Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 769

Systems Group, Department of Computer Science, ETH Zurich

Shared Scans on Main Memory Column Stores

by
Gustavo Alonso, Donald Kossmann, Tudor-Ioan Salomie, Andreas Schmidt

July 12, 2012

inf | Informatik
Computer Science

Shared Scans on Main Memory Column Stores

G. Alonso, D. Kossmann, T.-I. Salomie
Systems Group, Dept. of Comp.Science, ETH Zurich
Email: {alonso,donaldk,tsalomie}@inf.ethz.ch

A. Schmidt
Karlsruhe Institute of Technology
Email: andreas.schmidt@kit.edu

***Abstract*—Column stores and shared scans have been found to be effective techniques in order to improve performance for many workloads. Another recent hardware trend makes it possible to keep most data in main memory. This paper builds upon these trends and explores how to implement shared scans on column stores in main memory efficiently. In particular, this paper proposes new approaches to avoid unnecessary work and to best implement position lists in such a query processing architecture. Performance experiments with real workloads from the travel industry show the advantages of combining column stores and shared scans in main memory over traditional database architectures.**

I. Introduction

The last years have seen a growing realization that the textbook architecture of database engines is outdated [1]. As part of this trend, and to support OLAP workloads and *Operational BI* workloads, two crucial techniques have evolved: (a) column stores and (b) shared scans. Furthermore, hardware trends have made it possible to serve most (often all) the data from main memory. Quoting Jim Gray and James Hamilton [2], disk is tape and main memory is the new disk. The goal of this paper is to show how column store technology and shared scans can be combined effectively in order to achieve high performance in main memory database systems.

Both column stores and shared scans have been studied extensively in various contexts (disk and main memory) in the past; e.g., [3], [4]. Column stores are particularly attractive for read intensive, complex query workloads because they often involve *scanning* a large number of tuples. With column stores, only a fraction of the data must be read and processed. Examples of column store systems include Sybase IQ, MonetDB, Vertica and SAP's TRex accelerator.

Shared scans are also attractive for OLAP and Operational BI workloads. The key idea is to batch queries that operate on the same (fact) table and execute the scan on that table only once for all queries of the batch. Shared scans are a special case of the more general concept of multi-query optimization [4]. Arguably, they are the most attractive incarnation of that concept because they optimize the most expensive operation and the optimization is almost trivial. Shared scans have been adopted by a number of database systems including RedBrick [5], IBM Blink [6], and Crescando [7]. They have also been studied in MonetDB [8].

While these two techniques have proven to be effective in isolation, they have never been integrated and exploited in a single system as far as we know. This does not come as a surprise because it is not obvious how these two techniques can be combined best. Indeed, the most compelling advantages of a column store seem to disappear if queries share the scan: While it is unlikely that a single query uses more than a few columns, a batch of, say, thousand queries is likely to involve most of the columns of a table. It seems that in main memory, the advantages of both techniques are even further diminished as the cost for data access and the penalty for poor data locality is diminished. This paper shows that these perceptions are not correct. Column stores and shared scan technology can be combined effectively and doing so matters in main-memory databases.

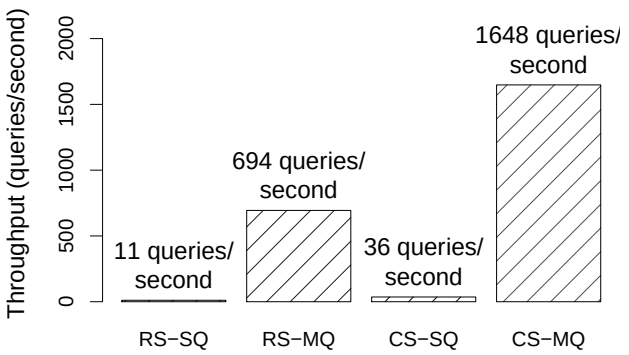

Fig. 1. Query Throughput, Real Workload

The main result of this paper is captured by Figure 1. Figure 1 shows the query throughput for a real workload from the travel industry (i.e., printing passenger lists in an airline reservation system). As detailed in Section V, this workload involves processing thousands of queries and hundreds of updates per second. Figure 1 shows the query throughput using four different systems: (a) a row store without shared scans (RS-SQ); (b) a row store with shared scans (RS-MQ); (c) a column store without shared scans (CS-SQ); and (d) a column store with shared scans (CS-MQ). In all four cases, the database resides entirely in main memory. As the figure indicates, the best performance is achieved by CS-MQ, the technique proposed in this work.

This paper makes two contributions: First, it explores the design space of alternative approaches to implement shared scans on column stores. Second, it describes the results of a comprehensive performance study that analyzes the trade-offs

of the alternative approaches and revisits the question whether and why column stores are better than row stores, if shared scans and multi-query processing in main memory are taken into account. The results confirm that indeed column stores are superior, but for different reasons than those stated in the literature so far (e.g., [9]). The experiments also confirm that the proposed approach significantly outperforms state-of-the-art main memory database systems such as MySQL (with its main memory option) or MonetDB. Somewhat surprisingly, the results of the, experiments with MySQL show that shared table scans outperform index scans even for workloads with many highly selective queries and point queries in which (B-tree) indexing is attractive. So far, shared scans have been mostly studied in OLAP contexts in which large volumes of data had to be processed and indexing was not an option.

This work was carried out as part of the Enterprise Computing Center at ETH Zurich (ECC). In collaboration with companies such as Amadeus, Credit Suisse, and SAP, the ECC studies new database architectures for challenging workloads whose requirements cannot be met using off-the-shelf database systems and conventional database technologies. This work specifically was motivated by the passenger information management system from Amadeus, a complex read and update-heavy Operational BI workload with strict data freshness and query latency guarantees. The characteristics of this workload have been described in detail [7] and are briefly revisited in Section V.

The remainder of this paper is organized as follows: Section II reviews the state of the art. Section III describes the architecture and key aspects of the systems compared. Section IV discusses how to implement shared scans on column stores. Sections V, VI, and VII contain the results of performance experiments that studied a variety of workloads and parameter settings. Section VIII concludes the paper.

II. Related Work

This work is based on recent developments in (a) column stores and (b) shared scans. What makes our work unique is that it presents techniques that enable to combine the developments in these two areas in the context of main-memory databases. To the best of our knowledge, we are the first to do so.

A good overview of the state-of-the-art on column stores is given in [10]. In general, there has been work on column stores for disk-resident data and for main-memory database systems. C-Store [11] is one of the most prominent projects that have studied column stores on disk. A recent representative of this class of systems is Dremmel [12]. MonetDB [13] is one of the most prominent projects that have studied column stores in main-memory. Both of these projects have studied a number of different techniques in order to improve the performance of column stores; examples are compression [14], intermediate result sharing [15], and tuple materialization and reconstruction [16]. Compression and tuple materialization for column stores have also been studied in other papers; e.g., [17], [18]. Predicate evaluation techniques for column stores have been studied in [19]. Our work was carried out in the context of *main-memory column stores* because that was the right choice for the workload of the travel industry that motivated this work and respects recent hardware trends. Accordingly, we implemented the state-of-the-art techniques for this kind of system.

