Bulletin of the Technical Committee on

Data Engineering

June 2005 Vol. 28 No. 2

IEEE Computer Society

Letters

Special Issue on Databases for New Hardware

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

There are two Data Engineering Bulletin web sites: http://www.research.microsoft.com/research/db/debull and http://sites.computer.org/debull/.
The TC on Data Engineering web page is
http://www.ccs.neu.edu/groups/IEEE/tcde/index.html.

Letter from the Editor-in-Chief

The Current Issue

The characteristics of the hardware upon which modern database systems run have change dramatically since the early years of relational database systems. We have gone from uniprocessor machines of a few MIPS to SMP machines and dual core machines executing at several GIPS. Disks have grown from hundreds of megabytes to hundreds of gigabytes. Main memory has grown from the megabyte range to the gigabyte range. Memory latency has grown from a few machine cycles to over a hundred cycles. Disk latency in machine cycles, has exploded from thousands of machine cycles to millions of cycles.

It is a tribute to the architecture of database systems, which has changed very little, that they continue to execute "reasonably well" on modern hardware. Larger memories have been exploited for larger database caches so as to reduce the number of disk accesses. We have worked hard to make more I/O sequential. Database page size has increased modestly. These simple things have worked fairly well, and have been successful in causing database system performance to increase enormously as hardware capabilities have grown. Nonetheless, the ratio of peak machine performance to realized machine performance has has grown.

We are simply not exploiting modern machines as well as we had exploited the hardware of the 1970's on which the first database systems were implemented. This performance "gap" has attracted the attention of database systems providers as well as the research community. The research community has been very active from quite early in doing the more adventuresome "tuning", such as for processor cache characteristics, e.g AlphaSort, a paper on which was published in SIGMOD'94. Work in this area continues with papers appearing regularly in the leading database conferences. Indeed there is a community of database researchers who have focused on the problem, with interesting and useful results.

I was enthusiastic about an issue on this topic when Gustavo Alonso, this issue's editor, first proposed it. Gustavo has solicited and assembled an interesting collection of papers on this important topic which are well worth serious study. I want to thank Gustavo for his efforts and I encourage you to read the issue. I am confident that you will find the papers presented here both interesting and useful.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Databases have been around for quite a while now. A great success story, today's databases are built on very solid theoretical foundations and an amazing wealth of engineering knowledge. Yet, for all their success, some of the fundamental assumptions behind the architecture of modern database systems are being challenged by the evolution of the underlying hardware. In this issue of the Bulletin, several authors explore the problem in great detail and provide valuable insights on how to exploit new features at the hardware to improve database performance.

The first article, by Jim Gray, reviews the history of a number of database benchmarks. It points out that the rate of improvement in speed for several key benchmarks has slowed down in the last years. He conjectures that the reason for this slowed improvement is due to the benchmarks becoming bound by memory latency, and the slow progress in speed merely reflecting the slow improvement in memory latency. The second article, by Ken Ross et al., discusses how different aspects of modern processors can be exploited by database engines. It provides a very good overview of the different issues involved and of a number of techniques proposed over the years to adapt database processing to the characteristics of the underlying hardware. While Ross' article describes a collection of techniques, the third article in the issue -by Harizopoulos and Ailamaki- proposes a novel database architecture that optimizes data and instruction locality at all levels of the memory hierarchy. The relevance of this contribution is that it shows what needs to be changed in the way databases are built to be able to function efficiently on modern processors. The fourth article in the issue, by Zukowsky et al., explores the problem of minimizing the amount of instructions needed to process a query. They show that the performance of today's databases is order of magnitudes worse than hand coding the solution using C++. Then they go on to describe a new query execution engine, X100, that uses in-cache vectorized processing to reach very significant performance improvements. The last article, by Bandi et al., tackle the problem of using new hardware such as MEMS storage devices to improve the performance of certain database operations. The article shows how to effectively use MEMS to process spatial queries and they show how to incorporate this new hardware into existing database products.

Taken together, these articles offer a fascinating perspective on how some of the basic assumptions behind the architecture of existing database systems are being challenged by advances in hardware. If we consider the changes that are taken place in many other areas (e.g., use of clusters for scalability, or the advent of aspect oriented programming) it becomes clear that databases must radically change their architecture to avoid becoming niche systems usable only in a very small set of applications. I am sure some of these topics will soon become the main theme of new issues of the Data Engineering Bulletin.

Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETH Zurich) Zurich
Switzerland

A “Measure of Transaction Processing” 20 Years Later

Jim Gray
Microsoft Research
April 2005

”A Measure of Transaction Processing Power” [1] defined three performance benchmarks: DebitCredit: a test of the database and transaction system, Sort: a test of the OS and IO system, and Copy: a test of the file system.

DebitCredit morphed into TPC-A and then TPC-C. In 1985, systems were nearing 100 transactions per second, now they deliver 100,000 transactions per second, and a palmtop can deliver several thousand transactions per second (www.tpc.org, and [2]). Price-performance (measured in dollars/tps) has also improved dramatically as shown in Figure 1.

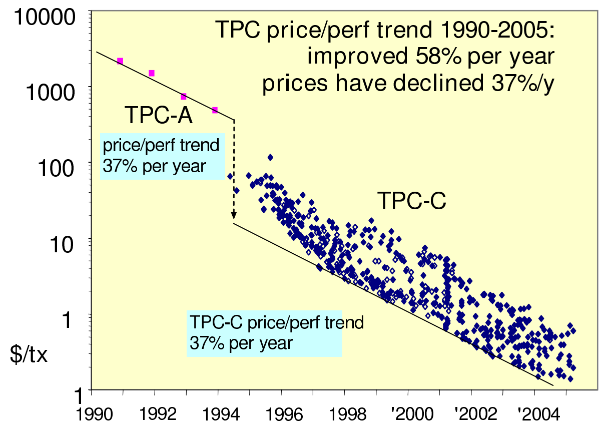


Figure 1: (by Charles Levine from [2]): Price/performance trend lines for TPC-A and TPC-C. The 15-year trend lines track Moore’s Law (100x per 10 years.)

The sort benchmark has seen similar improvements. The traditional Datamation ”sort 1M records” now runs in a fraction of a second and has been replaced by PennySort (sort as much as you can for a penny), MinuteSort (sort as much as you can in a minute), and Terabyte sort (sort a trillion records). Each of these three benchmarks has a Daytona (commercial), and Indy (benchmark special) category [3].

This year Jim Wyllie of IBM Almaden Research won both Terabyte Sort and MinuteSort medals with his SCS (SAN Cluster Sort) [4] using SUSE Linux (SLES8) and IBM’s General Parallel File System (GPFS) on an IBM 40-node 80-Itanium cluster with a SAN array of 2,520 disks. The machine delivered up to 14 GBps of IO bandwidth and sorted a terabyte in 437 seconds. SCS was able to sort 125 GB within a minute (wow!). It is hard to know the price of this system (now at UCSD) but the ”list” price is at least $9M.

At the other end of the spectrum, Robert Ramey with his PostmansSort used a $950 Wintel box (3.2 GHz Pentium4, 2 Maxtor SATA disks, WindowsXP) to sort 16.3 GB in 979 seconds - setting a new Daytona Pennysort record [5].

Figure 2 shows that price-performance improved about 68%/year each year since 1985, handily beating Moore’s 58%/year law. Sort speed (records sorted per second) doubled every year between 1985 and 2000. That doubling in part came from faster hardware, in part from better software, and in part from the use of LOTS more hardware (the year 2000 system used 1,962 processors and 2,168 disks.) In the last 5 years, peak sort speed has only improved 2.4x (about 20%/year improvement).

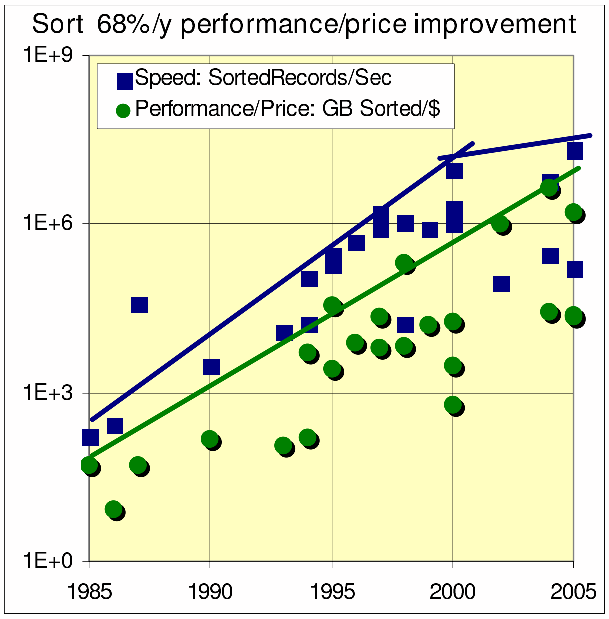


Figure 2: Sort speed doubled every year from 1985 to 2000; but it only improved 2.4x since then (a 20%/y improvement). Price-performance has steadily improved at 68%/y.

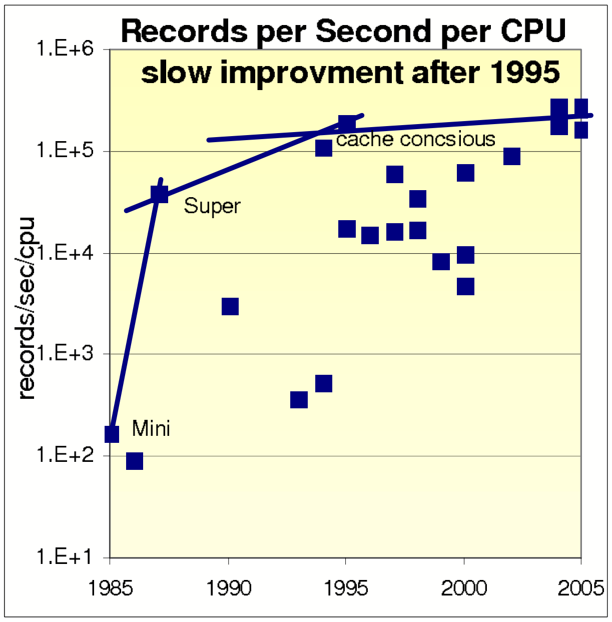


Figure 3: Sort speed per processor improved 100x at first, then 10x more with cache-conscious algorithms. But for the last 10 years improvement has been about 20% year - probably reflecting the improvement of bulk RAM latency.

Performance improvements have been accomplished with multi-processors (hundreds of them). Price-performance improvements have come from cheaper and faster disks and from cheaper processors. But as Figure 3 shows, per-processor speeds seem to have plateaued. Sorted-records/second/processor (r/s/p for short) improved ten fold between 1985 and 1995. But speed has improved only 2.7 fold in the last decade.

The 1980's saw 200 r/s/p on a minicomputer to 38.5 k r/s/p on a Cray. In 1994, AlphaSort showed the importance of cache-conscious sorts and got to 111 k r/s/p. Since then, there has been a slow climb to 280 k r/s/p (e.g., Jim Willey's SCS Itanium TerabyteSort and the Pentium4 SkeenSort.)

I conjecture that this relatively slow improvement reflects the slow improvement in memory latency. All the algorithms are now cache conscious, so they are all limited by the speed of bulk memory - processor speed (and even cache speed) is not relevant here since sort cache misses are essentially random during the "comparison" and merge phase. But, that is just my guess. It would make an interesting study for the hardware architects – speed problems may lie elsewhere. But, my guess is: Remember! It's the memory.

References

[1] Anon et. al.. "A Measure of Transaction Processing Power" Datamation, 1 April, 1985. Also at: *http://research.microsoft.com/ gray/papers/AMeasureOfTransactionProcessingPower.doc*

[2] Jim Gray and Charles Levine. Thousands of DebitCredit Transactions per Second: Easy and Inexpensive. MSR TR 2005-39. *http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2005-39*

[3] Sort Benchmark Website *http://research.microsoft.com/barc/SortBenchmark/*

[4] Jim Wyllie Sorting on a Cluster Attached to a Storage Area Network April, 2000 *http://research.microsoft.com/barc/SortBenchmark/2005_SCS_Wyllie.pdf*

[5] Robert Ramey. 2005 Performance/Price Sort and PennySort. *http://research.microsoft.com/barc/ SortBenchmark/2005_PostMansSort.pdf*

Architecture Sensitive Database Design: Examples from the Columbia Group

Kenneth A. Ross[*] — Columbia University
John Cieslewicz — Columbia University
Jun Rao — IBM Research
Jingren Zhou — Microsoft Research

In this article, we discuss how different aspects of modern CPU architecture can be effectively used by database systems. While we shall try to cover most important aspects of modern architectures, we shall not aim to provide an exhaustive survey of the literature. Rather, we will use examples from our own recent work to illustrate the principles involved. While there is related work on using other components of modern machines such as graphics cards and intelligent disk controllers for database work, we limit our discussion here to just a single CPU and the memory hierarchy (excluding disks).

