Cache-Conscious Buffering for Database Operators with State

John Cieslewicz
Columbia University
johnc@cs.columbia.edu

William Mee
Columbia University
wjm2107@columbia.edu

Kenneth A. Ross[*]
Columbia University
kar@cs.columbia.edu

ABSTRACT

Database processes must be cache-efficient to effectively utilize modern hardware. In this paper, we analyze the importance of temporal locality and the resultant cache behavior in scheduling database operators for in-memory, block oriented query processing. We demonstrate how the overall performance of a workload of multiple database operators is strongly dependent on how they are interleaved with each other. Longer time slices combined with temporal locality within an operator amortize the effects of the initial compulsory cache misses needed to load the operator's state, such as a hash table, into the cache. Though running an operator to completion over all of its input results in the greatest amortization of cache misses, this is typically infeasible because of the large intermediate storage requirement to materialize all input tuples to an operator. We show experimentally that good cache performance can be obtained with smaller buffers whose size is determined at runtime. We demonstrate a low-overhead method of runtime cache miss sampling using hardware performance counters. Our evaluation considers two common database operators with state: aggregation and hash join. Sampling reveals operator temporal locality and cache miss behavior, and we use those characteristics to choose an appropriate input buffer/block size. The calculated buffer size balances cache miss amortization with buffer memory requirements.

1. INTRODUCTION

As the cost per byte of random-access memory continues to decline, large database operations may be performed entirely in main-memory. It has been observed that relatively high latency accesses to main memory account for a significant portion of in-memory database query processing time [1]. Much research has focused on improving the cache behavior of specific database operations, e.g., hash join, aggregation, partitioning, index maintenance and traversal, etc. (see the discussion of related work in Section 2). Rather than improving the cache behavior of a specific operation, this paper examines the crucial higher-level question of how different operators interact with each other in the cache.

To motivate this problem, consider the following simple example. A database has two operators, A and B, which it must run. These operators may be from a single query, or from different concurrently running queries. Each operator uses a relatively large, private data structure such as a hash table. If operator A's data structure fits within the cache, after processing enough tuples, its data structures will become cache resident and subsequent accesses will be cache hits, which will result in faster processing times for those tuples. If, however, operator B's execution is interleaved with operator A's, then B's data structure may evict A's from the cache. This means that when A resumes processing, it will again suffer cache misses to load its data structure into the cache. If too few tuples are processed by each operator, neither operator may reach a point at which its data structure is cache resident. Even though the operators may have been designed to carefully use the cache in isolation, when their execution is interleaved neither benefits from that careful design.

Our approach is to use large enough batches of work for each of A and B so that the initial compulsory cache misses are amortized over many input tuples. To batch the work, we insert buffers into a query plan to hold a batch of intermediate results that are inputs to an operator. For nonblocking operators, we also insert a buffer to hold the output from the operator. Each operator is scheduled so that it processes its entire input buffer in a single batch.

We use hardware performance counters in an on-line fashion to measure the number of cache misses suffered by an operator. This choice gives the system accurate numbers with negligible overhead and without needing to know how an operator works. The system calculates space/time trade-off curves of each operator in the system. This curve is derived from the L2 data cache miss counts measured during the initial runs of the operator. Based on these curves, the system can estimate the payoff for increasing (or decreasing) the buffer size for each operator.

On a multi-core system, we adopt the approach that all available threads should be devoted to a single operator during a time-slice, so that all threads can share common data structure elements in the cache and effectively utilize the full shared L2 cache and the instruction caches. If we allowed each thread to perform independent work, then there would be much less effective cache per thread, and cache interference would be much more likely [19].

We experimentally evaluate our system using a Sun UltraSPARC T1 machine that has 32 on-chip hardware thread contexts. We study the two most common database operators with state, namely aggregation and joins. We show that operator interleaving can more than double the execution time when compared with a plan in which operators are not interleaved. We demonstrate the importance of

[*]This research was supported by the National Science Foundation under Grant IIS-0534389 and by a United States Department of Homeland Security Graduate Research Fellowship.

Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN 2009) June 28, 2009, Providence, Rhode Island.
Copyright 2009 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

identifying operators with state that fits in the cache and then processing enough tuples with those operators in order to amortize the cost of loading that state into the cache. When memory is plentiful, large buffers between operators give good performance, but when memory is limited, more buffer space should be allocated to those operators that benefit the most from temporal locality in accesses to their state. Further, we show that in a mixed operator environment where total buffer space is limited, allocating buffer space based on measured cache miss characteristics of operators can result in a 3% to 8% improvement over a naive allocation of buffer space.

2. RELATED WORK

A large body of related work has sought to make specific database operations cache conscious in order to improve performance by minimizing the processor to memory bottleneck. Examples of this approach include making data structures more cache-conscious by designing them to be either more compact or otherwise harness temporal or spatial locality [15, 16, 11, 3, 5, 9].

Processing data in batches to improve cache behavior is proposed by Padmanabhan et al. [14]. Their study did not explicitly consider interleaving of operators, and the size of their aggregate operator's state was typically small enough to fit in the L1 data cache. As a result, their optimal block size was small, about half the size of the L1 data cache. Padmanabhan et al. demonstrate that such memory optimizations make a difference even in a disk-based database system.

Buffering between operators in a query plan to reduce instruction cache misses was proposed by [21], but the question of buffering to reduce cache misses for operators with state is not addressed. In an OLTP workload where there may not be enough tuples within one plan to make buffering work, instruction cache misses are reduced by context switching between different transactions in order to run a different transaction over the same instructions [7]. In this paper our focus is on data cache misses for OLAP workloads.