In the past, there have also been related studies that compare the performance of column and row stores. Like this work, most of these studies conclude that column stores are superior for OLAP and Operational BI workloads [9], [10]. [20] argues that most of the techniques that make column stores on disk effective can also be implemented as part of a (disk-based) row store. The idea is to use so-called *super-tuples* in order to reduce I/O and to keep tuples sorted using a column abstraction. According to [20], this way to implement row stores is (almost) as efficient as column stores for read-intensive workloads.

Shared scans, the other technique that this work is based on, has also been studied extensively in the past. Again, there were two lines of work on shared scans: (a) shared scan on disks and (b) shared scan in main memory. The first commercial system that employed (disk-based) shared scans was RedBrick [5]. The best known technique to implement disk-based shared scans was devised in [8]. [8] proposes a scheme that makes optimal use of the (main memory) buffers for shared scans. For main-memory databases, shared scans have been studied in [21] and [7]. [21] shows how to schedule operations in the CPU in the presence of such shared scans in the Blink system. [7] proposes a specific shared scan operator called ClockScan. ClockScan is based on indexing *queries* of a batch of queries and can, therefore, scale to batches of thousands of concurrent queries. ClockScan is also the approach we adopt in this paper to implement shared scans on main-memory column stores.

In addition to scans, the sharing of other operators has also been studied. [22] proposes the CJOIN operator which allows concurrent evaluations of joins for many queries. The system runs a single "always on" query plan to which incoming queries can attach. That line of work is orthogonal to the work presented in this paper and can nicely be combined into a more comprehensive multi-query optimization framework.

Another important technique that our work is based on is *vectorization* (sometimes also called hyper-pipelining). Vectorization inspired the Zigzag technique described in IV-B. Vectorization was pioneered in the MonetDB/X100 and StageDB systems [23], [24]. In those systems, it is used to improve the locality of the processor's *instruction cache*. Both these systems do not make use of shared scans. As shown in Section IV-B, the Zigzag technique proposed in this work allows to avoid wasted work for shared scans in addition to improving instruction cache locality.

III. System architecture

A. System configuration

Figure 2 gives an overview of the system that we used to implement the four different variants: (a) row stores, single query at a time (RS-SQ); (b) column stores, single query

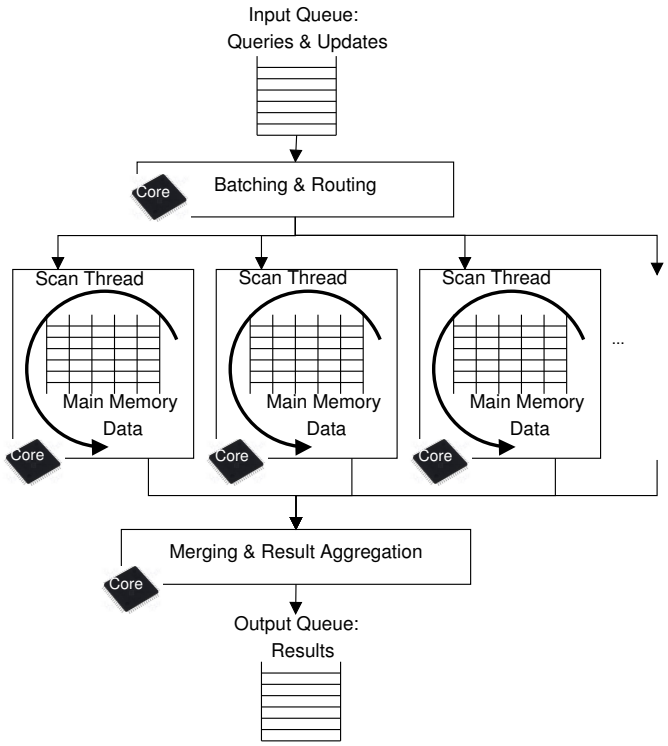

Fig. 2. Overview of the system architecture

at a time (CS-SQ); (c) row stores, many queries at a time, using shared scans (RS-MQ); and (d) column stores, many queries at a time, also using shared scans (CS-MQ). The system was specifically designed to work well on modern hardware with many CPU cores and large main memory. As a result, the database is horizontally partitioned and a separate core (with a scan thread) is dedicated to process queries and updates on each partition. This design follows the tradition of *shared nothing databases* within a single machine [25]. To work well with NUMA machines, each partition is located on main-memory banks close to the core that scans and updates that partition. Depending on the partitioning scheme, some cores may not be affected by a particular query or update statement. In this work, however, we considered a round-robin partitioning scheme only, thereby making sure that all partitions have the same size. As a result, all partitions and all cores are needed to process a query or update statement.

In addition to the *data cores* which scan and update the horizontal partitions of the database, there are two dedicated cores for *Batching & Routing* and for *Merging & Result Aggregation*. The *Batching & Routing* core reads queries and updates from the input queue and forwards them to the *data cores* as soon as they are ready. The difference between the SQ (single query) and MQ variants (shared scans) lies exactly in this component: In the SQ variants, each query is processed individually so that each data core processes at most one query at a time. In the MQ variants, a batch of potentially many queries and updates (up to several thousand) is taken from the input queue and propagated to the data cores. The *Merging & Result Aggregation* core aggregates the results produced by the data cores (typically, a simple *union*) and feeds them to the output queue of query results so that they can be consumed by the client applications.

Again, the difference between the four different variants studied in this work lie in the way the data is laid out in main memory (column vs. row store) and whether queries are batched or processed one query at a time (single query vs. shared scans). These differences are discussed in more detail in the remainder of this section.

B. RS-SQ: Row Store, Single Query

Most traditional relational database systems fall into this class of systems. In this architecture, the data is layed out in *rows* in main memory and the *Batching & Routing* process forwards only a query or update at a time to the *data cores*.

In Section V we present results for two different RS-SQ systems. The first is our own implementation based on the architecture given in Figure 2. In this RS-SQ implementation, there are no indexes and all queries and updates are processed by scanning the entire database. As a second representative of this class of systems, we studied MySQL. Obviously, MySQL makes heavy use of indexes and we have tuned it in the best possible way for our performance experiments.

C. CS-SQ: Column Store, Single Query

Main-memory column stores have recently gained a great deal of attention. Again, we studied two representative as part of our performance experiments. The first is MonetDB, a popular open source system. The second is our own implementation of a CS-SQ system based on the architecture shown in Figure 2. The remainder of this section briefly sketches the design decisions made in our implementation of a column store. Details of MonetDB can be found in [13].