In recent years the primary focus of database performance research has fundamentally shifted from disks to main memory. Systems with a very large main memory allow a database's working set to fit in RAM. At the same time processors have continued to double in performance roughly every eighteen months in line with "Moore's Law," while memory speed improvements have been marginal. The CPU now spends many hundreds of cycles waiting for data to arrive from main memory, and database workloads are often memory-bound. Because of this bottleneck, database research at Columbia and elsewhere has focused on effectively using the cache hierarchy to improve database performance.

Changes in processor technology have also influenced database research. Modern processors feature a very long pipeline, which helps to provide better performance. It also results in a higher branch misprediction penalty, because the pipeline must be flushed and restarted when a branch is mispredicted.

Modern CPUs provide the database programmer with a rich instruction set. Using specialized instructions, such as SIMD instructions, database performance can be improved by allowing a degree of parallelism and eliminating some conditional branch instructions, thus reducing the impact of branch mispredictions.

1 The Data Cache

In traditional B^+-Trees, child pointers are stored explicitly in each node. When keys are small, child pointers can take half of the space within each node, significantly reducing the cache-line utilization. The idea of Cache-Sensitive Search Trees (CSS-Trees) [7] is to avoid storing child pointers explicitly, by encoding a complete n-ary tree in an array structure. Child nodes can be found by performing arithmetic on array offsets. As a result, each node in a CSS-Tree can store more keys than a B^+-Tree, and thus reduce the number of cache-line accesses when traversing a tree. The small amount of CPU overhead of computing child node addresses is more than offset by the savings of fewer cache misses. CSS-Trees are expensive to update, and are therefore only suited for relatively static environments where updates are batched.

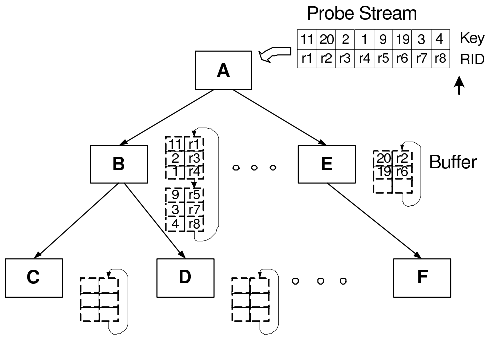
[*]Supported by NSF grant IIS-01-20939. The work described here was performed while the authors were at Columbia University.

To overcome this limitation of CSS-Trees, we developed another indexing technique called Cache Sensitive B^+-Trees (CSB^+-Trees) [8]. It is a variant of B^+-Trees that stores all the child nodes of any given node contiguously. This way, each node only needs to keep the address of its first child. The rest of the children can be found by adding an offset to that address. Since only one child pointer is stored explicitly in each node, the utilization of a cache line is almost as high as a CSS-Tree. CSB^+-Trees support incremental updates in a way similar to B^+-Trees, but have somewhat higher cost because when a node splits, extra copying is needed to maintain node contiguity. In summary, CSB^+-Trees combine the good cache behavior of CSS-Trees with the incremental maintainability of B^+-Trees, and provide the best overall performance for many workloads with a mix of queries and updates.

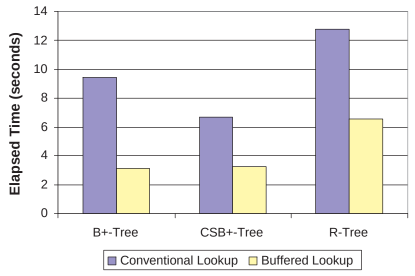
CSS-Trees and CSB^+-Trees focus on better utilization of each cache line and optimize the search performance of a single lookup. For applications in which a large number of accesses to an index structure are made in a small amount of time, it is important to exploit spatial and temporal locality between consecutive lookups.

Cache memories are typically too small to hold the whole index structure. If probes to the index are randomly distributed, consecutive lookups end with different leaf nodes, and almost every lookup will find the corresponding leaf node not in the cache. The resulting *cache thrashing* can have a severe impact on the overall performance. Our results show that conventional index structures have poor cache miss rates for bulk access. Even cache-sensitive algorithms such as CSB^+-Trees also have moderately high cache miss rates for bulk access, despite their cache-conscious design for single lookups.

We propose techniques to buffer probes to memory-resident tree-structured indexes to avoid cache thrashing [12]. As a lookup proceeds down the tree, a record describing the lookup (containing a probe identifier and the search key) is copied into a buffer of such records for an index node. These buffers are periodically emptied, and divided among buffers for the child nodes. While there is extra copying of data into buffers, we expect to benefit from improved spatial and temporal locality, and thus to incur a smaller number of cache misses.



(a) Variable-Size Buffered Tree

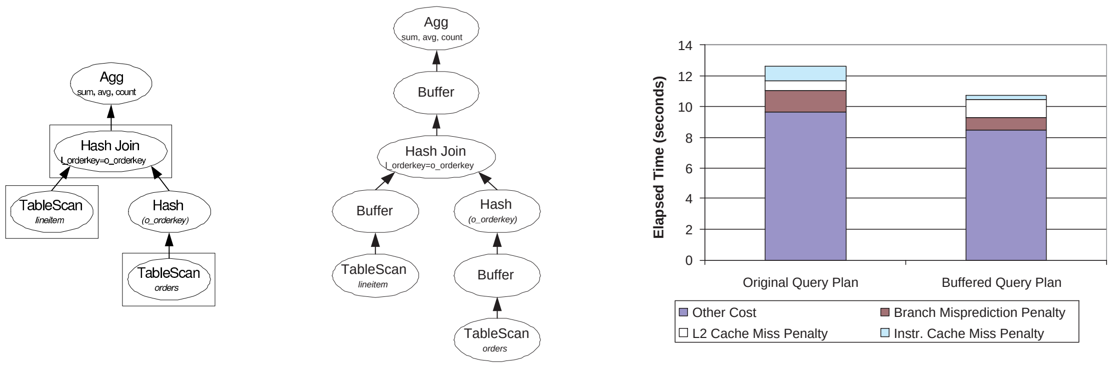


(b) Overall Performance

Figure 1: Buffering Accesses to Indexes

Figure 1(a) shows a variable-size buffered tree. The root node A is used to process a large number of probes. Buffers for A's children are created, and filled with the corresponding probes. Eventually, the buffer of node B gets flushed, buffers for its children are created, and probes are distributed between the child buffers. When the leaf level is reached, the probes generate join results. When multiple levels of the tree can fit in the cache, we do not buffer at every level. Instead, we buffer once for the maximum number of levels that fit in the cache. For more details, see [12].

Figure 1(b) shows the results of buffering for various index techniques. (See [12] for the experimental setup.) Buffered lookup is uniformly better, with speedups by a factor of two to three. Interestingly, the performance of B^+-trees and CSB^+-trees differ little once buffering techniques are applied. The buffering overhead is the same; the size of nodes is relatively unimportant for bulk access.



(a) Original Plan (b) Buffered Plan (c) Execution Time Breakdown

Figure 2: Buffered Plans

2 The Instruction Cache

As demonstrated in [1, 6, 5, 2], database workloads exhibit instruction footprints that are much larger than the first-level instruction cache. A conventional demand-pull query execution engine generates a long pipeline of execution operators. An operator returns control to its parent operator immediately after generating *one* tuple. The instructions for this operator get evicted from the cache to empty cache memory space for the parent operator. However, the evicted instructions are required when generating the next tuple, and have to be loaded into the cache again. The resulting *instruction cache thrashing* may reduce the overall query performance significantly.

To avoid the instruction cache thrashing problem, we proposed implementing a new light-weight buffer operator using the conventional iterator interface [13]. A buffer operator simply batches the intermediate results of the operator(s) below it. We do not require modifications to the implementations of any existing operators. Buffers are placed between groups of operators, where a group is the largest set of consecutive operators that together fit in the instruction cache.

Like other operators, when the GetNext() function of a buffer operator is called for the first time, it begins to retrieve a tuple from the child operator. However, instead of returning the tuple immediately, a buffer operator saves the pointer to the previous tuple and continues to retrieve another tuple. A buffer operator won't return any tuple until either end-of-tuples is received from the child operator or it collects a full array of tuples. Subsequent requests will be filled directly from its buffered array until all the buffered tuples have been consumed and the operator begins to retrieve another array of tuples. The benefit of buffering is that it *increases temporal and spatial instruction locality* below the buffer operator. Another possible benefit of buffering is better hardware branch prediction. More details can be found in [13].

We validate our techniques using an experimental prototype built on PostgreSQL 7.3.4. All of the queries are executed on a TPC-H benchmark database with scale factor 0.2 and can be answered within memory without I/O interference. We illustrate the buffering method on the following query:

```
SELECT sum(o_totalprice), count(*), avg(l_discount)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND l_shipdate <= date '1998-11-01';
```

Figure 2 shows the plans using a hash join and the query execution time breakdown. For the buffered plan, footprint analysis suggests three execution groups (marked with boxes). The "Hash" operator builds a hash table for the "orders" table on orderkey. The "HashJoin" operator implements the probe phase of the join. Both the build and probe phases are complex enough that they should be considered separately. The build operator, "Hash", is blocking and is not considered in any execution group. The combined footprint of a "TableScan" and either phase is larger than the L1 instruction cache. Therefore, a buffer operator is added for each "TableScan" operator. The buffered plan reduces the instruction cache misses by 70% and the branch mispredictions by 44%. Overall, the buffered plan improves query performance by 15%.

3 Branch Mispredictions

Modern CPUs have a pipelined architecture in which many instructions are active at the same time, in different phases of execution. Conditional branch instructions present a significant problem in this context, because the CPU does not know in advance which of the two possible outcomes will happen. Depending on the outcome, different instruction streams should be read into the pipeline.

CPUs try to *predict* the outcome of branches, and have special hardware for maintaining the branching history of many branch instructions. Such hardware allows for improvements of branch prediction accuracy, but branch misprediction rates may still be significant. Branches that are rarely taken, and branches that are almost always taken are generally well-predicted by the hardware. The "worst-case" branch behavior is one in which the branch is taken roughly half of the time, in a random (i.e., unpredictable) manner. In that kind of workload, branches will be mispredicted half of the time.

A mispredicted branch incurs a substantial delay. Branch misprediction has a significant impact on modern database systems [1]. As a result, one might aim to design algorithms for "kernel" database operations that exhibit good branch-prediction accuracy on modern processors.

Suppose we have a large table stored as a collection of arrays, one array per column, as advocated in [3]. Let's number the arrays `r1` through `rn`. We wish to evaluate a compound selection condition on this table, and return pointers (or offsets) to the matching rows. Suppose the conditions we want to evaluate are `f1` through `fk`. For simplicity of presentation, we'll assume that each `fi` operates on a single column which we'll assume is `ri`. So, for example, if `f1` tests whether the first attribute is equal to 3, then both the equality test and the constant 3 are encapsulated within the definition of `f1`. We assume that the condition we wish to test is a conjunction of basic conditions. For more general conditions, see [10].