Within one operation, buffering has been used to amortize cache misses incurred when loading state, such as a portion of an index. In [20], accesses to memory resident tree-based index structures are buffered at various points in the index. In this paper, we apply a similar concept of buffering to entire operators in a query plan without requiring explicit knowledge of their data structures or memory access patterns.

The question of allocating memory within database systems has also been studied. For instance, Nag and DeWitt examine memory allocation for decision support queries in environments where memory is constrained [13]. Our focus is on in-memory query processing and the specific problem of interquery buffer allocation to improve data cache performance for operators with state.

3. BACKGROUND

The total number of cache misses suffered by any one database operator is the aggregate of the cache misses caused by each of its memory-intensive subprocesses. A database operator which is sorting tuples, for instance, would read input, perform the sort (possibly by accessing an index tree structure) and write output. Typically these subprocesses each access specific data structures and thus exhibit individual cache miss patterns. Some of these subprocesses, such as sequential input scans, have little or no data locality, and the rate at which they generate cache misses is more or less invariant with time. Other subprocesses will display a decrease in the rate of new cache misses as a cache-resident working set is established and the cache becomes "warm." Examples of this latter pattern include tree traversal and cache-resident nested-loop joins. Figure 1

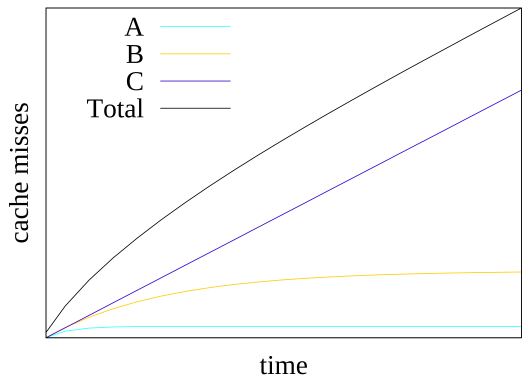


Figure 1: Cache Miss Patterns — the total cache misses suffered are a combination of the cache misses of the constituent subprocesses. In this example, these level out soon (A), somewhat later (B) or remain constant (C).

shows this concept of the aggregation of individual patterns. In this figure, pattern A might represent a small data structure, pattern B a larger data structure that still fits in the cache, and pattern C a data structure (such as a scanned input array) with no temporal locality.

In order to provide fair access for all operators, each would be given some processor time in turn in the form of a "time slice". This interleaving of the operators ensures that none of them are starved of processor time, and the average latency is acceptable. As long as the time slice is not too large, scheduling algorithms can balance latency and throughput in various ways [17, 8]. Operating systems, for example, commonly use a time-slice of 10-100ms [17].

In this paper, we will be dealing with smaller-scale time-slices for database operators with a somewhat different goal in mind. In particular, if our database operator exhibits some temporal locality, as in curves A and B in Figure 1, there is an incentive to run it using time-slices that are long enough to amortize the cache misses over a larger number of records. In modern architectures, data cache misses are expensive and a primary performance bottleneck. If the time-slice is too small, these misses will be incurred again and again; the operator will not benefit from a warm cache. The cache access pattern of an operator may depend on its input; it is therefore important to be able to determine a time-slice tailored to the cache behavior of each operator instance, rather than adopting a one-size-fits-all approach.

4. QUERY PROCESSING MODEL

We do not propose any structural changes to the query optimization process. A query is supplied to the query optimizer, and a plan is generated in the conventional manner.[1] Buffers are inserted before operators with state such as joins and aggregations, in a fashion similar to [21]. Buffers are also added after such operators when those operators are nonblocking.

In a running system, there may be many buffers active at one time. A single query may employ many buffers, and many queries may be running concurrently. For each buffer, we measure the number of cache misses per tuple c using hardware performance counters. To compare the relative benefit of extra memory for different buffers, we need to multiply c by the rate at which tuples are

[1]It may be appropriate to modify the cost functions of certain operators to take account of the reduced cache misses provided by the present work.

generated per unit time, r. The product cr has units of cache misses per unit time.

It is relatively easy to calculate r by measuring how much time it takes for a buffer to fill. In a practical system, r will vary due to (a) changes in the phase of the query plan, such as a transition from the build phase to the probe phase of a hash join, and (b) the admission of new queries and the completion of existing queries. Exactly how one would respond to changes in r is a complex question. Responding to every fluctuation may introduce overhead as buffers are repeatedly enlarged and shrunk. On the other hand, ignoring changes in r altogether would lead to suboptimal memory allocation in changing workloads. We leave the question of how to respond to changes in r to future work. For this paper, we will assume that r is fixed and equal for each active buffer.

Consider, for example, a query that performs a sequence of foreign key hash joins between a fact probe table F and dimension build tables D_1 and D_2, followed by an aggregation. The net rate at which tuples flow into each of the three operators (two joins and an aggregation) is the same, meaning that they each have roughly the same value of r.[2]

For the initial join between F and D_1 there is no input buffer needed because F is a base relation. An output buffer B_1 is required, which also functions as the input buffer for the join with D_2. The output buffer from this join, B_2, is the input buffer for the aggregation. The flow of data is summarized in Figure 2.

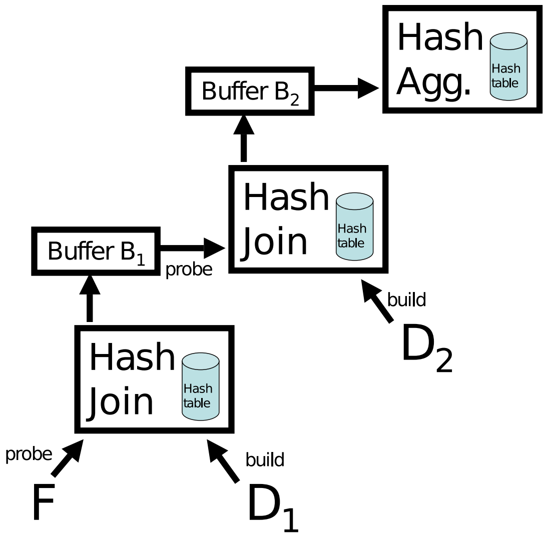


Figure 2: A query plan with buffers.