First, a way of mapping values from different columns to the original tuple needs to be defined [11]. Logically, a column store implements joins between the columns in order to reconstruct tuples. The naïve way to implement this join is to attach a tuple identifier to each value in each column. This approach allows data to be re-organized in the column (e.g., sorted), but it requires more space and makes it hard to directly access a value based on its tuple id. The state-of-the-art approach identifies tuples implicitly by the *position* of their values in a column [11]; correspondingly, we adopted this approach in our implementation of a column store.

Second, intermediate query results need to be maintained. To this end, a data structure called a *position list* is employed. The position list keeps track of the *positions* of all candidate tuples that potentially match a query. After traversing all columns, the position list contains the *positions* of all query results. There are two ways to compute query results from a position list [16]. The first, *early* materialization, stores the values of the result tuples within the position lists. The second, *late* materialization, keeps only the *positions* and makes a second pass through the data in order to fetch the values. We chose late materialization in our implementation because that is the more space-efficient approach. Furthermore,

we experimented with different data structures to implement position lists. These data structures are described in more detail in Section IV-C.

D. RS-MQ: Row Store, Many Queries

Shared scans have been studied in detail in [21], [7], [8]. In shared scans, queries and updates are batched and each batch is processed as a whole at once. In the architecture of Figure 2, the batching is carried out by the *Batching & Routing* component. While different batching strategies are conceivable that select a subset of queries and updates from the input queue, our implementation of batching simply batches *all* the pending queries and updates of the input queue. This strategy may not be optimal in all situations, but it showed good performance in all experiments that we conducted.

Our implementation of RS-MQ is based on the ClockScan algorithm presented in [7]. We chose this algorithm because it seems to be the best known algorithm for the implementation of shared scans in main memory. ([8], in contrast, is particularly effective for shared scans on disk.) The ClockScan algorithm is based on volatile indexing of query predicates. That is, the predicates of a batch of queries and updates are indexed, just as in a publish/subscribe or data stream processing system [26]. During a scan, each tuple is probed using these query indexes in order to find queries that match that tuple, again just as in a publish/subscribe system. For row stores, one-dimensional and multi-dimensional query indexes can be used; multi-dimensional indexes can index conjunctions of several predicates of a query. Just as data indexing in traditional databases, query indexing has limitations. It is not always beneficial to index *all* predicates found in a batch of queries. Thus, the ClockScan algorithm also supports the processing of *unindexed predicates* [7].

One nice property of the ClockScan algorithm is that it allows to index and process queries and updates in the same, uniform way. The only difference is that updates must be applied *before* the queries during a scan of the data. As stated in [7], this approach guarantees Snapshot Isolation consistency in a row store. As shown in Section IV-D, the ClockScan algorithm must be extended significantly for column stores in a CS-MQ architecture.

IV. CS-MQ: Shared scans on column stores

This section presents the main contribution of this work: alternative ways to process shared scans on main-memory column stores. As in a traditional column store, the goal is to avoid wasted work of scanning data that is not relevant for a batch of queries. This task is not trivial as a large batch of, say, 1000 queries is likely to involve most of the columns of a table. In order to avoid wasted work, we propose a special execution model called "Zig-zagging" (Section IV-B). Zig-zagging is inspired by work on vectorization [24], [23], but it serves a different purpose here. A second goal is to process all queries, tuples, and attribute values as efficiently as possible. Conceptually, processing a batch of queries over a column store involves iterating over the set of columns, iterating over each value of a column, and iterating over the predicates of all queries. This section describes how these three iterations can be carried out efficiently by indexing the queries and keeping intermediate state efficiently in position lists. Query indexes are described in Section IV-A; alternative ways to implement position lists are described in Section IV-C. Finally, this section shows how concurrent reads and updates can be processed (Section IV-D).

Algorithm 1: Basic scan thread

```
input : queries, qSelectiv, qIndexes
output: resultTuples
1  ResetPositionList (plist);
2  predAttr ← GetPreds (queries).Sort (qSelectiv);
3  foreach col ∈ ( qIndexes ∩ predAttr ) do
4  |  col.PopulateIndex (queries);
5  |  foreach val ∈ col do
6  |  |  col.Probe (val,plist);
7  |  end
8  end
9  foreach col ∈ ( predAttr − qIndexes ) do
10 |  activeQueries ← GetActiveQOnCol
   |  (queries,col);
11 |  foreach q ∈ activeQueries do
12 |  |  foreach tuple ∈ plist [q] do
13 |  |  |  col.EvalPredicate (q,tuple,plist);
14 |  |  end
15 |  end
16 end
17 foreach q ∈ queries do
18 |  resultTuples += GetResultTuples (plist [q]);
19 end
20 return resultTuples;
```

A. Query Processing and Indexing

Algorithm 1 details the proposed algorithm to process a batch of queries on a column store. This algorithm iterates through the set of columns. For each column, it finds the pairs of positions p and queries q, such that the value at position p of the column matches the predicate of query q on that column. For CS-MQ, therefore, the position list is a two-dimensional data structure (query and position, rather than position only).

Again, keep in mind that the big overall goal is to avoid wasted work; that is, minimize the number of comparisons that need to be carried out in order to evaluate query predicates. As shown in Algorithm 1, this goal is achieved in two ways: First, the columns are ordered so that those columns are considered first on which many queries of a batch have highly selective predicates. For instance, if many queries of the batch involve a predicate on *date* and typically only a few result tuples match those *date* predicates, then the *date* column would be considered first. This way, intermediate results that need to be maintained in the position lists are kept small. This idea

has been exploited in many different ways in traditional query processing (e.g., join ordering or the optimization of expensive predicates in conjunctive queries); the novelty here is that we apply this idea for a potentially large batch of queries simultaneously.

The second idea that helps to avoid wasted work is to index query predicates. For instance, if *date* is one of the high selectivity columns, then a query index on all *date* predicates could be built in order to identify matching queries quickly while scanning through the *date* column. The pseudo-code of building and probing such a query index is shown in Lines 4-7 of Algorithm 1. Figure 3 illustrates the three steps of Lines 4-7. First, the query index is built based on the predicates of a batch of queries. The result is a query index as shown in the left bottom half of Figure 3. Second, the values of the column are scanned. Third, for each value the query index is probed.