The straightforward way to code the selection operation applied to all records would be the following. The result is returned in an array called `answer`. In each algorithm below, we assume that the variable `j` has been initialized to zero.

```
for(i=0;i<number_of_records;i++) {
  if(f1(r1[i]) && ... && fk(rk[i])) { answer[j++] = i;} }
```

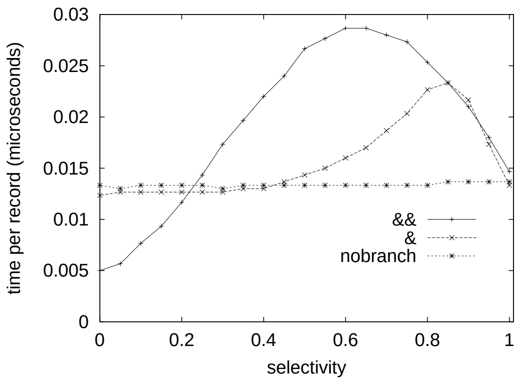
The important point is the use of the `C` idiom "`&&`". This implementation saves work when `f1` is very selective. When `f1(r1[i])` is zero, no further work (using `f2` through `fk`) is done for record `i`. However, the potential problem with this implementation is that its assembly language equivalent has `k` conditional branches. If the initial functions `fj` are not very selective, then the system may execute many branches. The closer each selectivity is to 0.5, the higher the probability that the corresponding branch will be mispredicted, yielding a significant branch misprediction penalty. An alternative implementation uses logical-and (`&`) in place of `&&`:

```
for(i=0;i<number_of_records;i++) {
  if(f1(r1[i]) & ... & fk(rk[i])) { answer[j++] = i;} }
```

Because the code fragment above uses logical "`&`" rather than a branching "`&&`", there is only one conditional branch in the corresponding assembly code instead of `k`. We may perform relatively poorly when `f1` is selective, because we always do the work of `f1` through `fk`. On the other hand, there is only one branch, and so we expect the branch misprediction penalty to be smaller. The branch misprediction penalty for that one branch may still be significant when the combined selectivity is close to 0.5. The following loop implementation has *no* branches within the loop.

```
for(i=0;i<number_of_records;i++) {
  answer[j] = i;   j += (f1(r1[i]) & ... & fk(rk[i])); }
```

To see the difference between these three methods, we implemented them in C for k = 4 and ran them on a 750Mhz Pentium III under Linux. (See [10] for details of the experiment.) The graph on the right shows the results. *Each of the three implementations is best in some range, and the performance differences are significant. On other ranges, each implementation is about twice as bad as optimal.* There are, in fact, many additional plans that can be formed by combining the three approaches. An example that combines all three is the following (with the loop code omitted):

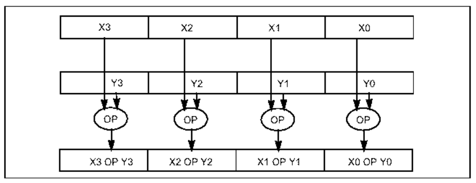


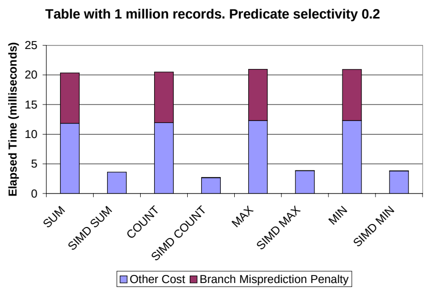
```
  if((f1(r1[i]) & f2(r2[i])) && f3(r3[i]))
    { answer[j] = i;  j += (f4(r4[i]) & ... & fk(rk[i])); }
```

Several of these combination plans turn out to be superior to the three basic methods over some selectivity ranges. In [10, 9] we formulate a cost model that accurately accounts for the cost of branch mispredictions in query plans for conditional scans. We also derive exact and heuristic optimization algorithms that can find good ways of combining the selection conditions to minimize the overall cost.

4 SIMD Instructions

Modern CPUs have instructions that allow basic operations to be performed on several data elements in parallel. These instructions are called SIMD instructions, since they apply a single instruction to multiple data elements. SIMD technology was initially built into commodity processors to accelerate multimedia applications. SIMD technology comes in various flavors on a number of architectures. We focus on a Pentium 4 because the SIMD instructions are among the most powerful of mainstream commodity processors, including 128-bit SIMD registers and floating point SIMD operations.





(a) Packed Operation (b) Aggregation and Branch Misprediction

Figure 3: Using SIMD Instructions

We illustrate the use of SIMD instructions using the packed single-precision floating-point instruction available on Intel SSE technology chips, and shown in Figure 3(a). Both operands are using 128-bit registers. Each source operand contains four 32-bit single-precision floating-point values, and the destination operand contains the results of the operation (OP) performed in parallel on the corresponding values (X0 and Y0, X1 and Y1, X2 and Y2, and X3 and Y3) in each operand. SIMD operations include various comparison, arithmetic, shuffle, conversion and logical operations [4]. A SIMD comparison operation results in an *element mask* corresponding to the length of the packed operands. For this operator, the result is a 128-bit wide element mask containing four 32-bit sub-elements, each consisting either of all 1's (0xFFFFFFFF) where the comparison condition is true or all 0's (0x00000000) where it is false.

Suppose we wish to sum up `y` values for rows whose `x` values satisfy the given condition. SIMD instructions can compare 4 `x` elements at a time and generate an element mask. Suppose that we have a SIMD register called `sum` that is initialized to 4 zero words. The element mask can be used to add up qualified `y` values in two SIMD operations `sum[1..4]=SIMD_+(sum[1..4], SIMD_AND(Mask[1..4], y[1..4]))`. The idea is to convert non-matched elements to zeroes. After finishing processing, we need to add the 4 elements of *sum* as the final result. Similar techniques also apply to other aggregation functions [11].

Figure 3(b) shows the performance and branch misprediction impact of a query corresponding to "SELECT AGG(y) FROM R WHERE $x_{low} < x < x_{high}$" for different types of aggregation functions. Table R has 1 million records and the predicate selectivity is 20%. There are no conditional branches in the SIMD-optimized algorithms. As a result, they are faster than the original algorithms by *more* than a factor of four. Forty percent of the cost of the original algorithms is due to branch misprediction.

The use of SIMD instructions has two immediate performance benefits: It allows a degree of parallelism, so that many operands can be processed at once. It also often leads to the elimination of conditional branch instructions, reducing branch mispredictions. SIMD instructions can also be useful for sequential scans, index operations, and joins [11].

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of International Conference on Very Large Data Bases*, 1999.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of International Symposium on Computer Architecture*, 1998.

[3] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of International Conference on Very Large Data Bases*, 1999.

[4] Intel Inc. IA32 intel architecture optimization reference manual. 2004.

[5] 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. of the 25th Int. Symposium on Computer Architecture*, 1998.

[6] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[7] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of International Conference on Very Large Data Bases*, 1999.

[8] J. Rao and K. A. Ross. Making B+ trees cache conscious in main memory. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2000.

[9] K. A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2002.

[10] K. A. Ross. Selection conditions in main memory. *ACM Transactions on Database Systems*, 29(1):132–161, 2004.

[11] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2002.

[12] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proceedings of International Conference on Very Large Data Bases*, 2003.

[13] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2004.

StagedDB: Designing Database Servers for Modern Hardware

Stavros Harizopoulos, Anastassia Ailamaki
Carnegie Mellon University
{stavros,natassa}@cs.cmu.edu

Abstract

Advances in computer architecture research yield increasingly powerful processors which can execute code at a much faster pace than they can access data in the memory hierarchy. Database management systems (DBMS), due to their intensive data processing nature, are in the front line of commercial applications which cannot harness the available computing power. To prevent the CPU from idling, a multitude of hardware mechanisms and software optimizations have been proposed. Their effectiveness, however, is limited by the sheer volume of data accessed and by the unpredictable sequence of memory requests.

In this article we describe StagedDB, a new DBMS software architecture for optimizing data and instruction locality at all levels of the memory hierarchy. The key idea is to break database request execution in stages and process a group of sub-requests at each stage, thus effortlessly exploiting data and work commonality. We present two systems based on the StagedDB design. STEPS, a transaction coordinating mechanism demonstrated on top of Shore, minimizes instruction-cache misses without increasing the cache size, eliminating two thirds of all instruction misses when running on-line transaction processing applications. QPipe, a staged relational query engine built on top of BerkeleyDB, maximizes data and work sharing across concurrent queries, providing up to 2x throughput speedup in a decision-support workload.

1 Introduction

Research shows that the performance of Database Management Systems (DBMS) on modern hardware is tightly coupled to how efficiently the entire memory hierarchy, from disks to on-chip caches, is utilized. Unfortunately, according to recent studies, 50% to 80% of the execution time in database workloads is spent waiting for instructions or data [1, 2, 10]. Current technology trends call for computing platforms with higher-capacity memory hierarchies, but with each level requiring increasingly more processor cycles to access. At the same time, advances in chip manufacturing process allow the simultaneous execution of multiple programs on the same chip, either through hardware-implemented threads on the same CPU (simultaneous multithreading—SMT), or through multiple CPU cores on the same chip (chip multiprocessing—CMP), or both. With higher levels of hardware-available parallelism, the performance requirements of the memory hierarchy increase. To improve DBMS performance, it is necessary to engineer software that takes into consideration all features of new microprocessor designs.

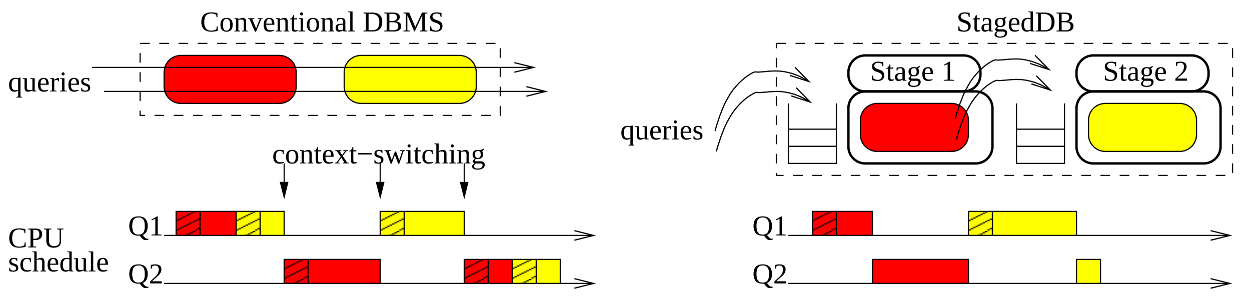


Figure 1: Example of conventional and Staged DBMS architecture with two system modules. Striped boxes in the query schedule (assuming single CPU) correspond to time spent accessing instructions and data common to both queries; these need to be reloaded when modules are swapped. StagedDB loads a module only once.

1.1 Techniques to improve memory hierarchy performance

Since each memory hierarchy level trades capacity for lookup speed, research has focused on ways to improve locality at each level. Techniques that increase reusability of a page or a cache block are referred to as *temporal locality* optimizations, while *spatial locality* optimizations are improving the utilization within a single page or cache block. Database researchers propose relational processing algorithms to improve data and instruction temporal locality [16, 14], and also indexing structures and memory page layouts that improve spatial locality [15]. Buffer pool replacement policies seek to improve main memory utilization by increasing the reusability of data pages across requests [5, 12]. Computer architecture researchers propose to rearrange binary code layout [13] to achieve better instruction spatial locality. A complementary approach to improve performance is to overlap memory hierarchy accesses with computation. At the software level, recent techniques prefetch known pages ahead of time [4], while at the microarchitecture level, out-of-order processors tolerate cache access latencies by executing neighboring instructions [9]. Despite the multitude of proposed optimizations, these are inherently limited by the sheer volume of data that DBMS need to process and by the unpredictable sequence of memory requests.

1.2 StagedDB design: exploiting concurrency to improve locality

Most of the research to date for improving locality examines data accessed and instructions executed by a single query (or transaction) at a time. Database systems, however, typically handle multiple concurrent users. Request concurrency adds a new dimension for addressing the locality optimization problem. By properly synchronizing and multiplexing the concurrent execution of multiple requests there is a potential of increasing both data and instruction reusability at all levels of the memory hierarchy. Existing DBMS designs, however, pose difficulties in applying such execution optimizations. Since typically each database query or transaction is handled by one or more processes (threads), the DBMS essentially relinquishes execution control to the OS and then to the CPU. A new design is needed to allow for application-induced control of execution.

StagedDB [6], is a new DBMS software architecture for optimizing data and instruction locality at all levels of the memory hierarchy. The key idea is to break database request execution in *stages* and process a group of sub-requests at each stage, thus effortlessly exploiting data and work commonality. The StagedDB design requires only a small number of changes to the existing DBMS codebase and provides a new set of execution primitives that allow software to gain increased control over what data is accessed, when, and by which requests. Figure 1 illustrates the high-level idea behind StagedDB. For simplicity, we only show two requests passing through two stages of execution. In conventional DBMS designs, context-switching among requests occurs at random points, possibly evicting instructions and data that are common among requests executing at the same stage (these correspond to the striped boxes in the left part of Figure 1). A StagedDB system (right part of Figure

1) consists of a number of self contained modules, each encapsulated into a stage. Group processing at each stage allows for a context-aware execution sequence of requests that promotes reusability of instructions and data. The benefits of the StagedDB design are extensively described elsewhere [6].