The per-tuple performance (measured as c above) of the initial join will depend on the size of B_1, and how well this buffer amortizes the cost of reading the hash table on D_1. The performance of the second join will depend on the size of the two buffers. The smaller buffer will be the tighter constraint on how many iterations of the operator are possible during a single batch. This number of iterations should (hopefully) amortize the cost of reading the hash table on D_2. The performance of the aggregation will depend on the size of B_2 and how well this buffer amortizes the cost of accessing the hash table of group-by keys and values.

There are some subtleties caused by the fact that the output buffer of one operator is the input of the next. For example, the D_2 join may consume part of its input buffer B_1, filling B_2. If the initial join were to be run now, it would have only a fraction of B_1 available, and would not fully amortize its cache misses. Instead, the aggregation should be run to empty B_2, and then the D_2 join run again to drain more tuples from B_1. The following simple heuristic achieves good scheduling behavior for chains of operators.

[2]The instantaneous rate at which operators consume and/or produce tuples may vary. However, an operator that runs fast will then have to wait longer for its input buffer to fill or for its output buffer to empty.

Let $B_1, \ldots, B_n$ be n buffers in a chain, and let $O_0, \ldots, O_n$ be operators such that O_i consumes input from B_i, $i = 1, \ldots, n$, and O_i outputs data to B_{i+1}, $i = 0, \ldots, n-1$. We call O_0 the *source* operator, and O_n the *sink* operator. We say an operator O_i is *available for scheduling* if its input buffer (if it exists) is at least half full, and its output buffer (if it exists) is at least half empty.

LEMMA 1. *In any chain of buffered operators, there always exists an operator that is available for scheduling.*

PROOF. *If O_0 is not available, then B_1 must be more than half full. By induction, if O_i is not available then B_{i+1} must be more than half full, for $i = 1, \ldots, n-1$. But if B_n is more than half full, O_n is available.* $\square$

As a result of Lemma 1, it is always possible to order the operators so that each is run when it has substantial input available and substantial output space available. That way, the batch size will always be relatively large, and operators will not need to be interrupted frequently because the input is exhausted quickly or the output fills up quickly.

For a worst-case analysis, one can estimate the c value for each operator by assuming that half its input buffer and half its output buffer is available. Suppose that we get c values of c_1, c_2, and c_3 for the three operators respectively, given certain buffer sizes for B_1 and B_2. The overall cost is thus $r(c_1 + c_2 + c_3)$ for this query.

If we add a unit of memory to B_1, c_1 and c_2 may be reduced to c_1' and c_2' respectively, giving a new overall cost of $r(c_1' + c_2' + c_3)$. Alternatively, we could add a unit of memory to B_2, at which point c_2 and c_3 may be reduced to c_2'' and c_3'' respectively giving a new overall cost of $r(c_1 + c_2'' + c_3'')$. Whichever is the smaller cost is the better choice of the two. If a better improvement is possible for the same memory in another query, then that other query should get the unit of memory.

In some cases, such as a simple aggregation of a base table, no buffers are needed. It may appear tempting to let such an operator run for a very long time slice, since there is no memory cost. However, in extreme cases such a choice could cause starvation for other queries, and an absolute time-slice limit should be enforced. An analogous choice would be needed if there was such an abundance of memory that huge buffers were feasible.

5. RUNTIME SAMPLING

To handle arbitrary operators on arbitrary input, we need to measure cache misses for operators in a black-box fashion, without knowledge of how they work. Fortunately, most modern platforms provide a hardware-based mechanism to measure events such as cache or TLB misses. On the UltraSPARC T1, the performance counters may be accessed programatically by using the CPC counter libary, which enables a per-thread measurement of cache misses and other processor events. Accessing the performance counters on the T1 via the CPC library results in a negligible performance penalty and we use them only when necessary.

One thread is responsible for operator scheduling and the processing of statistics. The remaining 31 threads do the real work. When an operator is scheduled, we devote all 31 threads to process the operator in parallel. As we have previously mentioned, this choice allows all threads to effectively utilize the full shared cache, rather than having to do independent work using a small fraction of the cache. Additionally, this choice makes the performance of the operator, including the sampled number of cache misses, more predictable. Performance does not depend on what else might be running on some of the threads.

We interrogate the performance counters just before and just after an operator call, in order to obtain the cache miss count for processing a given number of records. The cache miss counts from all threads are accumulated together. These cache miss counts, when divided by the number of tuples processed, correspond to the c cost factor described in Section 4.

In practice, these numbers are obtained for the first few batches of tuples in the actual running code. We start with an initial buffer of moderate size B. During the initial profiling, we run the operator without pre-emption on successively larger segments of the buffer, calculating cumulative cache miss statistics along the way. That way, in one pass through the buffer, we get performance statistics not just for size B, but for a range of sizes smaller than B.

As an example, consider the choice of B where $B = 2^k$. We start by sampling 1, then 2, then 4, then 8, up to 2^{k-1} tuples at a time, recording the number of cache misses for each batch. In the i^{th} iteration, all $i - 1$ iterations worth of tuples have "warmed" the cache. After each iteration we compute an approximation of second derivative of the number of cache misses per tuple processed. When this gets close enough to zero (an empirically derived 'threshold' that we compare against), we decide that the buffer is successfully amortizing cache misses. We use the second derivative because there are compulsory cache misses, such as reading input, which means that the first derivative will likely be non-zero.

