Main-Memory Scan Sharing For Multi-Core CPUs

Lin Qiao Vijayshankar Raman Frederick Reiss Peter J. Haas Guy M. Lohman
IBM Almaden Research Center, San Jose, CA 95120, U.S.A.
{lsqiao,ravijay,frreiss,phaas,lohmang}@us.ibm.com

ABSTRACT

Computer architectures are increasingly based on multi-core CPUs and large memories. Memory bandwidth, which has not kept pace with the increasing number of cores, has become the primary processing bottleneck, replacing disk I/O as the limiting factor. To address this challenge, we provide novel algorithms for increasing the throughput of Business Intelligence (BI) queries, as well as for ensuring fairness and avoiding starvation among a concurrent set of such queries. To maximize throughput, we propose a novel FullSharing scheme that allows all concurrent queries, when performing base-table I/O, to share the cache belonging to a given core. We then generalize this approach to a BatchSharing scheme that avoids thrashing on "agg-tables"—hash tables that are used for aggregation processing—caused by execution of too many queries on a core. This scheme partitions queries into batches such that the working-set of agg-table entries for each batch can fit into a cache; an efficient sampling technique is used to estimate selectivities and working-set sizes for purposes of query partitioning. Finally, we use lottery-scheduling techniques to ensure fairness and impose a hard upper bound on staging time to avoid starvation. On our 8-core testbed, we were able to completely remove the memory I/O bottleneck, increasing throughput by a factor of 2 to 2.5, while also maintaining fairness and avoiding starvation.

1. INTRODUCTION

Historically, business intelligence, or BI, has been an I/O-bound workload. Business data is stored on the disks of a data warehouse, and retrieving data from these disks is the main cost in query execution. The state of the art in BI is defined by this I/O bottleneck: Low-end systems spend most of their time waiting for disk I/O; while high-end systems use large numbers of disks to achieve high throughput at great financial cost.

Researchers have developed several techniques to alleviate this bottleneck by reducing amount of data a query processor needs to touch. These techniques include aggressive compression [2, 17], column stores [11], and materialized views [19]. With the advent of large main memories, these techniques often allow the entire working set of a BI system to fit in RAM, bypassing the traditional disk I/O bottleneck. For the first time, BI has become CPU-bound.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand

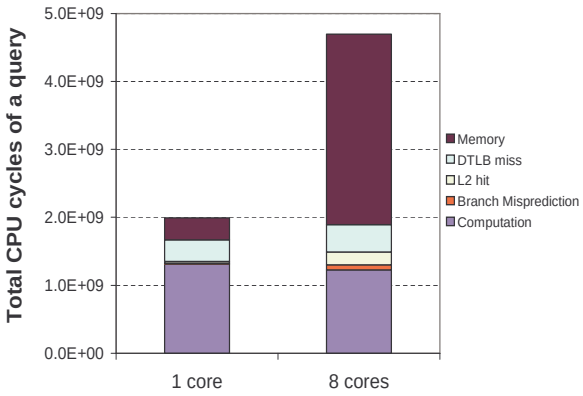


Figure 1: Breakdown of total CPU cycles consumed for the query from Figure 2, when running on an 8-core server. The stacked bar on the left is generated by pinning all threads to a single core. When all cores are used, main memory bandwidth becomes the performance bottleneck.

However, recent trends in hardware are bringing this new era quickly to an end. Processor manufacturers are putting ever-increasing numbers of cores onto a CPU die, and main memory bandwidth is not keeping pace.

A simple experiment demonstrates the performance effects of the recent trend towards multicore CPUs. We take a typical BI query (Figure 2) and run it on the Blink query processor [18], on an Intel Xeon server with 2 quad-core CPUs. Blink is a compressed main-memory database targeted at BI applications; we describe the system in more detail in Section 2. We used the `oprofile` whole-system profiler and the Xeon's hardware performance counters to break the cycles spent on this query (across all 8 cores) into five components: cycles spent in computation, cycles wasted on pipeline stalls due to branch mispredictions, cycles spent on level-1 (L1) cache misses, translation lookaside buffer (TLB) misses, and main memory access (L2 cache misses).

As Figure 1 shows, when the query is bound to a single core, the system is CPU-bound, with the majority of time going to computation. But when we allow the query to use all 8 cores on the machine, accessing main memory becomes the bottleneck. With manufacturers soon to put 6 and 8 cores on a single chip, this problem will only become worse.

In this paper, we address the memory bandwidth bottleneck with a novel scan sharing technique for main-memory query processors. We observe that much of the memory bandwidth used in BI queries goes towards scanning tables and indexes, and that these scans are

often repeated across multiple queries. We attach multiple queries to a single scan, amortizing the overhead of reading the data from main memory into the processor's cache.

Shared scans have been used in the past to overcome disk I/O bottlenecks [16, 21], but bringing this technique to main-memory DBMS's poses significant challenges. Disk-based systems use programmable buffer pools and dedicated I/O threads to implement scan sharing. Different queries share data via the buffer pool, and a buffer manager choreographs the reading of data into the pool.

In a main-memory DBMS, however, the processor cache takes the place of the buffer pool, with the cache controller hardware determining the data that resides in cache. In such an environment, scan sharing requires careful scheduling of low-level query operations, ensuring that data is resident in the cache when it is needed.

This scheduling is complicated by the fact that processor caches are significantly smaller than buffer pools. The working set (auxiliary data structures like hash tables and dimension tables) of a small group of queries can easily exceed the size of cache, leading to thrashing. An implementation of scan sharing needs to estimate the working set sizes of queries and to avoid grouping too many queries together. Efficiently predicting the working set size of a query, e.g, by sampling, is a non-trivial problem. For example, with a group-by query, if we adopt a simple bound on the working set – the number of distinct group-by values, we have to solve the infamous sample-based distinct count estimation problem [10].

1.1 Contributions

In this paper, we present a query scheduling algorithm that implements shared scans on a main-memory DBMS. We show that the naive application of this algorithm can lead to thrashing as the working set size of queries exceeds cache. We then develop extensions to our algorithm to group queries in a way that avoids thrashing. As part of this extended algorithm, we provide a new technique for efficiently and robustly estimating the working-set size of group-by queries. This technique rests on the observation that the working-set size is less than the number of distinct values due to skew in access to the hash table buckets, so that we can exploit the well-studied statistical notion of *coverage* [9]; coverage is a more tractable quantity to estimate than the distinct value count. We also develop techniques for guaranteeing a fair allocation of shared resources.

We evaluate our techniques using a thorough set of experiments on a real-world dataset. We demonstate that our techniques significantly improve the scaling of multi-query workloads on multicore processors. On our 8-core testbed, we obtain near linear scaling of throughput with cores, a performance improvement of up to 2.5 times of that which is attained without these techniques.

Paper Outline

Section 2 provides architectual and system background. Section 3 describes related work. In section 4, we propose a novel approach to achieve appropriate I/O sharing inside a cache. In section 5, we provide a robust technique to estimate query parameters. Section 6 presents experimental results, and we conclude in Section 7.

2. BACKGROUND

This section presents some material that is useful background for this paper. We give an overview of modern hardware architectures, and then describe Blink, the main memory database in which our techniques are implemented and our experiments are run.

2.1 Modern Hardware Architectures

Today, major processor vendors are shipping processors equipped

```
select
  X2.MATL_GROUP, X1.INDUSTRY, DU.SID_BASE_UOM,
  DU.SID_STAT_CURR, DT.SID_CALMONTH
  sum(F.RTNSQTY), sum(F.RTNSVAL), sum(F.INVCD_CST3), sum(F.INVCD_QTY3), sum(F.INVCD_VAL3),
  sum(F.OPORDQTYBM3), sum(F.OPORDVALSC3), sum(F.ORD_ITEMS3), sum(F.RTNSCST3),
  sum(F.RTNSQTY3), sum(F.RTNSVAL3), sum(F.RTNS_ITEMS3), sum(F.CRMEM_CST4),
  sum(F.CRMEM_QTY4), sum(F.CRMEM_VAL4), sum(F.CST4), sum(F.QTY4),
  sum(F.VAL4), sum(F.INVCD_CST4)
from
  "/BIC/FLARGE" F, "/BIC/DLARGE1" D1 "/BIC/XCUSTOMER" X1, "/BIC/DLARGE2" D2,
  "/BI0/XMATERIAL" X2, "/BIC/DLARGEP" DP, "/BIC/DLARGET" DT, "/BIC/DLARGEU" DU
where
  F.FKEY1 = D1.DIMID and D1.SID_0SOLD_TO = X1.SID
  and F.FKEY2 = D2.DIMID and D2.SID_0MATERIAL = X2.SID
  and F.FKEYP = DP.DIMID and F.FKEYU = DU.DIMID
  and F.FKEYT = DT.DIMID
  and DP.SID_0CHNGID = 0 and X2.MATL_GROUP IN (9, 8, 7, 6, 5, 4, 3, 2)
  and DP.SID_0RECORDTP = 0 and DP.SID_0REQUID ≤ 536
  and X1.INDUSTRY IN (9, 8, 7, 6, 5, 4, 3, 2) and D5.SID_0VTYPE IN (2, 3, 4, 5, 6)
  and X2.OBJVERS = 'A' and X1.OBJVERS = 'A'
group by
  X2.MATL_GROUP, X1.INDUSTRY, DU.SID_BASE_UOM, DU.SID_STAT_CURR, DT.SID_CALMONTH;
```

Figure 2: An example BI query, reporting various sales statistics, broken down by industry and time.