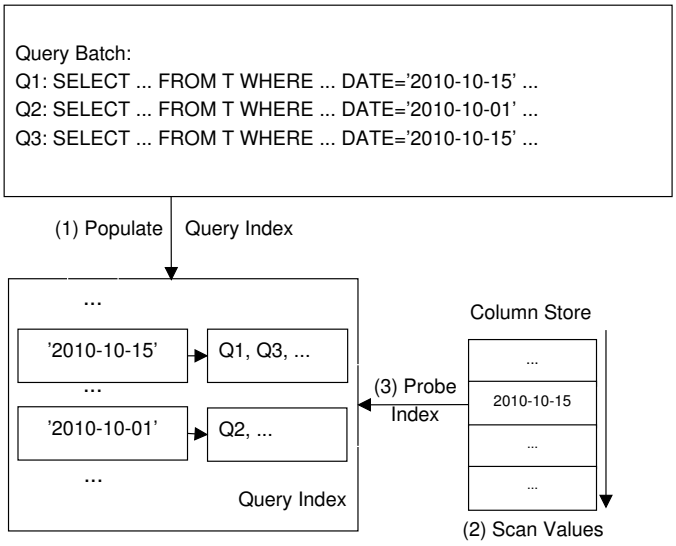

Fig. 3. A Query Index

As mentioned in Section III-D, it is not beneficial to carry out query indexing for *all* columns because constructing such a query index is expensive. The hope is that after a few columns, the set of candidate tuples is so small that the effort to evaluate the remaining query predicates (on columns that have not yet been considered) is smaller than the effort to build query indexes. How to probe these values is shown in Lines 9-16 of Algorithm 1. The next subsection will describe these lines of Algorithm 1 in more detail. Heuristics that determine for which columns query indexes should be built have been proposed in [7]. Even though that work only addresses row stores, the proposed heuristics of [7] for selecting the right query indexes for a batch of queries are directly applicable to our work on shared scans on column stores. The query indexes that we have used for our experiments are multi-maps [27]. These multi-maps associate to each key a list of query identifiers. We use the multi-map data structures of [27] because they have been specifically optimized for in-memory computation.

B. Skipping and Zig-Zagging

Lines 9-16 show how columns are processed for which the query predicates have not been indexed. At this point of processing Algorithm 1, two facts can be exploited. First, columns considered in Lines 9-16 of Algorithm 1 are likely to be relevant for only a few queries; the columns that are relevant for many queries are likely to be indexed and considered in the first phase of Algorithm 1 (i.e., Lines 3-8). In order to find out the set of queries that are relevant for a query, the *GetActiveQOnCol()* function is called in Line 10 of Algorithm 1. That way, only a subset of the queries will be considered while scanning through a column.

Second, the position list is sparse when the processing reaches Line 9 of Algorithm 1. Recall that the position list data structure records pairs of *positions* and *query-ids*. The positions represent tuples in the database; for instance, the 100th value of the *salary* column and the 100th value of the *starting date* column belong to the same *Employee* tuple. Initially, *all* tuples are potential candidates for *all* queries; that is, the position list is (logically) initialized to contain all possible *position* and *query-id* combinations. Fortunately, most queries are conjunctive queries so that the set of candidate tuples for a query is pruned with every column that is scanned. Again, the columns that involve high selectivity predicates are scanned in the first phase of Algorithm 1 so that by the time processing the second phase starts the position list is expected to involve only few $\langle$*position, query*$\rangle$ pairs. This fact is exploited in Line 12 of Algorithm 1 by considering only those values of the columns that are still part of candidate queries. Conceptually, Algorithm 1 *skips* to the right positions in the column, thereby avoiding wasted work on values that belong to tuples which are not candidates for any query result.

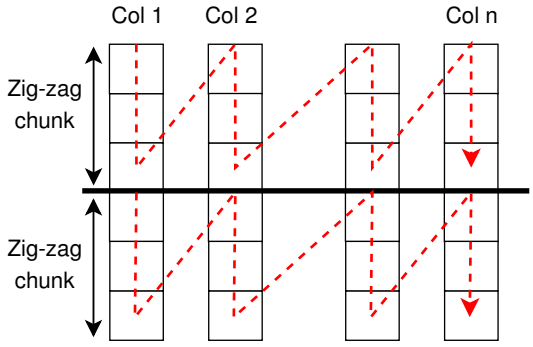

Fig. 4. Zig-zag Traversal

Figure 4 shows a technique that helps even further optimize the scanning of columns. This technique is called *Zig-zagging* and is not reflected in Algorithm 1. Zig-zagging partitions columns into chunks and only considers chunks if they are still relevant for the batch of queries. This way, most chunks need not be considered at all, even though the column (as a whole) is still relevant for a batch of queries. This way, Zig-zagging is a particularly efficient way to make use of *skips*. Furthermore, Zig-zagging bounds the size of position lists. With the right

Chunksize, it can be guaranteed that the position list fits into the L1 cache of a CPU core because the position list records matching tuples of the current chunk of tuples. In Figure 4, for instance, the position list must only record information about three tuples. The *Chunksize* parameter should, thus, be chosen carefully. It should be small enough so that the position lists fit in main memory. On the other hand, it should be large enough to limit the overhead or resetting the position list between chunks. Fortunately, the math for finding the right setting based on the L1 cache size is simple, and our performance experiments (Section VII) indicate that performance is not sensitive to the exact setting of the *Chunksize* parameter and setting it to, say, 15000 tuples always gives good results.

C. Position Lists

The implementation of position lists is critical for good performance. The requirements are as follows: First, position lists should be space-efficient so that they fit in L1 cache even for a large *Chunksize*. Second, the basic operations of initializing, adding entries, and deleting entries should be efficient. Third, they should support *skipping* to the relevant values of a column efficiently (Line 12 of Algorithm 1). With Zig-zagging, it is common that a column can be skipped altogether for a chunk and this situation should be detected quickly.

Previous research on column stores [16] proposed three types of (one-dimensional) data structures for position lists: lists of ranges, bitmaps, and arrays. All these three data structures can be extended to accommodate the *query* as a second dimension as required for shared scans on column stores. (For column stores that process each query separately, this dimension is obviously not required.) In addition to the data structure, the way the position list is pivoted is an important consideration: The position list can be pivoted by the *position* dimension or by the *query* dimension. In this work, we studied the three most prominent variants. We sketch these three variants in the remainder of this subsection. As shown in Section VII, the *DDA* variant was the overall winner so that most experiments reported in this paper were carried out using that variant.

a) Bitmaps: Using bitmaps to implement position lists is straight-forward. For shared scans on column stores, two dimensional Bitmaps are used. A "1" indicates that the tuple at the corresponding position is still potentially relevant for the corresponding query. One advantage of Bitmaps is that they can be compressed well. Unfortunately, the workloads that we studied (in particular, the Operational BI workload from the travel industry) were CPU bound so that compression was not attractive. Furthermore, the bits had to be flipped frequently (potentially with every evaluation of a predicate) which hurt compression performance even further. Furthermore, it turned out that bit operations are fairly costly so that Bitmaps are not a good data structure for position lists in our context. The experiments reported in Section VII confirm this observation.

b) RDA: In this approach, the position list is organized as a dynamic array pivoted by tuples. That is, for each tuple (of a chunk in Zig-Zagging) a list of query-ids is kept that that tuple potentially matches. The candidate queries are initialized as part of scanning the first column; that way, we need not create an array of all query-ids for each tuple and start with a short list of query-ids for most tuples.

c) DDA: This approach uses dynamic arrays pivoted by queries. That is, for each query a list of *positions* are kept. It turns out that the RDA approach has better cache locality whereas DDA has lower maintenance cost (less dynamic arrays that need to be initialized because there are less queries than tuples). Furthermore, it is easier to find the right *skipping* position with DDA. Again, the experiments of Section VII study this tradeoff in more detail.