In this article we present an overview of two systems based on the StagedDB design. STEPS, a transaction coordinating mechanism, minimizes instruction misses at the first-level (L1-I) cache (Section 2), while QPipe, a staged relational query engine, maximizes data and work sharing across concurrent queries (Section 3).

2 *STEPS* for Cache-Resident Code

According to recent research, instruction-related delays in the memory subsystem account for 25% to 40% of the total execution time in Online Transaction Processing (OLTP) applications [2, 10]. In contrast to data, instruction misses cannot be overlapped with out-of-order execution, and instruction caches cannot grow as the slower access time directly affects the processor speed. With commercial DBMS exhibiting more than 500KB of OLTP code footprint [11] and modern CPUs having 16-64KB instruction caches, transactional code paths are too long to achieve cache-residency. The challenge is to alleviate the instruction-related delays without increasing cache size.

STEPS (for Synchronized Transactions through Explicit Processor Scheduling) is a technique that minimizes instruction-cache misses in OLTP by multiplexing concurrent transactions and exploiting common code paths [7]. At a higher level, STEPS applies the StagedDB design to form groups of concurrent transactions that execute the same transactional operation (e.g., traversing an index, updating a record, or performing an insert). Since DBMS typically assign a thread to each transaction, STEPS introduces a thin wrapper around each transactional operation to coordinate threads executing the same operation. Although transactions in a group have a high degree of overlap in executed instructions, a conventional scheduler that executes one thread after the other would still incur new instruction misses for each transaction execution (left part of Figure 2). The reason is that the code working set of transactional operations typically overwhelms the L1-I cache. To maximize instruction sharing among transactions in the same group, STEPS lets only one transaction incur compulsory instruction misses, while the rest of the transactions “piggyback” onto the first one, finding the instructions they need in the cache. To achieve this, STEPS identifies the points in the code where the cache fills up and performs a quick context-switch, so that other transactions can execute the cache-resident code (right part of Figure 2).

2.1 Implementation of STEPS

We implement STEPS inside the Shore database storage manager [3] by wrapping each transactional operation to form groups of threads. To achieve overhead-free context-switching between threads in the same group, we execute only the *core* context-switch code, updating only the CPU state, and postponing updates to thread-specific data structures until those updates become necessary. In our implementation, the fast context-switch

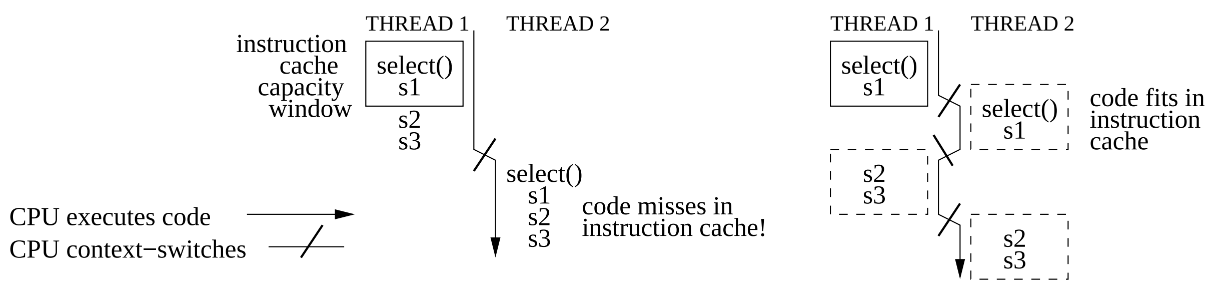
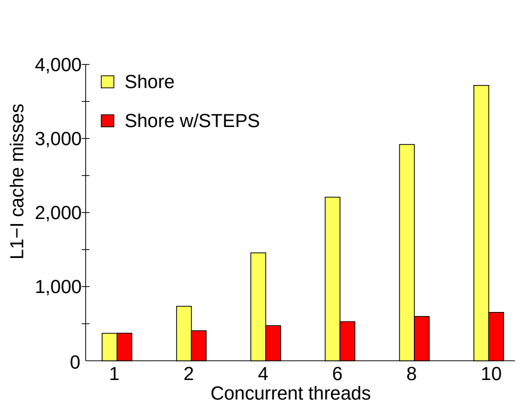
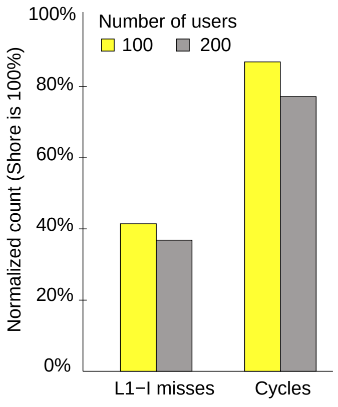


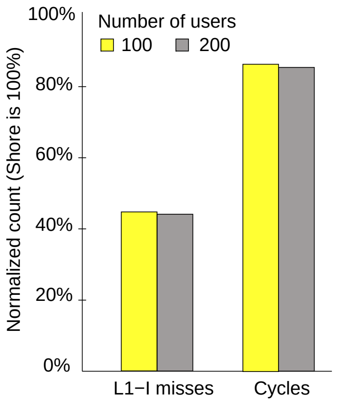
Figure 2: Illustration of instruction-cache aware context switching with STEPS.



(a)



(b)



(c)

Figure 3: (a) L1-I cache misses for Shore without and with STEPS, when running index select. (b) L1-I miss and execution cycles reduction with STEPS running a single TPC-C transaction, and, (c) a mix of four transactions.

code is 76 bytes and therefore only occupies three 32-byte or two 64-byte blocks in the instruction cache. We locate points in the source code to insert context-switch statements by using a cache simulation tool and also by measuring actual misses in the code using hardware counters. The rest of the code changes are restricted to the thread package, to make sure that every time a thread yields control (either because it needs to wait on I/O, or failed to acquire a lock, or for any other reason, such as aborting a transaction), all thread-package structures that were not updated during fast context-switches are checked to ensure system consistency.

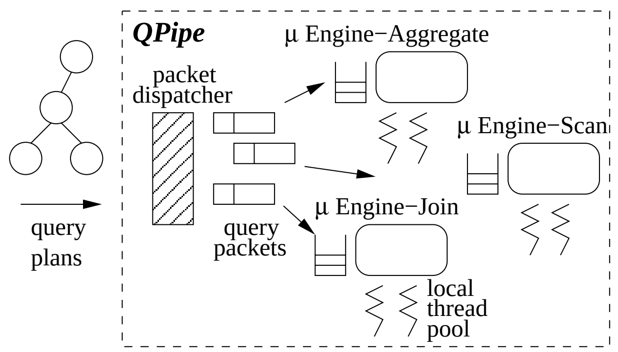
2.2 Evaluation

We conducted an extensive evaluation of STEPS, using both real hardware (two different CPU architectures) and full-system simulation, on both microbenchmarks and full workloads. The full results are presented elsewhere [7]. In this article we present microbenchmark and TPC-C results on a server with a single AMD AthlonXP processor, which has a large, 64KB instruction cache. Figure 3a shows that, when running a group of transactions performing an indexed selection with STEPS, L1-I cache misses are reduced by 96% for each additional concurrent thread. Figures 3b and 3c show that when running a single transaction (Payment) or a mix of transactions from TPC-C in a 10-20 Warehouse configuration (100-200 concurrent users), STEPS reduces instruction misses by up to 65% while at the same time produces up to 30% speedup.

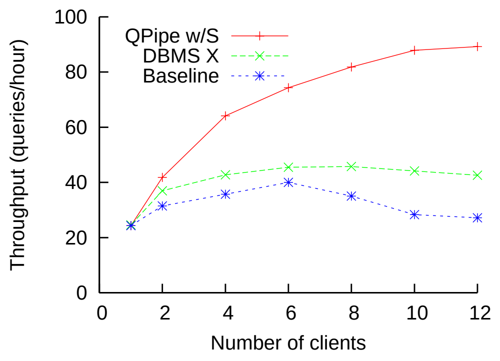
3 QPipe: A Staged Relational Query Engine

Modern query execution engines execute queries following the “one-query, many-operators” model. A query enters the engine as an optimized plan and is executed as if it were alone in the system. The means for sharing common data across concurrent queries is provided by the buffer pool, which keeps information in main memory according to a replacement policy. The degree of sharing the buffer pool provides, however, is extremely sensitive to timing; in order to share data pages the queries must arrive simultaneously to the system and must execute in lockstep, which is highly unlikely. Queries can share work through materialized views which, however, have a maintenance overhead and also require prior workload knowledge.

To maximize data and work sharing at execution time, we propose to monitor each relational operator for every active query in order to detect overlaps. For example, one query may have already sorted a file that another query is about to start sorting; by monitoring the sort operator we can detect this overlap and reuse the sorted file. Once an overlapping computation is detected, the results are simultaneously pipelined to all



(a)



(b)

Figure 4: (a) QPipe architecture: queries with the same operator queue up at the same micro-engine (for simplicity, only three micro-engines are shown). (b) Throughput results for a set of TPC-H queries.

participating parent nodes, thereby avoiding materialization costs. There are several challenges in embedding such an evaluation model inside a traditional query engine: (a) how to efficiently detect overlapping operators and decide on sharing eligibility, (b) how to cope with different consuming/producing speeds of the participating queries, and, (c) how to overcome the optimizer's restrictions on the query evaluation order to allow for more sharing opportunities. The overhead to meet these challenges using a "one-query, many-operators" query engine would offset any performance benefits.

To efficiently detect and exploit overlaps across queries, we introduce *QPipe*, a new query engine architecture which follows a "*one-operator, many-queries*" design philosophy [8]. Each relational operator is promoted to an independent micro-engine which manages a set of threads and serves queries from a queue (Figure 4a). A *packet dispatcher* converts an incoming query plan to a series of query packets. Data flow between micro-engines occurs through dedicated buffers - similar to a parallel database engine. QPipe achieves better resource utilization than conventional engines by grouping requests of the same nature together, and by having dedicated micro-engines to process each group of similar requests. Every time a new packet queues up in a micro-engine, all existing packets are checked for overlapping work. On a match, each micro-engine employs different mechanisms for data and work sharing, depending on the enclosed relational operator.

3.1 QPipe prototype

We implement QPipe on top of the BerkeleyDB storage manager, using native OS threads. The resulting prototype is a versatile engine, naturally parallel, running on a wide range of multi-processor servers. Each micro-engine is a different C++ class with separate classes for the thread-pool support, the shared-memory implementation of queues and buffers, the packet dispatcher, and the modules for detecting and exploiting overlapping work. The basic functionality of each micro-engine is to dequeue the packet, process it, optionally read input or write output to a buffer, and destroy the packet. Calls to data access methods are wrappers for the underlying database storage manager (BerkeleyDB) which adds the necessary transactional support, a buffer-pool manager, and table access methods. Client processes can either submit packets directly to the micro-engines or send a query plan to the packet dispatcher which creates and routes packets accordingly.

3.2 Evaluation

We experiment with a 4GB TPC-H database on a 2.6 GHz P4 machine, with 2GB of RAM and four 10K RPM SCSI drives (organized as software RAID-0 array), running Linux 2.4.18. We use a pool of eight representative TPC-H queries, randomly generating selection predicates every time a query is picked. Figure 4b shows the throughput achieved, when varying the number of clients from 1 to 12, for three systems. “Baseline” is the BerkeleyDB-based QPipe implementation with sharing disabled, “QPipe w/S” is the same system with sharing enabled, and “DBMS X” is a major commercial database system. Although different TPC-H queries do not exhibit overlapping computation by design, all queries operate on the same nine tables, and therefore there often exist data page sharing opportunities. For a single client, all systems perform almost the same since the disk bandwidth is the limiting factor. As clients increase beyond six, X is not able to significantly increase the throughput while QPipe with sharing enabled takes full advantage of overlapping work and achieves a 2x speedup over X. A more extensive evaluation of QPipe is presented elsewhere [8].

4 Conclusions

As DBMS performance relies increasingly on good memory hierarchy utilization and future multi-core processors will only put more pressure on the memory subsystem, it becomes increasingly important to design software that matches the architecture of modern hardware. This article shows that the key to optimal performance is to expose all potential locality in instructions, data, and high-level computation, by grouping similar concurrent tasks together. We presented two systems based on the StagedDB design. STEPS minimizes instruction cache misses of OLTP with arbitrary long code paths, without increasing the size of the instruction cache. QPipe, a novel query execution engine architecture, leverages the fact that a query plan offers an exact description of all task items needed by a query, and employs techniques to expose and exploit locality in both data and computation across different queries. Moreover, by providing fine-grained software parallelism, the StagedDB design is a unique match to future highly-parallel infrastructures.