Vendor CPU Name	Intel Xeon X5355	AMD Opteron 8347	Sun UltraSPARC T2
Num. Cores	4	4	8
L1 Cache	4 x 64 KB	4 x 128 KB	8 x 24 KB
L2 Cache	2 x 4 MB	4 x 512 KB	1 x 4 MB
L3 Cache	N/A	1 x 2 MB	N/A
Main Memory (4GB modules)	Up to 32 GB	Up to 32 GB	Up to 256 GB

Table 1: Architectural characteristics of three recent CPUs. The experiments in this paper were conducted on a machine with two Intel Xeon X5355 processors.

with 4 separate processing cores, with 6- and 8- core processors in the pipeline. Table 1 shows some statistics of processors from the current architectural generation.

Each core in a multi-core processor is an independent CPU; this CPU sits at the top of a memory hierarchy consisting of 2-3 levels of cache and a relatively slow main memory. Each core has a private level-1 (L1) cache that is very fast but very small. Larger level-2 (L2) and, often, level-3 (L3) caches provide slower access to larger amounts of memory Typically, the largest cache is shared across all cores on the processor die, while each processor maintains its own private cache at the higher caching layers. For example, the AMD Opteron processor in Table 1 has a shared L3 cache and private L1 and L2 caches.

At each level of the hierarchy, performance drops by one to two orders of magnitude. Storage capacity follows a different trajectory, increasing by a factor of 2-4 at each cache layer, with a dramatic jump in capacity at the main memory layer. Even the largest processor caches represent less than half of one percent of a modern computer's memory.

This cache/memory hierarchy is somewhat similar to the memory/disk hierarchy for which mainstream database systems were designed, with cache taking the place of the buffer pool and main memory taking the place of disk. However, there are are two important differences.

First of all, control of this memory hierarchy is implemented mostly in hardware, with the cache and memory controllers making most low-level decisions about which regions of memory reside in which level of the hierarchy. Modern CPUs provide a few instructions to "suggest" policy changes to the hardware (e.g., the x86-64 prefetch instructions [1]), but these mechanisms do not provide the flexibility and control that a typical database buffer pool enjoys. In addition, many of the low-level synchronization primitives needed

to implement a buffer pool within the L2 cache are themselves as expensive as a cache miss.

The second difference is one of scale. Even large L2 and L3 caches are typically less than 10 MB in size, which is smaller than database buffer pools have been for many years. Business intelligence (BI) queries are highly complex, and running them efficiently requires keeping a large "working set" in cache, including indexes, intermediate data structures, and executable code.

2.2 The Blink Query Processor

Blink is a query processor that operates on compressed main-memory tables. We now briefly describe the data format and query processing strategy used within Blink, focusing on the aspects that are relevant to the work in this paper. A more complete description of Blink can be found in [18].

2.2.1 Data Organization

Most columns are encoded with a dictionary code, where individual column values are replaced with *codewords*: offsets into a column-specific array of distinct values (the column *dictionary*). These offsets are fixed-length and bit-aligned, for extreme compression. Blink also exploits skew in data distribution by horizontally partitioning tables such that values with similar (marginal) frequencies go to the same partition, and using shorter offset-lengths for the partitions with more frequent values.

This dictionary compression scheme is further optimized in two ways to ensure that we do not have to decode the codewords when processing each tuple. First, the column dictionaries are stored in order-preserving fashion so that equality, range, and in-list predicates can be applied directly on the codewords. Second, numerical columns with high cardinality are encoded by simply subtracting each value from a base and representing the difference as a compact integer of appropriate bit-size.

From the standpoint of this paper, the result of these optimizations is that accesses to the dictionary are done once at the start of a query, and introduce almost no I/O after that.

Blink stores its input tables in de-normalized fashion. During load, Blink joins tables of an input schema along primary key - foreign key relationships to form a de-normalized table, so that query execution just involves scans and aggregation over this table.

2.2.2 Query Processing

The query processing component of Blink operates over fixed-size *blocks* of tuples from the denormalized fact table. Query processing proceeds in three stages.

In the first stage, the query processor scans the denormalized table, accessing the columns referenced in the query's WHERE clause and applies selection predicates that can be applied directly over the codewords, without decoding.

In the second stage, the query processor applies any remaining predicates that need decoding (such as shipDate $-$ receiptDate $>$ 30), only over tuples that passed the first stage. For each tuple that passes this second stage, Blink computes a unique *group code* from the compressed values of the columns referenced in the query's GROUP BY clause.

The third stage of processing uses each group code as a key for an in-memory hash table called the *agg-table*. Each entry in the agg-table holds a pointer to an array of running aggregates, the array holding one value for each clause in the query's SELECT list. The agg-table is implemented via open addressing with linear probing, because this provide good cache locality and avoids the overhead of pointers that happens with chaining.

To take advantage of multi-core CPUs, Blink's query processor is heavily multi-threaded. Blink maintains a pool of worker threads, one thread per core. Each thread "picks up" one query at a time and runs the query, scanning blocks of compressed tuples. When there are more threads than queries, the idle threads "steal work", attaching themselves to queries that are already executing. These additional worker threads split the compressed table among themselves, executing the query in parallel. To avoid locking overhead, each thread maintains a *private* copy of the agg-table for each query that it executes. At query completion time, these agg-tables are merged, if necessary, to produce the final query result.

The query processing steps that Blink uses have memory access characteristics similar to the low-level operations in a conventional relational database. From the memory subsystem's perspective, reading compressed tuples from the denormalized fact table is similar to a conventional table or index scan. Likewise, dictionary lookups are analogous to index joins on primary key - foreign key relationships. Finally, Blink's agg-table is similar to the data structures most DBMSs use for hash-based aggregation. Because of these low-level similarities, we expect that our results on Blink should also apply to other main-memory query processors.

3. RELATED WORK

DBMSs have always aimed to share the results of I/O among concurrent tasks via the buffer manager. Many recent systems explicitly synchronize concurrent queries to improve the amount of I/O that can be shared at the buffer pool, by grouping together queries that run at similar speeds [16, 21]. Unlike previous systems, the sharing in main-memory DBMSs must be done in L2 cache and not in memory. As we have discussed, this buffer pool model does not lend itself well to the implementation within the L2 cache. If large shared L3 caches become common, this approach is more promising. The much smaller cache sizes (when compared to memory) means that the combined working set of the queries often fails to fit. The thrashing of the working set leads to significant I/O that competes with the table I/O that we are trying to share.

An alternative approach to sharing is a data-driven approach in which a single pipeline of tuples feeds multiple concurrent queries. This is typically done using an operator that is shared among multiple queries, such as in a staged database system [12] or a continuous query processor [15, 5]. Our FullSharing scheme is similar to this approach. Recently, Johnson et al. [13] discuss the tradeoffs of work sharing in a multi-core processor using shared operators. In particular, they obtain the same observation as we do from a different perspective. That is, work sharing does not always improve throughput in a multicore system. This is similar to our statement that FullSharing is not always beneficial. However, we focus on solving issues related to hardware resource constraints, e.g. cache contention, while they focus on solving throughput degradation due to serialization. Our results on estimating the working set size and batching queries appropriately are applicable to any shared operator in a staged database system.

Another popular technique for avoiding the I/O bottleneck is to lay out records of a table in a column-major order. Harizopoulos et. al [11] quantify the extent of I/O savings in this layout for queries that access various fractions of a table. Such a layout is complementary to the techniques described in this paper; for example, our BatchSharing method can cluster queries according to the vertical partitions of the data they access.

Finally, another approach to scaling on a multicore system is intra-query parallelism. Recent papers on this topic have proposed specialized hash-based aggregation algorithms to avoid lock contention across cores [7]. This paper focuses on throughput and inter-query parallelism. Each thread uses a private hash table to

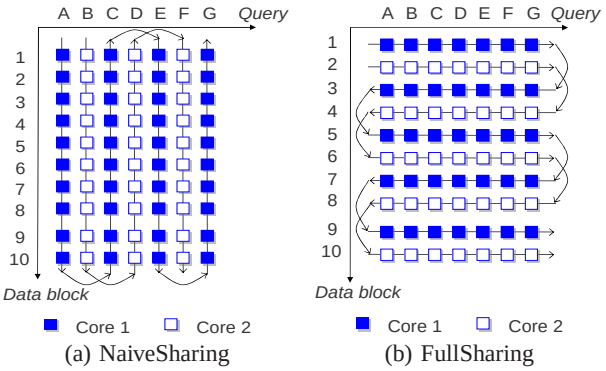


Figure 3: Illustration of the data access patterns of *NaiveSharing* and *FullSharing* on a two-core system. *FullSharing* inverts the traditional division of work among cores.