D. Updates

In an Operational BI workload (such as the workload from the travel industry that motivated this work), updates and queries must be processed concurrently. In principle, updates are processed in the same way as queries; that is, their predicates may be indexed in order to find quickly the tuples that are affected by the update. Once a matching tuple is found, it is updated *in-place*. (Studying other update schemes such as *versioning* is beyond the scope of this work and an interesting avenue for future work.) What makes the implementation of updates special are two observations:

- *Consistency:* As studied in [7], all updates must be executed in the order of arrival. Furthermore, a value that is updated in-place must be updated *before* it is read for the first time by a query (i.e., reader). This way, Snapshot Isolation can be guaranteed.
- *SQL Semantics:* SQL mandates that updates are processed in two phases in order to avoid convoys and the halloween effect [28]. Most updates can be optimized in a single pass if the updated columns are not correlated with the columns used in the predicates of the update statement. Unfortunately, this optimization does not work for column stores if an updated column is processed *before* a column involved in a predicate in Algorithm 1.

These observations have led us to scan each column twice as part of a shared scan for a batch of queries and updates. The first scan evaluates all the predicates of update statements in order to compute the target tuples of all updates. In the second scan, the updates are applied in-place first and then the queries are evaluated on the updated values. Obviously, this approach hurts the performance of shared scans on column stores. As shown in Figure 1 of the introduction and Section V, shared scans on column stores, nevertheless, outperform any other variant.

V. Experimental results

This section presents the results of performance experiments that compare the performance of shared scans on column stores with more traditional database architectures. As baselines for the comparison, MySQL and MonetDB are used.

A. Experimental Environment

1) Software and Hardware Used: We implemented the four architectural variants described in Sections III and IV in C++ on top of a storage manager that can be configured to be a column and a row store. Furthermore, we used several other open-source C++ libraries such as Google HashMap, FastBit, and Boost.

The RS-SQ, CS-SQ, and RS-MQ approaches were implemented in a straight-forward way, as described in Section III. For CS-MQ, we used DDA and Zig-Zagging for all experiments reported in this section. Furthermore, the *Chunksize* parameter was set to 15,000 and query indexes were built only for the first (most selective) column as this turned out to be the optimal plan for the workloads we studied. Experiments that study the alternative CS-MQ variants (e.g., RDA and Bitmaps) and the sensitivity of the right parameter settings are shown in Section VII.

As baselines, we used MySQL and MonetDB for all experiments. MySQL can be seen as a representative of a RS-SQ approach; and MonetDB can be seen as a representative of a CS-SQ approach. MySQL was used with its "Main Memory" storage manager in order to achieve best performance for the studied scenario in which the whole database fit in main memory. We also carried out experiments with MySQL and the widely used InnoDB storage manager, but we omit those results because InnoDB was outperformed by the "Main Memory" storage manager in all our experiments. Furthermore, we manually tuned MySQL in order to select the best set of indexes for each benchmark workload. MySQL, therefore, varies significantly from our own RS-SQ implementation which did not make use of any indexing. MonetDB was used in Version 5.22.0 as provided by the CWI download Web site [13]. That version of MonetDB does not support any kind of indexing. We contacted the MonetDB developers to ensure the best possible configuration of MonetDB and to confirm that the MonetDB results were correct and could not be improved.

Almost all experiments were carried out on a 8 core Nehalem L5520 machine with 24 GB of main memory. As shown in the next subsection, the benchmark database fits easily into main memory of this machine. The operating system was Debian Linux 5.0. In order to show the scalability of our approach and to show that it works on different architectures, we also used a 48 core AMD Magny Cours machine with 128 GB of main memory, running a 64-bit Ubuntu 10.04 server. The results of scalability experiments on this machine are presented in Section V-C.

2) Benchmark Database and Workloads: All experiments reported in this section were carried out with data, queries and updates from the Amadeus airline booking system. Amadeus provides the reservation service for many major airlines (e.g., AirFrance, British Airways, Lufthansa, Quantas, United, etc.). Each tuple of that database represents the booking of a passenger on a particular flight; e.g., passenger name, flight number, date, airline, dietary constraints, booking class, etc. In all, the schema involved 48 attributes and tuples have 305 bytes on an average. We varied the size of the database from 1 GB (about 3 million bookings) to 10 GB (about 33 million bookings). The queries involved printing passenger lists with varying criteria over the 48 attributes (e.g., all hon circle passengers flying out of Zurich today). Examples for updates were upgrades of passengers or registration of dietary constraints. The Amadeus workload has about 8 times as many queries as updates. The Amadeus service level agreements require that all queries must be executed within 2 seconds and that the data freshness must be 2 seconds.

If not stated otherwise, the database was partitioned so that each core scanned 1 GB of data. The partitioning scheme was such that all queries had to be executed on all partitions (i.e., the query predicates did not match the partitioning scheme) whereas updates could typically be directed to a single partition. This set-up is the same set-up as tested in [7] and is the set-up that is going to be used by Amadeus starting in August 2011. In some experiments, we changed the partitioning scheme in order to study the performance of large scans (e.g., a 10GB scan by a single core). Obviously, the Amadeus latency requirements could not be met using such settings.

Section VI presents the results of experiments in which we varied the selectivity of predicates and update rates. This section, however, only reports on experiments conducted with the real, live query and update traces from Amadeus.

B. Experiment 1: One Core, 1GB Data

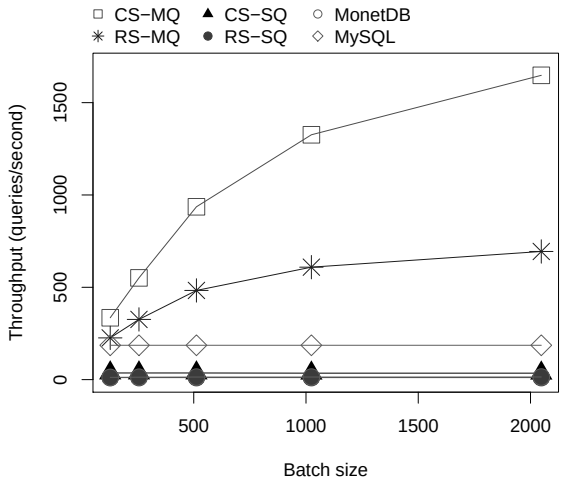

Fig. 5. Throughput (queries/sec): Real Workload, Vary Batchsize

1) Throughput Experiments: The main result of this paper is shown in Figures 5 and 6. Both figures show the throughput of the four different approaches and the two baseline database systems (MySQL and MonetDB) with varying batchsize. Figure 5 visualizes the results as graphs; Figure 6 gives the exact results in tabular form. As expected, the throughputs of all "query at a time" variants and MySQL and MonetDB are independent of the batch size. All these systems are run at their peak throughput in this experiment and process all incoming queries and updates as fast as they can. In contrast,

