJoin: A Reactively-Scheduled Pipelined Join Operator." In IEEE Data Eng. Bull. 23(2), 2000.

[25] A. N. Wilschut, and P. M. G. Apers. "Dataflow Query Execution in a Parallel Main-Memory Environment." In Proc. of PDIS, 1991.

[26] A. N. Wilschut, and S. A. van Gils. "A Model for Pipelined Query Execution." In Proc. MASCOTS, 1993.

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

[28] M. Zukowski, P. A. Boncz, and M. L. Kersten. "Cooperative Scans." CWI TR INS-E0411, 2004.