References

[1] A. Ailamaki, D. J. DeWitt, et al. DBMSs on a modern processor: Where does time go? In *VLDB*, 1999.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *ISCA*, 1998.

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

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

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

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

[7] S. Harizopoulos and A. Ailamaki. STEPS Towards Cache-Resident Transaction Processing. In *VLDB*, 2004.

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

[9] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan-Kaufmann, 1996.

[10] K. Keeton, D. A. Patterson, et al. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*, 1998.

[11] J. Lo, L. A. Barroso, S. Eggers, et al. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *ISCA*, 1998.

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

[13] A. Ramirez, L. A. Barroso, et al. Code Layout Optimizations for Transaction Processing Workloads. In *ISCA*, 2001.

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

[15] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling Memory Page Layout from Storage Organization. In *VLDB*, 2004.

[16] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*, 1994.

MonetDB/X100 - A DBMS In The CPU Cache

Marcin Zukowski, Peter Boncz, Niels Nes, Sándor Héman
Centrum voor Wiskunde en Informatica
Kruislaan 413, Amsterdam, The Netherlands
{M.Zukowski, P.Boncz, N.Nes, S.Heman}@cwi.nl

Abstract

X100 is a new execution engine for the MonetDB system, that improves execution speed and overcomes its main memory limitation. It introduces the concept of in-cache vectorized processing *that strikes a balance between the existing column-at-a-time MIL execution primitives of MonetDB and the tuple-at-a-time Volcano pipelining model, avoiding their drawbacks: intermediate result materialization and large interpretation overhead, respectively. We explain how the new query engine makes better use of cache memories as well as parallel computation resources of modern super-scalar CPUs. MonetDB/X100 can be one to two orders of magnitude faster than commercial DBMSs and close to hand-coded C programs for computationally intensive queries on in-memory datasets. To address larger disk-based datasets with the same efficiency, a new ColumnBM storage layer is developed that boosts bandwidth using* ultra lightweight compression *and* cooperative scans.

1 Introduction

The computational power of database systems is known to be lower than hand-written (e.g. C++) programs. However, the actual performance difference can be surprisingly large. Table 1 shows the execution time of Query 1 of the TPC-H benchmark run on various database systems and implemented as a separate program (with data cached in the main memory in all situations). We choose Query 1 as it is a simple single-scan query (no joins), that calculates a small aggregated result, thus mainly exposing the raw processing power of the system. The results show that most DBMSs are orders of magnitude slower than hand-written code.

DBMS "X"	MySQL 4.1	MonetDB/MIL	**MonetDB/X100**	hand-coded
28.1s	26.6s	3.7s	**0.60s**	0.22s

Table 1: TPC-H Query 1 performance on various systems (scale factor 1)

We argue that the poor performance of MySQL and the commercial RDBMS "X" is related to the Volcano iterator pipelining model [6]. While the Volcano approach is elegant and efficient for I/O bound queries, in computation-intensive tasks, its tuple-at-a-time execution makes runtime interpretation overhead dominate execution. That is, the time spent by the RDBMS on actual computations is dwarfed by the time spent interpreting

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

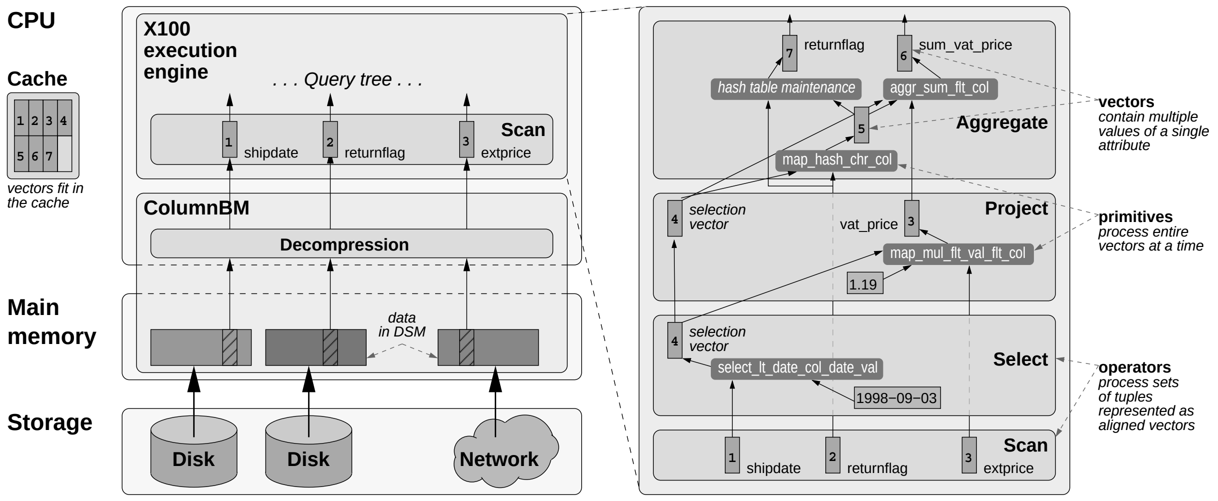


Figure 1: X100 – architecture overview and execution layer example

the expressions in a query tree, and looking up the needed attribute values in pages storing tuples in the N-ary storage model (NSM). As a result, quite a high number of CPU instructions are needed for each simple computational operation occurring in a query. Additionally, the instructions per cycle (IPC) efficiency achieved tends to be low, because the tuple-at-a-time model hides from the CPU most parallel execution opportunities, namely those provided by performing computations for *different* tuples in an overlapping fashion.

MonetDB [2], developed at CWI [1], is an alternative approach to building a query execution system. Its MIL algebra [3] minimizes the interpretation overhead by providing execution primitives that process data in a column-at-a-time fashion. Since MonetDB uses vertically decomposed storage model (DSM) [5], where columns are simple arrays, and each primitive performs only one fixed operation on the input data, the primitive implementation boils down to a sequential loop over aligned input arrays. Modern compilers can apply *loop pipelining* to such code, allowing MIL primitives to achieve high IPC on super-scalar CPUs. However, the column-at-a-time execution model introduces the extra cost of intermediate result materialization, which is only sustainable inside main memory, thus limiting the scalability of the system.

Figure 1 presents the architecture of X100, a new execution engine for MonetDB that combines the benefits of low-overhead column-wise query execution with the absence of intermediate result materialization in the Volcano iterator model. One of its distinguishing features is *vectorized execution*: instead of single tuples, entire *vectors* of values are passed through the pipeline. Another is *in-cache processing*: all the execution is happening within the CPU cache, using main memory only as a buffer for I/O and large intermediate data structures. These techniques make X100 execute efficiently on modern CPUs and allow it to achieve raw performance comparable to C code.

To bring the high-performance query processing of X100 to large disk-based datasets, a new *ColumnBM* storage manager is currently being developed. It focuses on satisfying the high bandwidth requirements of X100 by applying *ultra lightweight compression* for boosting bandwidth and *cooperative scans* for optimizing multi-query scenarios. Preliminary results show that it often achieves performance close to main memory execution without the need for excessive numbers of disks.

The outline of this article is as follows. In Section 2 we describe the hardware features important for the design of X100. Then, the system architecture is presented, concentrating on query execution in Section 3 and on the ColumnBM storage manager in Section 4. Finally we conclude and discuss future work.

[1]MonetDB is now in open-source, see `monetdb.cwi.nl`

2 Modern CPU features

In the last decade, a CPU clock-speed race resulted in dividing execution of a single instruction into an instruction pipeline of simple stages. The length of this pipeline continuously increases, going as far as 17 stages in AMD Opteron and 31 stages in Intel Pentium4. Another innovation found in modern super-scalar CPUs is the increase in number of processing (pipeline) units. For example, the AMD Opteron has 2 load/store units, 3 integer units and 3 floating point units, and allows for executing a mix of 3 instructions per cycle (IPC).

To fully exploit the pipelines, the CPU needs to know which instructions will be executed next. In the case of conditional branches, the CPU usually guesses which path the program will follow using a branch prediction mechanism and performs speculative execution of the chosen code. However, if a branch misprediction occurs, the pipeline needs to be flushed and the processing restarts with an alternative route. Obviously, with longer pipelines more instructions are flushed, increasing the performance penalty.

Another feature required to fully exploit available execution units is instruction independence. For example, when calculating `a=b+c;d=b*c;` both `a` and `d` can be calculated independently using two units. However, for `a=b+c;d=a*c;` processing of `d` has to be delayed until `a` is evaluated, resulting in just one processing unit being used. Figure 2 shows how *loop pipelining* eliminates this problem by interleaving the computation of multiple loop iterations. That is, `x[i]` is scheduled 8 instructions after the computation of `t0` started, making the result of `t0` immediately available for use. A similar effect can be achieved without compiler by a CPU that implements out-of-order execution, allowing it to execute instructions further up the stream (i.e. belonging to next loop iterations) while earlier instructions still wait on the result of another. Crucially, these techniques only work when a loop processes a large number of independent iterations. This property of the execution system, already present in MonetDB/MIL where one operation is executed on an entire column in a tight loop, is further improved in X100.

1) original code

```
for(i=0;i<n;i++) {
  t=a[i]+b[i];
  x[i]=t*c[i];
}
```

2) loop pipelining

```
for(i=0;i<n;i+=8) {
  t0=a[i]+b[i];
  t1=a[i+1]+b[i+1];
  ...
  t7=a[i+7]+b[i+7];
  x[i]=t0*c[i];
  x[i+1]=t1*c[i+1];
  ...
  x[i+7]=t7*c[i+7];
}
```

Figure 2: Loop Pipelining

Among the features of modern CPUs, cache memories received most attention from the DBMS community. A growing family of cache-conscious algorithms reduce access to main memory by tuning the access pattern of query processing algorithms and data structures [11, 1, 9, 8]. X100 goes even further by limiting its entire execution engine to the cache, as presented in the next section.

3 X100 architecture

The X100 engine is designed for *in-cache execution*, which means that the only "randomly" accessible memory is the CPU cache, and main memory is already considered part of secondary storage, used in the buffer manager for buffering I/O operations and large intermediate results. The other main principle behind X100 is *vectorized execution*, sketched in Figure 1:

- vertically decomposed tables are further partitioned horizontally into small chunks called *vectors*.
- a set of aligned vectors (one for each attribute), representing a set of tuples, is a single data unit in the execution pipeline.
- an optional *selection vector* contains the positions of the tuples currently taking part in processing. This removes the need for the extra after-selection projection steps present in MonetDB/MIL.
- the control logic of the operators is common for all data types, arithmetic functions, predicates etc.
- the actual data processing in the operators is performed by a set of *execution primitives* - simple, specialized and CPU-efficient functions.