	Batch size				
Queries:	128	256	512	1024	2048
Updates:	16	32	64	128	256
	Throughput (queries/second)				
CS-MQ	335	551	936	1326	1648
RS-MQ	226	326	483	609	694
CS-SQ	36	36	36	35	35
RS-SQ	11	11	11	11	11
MySQL	186	186	186	186	186
MonetDB	12	13	13	14	14

Fig. 6. Throughput (queries/sec): Real Workload, Vary Batchsize

the shared scan approaches (i.e., CS-MQ and RS-MQ) benefit greatly from an increased batchsize. The more queries and updates are processed concurrently, the higher the benefit from sharing the scans. As shown in Figure 5 the CS-MQ and RS-MQ curves flatten out at about 1000 concurrent queries; at this point the advantages of further batching are outweighed by additional overhead to keep bigger query index structures in the processor caches.

Comparing row stores and column stores, the results shown in Figures 5 and 6 confirm the findings of all recent studies. In the SQ variants, column stores win by a factor of 3.3. Figure 7 studies this effect in more detail. As shown in Figure 7, CS-SQ moves about two orders of magnitude less data from main memory to the processor caches (L1 and L2) than RS-SQ. This observation confirms the traditional insight that column stores are superior for reading less data.

	Batch size			
Queries:	256		2048	
Updates:	32		256	
	L1 Data	**L2 Data**	**L1 Data**	**L2 Data**
CS-MQ	7.41	0.20	6.24	0.40
RS-MQ	12.40	1.57	8.91	1.80
CS-SQ	5.87	0.93	5.96	0.93
RS-SQ	439.14	207.23	477.37	207.16
MonetDB	95.69	9.29	76.18	6.60
MySQL	42.70	4.27	43.82	4.46

Fig. 7. L1, L2 Data Cache traffic (MB/query): Real Workload

Using shared scans, column stores are also better than row stores, if the techniques devised in Section IV are used. Figure 6 shows that CS-MQ outperforms RS-MQ by a factor of about 2.5. Turning to the CS-MQ and RS-MQ lines in Figure 7, however, it can be seen that this result cannot be explained by L1 and L2 cache misses alone. Indeed, CS-MQ has better data locality than RS-MQ resulting in less misses and less data to be shipped from main memory into the L1 and L2 caches, but the effects are not as pronounced as for CS-SQ vs. RS-SQ. More significantly, the benefits of CS-MQ as compared to RS-MQ can be explained by avoiding wasted work and skipping large parts of the data due to the *Zig-zagging* approach presented in Section IV-B.

The low throughput of MonetDB (along with the high memory bandwidth utilization per query) is due to the use of full table scans per query (as confirmed by the MonetDB developers). MonetDB is optimized for OLAP and is less suitable for the kind of Operational BI workload we consider. As MonetDB is among the most representative main-memory column stores, we nevertheless use it as a baseline for our experiments.

Comparing CS-MQ to MySQL, it becomes clear that even for a workload with highly selective queries a non-index approach (such as CS-MQ) can significantly outperform a database system with indexes and that was specifically tuned for this kind of workload. This result came to us as a surprise. If the workload is large enough (i.e., a sufficient number of concurrent queries), then the benefits of sharing scans more than offset the extra work that needs to be done per query for using a full table scan rather than an index look-up. Mathematically, this result can be explained using Yao's formula [29]: The number of pages accessed by a set of independent index lookups grows almost linearly with the number of index lookups. That is, with a growing number of queries large portions of the base table need to be accessed anyway.

	Batch size			
Queries:	128	512	1024	2048
Updates:	16	64	128	256
	Min/Max Response time (seconds)			
CS-MQ	0.3	0.5	0.8	1.2
RS-MQ	0.6	1.2	1.9	3.2
CS-SQ	0.02/3.4	0.02/13.9	0.02/28.7	0.02/57.6
RS-SQ	0.11/11.2	0.11/45.2	0.11/90.6	0.11/181
MonetDB	0.07/11.4	0.06/36.7	0.06/73.0	0.06/143.5
MySQL	0.01/0.6	0.01/2.5	0.01/5.2	0.01/10.6

Fig. 8. Min/Max Response Time (sec): Real Workload, Vary Batchsize

2) Response Time Experiments: Figure 8 shows the average response times of queries with varying batchsizes for the real Amadeus workload. Obviously, the response times for CS-MQ and RS-MQ increase with a growing batch size: In those two approaches, all queries of a batch have the same response time. With an increasing batch size, more work needs to be done in order to execute the whole batch and, thus, the response time of each query increases.

For the "query-at-a-time" approaches, the response time depends on the arrival rate of the queries and queueing effects that might arise from overload situations if many queries arrive at the same time. Figure 8 shows the best possible case in which the queries (and updates) are processed serially and the worst possible case in which all queries and updates of a batch arrive at the same time. It can be seen that the variance between the best possible and worst case is huge, while the response time is fairly constant and predictable for CS-MQ and RS-MQ. In other words, the response times of all "query-at-a-time" approaches deteriorates quickly in overload situations, whereas shared scans degrade gracefully in overload situations.

Comparing the best cases, it can be seen that MySQL is the clear winner. If the system processes only a single query and there are no queueing effects, then MySQL can process a query extremely fast with only a few index lookups. The price that shared scans pay for their high throughput

and robust response times in overload situations is that they have comparably high response times in underload situations. From an operational perspective, however, a response time of 0.3 seconds is good enough and being faster does not help. Fortunately, 0.3 seconds response time is more than sufficient for the Amadeus application whose workload we used in these experiments.

C. Experiment 2: Vary Cores, Vary Data

	Number of cores			
	1	2	4	8
	Throughput (queries/second)			
CS-MQ	1648	3266	6481	12564
RS-MQ	694	1368	2734	5384
CS-SQ	35	72	146	292
RS-SQ	11	18	33	66
MonetDB	14	12	12	11
MySQL	185	328	443	690

Fig. 9. Throughput (queries/sec): Real Workload, Vary #Cores

Figure 9 shows how the throughput of the alternative approaches scales up with the number of cores used. The prototype database system that was used to implement CS-MQ, RS-MQ, CS-SQ, and RS-SQ was specifically designed to operate well on multi-core machines. Correspondingly, all these four approaches scale linearly with the number of cores. MySQL and MonetDB, in contrast, do not scale linearly. However, this observation is more an artifact of the particular implementation than of the RS-SQ and CS-SQ approaches that they represent. We would like to note, however, that it is in general difficult to achieve linear scale-ups on multi-core machines with database systems that rely on indexing, such as MySQL. So far, we are not aware of any database system that can achieve such linear scale-ups for index lookups.