The system can compare the statistics for this operator with those for other running operators as described in Section 4. If the buffer is too large, it can be shrunk. The statistics allow the direct choice of an appropriate buffer size. Because very small buffers lead to many other context-switching overheads besides cache misses, we impose a minimum buffer size for all operators.

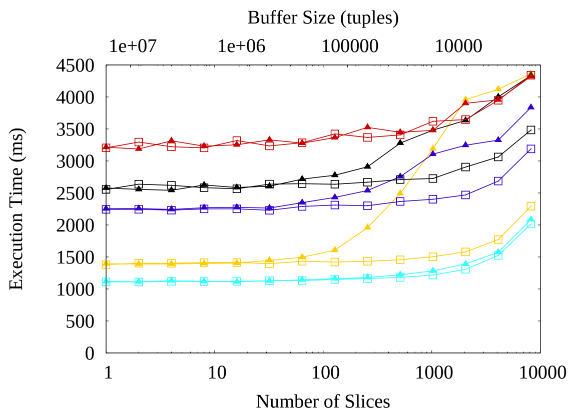
If the buffer is too small, there are several ways that it could grow. One could try to grow it incrementally in several steps, gathering more statistics along the way, until an appropriate size is reached. At the expense of temporarily using more memory, one could allocate a much larger buffer of size B', and accumulate statistics as before for a range of buffer sizes between B and B'. The incremental approach uses less memory, but may take longer to converge.

In some cases, an operator's initial performance may not be typical of its steady-state performance. For example, the adaptive aggregation method of [3] starts out by sampling its input. We therefore allow an operator to set a "Wait-I'm-Changing" flag, to inform the cache-miss monitoring system to hold off from measuring cache misses.[3]

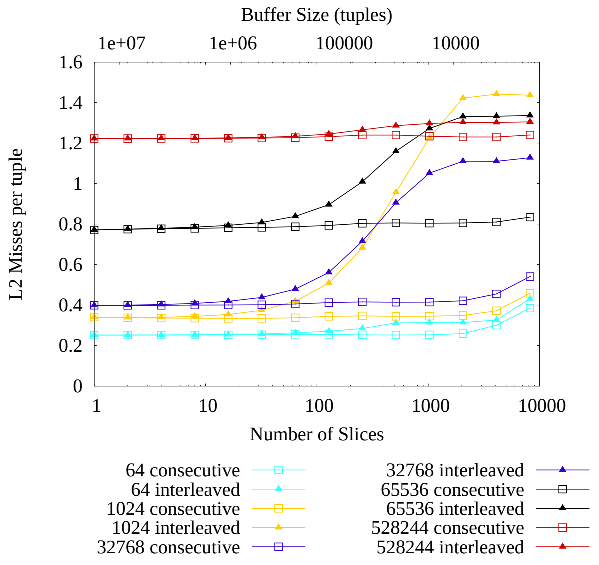
6. EXPERIMENTAL RESULTS

We conducted all of our experiments on real hardware, a Sun T1000 server with an UltraSPARC T1 processor. The operators we investigated were hash aggregation and hash join. The specifications for our experimental platform, input distributions, and operator implementations can be found in Appendix A.

[3]This flag may appear to violate the "black box" methodology we propose. One could alternatively wait a fixed time for startup overheads to have been paid, or to be more sophisticated and sense when the operator appears to have reached a steady state.



(a) Execution time



(b) Cache Misses

Figure 3: Execution time and measured L2 cache misses for interleaved and non-interleaved (consecutive) aggregation queries with various group-by cardinalities. There were 8 aggregation operators and 2^{24} uniformly distributed input tuples to each operator.

6.1 Cache Effect of Interleaving Queries

The first experiments we ran aimed to expose the effect of interleaving queries. 2^{24} (approx. 16 million) tuples were used as input for batches of eight identical aggregation queries. With an input buffer size of b tuples, the input was effectively processed in $2^{24}/b$ "slices". 31 of the available 32 threads were dedicated to processing a particular slice at once; one thread was used for scheduling and synchronization.

In one series of experiments, the slices were processed consecutively, i.e. all the slices of a single query were processed to the end before moving on to the next query. In a second series, the slices were interleaved so that after processing a slice from one query, a slice from the next would be processed.

In the non-interleaved case, the cache-resident items from work on a previous slice are available for work on the next slice. The cache is "warm" at the start of the slice, unless the slice is the first one in the query. In the interleaved case, the cache is likely to be "cold" for any single slice, because the intervening slices would

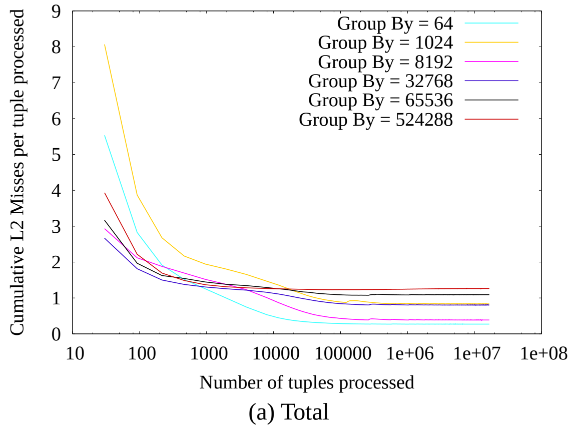


Figure 4: Cumulative cache misses per tuple for the aggregation workload. Note the logarithmic scale in (a).

have caused the query's data to be evicted from the cache. The slicing overhead (synchronization etc.) is the same in each scenario. We measured both execution times and L2D cache misses (using the T1's CPC counters) in the experiments.

Figure 3(a) shows how execution time varies with the number of slices for various group-by cardinalities. The group-by cardinality determines the size of the hash table that the aggregation operator needs to maintain. Additionally, the aggregation operator internally uses different algorithms for different group-by cardinalities.