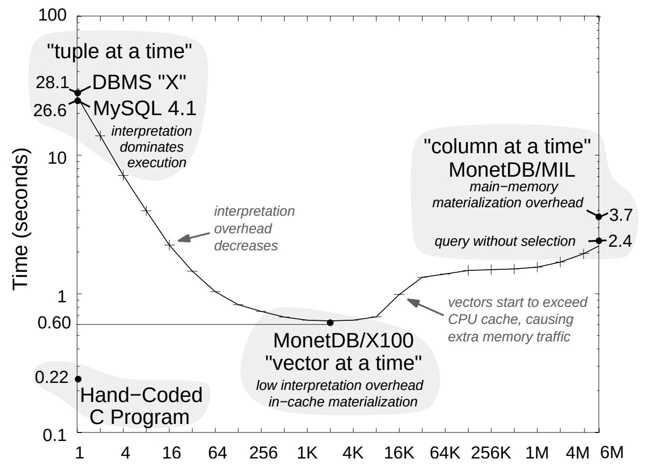


Figure 3: TPC-H Query 1 performance in X100 with varying vector size (in tuples, horizontal axis)

input count	time (us)	avg. cycles	**X100 primitive**
6M	13307	3.0	`select_lt_usht_col_usht_val`
5.9M	10039	2.3	`map_sub_flt_val_flt_col`
5.9M	9385	2.2	`map_mul_flt_col_flt_col`
5.9M	9248	2.1	**`map_mul_flt_col_flt_col`**
5.9M	10254	2.4	`map_add_flt_val_flt_col`
5.9M	13052	3.0	`map_uidx_uchr_col`
5.9M	14712	3.4	`map_directgrp_uidx_col_uchr_col`
5.9M	28058	6.5	`aggr_sum_flt_col_uidx_col`
5.9M	28598	6.6	`aggr_sum_flt_col_uidx_col`
5.9M	27243	6.3	`aggr_sum_flt_col_uidx_col`
5.9M	26603	6.1	`aggr_sum_flt_col_uidx_col`
5.9M	27404	6.3	`aggr_sum_flt_col_uidx_col`
5.9M	18738	4.3	`aggr_count_uidx_col`

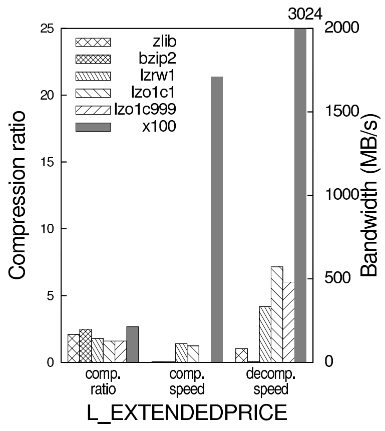
Table 2: MonetDB/X100 performance trace during TPC-H Query 1 (primitives)

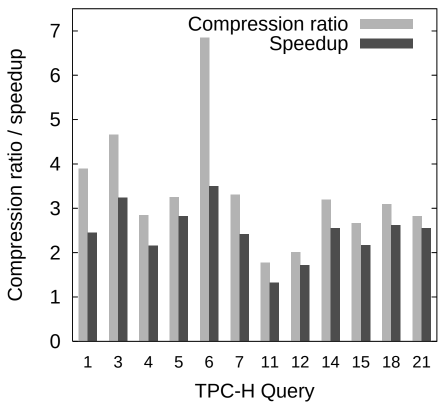
The execution primitive for multiplication of two vectors of floating point numbers might look like this:

```
int map_mul_flt_col_flt_col(int n, flt* res, flt* col1, flt* col2, int *sel)
{
  for(int i=0; i<n; i++)
      res[sel[i]] = col1[sel[i]] * col2[sel[i]];
  return n;
}
```

The simplicity of these primitives allows compilers to produce code that achieve IPC of over 2 and, as Table 2 shows, spend only a few cycles per tuple. In the case of a multiplication of two values, X100 spends 2 cycles per tuple, whereas MySQL spends 49 [4]. Complex expressions must be executed in X100 by calling multiple primitives that each materialize intermediate results. While this materialization is in-cache, hence highly efficient, it still causes a high ratio of load-store instructions. For example, the multiplication primitive reads from `sel`, `col1` and `col2`, and writes in `res`, thus needing 4 load/stores for 1 ”work” instruction. A hand-coded C program does not need these load/stores as subexpression results are passed through CPU registers. To overcome this problem, primitives in X100 are *generated* from a so-called signature request and a code pattern. This allows X100 to generate *compound primitives* that execute an entire expression subtree (e.g. `(a*1.19+b)*0.95`). Currently, primitive generation is predefined, but the ultimate goal is to allow a query optimizer to generate and compile compound primitives in a just-in-time fashion.

A key tuning factor in X100 is the size of the vectors, which can be chosen at run-time. Figure 3 shows the performance of MonetDB/X100 on TPC-H Query 1 with vector sizes varying from 1 (tuple-at-a-time) to 6M (the entire table: column-at-a-time). For better understanding, we also plotted the results of MySQL, DBMS ”X”, MonetDB/MIL and the hand-coded C program from Table 1. The performance of X100 at vector size 1 is highly similar to that of the relational DBMS systems. As the vector size is increased, performance of MonetDB/X100 improves by two orders of magnitude, clearly showing that interpretation overhead was the bottleneck. With further increase, however, the cache size boundary is crossed and cache misses start to occur. In other words, the materialized intermediate results become too big, making the query memory bandwidth bound. At size 6M, we indeed observe the performance of MonetDB/X100 to be suboptimal and highly similar to MonetDB/MIL. MonetDB/MIL is slower mainly due to materialization the of 99% selection of Query 1, which is avoided in X100 thanks to selection vectors. With this selection omitted, the performance of X100 exactly coincides with MIL.





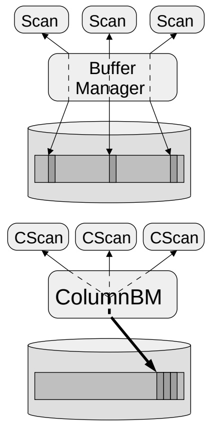


Figure 4: Performance of various compression algorithms on example TPC-H data

Figure 5: Compression ratio and performance improvement on a subset of TPC-H queries

Figure 6: Scan processing in a traditional system and in the ColumnBM

4 ColumnBM

While the X100 execution engine is efficient in main memory scenarios, achieving similar performance for disk-based data is a real challenge. Due to its raw computational speed, MonetDB/X100 exhibits an extreme hunger for I/O bandwidth. As an extreme example, TPC-H Query 6 uses 216MB of data (SF=1), and is processed in MonetDB/X100 in less than 100 ms, resulting in a bandwidth requirement of ca. 2.5GB per second. For most other queries this requirement is lower, but still in the range of hundreds of megabytes per second. Clearly, such bandwidth is hard to achieve except by using expensive storage systems consisting of large numbers of disks. ColumnBM combines three techniques (DSM, compression and cooperative scans) to combat this problem.

DSM. ColumnBM vertically fragments tables on disk using DSM, which saves bandwidth if queries scan a table without using all columns. Its main disadvantage is an increased cost of updates: a single row modification results in one I/O per each influenced column. ColumnBM tackles this problem by treating data chunks as immutable objects and storing modifications in the (in-memory) delta structures, periodically updating the chunks [4]. During the scan, data from disk and delta structures are merged, providing the execution layer with a consistent state. While ColumnBM uses DSM, this is not strictly required by the X100 execution model. The rationale behind the in-cache column-wise layout (i.e. vectors) in X100 is not optimizing memory storage or reducing I/O bandwidth, but to allow a compiler to detect loop-pipelining opportunities in its execution primitives. To store data with a high-update rate, ColumnBM will also support the PAX [1] storage scheme, which stores entire tuples in disk blocks, but uses vectors to represent the columns inside such blocks.

Compression. While compression in databases was proposed by many researchers [7, 10], we introduce two novel techniques: *ultra lightweight compression* and *into-cache decompression*. Traditional compression algorithms usually try to maximize the compression ratio, making them too expensive for use in a DBMS. X100 introduces a family of new highly CPU-efficient compression algorithms, specifically designed to create a (de)compression kernel, that compiles into a pipelinable loop by making it simple, predictable and eliminating all if-then-else constructs. As Figure 4 shows, these new algorithms achieve a throughput of over 1 GB/s during compression and a few GB/s in decompression, beating speed-tuned general purpose algorithms like LZRW and LZO, while still obtaining comparable compression ratios. Preliminary experiments presented in Figure 5 show the speedup of the decompressed queries to be close to the compression ratio, which in case of TPC-H allows for a bandwidth (and performance) increase of a factor 3.

Most database systems employ decompression right after reading data from the disk, storing buffer pages in uncompressed form. This solution requires data to cross the memory-cache boundary three times: when it is delivered to the CPU for decompression, when uncompressed data is stored back in the buffer, and finally when it is used by the query. Since such approach would make X100 decompression routines memory-bound, ColumnBM stores disk pages in a compressed form and decompresses them just before execution, on a per-vector granularity. Thus (de)compression is performed on the boundary between CPU cache and main memory, rather than between main memory and disk. This approach nicely fits the delta-based update mechanism, as merging the deltas can be applied after decompression, and chunks need to be re-compressed only periodically.

Cooperative Scans. While compression is tuned to improve performance of isolated queries, it is often the case that *multiple* queries are running at the same time, often fighting for disk bandwidth. If they read the same columns, ColumnBM applies *cooperative scans* [12], a technique in which queries, instead of reading data in a fixed sequential order, try to reuse data already buffered by other queries. Disk accesses are scheduled to satisfy the largest possible number of ”starving” queries. Preliminary results show that X100 on a single CPU can sustain a load of 30 I/O bound queries (TPC-H Query 6) without performance loss.

5 Conclusions and future work

In this article we presented X100, a novel execution engine for MonetDB. By applying vectorized in-cache execution, it efficiently exploits the features of modern CPUs and allows for raw in-memory performance one to two orders of magnitude higher than other systems. Additionally, we discussed ColumnBM, a new storage system that enables scaling MonetDB to large disk-based datasets.

X100 and ColumnBM are still in an experimental stage. Both components will play a role in our future research into architecture-aware query processing, e.g. looking at new CPU features such as SMT and CMT.

From a software point of view, X100 is part of the MonetDB software family so it will re-use its API infrastructure and query front-ends (SQL and XQuery). X100 is already being used in multimedia information retrieval on the TREC-VIDEO collections. Other applications on our agenda for X100 are data mining and analysis of astronomy datasets.

References

[1] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, 2001.

[2] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[3] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB J.*, 8(2):101–119, 1999.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, 2005.

[5] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, 1985.

[6] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[7] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, 1991.

[8] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE TKDE*, 14(4):709–730, 2002.

[9] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proc. SIGMOD*, 2000.

[10] M. Roth and S. van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, September 1993.

[11] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proc. VLDB*, 1994.

[12] M. Zukowski, P. A. Boncz, and M. L. Kersten. Cooperative scans. Technical Report INS-E0411, CWI, December 2004.

New Hardware Support for Database Operations [*]

Nagender Bandi†, Chengyu Sun‡, Hailing Yu◇, Divyakant Agrawal†, Amr El Abbadi†
† Department of Computer Science, University of California, Santa Barbara
‡ Department of Computer Science, California State Universtity, Los Angeles
◇ Oracle Corporation, Redwood Shores, California

Abstract

Query processing cost in database applications can be divided into two parts: the I/O cost *and the* computational cost*. Traditionally DBMS researchers have focused on the issue of reducing the* I/O cost *as it is the primary source of bottleneck in most of the database operations. As the computer systems which run the DBMSs evolved, this problem worsened because the speeds of secondary storage devices increased at a much slower pace compared to the memory access and processing speeds. Further, as databases become increasingly accepted in areas such as spatial databases, we encounter new challenges. For example, the* computational cost *has been known to be the bottleneck in certain spatial operations. In this paper, we present a survey of our work where we present novel solutions which use hardware to alleviate the* I/O cost *bottleneck for traditional database applications and the* computational cost *bottleneck for non-standard database applications such as spatial databases. In particular, we use the novel MEMS-based storage devices to develop efficient data placement solutions for relational databases and two-dimensional datasets. Using commodity graphics hardware, we develop novel spatial solutions which outperform conventional software techniques. We also integrate these techniques into a popular commercial spatial database and demonstrate significant improvement in query performance.*

1 Introduction

The cost of a DBMS query consists of two factors: *I/O cost*, the time spent in loading the data from the secondary storage into the main memory, and *computational cost*, the time spent by the DBMS in processing this data and returning the result. Traditionally *I/O cost* has been known to be the major bottleneck due to the high cost of disk access and algorithms were developed to reduce the I/O cost [10]. Although the past two decades witnessed tremendous improvement in the processing and memory speeds, the disk access speeds have not substantially changed thus worsening the I/O cost bottleneck. Furthermore, increasing acceptance of DBMSs in non-traditional areas, such as spatial databases, presents new interesting challenges. For example, *computation cost* accounts for most of the query cost for certain spatial operations [12]. Although many solutions have been proposed over the last decade to alleviate this *computation cost*, the problem still persists. Most of these solutions have been software based, however, recently, more interest has been directed to using the impressive

[*]This research has been supported by the NSF grant IIS-02-20152.

advances in hardware to reduce both the I/O and the computational costs. We believe that novel architectures and hardware devices promise to provide efficient solutions to some of the significant bottlenecks encountered by DBMSs. In this paper, we present a survey of our work which provide efficient hardware-based solutions that alleviate the *I/O cost* in the case of relational databases and *computation cost* in spatial applications.

In Section 2, we present efficient data placement and retrieval solutions for relational databases and two-dimensional datasets. Our research is based on the novel MEMS-based storage devices [1, 18, 2] being developed towards replacing magnetic disks as secondary storage media. We exploit the inherent two-dimensional layout of the MEMS-based storage to develop a data placement scheme for relational databases which caters for both OLAP and OLTP applications in most efficient way [19]. We also apply the high bandwidth parallel transfer provided by thousands of synchronous tips of a MEMS-device to develop mappings for two-dimensional datasets which enable highly optimal retrieval for window queries [20].