Dataset size	Number of cores			
	1	2	4	8
	Throughput (queries/second)			
1GB	1648	3266	6481	12564
2GB	801	1644	3261	6400
5GB	308	614	1220	2370
10GB	149	294	575	1180

Fig. 10. Throughput (queries/sec): CS-MQ, Real Workload Vary #Cores, Vary Dataset Size

Figure 10 assesses the scalability of CS-MQ in more detail along two dimensions: the number of cores and the size of the database. As expected, this experiment confirms that CS-MQ indeed scales linearly along both dimensions. With the same number of cores, it takes about ten times as long to process the queries on 10GB than on 1GB of data (resulting in a tenth of the throughput). Overall, the performance of CS-MQ is highly predictable: The amount of data that needs to be processed by a single core determines both the (maximum) query latency and the query throughput that can be sustained. For Amadeus, 1GB per core is the rule of thumb because that allows to met the 2 seconds query latency and data freshness guarantees.

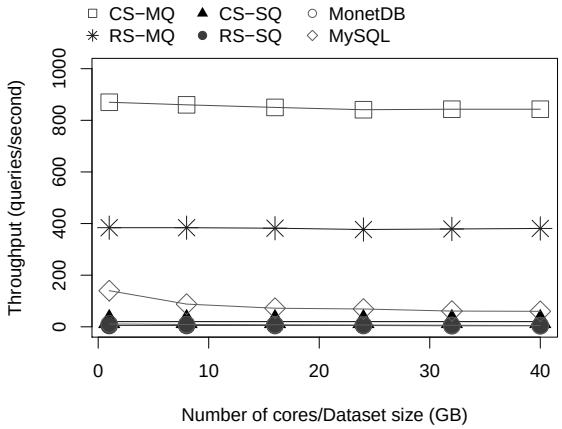

Fig. 11. Throughput (queries/sec): Real Workload Vary #Cores, Vary Dataset Size (1 core/GB)

	Number of cores / Dataset size (GB)				
	1	8	16	32	40
	Throughput (queries/second)				
CS-MQ	870	860	850	843	843
RS-MQ	384	384	382	379	381
CS-SQ	20	20	20	20	20
RS-SQ	5	5	5	4	4
MonetDB	11	9	8	6	5
MySQL	140	88	72	61	60

Fig. 12. Throughput (queries/sec): Real Workload Vary #Cores, Vary Dataset Size (1 core/GB)

In order to experiment with different hardware and show the scalability of the approaches with regard to larger data sets and many cores, we also carried out experiments with a machine with 48 cores. In these experiments, we varied the number of cores used from 1 to 40 cores and at the same time varied the size of the database from 1 GB to 40 GB. Following the Amadeus rule of thumb, the database was 1 GB when only one core was activated and, correspondingly, 40 GB with 40 cores. The real, original Amadeus workload was studied with batches of 2048 queries and 256 updates. The results are shown in Figure 11. This figure shows that all approaches scale approximately linearly in this experiment. This result is not surprising because all systems (except MySQL) are based on table scans and such table scans obviously scale-up linearly. The performance of MySQL relies heavily on indexing. B-tree indexes do not scale as nicely with the number of cores for an update-intensive workload such as the Amadeus workload. Accordingly, the throughput of MySQL drops slightly as the number of cores is increased (and the database size is scaled up at the same time). Comparing the alternative approaches, CS-MQ wins again and for the same reasons as in the previous experiments. Overall, the throughputs are lower than in the previous experiments for all approaches because the AMD Magny Cours machine has a less powerful main-memory system as compared to the Intel machine used in all other experiments.

VI. Experiments with Different Workloads

This section presents the results of experiments that study the behavior of the alternative systems with a number of different workloads, thereby vary the update rates, the number of attributes in the project lists of queries, and the selectivity of predicates. Overall, the results of these experiments confirm the findings made in the previous section with the Real Workload of Amadeus.

A. Vary Updates

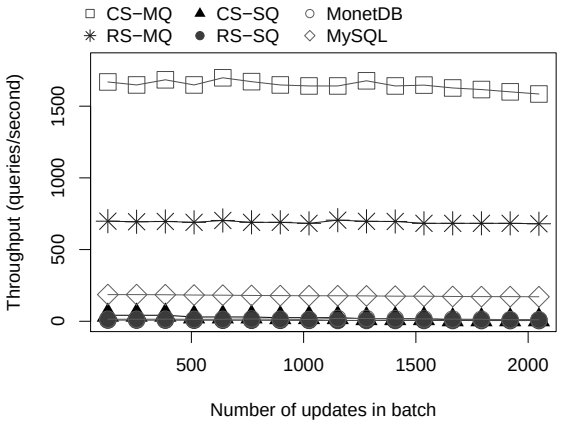

Fig. 13. Throughput (queries/sec): 2048 Q. Batchsize, Vary Updates

Figure 13 shows the query throughput of the alternative approaches and systems with a varying number of concurrent updates while processing a batch of 2048 queries. For these experiments, we took the original queries and updates from the Amadeus traces, but we varied the mix of queries and updates; i.e., pushed or delayed updates from the traces in order to get a different query/update ratio. In the extreme case, more than 2000 updates are executed (i.e., one update with every query). It can be seen that the throughputs of all systems and approaches are not affected significantly by an increasing concurrent update workload.

MySQL's performance does not degrade significantly with an increasing update workload because we are using the "Main Memory" storage engine which offers no transactional guarantees (no consistency). For a lock-based storage manager such as InnoDB, the performance would degrade significantly with a growing update rate.

B. Vary Projections

Figure 14 shows the effects of a varying number of attributes in the `SELECT` clause of queries. In this experiment, again, we used the original Amadeus traces, but we padded the `SELECT` clause of the queries. As a baseline, the `Real` column shows the throughput of the original Real Workload.

As can be seen, the size of projection lists do not impact the performance significantly in this experiment. In general, it can be expected that the more attributes are retrieved by a query, the worse the performance of a column store gets while the performance of a system based on a row store should be constant. Indeed, all row stores (RS-MQ, RS-SQ, and MySQL) have (almost) constant throughput in this experiment. However, unlike expectations, the column stores (CS-MQ, CS-SQ, and MonetDB) are fairly stable as well. That is, the execution of inner joins is fairly cheap as compared to the execution of scans on the data; in particular as most of the queries are highly selective so that the inner join must only be computed on a small number of query results.