Figure 3(b) shows the corresponding measurements for the L2 data cache misses, which are strongly correlated with the execution times. There is a baseline of 0.25 misses per tuple that comes from scanning the input tuples, which each occupies 0.25 cache lines. The remaining misses are for the aggregate operator's hash tables.

The measurements for interleaved and non-interleaved queries are nearly identical for large buffers. However, with decreasing buffer size, the graphs show that the number of cache misses for the interleaved queries increases significantly and the total execution time grows correspondingly. The cache misses and execution time of the non-interleaved queries, in contrast, are relatively stable. The non-interleaved performance starts to worsen beyond a buffer size of 2^{13} tuples due to context-switching overheads that become significant. As a result, a minimum buffer size of 2^{13} tuples would be appropriate.

A closer examination reveals that for group-by cardinalities that are small (64) and large (528244), large buffers do not significantly improve the cache miss profile or the overall performance. This effect is to be expected. For very small group-by cardinalities, even a small buffer is enough to amortize the cache misses, since there are so few of them. For very large group-by cardinalities, almost every access will be a cache miss whether or not the input is buffered.

For the intermediate group-by cardinalities, the performance impact of buffering is significant. For a group-by cardinality of 1024, changing a buffer from 2^{13} tuples to 2^{16} tuples halves the overall execution time, and even larger buffers can be used to get an additional 25% improvement. For 1024 and fewer group-by values, the adaptive aggregation operator chooses to employ a two-level hashing scheme, with each thread having a small local hash table that overflows into a large shared hash table. At 1024 group-by values, specifically, it takes a relatively large number of accesses to bring all of the local tuples into the cache and hence the benefit from a larger buffer size.

The group-by cardinality of 32768 shows a case where the aggregation algorithm uses a single shared hash table that just fits within the L2 cache. Changing the buffer from 2^{13} tuples to 2^{16} tuples reduces the overall execution time by 20%. The group-by cardinality of 65536 shows a case where the aggregation algorithm uses a single shared hash table that just exceeds the L2 cache size. As a result, the number of cache misses per tuple is higher than the 32768 cardinality case.

Figure 4 presents the same data, interpreted cumulatively. The graph shows how well the total number of cache misses incurred up to a certain number of records is amortized over those records. Figure 4(b) shows the initial region up to a buffer size of 60,000 records. The group-by cardinality of 64 quickly amortizes its relatively small number of cache misses, and asymptotes to 0.25, which is the number of misses required to process the input.

The group-by cardinality of 524,288 never manages to amortize its relatively large number of cache misses. It asymptotes to 1.25, which is the number of misses required to process the input and one hash cell per input record. Intermediate cardinalities have intermediate behavior. The curves cross, because the aggregation operator is employing different algorithms for the different group-by cardinalities. By 60,000 records, all operators are close to their asymptotic number of cumulative misses per tuple.

In Figure 3(a), it appears that using 100 slices (with a buffer size of about 160,000 records) allows the system to amortize almost all of the cache miss penalty. Individual slices are taking between 10ms and 35ms. These are relatively small slices in terms of absolute time, and will not cause scheduling delays for fast queries waiting for slow queries to complete their allotted timeslices.

6.2 Variable Buffer Size

In a mixed workload, some queries might be more sensitive to the buffer size than others. We experiment with a workload containing ten aggregation queries having a group-by cardinality of 64, and ten queries with a group-by cardinality of 1024. As we saw in Figure 3, a query with 64 group-by values performs well even with small buffers, while a query with 1024 group-by values can benefit more from larger buffers.

It is clear from the previous experiments that, if memory is available, it could (and should) be used to improve temporal locality. In Figure 5, we investigate the scenario where the total memory budget is fixed; all that can be varied is how the memory is allocated to the various operators. The black curves show the performance of the mixed workload when interleaved and when not interleaved, assuming an equal distribution of memory to all operators. The red curves show an allocation in which the memory is preferentially

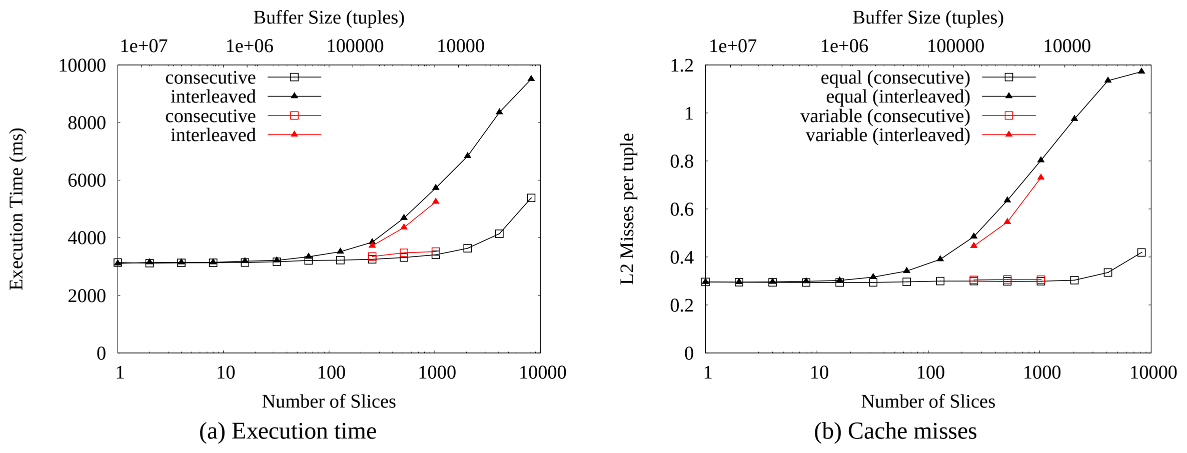


Figure 5: Execution time and cache misses, mixed workload.

allocated to the 1024 group-by value queries. In particular, the 64 group-by value queries each get buffers with 8192 records (the minimum value, as discussed above). The remaining memory is divided equally among the 1024 group-by value queries. Each data point is plotted at the axis coordinate corresponding to the overall average buffer size, so that the points on the red and black curves are comparable.