In Section 3, we discuss novel spatial database techniques [17, 6] where we exploit the computation power of off-the-shelf graphics hardware towards achieving significant performance improvement over most sophisticated techniques. We also integrate our techniques into a popular commercial database and show that significant reduction in the computation cost can be achieved through this integration.

2 MEMS-based Storage

In this section, we describe the basic features of MEMS and how we use MEMS to develop efficient data placement algorithms for relational databases and two-dimensional datasets. MEMS are extremely small mechanical systems formed by the integration of mechanical elements, actuators, electronics, and sensors. They are fabricated on silicon chips using photo-lithographic processes similar to those employed in manufacturing standard semiconductor devices. As a result, MEMS-based storage can be manufactured at a very low cost. They represent a compromise between slow traditional disks and expensive storage based on EEPROM technologies. Due to the great potential for applications MEMS present, several major research labs such as IBM [18] and HP [2], and universities such as Carnegie Mellon University [1] have been involved in their development. MEMS devices from these research groups differ largely in the types of actuators used to position the media and the media used to record data in the medium. Figure 1 shows a high level view of a MEMS-based storage device from CMU CHIPS project [15]. The media sled is organized into rectangular regions at the lower level. Each of these rectangular regions contains over millions of bits and is accessible by one tip. Because of heat-dissipation constraints, in the CMU CHIPS model, not all tips can be activated simultaneously to access data even though it is theoretically feasible.

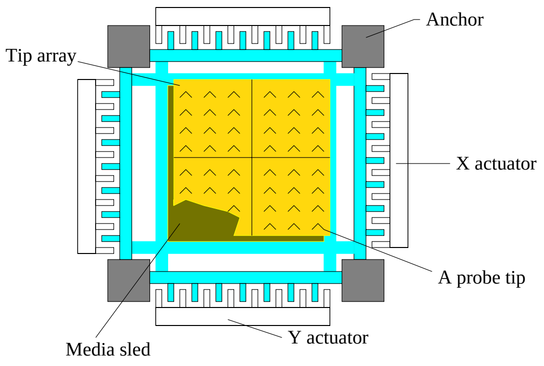


Figure 1: The architecture of CMU CHIPS

MEMS-based storage has several advantages over traditional magnetic disk technology. Since MEMS do not contain large mechanical components, the access latency is significantly smaller. Furthermore, since MEMS are based on conventional silicon-based technology, they can leverage from the large-scale integration resulting in much smaller form factor and economies of scale. Another advantage of MEMS over disks is that the power consumption is much smaller and they do not incur long latencies when powered-up or powered-down. Recent efforts in computer science research have focused on taking advantage of MEMS-based storage devices from different perspectives. One such aspect is how to integrate MEMS-based storage into current systems and to develop novel solutions using these devices. In our studies, we exploit the inherent characteristics of MEMS-based storage devices and design novel data layout designs for different applications [19, 20]. In particular, we exploit the property of MEMS-based storage devices that they are inherently two-dimensional and can provide high-bandwidth concurrent data access through thousands of concurrent tips. In the following subsections, we describe our data placement schemes for relational databases and two-dimensional datasets.

2.1 MEMS-based Storage Architecture for Relational Databases

Databases have always suffered from the increasing gap between economically viable non-volatile hard disks and expensive but fast transistor-based in-memory chips. One of the main challenges in database research is to find efficient methods to support operations that have conflicting needs in terms of storage and page layouts. In particular, due to the update characteristics of OLTP transactions, relations need to be accessed in a row-wise manner. In contrast, for OLAP queries where only a subset of attributes are of interest, the I/O system should facilitate data retrieval in a column-wise manner. Hence OLTP workloads perform better with row-wise placement of relations while OLAP workloads perform better with column-wise allocation. However the traditional data layouts over disk devices for relational data neither favor the OLTP applications nor the OLAP applications. Since a disk is modeled as an one-dimensional array, achieving row-wise access and column-wise access are conflicting goals. Recent proposals have been made to address this problem at both the cache and disk levels. Ramamurthy et al. [14] propose multiple representations of the same relation on disks to support such conflicting requirements. Even though multiple representations of the same relation alleviates this problem, two copies of each relation have to be stored. In order to address cache performance, Ailamaki et al. [4] propose PAX that reorganizes the data in each page so as to reduce cache pollution. The essence of PAX is not to pollute cache with unnecessary data, but no I/O time is saved.

Based on these observations, we take advantage of the inherent properties of MEMS-based storage and propose to replace disks in relational database systems. In [19], we propose a new strategy for placing relational data on MEMS-based storage devices [19]. The new data placement scheme exploits the two-dimensional nature of MEMS-based storage devices to support efficient query processing for different types of workloads. In other words, the new scheme supports both row-wise and column-wise access to relations, hence the unnecessary data will not be brought into main memory and a lot of I/O time is saved. We also exploit the parallel retrieval property of MEMS devices to efficiently reduce cache pollution. As a result, our data placement strategy for placing relations on MEMS-based storage improves I/O time as well as reduces the processor and main memory gap. Our theoretical analysis and experimental results show that the new strategy can result in significant improvements for relational DBMSs. The TPC-H queries can be improved up to 5 times compared to disk devices. We also develop query processing algorithms for different types of queries and address the issues of indexes, because MEMS-based storage devices have a different interface from disks in our design.

2.2 Slicing and Dicing Two-dimensional Datasets

Multi-dimensional datasets have received a lot of attention due to their wide applications, such as relational databases, images, GIS, and spatial databases. Range queries are an important class of queries for multi-dimensional data applications. In recent years, the size of datasets has been rapidly growing, hence, fast retrieval of relevant data is the key to improve query performance. To alleviate this problem, two-dimensional datasets are often uniformly divided into tiles. The tiles are then declustered over multiple disks to improve query performance through parallel I/O. In [3], it is shown that strictly optimal solutions for range queries exist in only a small number of cases. However, all prior research, including the optimal cost formulation itself, is based on the assumption that the retrieval of each tile takes constant time and is retrieved independently (random I/O involving both seek and transfer time). This condition does not hold when considering current disk technology. Recent trends in hard disk development favor reading large chunks of consecutive disk blocks (sequential access). Thus, the solutions for allocating two-dimensional data across multiple disk devices should also account for this factor.

In [20], we define the new optimal cost and prove that it is impossible to achieve the new optimal cost for the general case, since the new optimal cost of range queries requires two-dimensional sequential access. This is not feasible in the context of disks, since disks are single-dimensional devices. In order to improve the performance of range queries over two-dimensional datasets, we explore data placement schemes for two-dimensional data over MEMS-based storage to facilitate efficient processing of this type of queries. We develop a new data placement scheme for MEMS-based storage that takes advantage of its physical properties, where the two-dimensional sequential access requirement is satisfied. We utilize the Y-axis sequential access of MEMS-based storage devices to achieve the Y dimensional sequential access over a dataset and take advantage of intra-device parallelism to achieve the X dimensional sequential access. The new scheme allows for the reorganization of query ranges into new tiles which contain more relevant data, thus maximizing the number of concurrent active tips to access the relevant data. In our new scheme, we tile two-dimensional datasets according to the physical properties of I/O devices. In all previous works, the dataset is divided into tiles along each dimension uniformly. Through the performance study, we show that tiling the dataset based on the properties of devices can significantly improve the performance without any adverse impact on users, since users only care about query performance, and do not need to know how or where data is stored. Our theoretical analysis and experimental results show that our new scheme can achieve almost optimal performance.

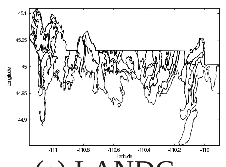
3 Hardware Acceleration for Spatial Database Operations

In this section, we address the problem of alleviating the computational cost bottleneck for spatial database applications. We observe that spatial database operations lend themselves well to hardware acceleration. Specifically, both spatial databases and graphics hardware abstract real world objects as geometries such as points, polylines, and polygons, and process them based on their topological properties and relationships. In our recent work [17, 6], we developed a hardware intermediate filter for spatial database primitives, and showed that this technique can be easily integrated into commercial databases without changes to existing storage and index structures.

3.1 Spatial Query Evaluation

Spatial database queries are typically evaluated in two steps: the *filtering step* and the *refinement step*. In the filtering step, the minimal bounding rectangles (MBRs) of the objects and spatial indexes such as R-trees [11] are used to quickly determine a set of candidate results. In the refinement step, the final results are determined by retrieving the actual geometries of the candidates from the database, and comparing them to either a query geometry or to each other. For complex geometries such as polygons, the cost of the refinement step usually dominates the query cost due to the complexity of the underlying computational geometry algorithms.

The cost of the refinement step consists of two factors: the *I/O cost* of loading the geometries from disk to main memory, and the *computational cost* of geometry-geometry comparison. The ratio of the computational cost over the I/O cost varies significantly depending on the types of spatial queries and the complexity of the geometries, which can be roughly characterized by the number of vertexes of a geometry. Generally speaking, the more complex the data, the more significant the computational cost. For instance, a recent study on spatial selections [12] shows that for point geometries, the I/O cost is the dominant factor, but for polygon geometries, both costs are significant. In the case of a spatial join, the computational cost could be orders of magnitude higher than the I/O cost, because once a geometry is loaded, it is buffered in the main memory and compared to many other geometries.


(a) LANDC

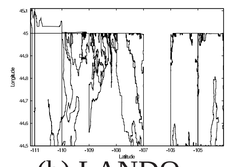

(b) LANDO

Figure 2: Sample Objects from Two Dataset

The high computational cost of the spatial operations comes from the complexity of the data in the real world, where it is not uncommon that a polygon has tens of thousands of vertexes. Furthermore, the shapes of the polygons can be arbitrarily complex, as can be seen from Figure 2, which shows the first 100 polygons in the Wyoming land cover dataset (LANDC) and the Wyoming land ownership dataset (LANDO). In many cases, the polygons are concave, and sometimes, even non-simple[1]. Processing these types of polygons is very expensive. For instance, assuming the numbers of vertexes of two polygons are n and m, the complexity of the commonly used intersection test algorithm is $(O((n+m)\log(n+m))$ [9], and the distance calculation algorithm is even more expensive with $O(n \times m)$ worst case complexity [8]. Since the I/O cost remains linear, the computational cost quickly outweighs the I/O cost as the complexity of data increases.

Over the last decade, many techniques have been proposed to improve spatial query processing, and most of them rely on an *intermediate filtering* step [7, 21, 5, 12, 8], where false or positive hits can be identified without performing the costly geometry-geometry comparison. These techniques have been shown to be very effective for improving query performance, but suffer from two drawbacks. First, the computation required to pre-process the data is usually expensive, which degrades update performance, and cannot be applied to cases where one of the datasets is an intermediate dataset, e.g. the results of a map overlay operation. Second, the pre-computed polygon approximations or auxiliary data structures have to be stored on disk. The extra storage and I/O costs may not be significant, but it still requires changes of existing storage or index structures, therefore it is more difficult to incorporate these techniques into commercial DBMSs [13].

3.2 Hardware-based Intermediate Filtering

Our technique is based on the observation that most low-level algorithms spatial databases rely on have been well studied by the computational geometry community. Under current computer architectures, it is unlikely that algorithmic advances will significantly reduce the cost of geometry-geometry comparison. On the other hand, the last few years have seen tremendous advances in graphics hardware technologies. Off-the-shelf graphics cards are now capable of handling thousands of polygons in real time, and are widely used in computer games, 3D modeling, and virtual reality applications. Since both graphics hardware and spatial databases work on geometries such as points, lines, and polygons, and they both deal with geometric relations such as intersection and containment in a 2D or 3D space, it is only natural to exploit the computational power of graphics hardware to speed up spatial database operations.

The general strategy to test for intersection of two polygons with graphics hardware is quite straight-forward:

- Render the first polygon with color c_1
- Render the second polygon with color c_2

[1]Non-simple polygons are polygons with self-intersecting edges or with vertexes that have degrees greater than 2 (more than 2 edges incident to a vertex).

- Search the frame buffer for overlapping pixels with color $c_1 + c_2$. If such pixels exist, these two polygons intersect each other.

An example of this strategy is illustrated in Figure 3(a), where the two polygons are rendered with color *gray*, and the overlapping pixels are *black*.

