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.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

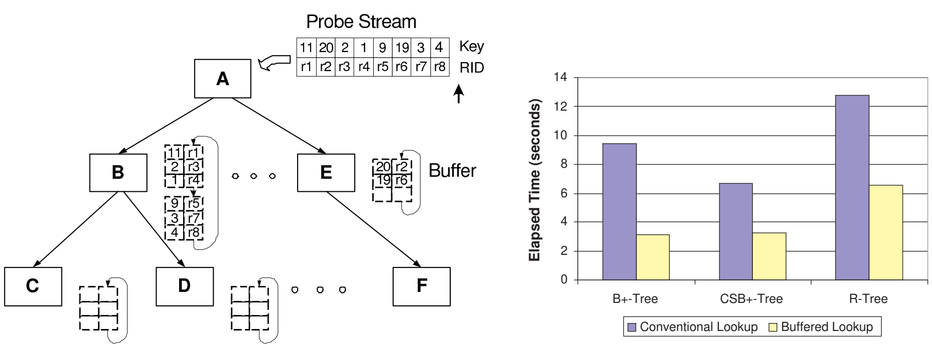
*Supported by NSF grant IIS-01-20939. The work described here was performed while the authors were at Columbia University.

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.

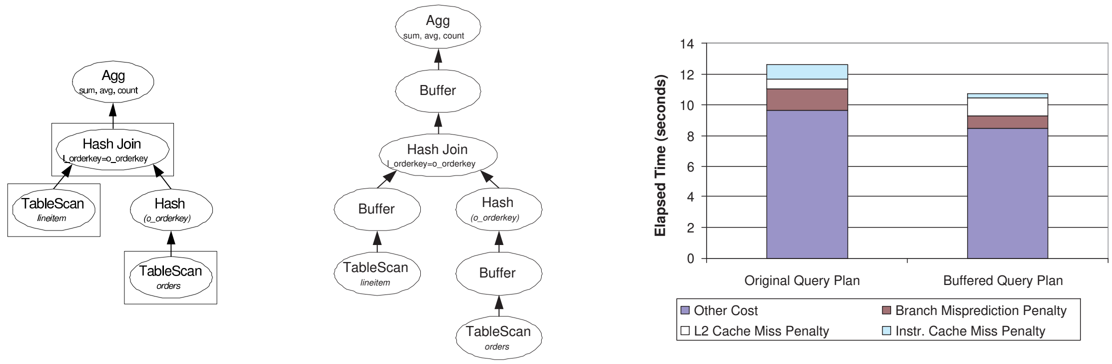


(a) Variable-Size Buffered Tree (b) Overall Performance

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;} }
```

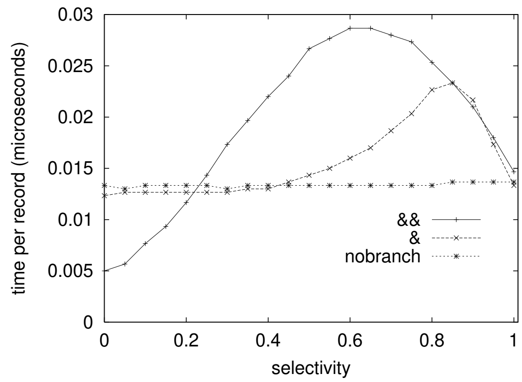
The important point is the use of the `C` idiom "`&&`". This implementation saves work when `f1` is very selective. When `f1(r1[i])` is zero, no further work (using `f2` through `fk`) is done for record `i`. However, the potential problem with this implementation is that its assembly language equivalent 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):

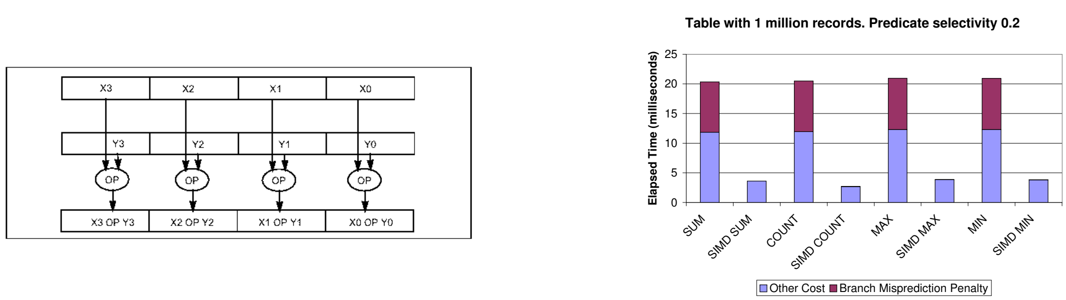



```
if((f1(r1[i]) & f2(r2[i])) && f3(r3[i]))
  { answer[j] = i;  j += (f4(r4[i]) & ... & fk(rk[i])); }
```

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.