Allocating the memory non-uniformly makes a modest, but measurable difference to overall performance, between 3.3% and 8.4% in the range shown in Figure 5.

We experimented with the space allocation methods described in Sections 4 and 5, in order to generate buffers whose size is appropriate to the locality displayed by the operator. As a baseline, we used the default choice of allocating the available space equally among all operators. The informed allocation performed better than the default allocation, but typically by only a few percent. The 3% to 8% results reported in Figure 5 above were among the best improvements generated.

The reason for this behavior is twofold: If all operators in the workload need memory for locality, then an even allocation will probably do a reasonably good job. If only some of the operators need memory for locality, then it does pay to give those more memory. Nevertheless, Amdahl's law means that the apparent benefit of that memory is smaller, being "diluted" by the time spent on buffer-size-insensitive work. The results of this section suggest that equal allocation performs well as a default allocation strategy, extra effort yielding an improvement of only 8.4%.

An exception to this rule would be if the r values, i.e., the rate at which tuples flow into an operator, is significantly different for different operators. Studying operators with differing (possibly varying) r values is left to future work.

6.3 Alternative Distributions

The locality of an operator can depend on its input distribution. For example, if some of the group-by values of an aggregation operator are very common, then there is more temporal locality than a uniformly distributed set of group-by values of the same cardinality. We ran the aggregation algorithm on the various distributions described in [3], and measured the cache misses and performance for various buffer sizes.

The results were in line with the expected changes in temporal locality. Figure 6 shows the results for a self-similar distribution in which, at every scale, 80% of the accesses go to 20% of the data [6]. Note that the absolute performance is not directly comparable to the performance for uniform data because the aggregation operator is using different algorithms in each case [3]. Nevertheless, because of skew in the data access pattern, there is increased temporal locality at all group-by sizes. This enhanced locality is apparent in the gap between interleaved and non-interleaved performance. At a group-by size of 1024, the gap is smaller than for uniform, because there is less pressure on the cache. At a group-by size of 528,244, the gap is larger than for uniform, because many accesses that used to be cache misses (even with large buffers) are now cache hits.

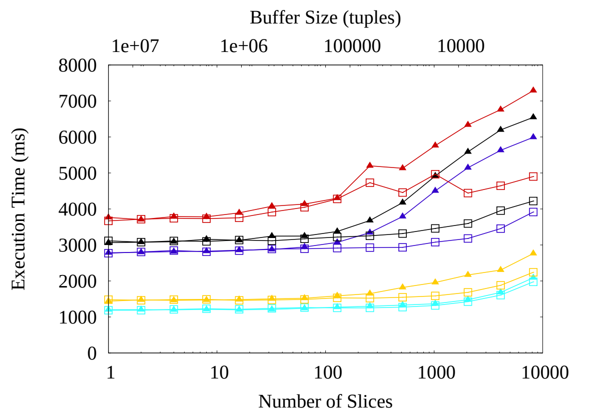
6.4 Joins

We also validated our observations on a hash join operator. A hash join operator's hash table represents a large amount of state, similar to the hash table used for aggregation described earlier.

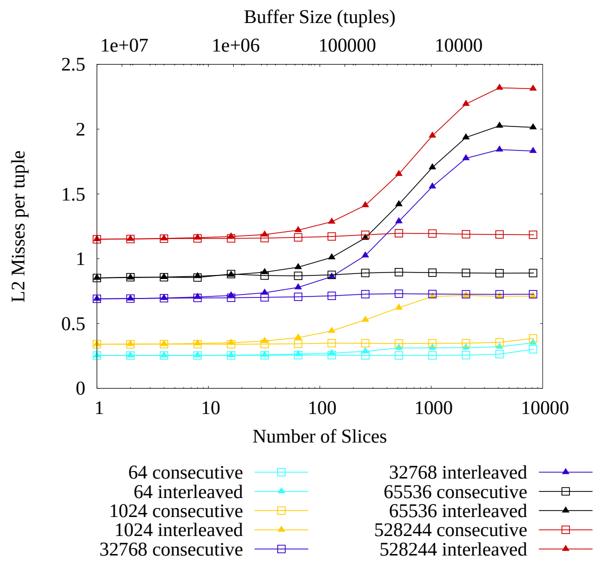
We constructed the hash join workload to be a foreign key join in which each tuple in the probe relation joins with one and only one tuple in the build relation. The number of unique keys in the probe input was varied in a manner analogous to the varied group-by cardinalities used in the aforementioned aggregation experiments. The build relation was always sized to exactly match the number of unique keys in the probe input, containing each key once. Therefore, the size of the hash table was directly related to the number of unique keys in the probe input.

Based on the size of the hash table's components, with a load factor of .5, a hash table containing around 150,000 tuples will fit within the L2 cache. Figure 7 shows execution time and cache miss results for various hash join workloads. In these experiments, the input contained different numbers of unique keys that were uniformly distributed.

The join experiments confirm the aggregation results. Figure 7(a) shows that join performance suffers when smaller buffers (more frequently interleaving) is used. This is particularly true for the input distributions with hash tables that use a lot of the cache, but do fit. Figure 7(b) shows that the 32768 and 65536 inputs for example benefit from locality when accessing the hash table, but only if enough tuples are processed so that a cache line can be reused. The 528244, conversely, has a hash table that does not fit in the cache, so it experiences a high number of cache misses whether execution is interleaved or not. In the case of aggregation with a group by cardinality of 64 or hash join input with only 64 unique keys, the respective hash tables are so small that they fit within the L1 cache and L2 misses are almost exclusively caused by reading input after the hash table is initially loaded into the cache.