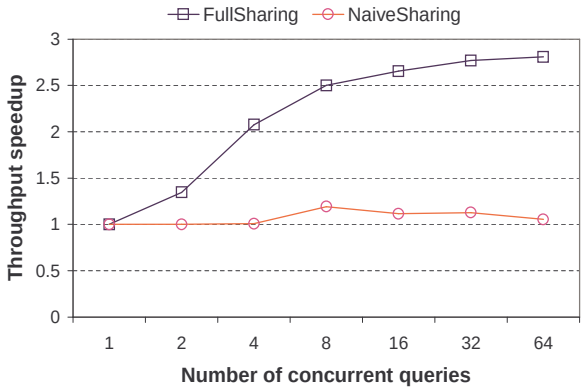


Figure 4: NaiveSharing vs FullSharing

store its running aggregates, so lock contention is avoided.

Recently, the OS community has begun to consider the problem of sharing a cache among multiple threads on chip multi-processors. Chang et al. [4] proposed a cooperative cache partitioning scheme to achieve high throughput using multiple threads. Kim et al. [14] addressed the issue of fairness in cache sharing. Chen et al. [6] presented a constructive cache sharing scheduler that allows threads to share an overlapping working set. These techniques are orthogonal to our work, and applying them in a shared cache can further improve throughput and fairness of our system.

4. SCAN-SHARING

Query processors that run concurrent queries usually operate in a multi-threaded fashion, where each thread handles a query at a time. When this model is applied to a main-memory, multicore system, each thread runs on a core and scans data from memory. The challenge of I/O sharing is to optimize the memory access so that the threads are always busy doing work, and are not bound by memory bandwidth. As we discussed in Section 2, main-memory DBMSs lack buffer pools, instead relying on hardware to read data into the processor's caches.

Even in the absence of a buffer pool, main-memory DBMS's can attain some speedup through "incidental" I/O sharing, which occurs because of the convoy phenomenon [3]. Consider the case when multiple queries, running on different cores, start scanning a table at approximately the same time. The first query will incur a cache miss to read each tuple from main memory. The remaining queries, however, can take advantage of the data that the "trailblazer" query has read into the processor's shared L2 or L3 cache. The queries form a "convoy" behind whichever query is furthest along in scanning the table; slower queries can catch up while faster queries wait for the memory controller to respond. Throughout the rest of this paper, we use the term *NaiveSharing* to describe the traditional multithreaded approach to scheduling query execution, which achieves limited I/O sharing via the convoy phenomenon.

4.1 FullSharing

In the rest of Section 4, we develop techniques that obtain significantly more I/O sharing — and hence better performance — than *NaiveSharing*. Our first technique is called *FullSharing*. Here, each processing thread executes a separate table scan. A given thread feeds each block of tuples through every query before moving on to the next block. Figures 3(a) and 3(b) show how *FullSharing*'s query scheduling contrasts with that of *NaiveSharing*. *FullSharing* inverts the traditional division of work within the database: instead of processing all blocks of an entire query at a time, each thread "processes" a block of data at a time across all queries.

The benefits of *FullSharing* over *NaiveSharing* are demonstrated in Figure 4. We constructed a batch query workload consisting of multiple copies of the query from Figure 1 and ran it on an 8-core server, first using *NaiveSharing* to schedule the 8 cores, and then using *FullSharing*. We repeated the experiment multiple times, varying the number of queries in the workload. For each run of the experiment, we compared overall throughput against the throughput of the one-query workload.

As the number of queries in the system increases, *FullSharing* is able to amortize memory I/O across the entire group of queries, more than doubling its query throughput. Beyond 4 concurrent queries, *NaiveSharing* achieves some speedup through I/O sharing. However, the speedup is negligible compared to that of *FullSharing*. Even though all the queries in the workload are identical and start at the same time, the convoy effect is not sufficient to induce effective sharing of memory I/O.

4.2 Implementing FullSharing

In Blink, *FullSharing* is implemented easily, with only modest code changes. Recall from Section 2.2 that Blink queries scan a compressed denormalized fact table. Blink divides this table horizontally into *blocks*, so that multiple cores can work on the same query simultanesouly. To support this intra-query parallelism, Blink has a global scheduler that tells each thread which queries to run over which blocks. Our implementation of *FullSharing* in Blink replaces this scheduler component, leaving all other portions of the system unchanged.

Our new scheduler works as follows: When we want to run a workload Q of queries, we create a pool of work-units, where each work-unit corresponds to a block. Each thread steals work from this pool as follows:

Repeat until the pool is empty:

- Pick a block from the pool of work-units.
- Scan this block.
- For every query $q \in Q$, apply q on this block.

4.3 Agg-Table Thrashing

The overall goal of scan-sharing in a main-memory DBMS is to reduce the number of cache misses. The *FullSharing* technique achieves this goal by loading tuples into cache once, then sharing them among multiple queries. However, applying *FullSharing* too aggressively can lead to *more* cache misses, due to an effect we call *agg-table thrashing*. In this subsection, we explain why agg-table

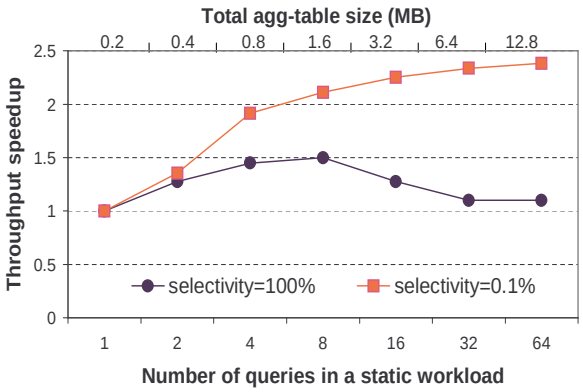


Figure 5: Performance improvement due to the *FullSharing* technique. Unless selectivity is below 0.1%, throughput degrades when the working set of the queries, combined with the current block of tuples, exceeds L2 cache size. We call this phenomenon *agg-table thrashing*.

thrashing occurs. In the sections that follow, we use this knowledge to develop scan-sharing techniques that are immune to the problem.

A query that scans a table typically streams the results of the scan into another operation, such as index nested-loops join, or grouped aggregation. To run efficiently, these operations require fast access to a "working set" of data structures, such as indexes or hash tables. If too many queries share a scan, their working sets can overflow the cache. Once this situation occurs, the queries start to thrash, incurring frequent cache misses to fetch portions of their working sets. The resulting accesses to main memory can easily negate the benefits of scan-sharing. The working set of a Blink query consists primarily of the agg-table data structure (see Section 2.2); hence, we use the name "agg-table thrashing" to describe this effect.

We have conducted detailed experiments to determine the conditions in which agg-table thrashing can occur; full results can be found in Section 6. To aid the discussion in the current section, we have created Figure 5, which shows a small subset of our results. The experiments behind Figure 5 used *FullSharing* to share a single scan between multiple copies of a given query. We varied the number of simultaneous queries from 1 to 64 and measured the resulting throughput improvement. The two lines in the graph show the performance improvement for two variants of the query in Figure 2. We produced these variants by modifying the WHERE clause of the query, changing the query selectivies to 100% and 0.1%, respectively. The high-selectivity query experiences agg-table thrashing, suffering a performance reduction when more than 8 queries run simultaneously.

Our experiments identified two factors that determine whether agg-table thrashing will occur: query selectivity and working set size. The results in Figure 5 illustrate these two factors. The effects of selectivity are most readily apparent: The high-selectivity query thrashes, while the low-selectivity query does not. Overall, we have found that queries with selectivities of 0.1% or less do not exhibit agg-table thrashing.

The effects of working set size can also be seen by focusing on points at which thrashing occurs: in the case of Figure 5, at all points beyond 8 queries. The agg-tables for the queries shown here take up 200KB of memory each. (Recall that, in the Blink query processor, the agg-table data structure is the dominant part of a query's working set.) Note the secondary scale across the top of the graph; this scale shows the total size of the agg-tables across all queries. Our test machine has two 4MB L2 caches, each split between two cores. Effectively, each core has 2MB of cache. The block size was 400K, leaving 1.6MB of space per core for the queries' working sets When the total agg-table size exceeds 1.6MB, the queries start to thrash. Our experiments have verified this result across queries with selectivities from 1 to 100 percent and agg-table sizes ranging from 30KB to 3.2MB.

To summarize, a scan-sharing technique must avoid agg-table thrashing in order to achieve the benefits of shared scans. The two factors that determine whether thrashing will occur are query selectivity and working set size. In the rest of Section 4, we use this knowledge to develop techniques to avoid thrashing.