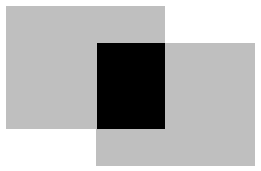


Figure 3: Hardware Intersection Test

It turns out that implementing this strategy is quite difficult. First of all, in order to achieve maximum performance for the most common tasks, graphics hardware is designed to support only a limited set of geometries, which include points, line segments, and convex polygons. Concave polygons, which are common in spatial datasets, must be triangulated before they can be rendered by hardware. Polygon triangulation, which has to be performed in software, is much more expensive than hardware operations. Secondly, the coordinates of publicly available GIS data typically have 4 to 6 digits accuracy, or about 11 to 18 bit accuracy in binary form. On the other hand, the resolution of a rendering window is rather limited. For instance, a 32×32 rendering window only provides 5 bit accuracy in each dimension. This disparity between the data resolution and the window resolution means that the hardware approach may produce *false* results, which are usually not acceptable for spatial database applications.

Taking these issues into account, we developed a polygon intersection test algorithm [17] that uses hardware segment intersection test as an additional filtering step. This algorithm has several properties:

- The algorithm only renders the boundary of a polygon, thereby is highly efficient regardless of the convexity of the polygon.
- The algorithm guarantees no false results based on the anti-aliasing property of OpenGL [16],
- The algorithm can be easily extended to handle distance test.

3.3 Integration with Oracle Spatial

In order to evaluate the performance of the hardware-based spatial solutions against sophisticated software-based techniques, we integrated our techniques into Oracle, a popular commercial spatial database [6]. We developed a general framework for integrating novel hardware-based techniques such as ours into a commercial database. We integrate the hardware-accelerated spatial solutions as external procedures where the query processing which accesses the graphics hardware happens in a process external to the address-space of the DBMS. This integration enables a comparison of the run-time hardware-based techniques against both run-time and preprocessing-based software techniques provided by Oracle. It also presents how our technique seamlessly integrates with existing software-based indexes with out requiring any modifications as in the case of preprocessing-based solutions.

3.4 Performance Results

Experimental evaluation over real world datasets show that the hardware-accelerated spatial operations achieve significant performance improvements over sophisticated software-based algorithms. Comparison of hardware-based intersection solutions against run-time software solutions [6] using R-tree indexes show an average case improvement of 3-4 times for spatial selection queries. In the case of spatial joins, we achieve an improvement of 3-4 times for intersection joins and 5.9 times for with-in distance joins. We also compare the performance of the hardware-based solutions with preprocessing-based software solutions. Preprocessing-based solutions in Oracle utilize Quad-tree indexes and the preprocessing accuracy is determined by the Quad-tree tiling level. Preprocessing cost increases exponentially with the tiling level both in terms of index creation time and index storage cost where a typical index creation for a real world megabyte-sized dataset can take hours of creation cost and gigabytes of index space. Our evaluation shows that the hardware-based solutions not only outperform the preprocessing-based solutions at lower tiling levels but also performs as well if not better at higher tiling levels with out incurring any of the preprocessing overheads.

4 Conclusion

In this paper, we briefly highlighted some of our recent research exploiting modern hardware to improve the overall performance of DBMS operations. Software solutions are restricted in terms of how much improvements they can provide. Hardware, on the other hand, has been achieving incredible advances in recent years. In our MEMS work, we are exploiting a new and futuristic technology, with the goal of reducing the I/O cost of retrieving data. As material and mechanical engineers develop new and more efficient MEMS-based storage devices, we believe the role of the computer scientist, in general, and the database researcher, in particular, is to develop efficient ways to integrate such devices into computer systems. Our work represents an initial step in this direction for data placement, and has shown the great potential MEMS has for computer systems. On the other hand, our research with graphics hardware exploits a well established and widely available hardware device. In fact, due to its widespread availability on most modern day computers, we strongly believe that graphics cards should be exploited as much as possible to improve the performance of large software systems, and in particular databases. Our research has clearly demonstrated the successful integration of graphics hardware with commercial databases, resulting in substantial performance improvements. We are continuing, at UC Santa Barbara, to explore such trends, namely using both futuristic devices with high potential, as well as existing off the shelf hardware for new and novel applications.

References

[1] CMU CHIP project. 2002. http://www.lcs.ece.cmu.edu/research/MEMS.

[2] Hewlett-packard laboratories atomic resolution storage, 2003. http://www.hpl.hp.com/research/storage.html.

[3] K.A.S. Abdel-Ghaffar and A. El Abbadi. Optimal allocation of two-dimensional data. *In International Conference on Database Theory*, pages 408–418, January 1997.

[4] A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis. Weaving relations for cache performance. *In proceedings of the 27th Conference on Very Large Databases*, pages 169–180, September 2001.

[5] Wael M. Badawy and Walid G. Aref. On local heuristics to speed up polygon-polygon intersection tests. In *ACM-GIS '99, Proceedings of the 7th International Symposium on Advances in Geographic Information Systems*, pages 97–102, 1999.

[6] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. In *Proceedings of 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 1021–1032, 2004.

[7] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 197–208, 1994.

[8] Edward P.F. Chan. Evaluation of buffer queries in spatial databases. In *Proceedings of 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD 2001)*, pages 161–170, July 2001.

[9] Mark De Berg, Marc Van Kreveld, Mark Overmars, and O. Scharzkopf. *Computational Geometry: Algorithms and Applications*, chapter 2. Springer, 2000.

[10] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*, chapter 4. Prentice Hall, 1999.

[11] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, 1984.

[12] Ravi K. Kothuri and Siva Ravada. Efficient processing of large spatial queries using interior approximation. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD 2001)*, pages 404–421, 2001.

[13] Ravi K. Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and r-tree indexes in oracle spatial: A comparison using gis data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of DATA (SIGMOD'02)*, pages 546–557, 2002.

[14] R. Ramamurthy, D.J. DeWitt, and Q. Su. A case for fractured mirrors. *Proc. Int. Conf. on Very Large Data Bases*, pages 430–441, September 2002.

[15] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 1–12, 2000.

[16] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., April 1999.

[17] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration for spatial selections and joins. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 455–466. ACM Press, 2003.

[18] P. Vettider, M. Despont, U. Durig, W. Haberle, M.I. Lutwyche, H.E. Rothuizen, R. Stuz, R. Widmer, and G.K. Binnig. The "millipede"-more than one thousand tips for future afm storage. *IBM Journal of Research and Development*, 44(3):323–340, May 2000.

[19] H. Yu, D.Agrawal, and A. El Abbadi. Tabular placement of relational data on MEMS based storage devices. In *Proc. Int. Conf. on Very Large Data Bases*, pages 680–693, 2003.

[20] H. Yu, D.Agrawal, and A. El Abbadi. Declustering two-dimensional datasets over MEMS based storage. In *9th International Conference on Extending DataBase Technology*, pages 495–512, 2004.

[21] Geraldo Zimbrao and Jano Moreira de Souza. A raster approximation for processing of spatial joins. In *Proceedings of 24rd International Conference on Very Large Data Bases (VLDB'98)*, pages 558–569, 1998.

ICDE'06

Sponsored by the
IEEE Computer Society

CALL FOR PAPERS

22nd International Conference on Data Engineering

April 3 - April 7, 2006
JW Marriott Buckhead Hotel, Atlanta, Georgia, USA
http://icde06.cc.gatech.edu Mirror: http://icde06.ewi.utwente.nl

Data Engineering deals with the use of engineering techniques and methodologies in the design, development and assessment of information systems for different computing platforms and application environments. The 22nd International Conference on Data Engineering provides a premier forum for sharing and exchanging research and engineering results to problems encountered in today's information society.

We especially encourage submissions that make efforts on (1) Exposing practitioners how the research results and tools can contribution to their everyday problems in practice, and providing them with an early opportunity to evaluate these tools; (2) Raising awareness in the research community of the problems of practical applications of data engineering and promoting the exchange of data engineering technologies and experience among researchers and practitioners; (3) Identifying new issues and directions for future research and development in the data engineering field.

ICDE 2006 invites research submissions on all topics related to data engineering, including but not limited to those listed below:

1. Data Integration, Interoperability, and Metadata
2. Ubiquitous Data Management and Mobile Databases
3. Query processing, query optimization, and data structures
4. Data Privacy and Security
5. Data Mining
6. Semi-structured data and XML databases
7. Distributed, parallel, Peer to Peer databases
8. Web Data Management and Deep Web
9. Scientific and Biological Databases and Bioinformatics
10. Workflow, Web Services
11. Stream processing and sensor databases
12. Data Grids, Data Warehousing, OLAP
13. Temporal, Spatial, and Multimedia databases
14. Database Applications and Experiences
15. Database System Internals and Performance

AWARDS

An award will be given to the best paper. A separate award will be given to the best student paper. Papers eligible for this award must have a (graduate or undergraduate) student listed as the first and contact author, and the majority of the authors must be students. Such submissions must be marked as student papers at the time of submission.

INDUSTRIAL PROGRAM

The conference program will include a number of short papers and invited presentations devoted to industrial developments. Send your papers/proposals electronically, clearly marked as industrial track papers, to the Industrial Program Chair at <chris.bussler[AT]deri[.]org>.

PANELS

Panel proposals must include an abstract, an outline of the panel format, and relevant information about the proposed panelists. Send your proposals electronically by the submission deadline to the Panel Chair at <marek[AT]research[.]telcordia[.]com>.

ADVANCED TECHNOLOGY SEMINARS

Seminar proposals must include an abstract, an outline, a description of the target audience, duration (1.5 or 3 hours), and a short bio of the presenter(s). Send your proposals electronically to the Seminar Chair at <yzhang[AT]csm[.]vu[.]edu[.]au>.

DEMONSTRATIONS

Demonstration proposals should focus on new technology advances in applying databases or new techniques. Proposals must be no more than four double-columned pages, and should give a short description of the demonstrated system, explain what is going to be demonstrated, and state the significance of the contribution to database technology, applications or techniques. Send your proposals electronically to the Demonstration Chairs at <govi[AT]aztec[.]soft[.]net>, <leo.mark[AT]cc[.]gatech[.]edu>, and <arjen[.]de[.]vries[AT]cwi[.]nl>.

SUBMISSION INFORMATION

Research papers must be prepared in the 8/5"x11" IEEE camera-ready format, with a 12-page limit, and submitted electronically at http://icde06.cc.gatech.edu. All accepted papers will appear in the Proceedings published by the IEEE Computer Society.

IMPORTANT DATES

Abstract Deadline: June 14, 2005
Submission Deadline: June 21, 2005
Notification: September 15, 2005

GENERAL CHAIRS

Ramesh Jain, Univ. California, Irvine, USA
Calton Pu, Georgia Inst. of Technology, USA

PROGRAM CHAIRS

Ling Liu, Georgia Institute of Technology, USA
Andreas Reuter, European Media Lab.Germany
Kyu-Young Whang, KAIST, Korea

LOCAL ARRANGEMENTS

Brian Cooper, Georgia Inst. of Technology,USA
Daniel Rocco, West Georgia University, USA

PUBLICITY CHAIRS

Wei Tang, NCR Corp. USA
Andreas Wombacher, Univ Twente, The Netherlands

TREASURER

E. K. Park, Univ. of Missouri Kansas City, USA

INDUSTRIAL PROGRAM CHAIRS

Christoph Bussler, DERI, Ireland
Fabio Casati, HP, USA

PANEL CHAIR

Marek Rusinkiewicz, Telcordia, USA

SEMINAR CHAIR

Yanchun Zhang, Victoria Univ. Australia

WORKSHOP CHAIR

Roger Barga, Microsoft Research, USA
Xiaofang Zhou, Univ. of Queensland, AUS

DEMONSTRATION CHAIRS

Govi Govindarajan, Aztec, India
Leo Mark, Georgia Inst. of Technology, USA
Arjen de Vries, CWI, The Netherlands

AREA CHAIRS

Tamer Ozsu, Univ. of Waterloo, Canada
Ouri Wolfson, Univ. of Illinois at Chicago, USA
Alfons Kemper, Technische Univ. München, Germany
Elisa Bertino, Purdue University, USA
Johannes Gehrke, Cornell Univ. USA
Yannis Papakonstantinou, UC San Diego, USA
Beng Chin Ooi, National Univ. of Singapore
Paolo Atzeni, Università Roma Tre, Italy
Val Tannen, University of Pennsylvania, USA
Asuman Dogac, METU, Turkey
Donald Kossmann, ETH Zurich, Switzerland
Paul Watson, Univ. of Newcastle, UK
Ming-Syan Chen, National Taiwan Univ.
Masatoshi Yoshikawa, Nagoya Univ. Japan
Sang Kyun Cha, Seoul National Univ. Korea

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398