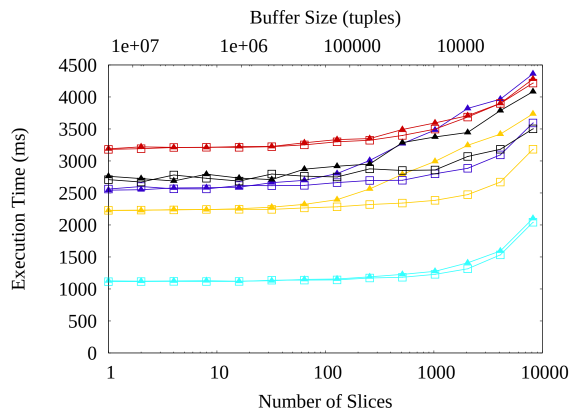


(a) Execution time

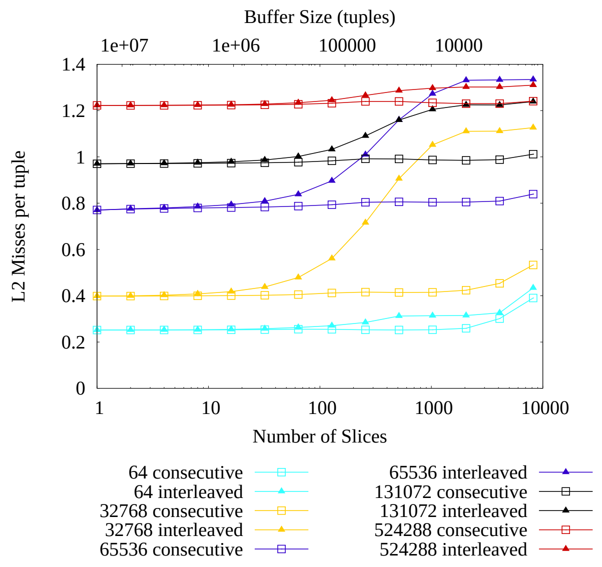


(b) Cache misses

Figure 6: Execution time and cache misses on a self-similar input distribution.



(a) Execution time



(b) Cache misses

Figure 7: Execution time and cache misses for the hash join workload with uniformly distributed probe input.

6.5 Finding Performance Bugs

During our initial experiments with the aggregation method of [3], we were surprised to find that there was a significant penalty to interleaved computation, even when the group-by cardinality was very high. This observation was unexpected, because at very high group-by cardinalities, almost all hash table references would be cache misses, whether the operator is interleaved or not.

After some investigation, we realized that the aggregation method in fact used *two* data structures for aggregation. The keys and aggregates were stored in hash table cells, while a set of “valid” flags was stored in a separate array, one byte per flag, to identify whether a hash cell stores any useful data. Even if this flag array fits into the cache, the time-slicing of the aggregation operator means that the flag array needed to be reloaded into the cache on every time slice. We were able to overcome this performance problem by redesigning the hash cell so that the valid flag was part of it, in a single cache line, meaning that only one cache miss per cell is needed.

In [3], we did not notice a significant performance impact, because this flag array was small enough to easily fit in the cache in most experiments. Nevertheless, the performance of aggregation on examples so large that the flag array did not fit in the cache did suffer, because two cache misses were needed when one would have been sufficient.

7. CONCLUSION

In main-memory databases and disk-based databases alike, cache misses are a major performance issue. When a database system is processing many concurrent queries, each with many operators, it will have to run each operator in some kind of time-sliced fashion. If a database operator has state, such as a hash table, compulsory cache misses will be encountered every time the operator resumes processing in a new time-slice.

We have studied the cache miss behavior of database operations with state, such as aggregations and joins. We have shown that it is desirable to allocate large buffers, enough to hold 10,000 to 100,000 or more records, in order to amortize an operator’s cache misses over many input records. In contrast to [14], which advocates buffers of about 10KB for hash-based aggregate operators, we show that buffers whose size is measured in megabytes can be worthwhile. In some cases, these buffers reduce the time taken to process a query by *more than a factor of two*. We have shown how one can instrument a system running on a multicore machine to measure cache misses in real-time on the actual input data, without knowing the algorithms or data structures employed by an operator. We have also shown how operators with intermediate-size state have the most to gain from large buffers: small data structures will become cache resident quickly, while very large data structures without temporal locality will always generate cache misses, even with buffering.

The disciplined use of buffers enables a database system to scale the number of concurrently running query operators, while maintaining a level of performance close to that obtained when each operator is run in isolation.

In future work, we plan to consider grouping together consecutive operators whose cache behavior is non-interfering into super-operators, to allow better pipelining [20, 21]. For example, if we had two consecutive joins with small build tables that together fit in the cache, there is no reason to buffer between those operators. A single larger buffer for the pair of operators would be better.

8. REFERENCES

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[2] Peter A. Boncz and Martin L. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[3] John Cieslewicz and Kenneth A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[4] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, pages 1–10, 2007.

[5] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious frequent pattern mining on a modern processor. In *VLDB*, pages 577–588, 2005.

[6] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[7] Stavros Harizopoulos and Anastassia Ailamaki. Improving instruction cache performance in OLTP. *ACM Trans. Database Syst.*, 31(3):887–920, 2006.

[8] D. Karger, C. Stein, and Joel Wein. Scheduling algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*. CRC Press, 1997.

[9] Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, pages 1275–1285, 2007.

[10] Roger MacNicol and Blaine French. Sybase IQ multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.