4.4 BatchSharing

To achieve the benefits of scan-sharing without inducing agg-table thrashing, we have developed a technique that we call *BatchSharing*. The intuition behind *BatchSharing* is simple: Prevent thrashing by grouping together smaller numbers of queries into *batches*. However, making this intuition work in practice is difficult, because it is hard to determine whether a given set of queries can share a scan without thrashing.

In this section, we describe the components of the *BatchSharing* technique. We start by discussing the problem of determining which queries can safely share a scan. Then we describe the query parametrization algorithm that we use to solve this problem. Finally, we explain how we implement *BatchSharing* in Blink.

For ease of exposition, this section describes a "static" version of *BatchSharing*. That is, we assume that the system is executing a single workload of queries (as in a report-generation scenario) all at once. Further, we assume that the goal of the system is to finish this entire workload as quickly as possible without regard for the relative running times of individual queries. This scenario is analogous to running daily reporting queries over a data warehouse. Later, in Section 4.5, we will relax these assumptions and extend *BatchSharing* to handle dynamic query arrival while ensuring a fair division of system resources among queries.

4.4.1 Query Parameter Estimation

In Section 4.3, we examined the phenomenon of agg-table thrashing. Our analysis identified two factors that, taken together, can be used to predict whether a batch of queries will thrash. These factors are query selectivity and query working-set size.

For queries in Blink, the working set is dominated by its agg-table. In general, there is no known efficient (i.e., sampling-based) method to estimate *a priori* the number of rows in an agg-table—i.e., the number of groups that the query's GROUP BY clause produces—with guaranteed error bounds [10]. However, by carefully defining the estimation problem, we can sidestep these issues so that a sampling-based technique will meet our needs.

Our algorithm hinges on three key observations:

OBSERVATION 4.1: Based on our characterization of agg-table thrashing (Section 4.3), we can classify queries into 3 categories:

- **Always share:** If a query has low selectivity (<0.1% as shown in our experiments), it can be grouped with any other query without thrashing.
- **Never share:** If a query's working set size exceeds the size of cache, adding that query to *any* batch will lead to thrashing.
- **Could share:** If a query does not fit into the previous two categories, then the system needs to estimate the query's working set size to know whether it can be safely added to a given batch.

OBSERVATION 4.2: Some parts of a query's agg-table are accessed very rarely, while others are accessed frequently; thus the working set can be viewed, approximately, as the set of groups that are nec-

essary to account for almost every access to the query's agg-table (here "almost every" is a tunable parameter). If this working set resides in cache, thrashing will not occur.

OBSERVATION 4.3: It is easier to estimate a query's working set size from a sample than it is to estimate the size of its agg-table, because hard-to-capture rare values impact the distinct-value count but not the working-set size. Working-set size is closely related to the classical statistical notion of "sample coverage," and techniques for estimating sample coverage are applicable.

These observations allow us to convert a potentially hard estimation problem into a tractable one:

> First, identify queries with selectivities of less than 0.1%, as well as queries with working sets that clearly exceed the size of cache. Then, for the *remaining* queries, estimate the working-set size.

In Section 5, we describe our query-parameter estimation algorithm in detail. For now, we give a synopsis of the algorithm and an intuition for how it works.

Our algorithm operates using two phases of sampling. Each phase operates over preallocated random samples of the table being scanned. The first phase identifies queries in the "always share" category. This phase proceeds by running the query over a sample of the table. If very few tuples pass the query's selection predicate, the query is marked as "always share." This phase works well because it is relatively easy to estimate predicate selectivities on the order of 0.1% or higher from a sample.

During its second phase, the algorithm feeds a sample of the table through the query while monitoring the number of distinct groups encountered thus far. The algorithm stops either when the groups encountered thus far account for almost every access to the agg-table (as measured by sample coverage) or when the groups encountered thus far would not fit into cache. In the latter case, the query is classified as "never share," whereas in the former case, the algorithm returns the number of groups encountered thus far as its estimate of the working-set size. This phase works well because the coverage estimator that we employ is accurate as long as the actual number of groups in the working set is sufficiently small relative to the number of tuples in the sample. By definition, every "could share" query meets this criterion, because a processor cache can only hold roughly 10,000 agg-table entries.

After the two phases of sampling, the algorithm has collected enough information to decide which queries can be safely batched together. In practice, we can obtain sufficiently accurate results for both phases with sample sizes of less than 100,000 tuples. Even when running a highly complex query, the Blink query processor can scan such a small sample in less than 5 msec.

4.4.2 Packing queries into batches

The result of the above estimation procedure is a quantification of the working set size w_q for each query q that the system needs to assign to a batch. For "always share" queries, this working set size is effectively zero; for "never share" queries, the working set size is effectively infinite. The next stage of *BatchSharing* uses this working set information to pack the queries into batches.

The goal of the packing algorithm is to minimize per-batch overheads by packing the queries into as few batches as possible; while avoiding agg-table thrashing. To prevent thrashing, we ensure that there is enough space in the cache for the working set of every query in a given batch. That is, if C is the size of the cache and B is the size of a block of data, then we guarantee that the queries in a batch have a total working set size of less than $C - B$.

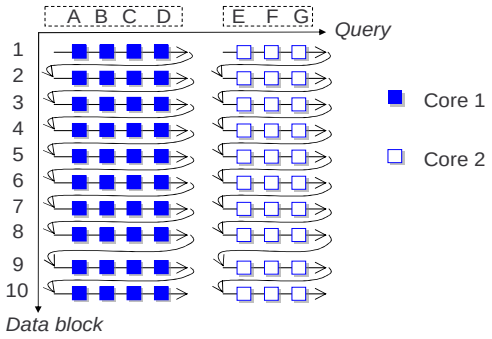


Figure 6: Query execution with *BatchSharing* on a two-core processor, running two batches; each core shares a single scan among the queries from one batch. If one core finishes its batch before the other, the idle core will steal work from the remaining batch.

This constraint is based on a conservative model of cache behavior. Let γ denote the fraction of memory accesses covered by each query's working set. We assume that the cache controller will keep the most popular γth percentile of memory in cache for each query. As long as this invariant holds, the overall cache miss rate across the queries in the batch is bounded from above by $1 - \gamma$. In reality, the controller will use a global replacement policy to allocate cache lines across all queries in a batch; this actual policy will achieve a lower miss rate than the simplified policy we assume.

More formally, our packing problem is: Given a set Q of queries and corresponding working set sizes w_q, find a partitioning:

$$Q = Q_1 \uplus Q_2 \uplus \cdots \uplus Q_p, \tag{1}$$

that minimizes p, subject to the constraint:

$$\sum_{q \in Q_i} w_q \leq C - B, \ \ \forall\, 1 \leq i \leq p, \tag{2}$$

where C is the size of the cache and B is the size of a block of tuples (Agg-table thrashing occurs when the total working set of the queries in a batch is greater than $C - B$ bytes).

This problem is identical to the standard bin-packing problem, with bin size $C - B$. We use the well known first-fit decreasing heuristic, which sorts queries by decreasing w_q and repeatedly packs a query into the first batch with sufficient space, starting a new batch if none is found. This heuristic is known to pack no worse than a factor of $11/9$ times the optimal solution [8].

4.4.3 Execution

We have implemented *BatchSharing* on the Blink query processor. As with our implementation of *FullSharing*, all of our changes to Blink were concentrated in the query scheduler. The general scheduling algorithm that we use is illustrated in Figure 6. The scheduler assigns a separate batch of queries to each core in the processor. Each core scans the table, feeding each block of tuples through its batch of queries.

Since *BatchSharing* assigns queries to batches purely according to their agg-table sizes, the batches can be very heterogeneous. As a result, the running times of different batches can vary significantly, and the division of work among cores can be uneven. To prevent cores from being idle, our implementation of *BatchSharing* uses the work-stealing features already present in Blink.

Recall from Section 2.2 that Blink's conventional query scheduler detects idle cores and assigns them to "help out" other cores

that have been assigned expensive queries. Our implementation of *FullSharing* disables this feature, since *FullSharing* naturally levels the load across threads. With *BatchSharing*, we re-enable work-stealing. If a thread finishes its batch of queries before the other threads, the thread can steal work (table blocks) from another batch. Thus, multiple threads work concurrently on the expensive batches, automatically achieving a balanced load.

4.5 Dynamic Query Grouping

The description of *BatchSharing* in the previous section assumed a single static workload of queries. In this section, we extend the technique to handle an online environment with dynamic query arrival, as in a data warehouse supporting a stream of analyst queries.

We still want to run queries in batches, with the combined working set of each batch fitting in the L2 cache to avoid agg-table thrashing. The basic methods from the previous section on estimating the agg-table size of each query and on packing queries into batches still apply. But we need to form and maintain batches for a dynamic query stream. We have two choices in how we form and maintain batches of queries for a dynamic query stream:

- If a batch X of queries is running and a new query q arrives, add q to X if the working sets of X and q together fit in cache.
- Once a batch of queries has started running, treat it as immutable.

The first option could potentially give higher throughput, but requires additional book-keeping to track which queries in a batch have operated on which blocks. To keep our design simple, we choose the second option.

Our dynamic approach to *BatchSharing* works as follows: At any point in time, the queries in the system fall into two categories: *active* and *unassigned*. Active queries have been assigned to query batches; these *active batches* are in the process of being executed over shared scans. Unassigned queries are not yet part of a batch; these queries reside in a special staging area until they are assigned to a batch.

Dynamic workloads arise primarily in interactive applications, with concurrent users submitting queries from their individual consoles. It is important for these users to see consistent query response times. To function correctly in such an environment, a query processor must schedule queries fairly and avoid starvation. Our dynamic *BatchSharing* implementation targets two kinds of fairness:

- **Fair scheduling:** On average, every active query receives an equal fraction of CPU time to within a constant multiplicative factor d.
- **No starvation:** As long as the system is not overloaded, the amount of time that a query can be in the unassigned state is strictly bounded.

4.5.1 Scheduling Queries Fairly

Since the queries in a given batch share a scan, it follows that every query in the batch must complete at the same time. Should a batch contain both fast and slow queries, the faster queries will receive a smaller slice of the CPU, violating our fair scheduling goal. To avoid this problem, we incorporate constraints on query running time into our bin-packing algorithm. A given pair of queries are allowed to share a batch only if their running times differ by a factor of less than d. We choose d experimentally in Section 6.4. Since we do table scans, query running times can be easily estimated from running the query on a sample.

Another obstacle to fairness is the relative weight of different batches in scheduling the activities of the CPU's cores. If two batches of unequal size receive equal slices of CPU time, the queries in the smaller batch will receive a greater share of CPU. To avoid such inbalances, we ensure that each batch receives an amount of CPU time proportional to its size.

We use lottery scheduling [20] to implement this allottment of CPU time. Each running batch receives a number of *lottery tickets* proportional to the number of queries in the batch. We store the mapping from tickets to batches in an array, where each entry represents a single ticket. We divide time into slices that are sufficiently large to amortize the overhead of flushing the processor's L2 cache. At the start of each time slice, every core chooses a lottery ticket uniformly at random and executes the corresponding batch for the remainder of the time slice. Overall, the expected amount of CPU time that each batch receives is proportional to its number of tickets, and hence to its number of queries.

4.5.2 Avoiding Starvation

To prevent starvation, our implementation of *BatchSharing* enforces an upper bound t_w on the amount of time a query can be in the unassigned state. At the same time, we want to keep queries in the staging area as long as possible, so as to maximize the opportunities for effective bin-packing. We achieve a compromise between these two factors by tracking the original arrival time of each unassigned query.

During query processing, the staging area is left untouched until one of the following occurs:

- No more active queries remain, or
- A query has spent more than t_w time in the staging area.

When either of these events happens, it triggers the following sequence of actions:

1. Pack all the unassigned queries into batches.

2. Activate any batch containing a query that has spent more than t_w time in the pool.

3. Activate a batch if there are still no active batches.

4. Return the remaining queries to the staging area.

5. WORKING-SET SIZE ESTIMATION

As discussed in Section 4.4.1, we need to classify queries as always-share, never-share, or could-share and, for the could-share queries, estimate the working-set (WS) size. We now describe our sampling-based classification and estimation algorithm.

We first introduce some notation. For a specified query q and real number $\gamma \in [0, 1]$, a *working set* $W_\gamma(q)$ is defined as a minimal set of rows in the agg-table—not necessarily unique—that accounts for $100\gamma\%$ of rows in the answer to q after predicates have been applied but prior to grouping. I.e., if the cache comprises the rows in $W_\gamma(q)$, then the cache-hit rate for query q (in isolation) will be $100\gamma\%$. (We use a value of $\gamma = 0.8$ in our prototype.) Given a value of γ, we wish to (1) classify a query as always-share if its selectivity σ is less than a threshold σ^*, (2) classify a query as never-share if the WS size will clearly exceed the space threshold $d^* = B - C$ allotted for the agg-tables, and (3) otherwise compute $|W_\gamma(q)|$ for purposes of bin packing.

To avoid expensive table scans, we sample the table T of interest, and merely estimate σ and $|W_\gamma(q)|$. The idea is to execute the classification steps (1)–(3) above, but modify each step to take into account the uncertainty due to sampling, using an "indifference-zone" approach, which we now describe. Set $x_i = 1$ if the ith row of T satisfies the predicates in q, and $x_i = 0$ otherwise, so that $\sigma = (1/|T|) \sum_{i=1}^{|T|} x_i$. Also set $\alpha^2 = (1/|T|) \sum_{i=1}^{|T|} (x_i - \sigma)^2$.

To determine whether $\sigma < \sigma^*$, we take a simple random sample of n rows from table T and apply the predicates in q to the sampled rows. Set $X_i = 1$ if the ith sampled row satisfies the predicates in q, and $X_i = 0$ otherwise. We then estimate σ by $\hat{\sigma}_n =$

$(1/n)\sum_{i=1}^{n} X_i$, and classify q as always-share if $\hat{\sigma}_n < \sigma^* - \epsilon_n$. The formulas for n and ϵ_n are given below, and are chosen so that the probability of a "type-1" or "type-2" error is less than a user-specified threshold p. A type-1 error occurs if $\sigma > \sigma^* + \delta_2$ but $\hat{\sigma} < \sigma^* - \epsilon_n$, where δ_2 is an "indifference" constant. That is, a type-1 error occurs if σ lies "significantly" above σ^*, as measured by δ_2, but our procedure, which uses $\hat{\sigma}$, incorrectly classifies query q as always-share. Similarly, a type-2 error occurs if $\sigma < \sigma^* - \delta_1$ but $\hat{\sigma} > \sigma^* - \epsilon_n$. If σ lies in the interval $[\sigma^* - \delta_1, \sigma^* + \delta_2]$, then we can tolerate a misclassification. In general, the repercussions of a type-1 error are much more serious than those of a type-2 error. Based on preliminary experiments, we found that suitable values of the foregoing constants are given by $\sigma^* = 0.001$, $\delta_1 = \sigma^*$, and $\delta_2 = 0.099$.

Specifically, we set

$$n = \left(\frac{2\alpha z_{1-p}}{\delta_1}\right)^2 \vee n_{\min},$$

and

$$\epsilon_n = \left(\frac{\alpha z_{1-p}}{\sqrt{n}} - \delta_2\right)^+$$

where $n_{\min} \approx 500$, z_x is the $100x\%$ quantile of the standard (mean 0, variance 1) normal distribution, $x \vee y = \max(x, y)$, and $x^+ = \max(x, 0)$. Note that the constant α appearing in the above formulas is unknown; in practice we use a small pilot sample of size $m = n_{\min}$ to estimate α by $\hat{\alpha}_m^2 = (m-1)^{-1}\sum_{i=1}^{m}(X_i - \hat{\sigma}_m)^2$. To see that use of the foregoing values achieves (approximately) the desired error control, observe that

$$\begin{aligned}
&P\{\text{ type-1 error }\} = P\{\,\hat{\sigma}_n < \sigma^* - \epsilon_n\,\} \\
&= P\left\{\frac{\hat{\sigma}_n - \sigma}{\alpha/\sqrt{n}} < \frac{\sigma^* - \sigma}{\alpha/\sqrt{n}} - \frac{\epsilon_n}{\alpha/\sqrt{n}}\right\} \\
&\le P\left\{\frac{\hat{\sigma}_n - \sigma}{\alpha/\sqrt{n}} < \frac{-\delta_2}{\alpha/\sqrt{n}} - \left(z_{1-p} - \frac{\delta_2}{\alpha/\sqrt{n}}\right)^+\right\} \\
&\le P\left\{\frac{\hat{\sigma}_n - \sigma}{\alpha/\sqrt{n}} < -z_{1-p}\right\} \approx p,
\end{aligned}$$

where the last $\approx$ follows from the central limit theorem (CLT) and the definition of z_{1-p}. (Our choice of $n_{\min}$ attempts to ensure the accuracy of the CLT approximation.) Similarly,

$$\begin{aligned}
&P\{\text{ type-2 error }\} = P\{\,\hat{\sigma}_n > \sigma^* - \epsilon_n\,\} \\
&= P\left\{\frac{\hat{\sigma}_n - \sigma}{\alpha/\sqrt{n}} > \frac{\sigma^* - \sigma}{\alpha/\sqrt{n}} - \frac{\epsilon_n}{\alpha/\sqrt{n}}\right\} \\
&\le P\left\{\frac{\hat{\sigma}_n - \sigma}{\alpha/\sqrt{n}} > \frac{\delta_1}{\alpha/\sqrt{n}} - z_{1-p}\right\} \\
&\le P\left\{\frac{\hat{\sigma}_n - \sigma}{\alpha/\sqrt{n}} > z_{1-p}\right\} \approx p,
\end{aligned}$$

To obtain a reasonable estimate of the working-set size for a query q, we incrementally maintain a uniform multiset sample W of grouping values by incrementally sampling table T; for each sampled tuple, we apply all of the predicates in q and, if the tuple survives, project it onto the grouping attributes before adding it to W. After each incremental sampling step, we estimate the *coverage* V of the set $D(W)$ of distinct grouping values in W. Denoting by T^* the reduced version of T obtained by applying the selection predicates in q, we define the coverage as $\sum_{i \in D(W)} \pi_i$, where π_i is the fraction of rows in T^* whose grouping values match the ith value in $D(W)$; see [9] for a discussion of coverage. As soon as $V \ge \gamma$, we stop the sampling process and use the number of rows in W as the estimate of the working-set size. The idea is that the most frequent grouping values will appear in W, so that W will be approximately minimal; more elaborate approaches are possible, but experiments indicate that our proposed technique is adequate for our task. As with query selectivity, the test of whether or not $V \ge \gamma$ is modified to take into account the uncertainty introduced by sampling, using an indifference-zone approach.