[11] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):709–730, 2002.

[12] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. Cache-conscious radix-decluster projections. In *VLDB*, pages 684–695, 2004.

[13] Biswadeep Nag and David J. DeWitt. Memory allocation strategies for complex decision support queries. In *CIKM*, pages 116–123, 1998.

[14] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574, 2001.

[15] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.

[16] Jun Rao and Kenneth A. Ross. Making B+-trees cache conscious in main memory. *SIGMOD*, pages 475–486, 2000.

[17] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 6th edition, 2003.

[18] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[19] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. Improving database performance on simultaneous multithreading processors. In *VLDB*, pages 49–60, 2005.

[20] Jingren Zhou and Kenneth A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, pages 405–416, 2003.

[21] Jingren Zhou and Kenneth A. Ross. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD*, pages 191–202, 2004.

APPENDIX

A. EXPERIMENTAL SETUP

We ran all experiments on a 1GHz Sun UltraSPARC T1 architecture, the details of which are summarized in Table 1. The T1 is a chip multiprocessor with 8 cores and 4 threads per core for a total of 32 hardware thread contexts. It has a small 8KB L1 cache for each core (i.e. shared amongst 4 threads) and a larger L2 cache (3MB in our case) shared between all hardware threads. Its cache line is 64 bytes. A large main memory (8GB) ensured that all the data and processing were in-memory. The T1's TLB supports multiple page sizes, and we forced the use of large, 256MB pages to ensure that TLB misses did not affect our measurements.

Experiments were run using both a hash-based aggregation operator and a hash join operator.

Processor	1 GHz UltraSPARC T1
Main Memory	8GB
L1 Instruction Cache	16KB/core
L1 Data Cache	8KB/core
L2 Data Cache	3MB
Hardware Threads	32
Cache Associativity	12
Data Cache Line Size	64

Table 1: Hardware Platform used for Experiments

Aggregation Implementation

The hash-based aggregation operator used in this paper is the multi-threaded, adaptive aggregation method presented in [3]. Based on lightweight sampling of the input, the operator chooses between different hash-based aggregation options. When there is no interthread contention or temporal locality in hash table accesses, the adaptive operator chooses to use one global hash table shared among all of the threads. This global table requires the use of a mutex or atomic operations in order to protect concurrent aggregate updates. When interthread contention or temporal locality is high, the adaptive operator chooses a two-level aggregation method. In this method, each thread has a small private hash table that will contain contentious and frequently accessed items. This private table spills excess entries into the global table described above. Finally, the adaptive operator is able to detect input with runs of consecutive keys which can be aggregated directly. This run-based aggregation is not used in any of the experiments in this paper. The implementation of the adaptive aggregation operator is described in more detail in [3].

Hash Join Implementation

The hash join implementation used in these experiments is similar to the hash join described by Manegold et al. [11], in which elements that hash to the same location are chained together using an array indexed by the record id of the element in the bucket. An array entry contains the record id (or index) of the next record in that bucket. Thus by following the chain of record ids, each record in a hash bucket may be located. Chaining, however, is a relatively rare occurrence because we sized the hash table to have an average load factor of $.5$.

As input, the hash join takes two relations whose tuples are the keys to be joined, and produces a join index as output. Because writing the join index must to be done in parallel by multiple threads, concurrent output writing is accomplished using the parallel buffer data structure described in [4]. The parallel buffer allows multiple threads to read or write a shared buffer with minimal interthread contention.

We study the performance of the probe phase of the join, once the hash table on the smaller input has been built.

Input Characteristics

In a main-memory setting, the transfer of data between the RAM and the cache is a precious resource. As a result, unnecessary or redundant reading or copying of data is likely to perform poorly. Such issues motivate the use of column-wise storage [10, 18, 2], so that only the required columns need to be read. Join algorithms that build a join index and resolve the pointers in a final stage are beneficial in a main-memory setting for similar reasons [12].

It is therefore appropriate to focus on situations in which the tuples flowing between operators are relatively narrow. In our evaluation, they will be 8 or 16 bytes. In the event that wider tuples are needed, the resulting extra cache misses could be partially hidden using prefetching.

In all experiments the input contained 2^{24} tuples. For aggregation, the input tuples used were two 64-bit unsigned integers, so each input tuple is 16 bytes long (or 0.25 cache lines). For the hash join, the input tuples consisted of 64-bit keys, so each tuple is 8 bytes long (or 0.125 cache lines). The record-id of a key is simply its position in the record sequence. The output join index consists of tuples with two 32-bit record- ids, so each output tuple is also 8 bytes long. The input distributions were generated using the techniques described in [6].

Thread Coordination

Coordination between the scheduler thread and the workers is via variables shared between each worker and the scheduler. When a worker thread is launched, its shared "pending" flag is set to false. Worker threads spin-wait for this flag to become true. The scheduler thread selects an operator to run by setting a shared variable to point to the current operator as well as by specifying a range of tuples for each thread to process. The scheduler thread starts the worker threads by setting the pending flag to true. It also resets to zero a counter shared by all worker threads. As each worker thread completes its assigned task, the worker thread sets its pending flag back to false and atomically increments the counter. The worker thread then goes back to waiting for its pending flag to change. In the mean time, the scheduler thread is waiting for the shared counter to equal the number of worker threads, at which time it knows that all threads have completed their assigned work and are ready for the next operator assignment. While waiting for workers to complete, the scheduler thread could perform preparatory work for the next scheduling task, but such optimizations are left to future work.

Scheduling of independent operators is done according to the following heuristic: schedule the operator that has processed the least input so far, excluding the operator that was run on the previous time-slice. When all operators have the same buffer size, a round-robin schedule results. When there are mixed buffer sizes, operators with smaller buffers will get scheduled more often than those with larger buffers.