In more detail, when W contains n elements, we estimate the coverage V by $\hat{V}_n = 1 - f_1/n$, where f_j $[1 \le j \le |D(W)|]$ is the number of distinct grouping values that appear exactly j times in W; see [9] for a discussion of this estimator, which is originally credited to Turing. Choose an indifference zone of the form $[\gamma - \delta_1', \gamma + \delta_2']$ and set

$$n' = \left(\frac{2\beta_n z_{1-p}}{\delta_2'}\right)^2 \vee n_{\min},$$

and

$$\epsilon_n' = \left(\frac{\beta_n z_{1-p}}{\sqrt{n}} - \delta_1'\right)^+,$$

where $\beta_n = (f_1/n) + 2(f_2/n) - (f_1/n)^2$. Then, provided that $|W| \ge n'$, declare that $V \ge \gamma$ if and only if $\hat{V}_{|W|} > \gamma + \epsilon_{|W|}'$. An argument similar to the one given above shows that, to a good approximation, the probability of a type-1 or type-2 error will be at most p. (The key difference from the prior argument is that we use a CLT for the coverage estimator due to Esty [9], rather than the standard CLT.) In our prototype, we use indifference-zone values of $\delta_1' = 0.05$ and $\delta_2' = 0.10$.

The overall technique is given as Algorithm 1. In the algorithm, the function DISTINCT(W) computes the number of distinct elements in W, and NUMWITHFREQ(W, i) computes the quantity f_i defined previously. The function SAMPLE(T, n) takes a simple random sample of n rows from table T, without replacement. The function INCREMENTSAMPLE(W, T, i) repeatedly samples from T until a sampled tuple survives the predicates in q. This tuple is then projected onto the grouping attributes and added to W. The sampling from T is incremental within and between function calls; the variable i records the cumulative number of tuples that have been sampled from T over all calls to INCREMENTSAMPLE.

We achieve efficiency by precomputing a sample T' of $100k$ rows, and storing them in random order, so that incremental sampling of T corresponds to a simple scan of T'. We set $n_{\max} = |T'|$, so that if the sample becomes exhausted at any point (lines 14 and 25), the algorithm terminates and conservatively categorizes query q as never-share. In practice, the same sample T' can be used for both the selectivity test (pilot and regular samples) and the WS size-estimation phase, without much adverse impact on the effectiveness of BatchSharing. Finally, note that, in line 26, DISTINCT(W) is essentially a lower bound on the size of the working set, so that the test in line 26 indeed identifies whether q is a never-share query.

6. EXPERIMENTS

We now present a detailed performance evalution of our scan-sharing algorithms. The evaluation proceeds in roughly the same order that we have described the algorithms:

Extent of Thrashing: Early in this paper we introduced a fairly simple algorithm, FullSharing, and then we argued that it leads to thrashing of agg-tables. Does it? How bad is the thrashing (Section 6.1)? Can we avoid this thrashing just by adding a simple admission-control scheme to FullSharing (Section 6.2)?

Mixed Workloads: BatchSharing is much more fancy than Full-

Algorithm 1 Query classification and WS size estimation

T, q: table and query under consideration
σ^*, γ: selectivity and WS-size cutoff values
$d^* = C - B$: agg-table threshold for never-share
$\delta_1, \delta_2, \delta_1', \delta_2'$: indifference-zone values
p: maximum allowed error probability
$n_{\min}$, $n_{\max}$: minimum and maximum sample sizes

// test selectivity
$m \leftarrow n_{\min}$
$T' \leftarrow \text{SAMPLE}(T, m)$ *// take pilot sample*
$\hat{\sigma} \leftarrow m^{-1} \sum_{i=1}^{m} X_i$
$\hat{\alpha} \leftarrow \left((m-1)^{-1} \sum_{i=1}^{m} (X_i - \hat{\sigma})^2\right)^{1/2}$
$n \leftarrow (2\hat{\alpha} z_{1-p}/\delta_1)^2 \vee n_{\min}$
if $n > n_{\max}$ **then return** "never-share"
$T' \leftarrow \text{SAMPLE}(T, n)$ *// take actual sample*
$\hat{\sigma} \leftarrow n^{-1} \sum_{i=1}^{n} X_i$
$\hat{\alpha} \leftarrow \left((n-1)^{-1} \sum_{i=1}^{n} (X_i - \hat{\sigma})^2\right)^{1/2}$
$\epsilon \leftarrow \left((\hat{\alpha} z_{1-p} n^{-1/2}) - \delta_2\right)^+$
if $\hat{\sigma} < \sigma^* - \epsilon$ **then return** "always-share" *// selectivity test*

// estimate WS size
$W \leftarrow \emptyset$; $i \leftarrow 0$ *// initialize*
while true do
  $\text{INCREMENTSAMPLE}(W, T, q, i)$
  if $i > n_{\max}$ **then return** "never-share"
  if $\text{DISTINCT}(W) > d^*$ **then return** "never-share"
  $f_1 \leftarrow \text{NUMWITHFREQ}(W, 1)$
  $f_2 \leftarrow \text{NUMWITHFREQ}(W, 2)$
  $\beta \leftarrow \left((f_1/n) + 2(f_2/n) - (f_1/n)^2\right)^{1/2}$
  $n' \leftarrow (2\beta z_{1-p}/\delta_2')^2 \vee n_{\min}$
  if $|W| \geq n'$ **then** *// is $|W|$ big enough for testing?*
    $\hat{V} \leftarrow 1 - (f_1/|W|)$
    $\epsilon' \leftarrow \left((\beta z_{1-p} n^{-1/2}) - \delta_1'\right)^+$
    if $\hat{V} > \gamma + \epsilon'$ **then return** $\text{DISTINCT}(W)$
  end if
end while

Sharing. Focusing purely on throughput, how good is it? Especially, what happens with a mixed workload of queries with small and large working sets? Is BatchSharing able to correctly batch the queries (Section 6.2)?

Interaction of Selectivity with Working Set: How does selectivity impact the thrashing of agg-tables? Were we right to model queries with low selectivity as not contributing to the working set of a batch (Section 6.3)?

Fairness: How well does BatchSharing maintain fairness among concurrent queries (Section 6.4)?

Putting it all together: We started this paper with a problem of scaling on a multicore system. Have we solved this problem (Section 6.5)?

Setup: Our test machine has two Intel Xeon quad-core CPUs (2.66 GHz/core). The memory hierarchy is 16GB of RAM, 4MB L2 cache (shared between two cores), and 64KB each of L1 data cache and instruction cache per core.

We use an actual customer dataset, CUST1 for our experiments. The denormalized table has 28M rows, and takes up 25GB uncompressed in a traditional commerical DBMS. Blink's compression reduces this to 2.8GB, an amount that comfortably fits in memory.

The queries used in our experiments have the following template (all column names are anonymized):
SELECT SUM(col_1), ... SUM(col_{12}) FROM table
WHERE conjunction of single column predicates
GROUP BY grouping cols

6.1 Tackling Agg-Table Thrashing

For our first experiment, we ran workloads with varying numbers of queries but homogeneous agg-table sizes. All the queries in a given workload have the same GROUP-BY clause, chosen so as to achieve a specific size for the query agg-table. The predicates were chosen to be non-selective (about 98% selectivity), so that almost all tuples participate in aggregation.

Figure 7(a) plots the throughput of each workload: increasing the number of queries on the x-axis, with a separate curve for each agg-table size. BS refers to BatchSharing and FS to FullSharing. For instance, "BS-50KB" is BatchSharing for queries with 50KB agg-table size. The total agg-table size for each workload is (number of queries) $\times$ (query agg-table size).

Observe that the throughput using FullSharing (the dotted curves) starts to drop when the total working-set size exceeds 1.6MB, which maps well to the 2MB L2 cache available per core (the remainder holds one block of the table being scanned). BatchSharing, on the other hand, does not exhibit any such thrashing. We have noted from logs that it behaves like FullSharing up to the thrashing point, and starts partitioning queries into batches thereafter.

Observe also that the throughput of BatchSharing does plateau, but at different numbers of queries for different agg-table sizes: about 16 queries for 800KB agg-tables, about 64 for 200KB agg-tables. By turning on logging, we have found that this corresponds to the point when the queries are partitioned into 8 batches: at this point each core gets its own batch and we cannot improve performance any further.

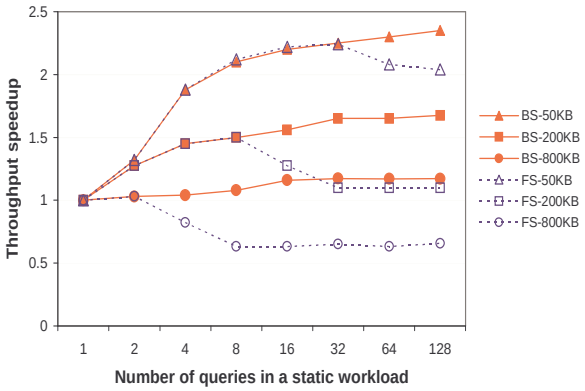
Figure 7(b) plots a more detailed version of the results for the same experiment, showing more query agg-table sizes. The x-axis shows the total agg-table size of the workload: notice that throughput keeps increasing well beyond the 1.6MB point at which FullSharing peaked.

6.2 Mixed workloads

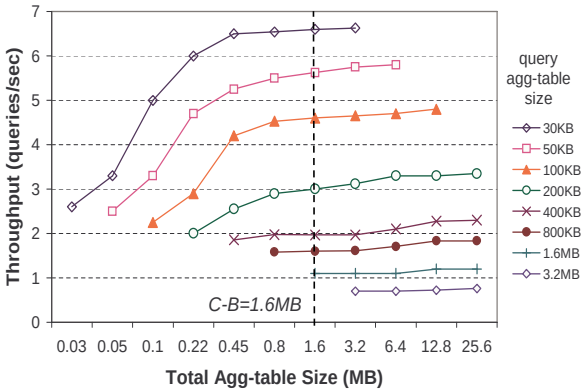
The homogeneous workload used in the last experiment is becoming less and less common as users consolidate different applications against the same DBMS. For example, queries whose results are to be shown to a human being usually have a small number of groups and hence small agg-table sizes, since they must fit on a screen, whereas drill-down OLAP queries involve fine-granularity grouping clauses that lead to large agg-table sizes. In real-world scenarios, the ratio of small-result queries to large-result queries varies across systems and over time.

So a natural question is whether BatchSharing is able to handle such mixed workloads. To test BatchSharing's robustness, we generate mixed workloads with queries of two agg-table sizes: 1.6MB and 50KB. Each workload has 100 queries, with the ratio of large to small varying from 1:1 to 1:32.

This experiment also tests another hypothesis. Based on the results in Figure 7(a), one might think that FullSharing can be fixed just by adding an admission control scheme that permits n concurrent queries at a time. Call such a scheme $FullSharing$-n. We show a full comparison among NaiveSharing, FullSharing, FullSharing-2, FullSharing-4, and BatchSharing in Figure 8. Observe that none of the FullSharing schemes with admission control is always better than the others. So there is no magic admission control value for FullSharing. Moreover, FullSharing with



(a) Comparing FullSharing with BatchSharing



(b) No Agg-table Thrashing using BatchSharing

Figure 7: BatchSharing Throughput

admission control is not a replacement for BatchSharing, because BatchSharing always outperforms such a scheme.

The other important point from Figure 8 is that the benefit of BatchSharing varies when the query ratio changes. The throughput improvement of BatchSharing decreases when there is a larger proportion of queries with a large working set. In practice, a BI system usually runs many more analytical queries than reporting queries; such a workload favors BatchSharing. Overall, BatchSharing outperforms other sharing schemes by up to a factor of 2.5.

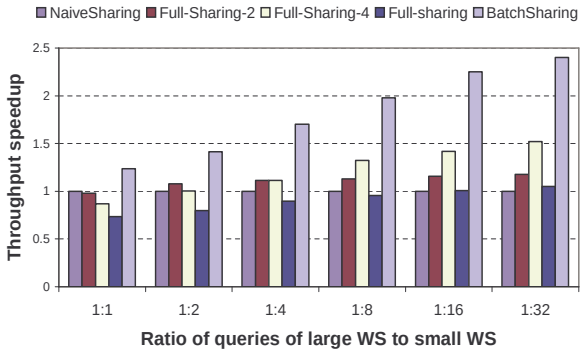


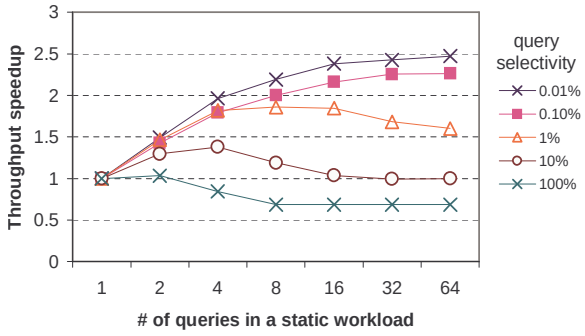
Figure 8: Throughput Speedup in a Mixed Query Workload

6.3 Impact of Selectivity on Thrashing

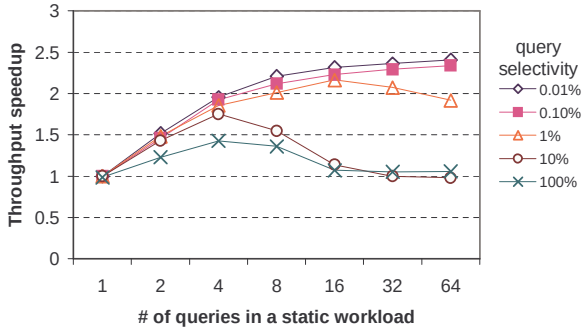
Our next goal is to validate the assumption in Section 5 that queries with low selectivity do not contribute to thrashing and can be ignored while estimating the working set size of a batch. We run queries with selectivity varying from 100% from 0.01%, using FullSharing. FullSharing does no batching, so this shows us the extent of agg-table thrashing at various selectivities.

Query selectivity	agg-table size in Fig. 9(a)	agg-table size in Fig. 9(b)
100%	0.9MB	0.22MB
10%	0.9MB	0.22MB
1%	0.84MB	0.22MB
0.1%	0.61MB	0.21MB
0.01%	0.19MB	0.12MB

Table 2: Working-set sizes in Figure 9



(a) 0.9MB working-set



(b) 0.2MB working-set

Figure 9: FullSharing vs query selectivity

Figure 9 shows the throughput speedup for queries under two situations: in Figure 9(a), the agg-table size before predicates are applied is 0.9MB, and in Figure 9(b) it is 0.2MB. But this size only applies to the 100% selectivity queries. When the WHERE clause is changed to reduce the selectivity, the agg-table size reduces because some groups get no tuples. Table 2 lists the agg-table sizes for various selectivities.

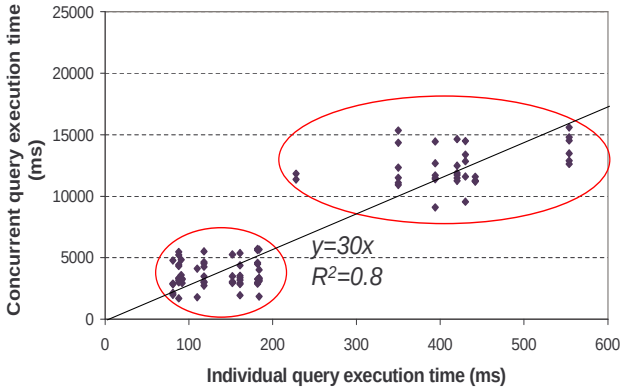
Observe that for the queries with 1% to 100% selectivity, the agg-table size does not differ much. But the peak throughputs are very different: e.g, in Figure 9(a), throughput peaks with 4 queries at 10% selectivity and with 2 queries with 100% selectivity. This phenomenon arises because queries with low selectivity do very few agg-table IOs for each base table I/O: most tuples fail the predicate and never reach the aggregation stage. So the extent of agg-table thrashing is low for such queries; at selectivities below 0.1%, the agg-table thrashing is negligible.

6.4 Fairness and Starvation Avoidance Using Dynamic Grouping Algorithm

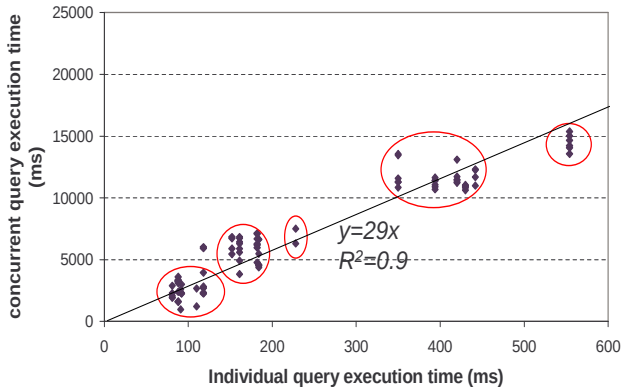
We now turn our attention from throughput to fairness and starvation avoidance. Instead of a static workload, we generate queries on the fly at an arrival rate of 6 queries/second and feed this stream into the dynamic BatchSharing scheduler. Queries are randomly generated, varying two parameters: the predicates are varied to achieve selectivities ranging from 0.01% to 98%, and the group-by columns are varied to achieve agg-table sizes ranging from 0.2MB to 0.9MB. This ensures a mix of: short-running queries (with low selectivity and small agg-table sizes), long-running queries (with high selectivity and large agg-table sizes), and medium queries (the rest). The individual query execution times vary from 81 milliseconds to 554 milliseconds.

As described in Section 4.5, to achieve fair scheduling, we use a multiplicative factor d to bound the (estimated) the execution times of queries within a batch. We use this experiment to study the sensitivity to the value of d and pick a good number. We report both fairness and throughput in the following.

6.4.1 Fairness and Starvation Avoidance



(a) $d = 2$



(b) $d = 1.25$

Figure 10: Fairness Using Dynamic Scan-sharing

To measure fairness, we look at the ratio of execution times for a query, when running concurrently vs when running standalone: if these ratios are identical for all queries we have perfect justice for all. Figure 10 is a scatter plot of individual (x-axis) and concurrent (y-axis) query execution time. Each point represents a query. To help visualize the degree of fairness, we also add a trendline, based on a linear regression fit passing through the origin. The equation of the regression line is displayed in each figure, along with the R^2 goodness-of-fit statistic. (Values of R^2 between 0.8 and 1 indicate a good fit.)

Figure 10(a) shows the result with $d = 2$. Notice that the data points are naturally clustered into two groups. Queries from this workload form two types of batches: one for short running queries and the other for long running queries. Across batches, the lottery scheduler ensures that the batches with short queries get the same share of the CPU as the batches with long queries. Figure 10(b) shows the result with a smaller $d = 1.25$. The data points form more clusters when compared to Figure 10(a). Moreover, the data points are more closely clustered around the trendline, with a R^2 of 0.9, compared with $R^2 = 0.8$ for $d = 2$.

At the same time, for both $d = 2$ and $d = 1.25$, all the queries are present in the plot, so no starvation occurs.

Overall, the results are as we would expect. Smaller values of d result in more fairness (values tightly clustered around a line), but both values of d avoid starvation.

6.4.2 Throughput

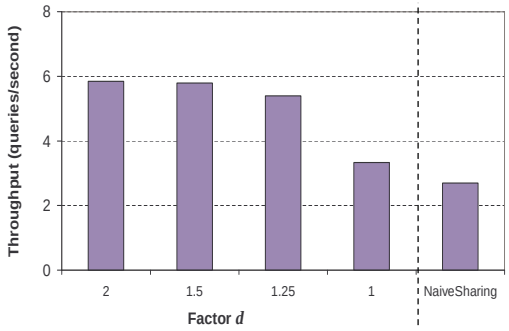


Figure 11: Dynamic Grouping Throughput vs Query Range $(1/d, d)$

We choose a good value of d based on throughput. Figure 11 reports throughput for dynamic grouping algorithm with d varying between 1 and 2. For the dynamic grouping algorithm, the system throughput decreases with smaller d, because that leads to a smaller pool of queries that are available to form a batch. However, throughput only differs 12% between $d = 2$ and $d = 1.25$, whereas it differs 50% between $d = 2$ and $d = 1$. So, we pick $d = 1.25$ as a sweet spot to reach a balance between throughput and fairness.

The same plot also shows a variant of NaiveSharing which takes a query stream as input. Comparing dynamic grouping with $d = 1.25$ to NaiveSharing, we see that the system throughput of the former is 2x of that of the latter.

6.5 Multi-core Scaling

Our last experiment repeats the query from Figure 2 of the introduction, but using BatchSharing. Our goal is to see if the I/O bottleneck has been removed.

Using the *oprofile* system profiler and Xeon hardware counters, we break the total CPU cycles into 5 components (Figure 12): computation, pipeline stall due to branch misprediction, L2 cache hit, DTLB miss, and resource stall due to memory loads (using Xeon counters: CPU_CLK_UNHALTED, RS_UOPS_DISPATCHED_NONE, MEM_LOAD_RETIRED (with mask 0x01 and 0x04), DTLB_MISSES, and RESOURCE_STALLS (with mask 0x10 and 0xf)).

The plot compares NaiveSharing and BatchSharing, run using 1 core and using 8 cores. With NaiveSharing on 8 cores much of

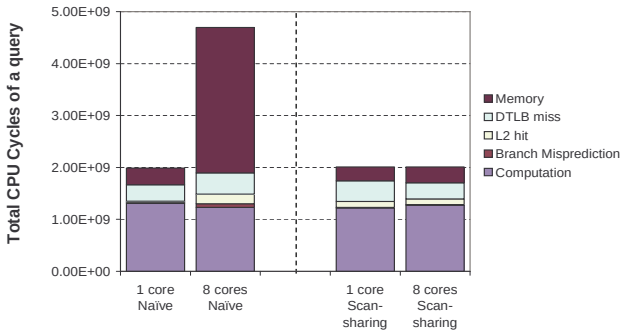


Figure 12: CPU breakdown of BatchSharing

the total CPU cycles go to memory access, because the bandwidth is saturated with all 8 cores issuing loads. For BatchSharing, the portion spent on memory access remains almost the same in going from 1 to 8 cores. BatchSharing also has more L2 cache hits than NaiveSharing, due to sharing of base table IOs.

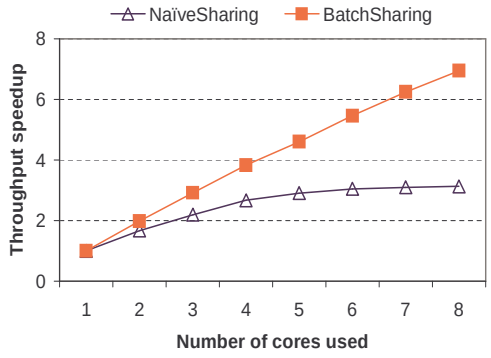


Figure 13: Performance scaling on multi-cores

Figure 13 plots the speedup in throughput as a function of the number of cores. NaiveSharing increases sub-linearly with increasing number of cores, saturating at 2.9. BatchSharing scales almost linearly, to a factor of 7 on 8 cores. Certain phases of query processing are not parallelizable in our system, such as query parsing and compilation, and the merging of partial agg-tables into a global agg-table (as described in Section 2.2). This explains why use of BatchSharing results in a speedup factor of 7, and not 8, for 8 cores.

7. CONCLUSION

The stories we have been hearing about the multicore trend have so far been mostly negative: clock speeds are decelerating, we have to write parallel programs, parallel programs do not scale easily, and enterprise software will perform poorly. The results of this paper suggest that the situation may not be so dire in all cases. In the context of a compessed database, we have shown a solution that achieves near-linear speedup of query throughput when running an 8-query workload on a server with 8 cores.

Looking forward, a board with two quad-core processors is a far cry from the GPUs with 100s of cores that are available today, or the CPUs with dozens of cores that are on the horizon. With such aggressively multicore architectures, the amount of cache available per core will decrease. An interesting direction for future work is to design aggregation tables that are tightly compressed, so that they fit in small caches, yet are efficiently updateable.

8. REFERENCES

[1] *Intel Architecture Software Developers Manual*, volume 2.

[2] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[3] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, 1979.

[4] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of Supercomputing (SC)*, pages 242–252, 2007.

[5] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[6] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of SPAA*, pages 105–115, 2007.

[7] J. Cieslewicz and K. A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *VLDB*, pages 339–350, 2007.

[8] G. Dosa. The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I)=(11/9)OPT(I)+6/9. In *ESCAPE*, 2007.

[9] W. W. Esty. A normal limit law for a nonparametric estimator of the coverage of a random sample. *Ann. Statist.*, 11(8):905–911, 1983.

[10] P. J. Haas and L. Stokes. Estimating the number of classes in a finite population. *J. Amer. Statist. Assoc.*, 93, 1998.

[11] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

[12] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: a simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[13] R. Johnson, N. Hardavellas, I. Pandis, N. Mancheril, S. Harizopoulos, K. Sabirli, A. Ailamaki, and B. Falsafi. To share or not to share? In *VLDB*, pages 351–362, 2007.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT*, pages 111–122, 2004.

[15] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The case for precision sharing. In *VLDB*, pages 972–984, 2004.

[16] C. A. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE*, pages 1136–1145, 2007.

[17] V. Raman and G. Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *VLDB*, pages 858–869, 2006.

[18] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, pages 60–69, 2008.

[19] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

[20] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, pages 1–11, 1994.

[21] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, pages 723–734, 2007.