	# Projected attributes					
	Real	1	20	30	40	48 (all)
	Throughput (queries/sec)					
CS-MQ	1648	1702	1659	1644	1629	1613
RS-MQ	694	650	644	640	685	688
CS-SQ	36	35	35	35	35	35
RS-SQ	9	9	9	9	9	9
MonetDB	14.2	14.3	14.3	14.3	14.2	14.2
MySQL	188	190	189	188	187	186

Fig. 14. Throughput (queries/sec): Vary Projections

C. Vary Predicates

	% of random predicates		
	1%	5%	10%
	Throughput (queries/sec)		
CS-MQ	1944	1624	896
RS-MQ	1119	381	215
CS-SQ	42	42	40
RS-SQ	9	9	9
MonetDB	11	11	11
MySQL	194	200	203

Fig. 15. Throughput (queries/sec): Vary Predicate Selectivities

Figure 15 shows the query throughput of the alternative systems, thereby varying the predicates of the `WHERE` clauses of the queries. Again, we used the original Amadeus traces; but, this time we swapped 1%, 5%, and 10% of the predicates from one query to another. This way, the selectivities of queries was randomized; i.e., queries had a higher variance of query result sizes with a growing percentage of randomization.

As a general trend, the MQ variants are sensitive to this parameter for two reasons. First, the overall throughput is dominated by the (expensive) queries who produce large results; as a result, some of the benefits of the MQ variants are less pronounced. Second, query indexing becomes less attractive if there are no highly selective predicates. Nevertheless, even with a high distortion of 10%, the MQ variants still outperform all other systems.

VII. Tuning CS-MQ

This section presents the results of experiments that analyze alternative variants to implement CS-MQ and the sensitivity to its parameter settings.

In all experiments reported in Sections V and VI, Direct Dynamic Arrays (CS-MQ-DDA) were used as an implementation of position lists. Figure 16 compares the throughput of DDA to Reverse Dynamic Arrays (CS-MQ-RDA) and Bitmaps

(CS-MQ-BM). As a baseline, Figure 16 shows the throughput of RS-MQ.

A. Vary Position Lists

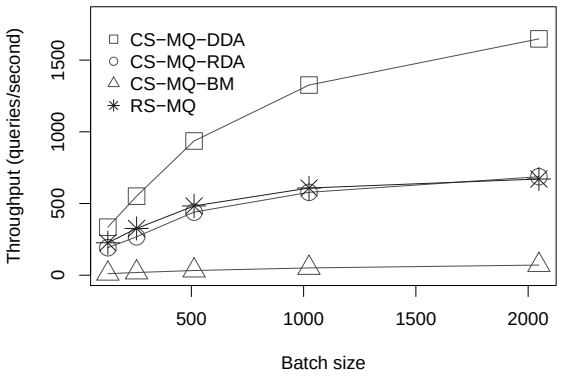

Fig. 16. Throughput (queries/sec): DDA vs. RDA vs. BM Real Workload, Vary Batchsize

As can be seen, only a column store with DDA significantly outperforms the row store in this experiment. Bitmaps consume a great deal of main memory in this experiment and, therefore, show poor cache locality. Furthermore, bit manipulation operations are fairly expensive if the bitmap is sparse. DDA is superior to RDA for two reasons. First, the DDA data structure has less rows because it is pivoted by "query" and not by "row" and there are fewer queries than rows. As a result, there is less maintenance effort (e.g., initializing the DDA). Furthermore, it is much cheaper to find the right skipping position with DDA than with RDA.

B. Vary Chunk Size

Figure 17 studies the throughput of CS-MQ (with DDA) as a function of the *Chunksize* parameter introduced in Section IV-B. As a baseline, the throughput of RS-MQ is shown. As stated in Section IV-B, there is a trade-off. With small chunk sizes, we need to reset the position lists frequently, resulting in high overheads for maintaining the position lists. On the positive side, small chunk sizes allow fine-grained decisions on skipping. Figure 17 depicts two important results:

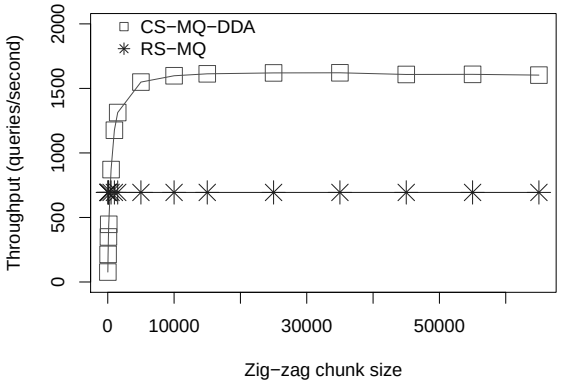

Fig. 17. Throughput (queries/sec): Real Workload, Vary Chunksize

- The overhead of resetting the position lists is prohibitive for small chunk sizes (below 1000 tuples). To explain why, we also measured the L1 cache misses (not shown for brevity) and found out that resetting the position lists incurs a large number of L1 cache misses.
- The sensitivity of the performance of CS-MQ to this parameter is low. In the range of 5000 to 60,000, we could observe hardly any change.

C. Vary Query Indexes

As mentioned in Section III-D, indexing queries is a key technique to achieve scalable and robust performance for shared scans. [7] proposes a multi-dimension index structure for shared scans in a row store. For shared scans on a column store, the situation is different: Each index must be probed individually as each column is scanned individually so that multi-dimensional indexing is less effective. In some sense, a column store simplifies the indexing because multi-dimensional indexes are not effective; in other ways, a column store limits the opportunities of multi-dimensional indexing.

Figure 18 shows the throughput of CS-MQ when varying the number of columns that were indexed. The most promising columns for query indexing were determined using the approach proposed in [7]. As baselines, we used a row store with multi-dimensional indexing (as proposed in [7]) and a row store with a single query index on the most promising column.

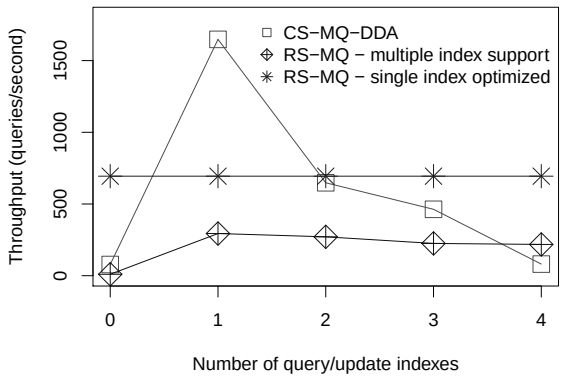

Fig. 18. Throughput (queries/sec): Real Workload, Vary #Indexes

It can be seen that indexing is indeed crucial. With 0 columns indexed, both CS-MQ and RS-MQ perform poorly. For this particular workload, indexing only one column yields the highest throughput. Indeed, most queries of the Amadeus workload involved the flight number of a booking and such a predicate on flight number was highly selective. As a result, almost no additional benefit could be achieved from indexing more columns. For the same reason, a row store with a single, one-dimensional index (on *flight-number*) outperformed the multi-dimensional indexing scheme proposed in [7]. Of course, these results are specific to this particular workload. In general, however, we believe that tuning the query indexing is not an issue and we were able to get good performance

by simply adopting the statistics and optimization approach proposed in [7] and limiting the number of indexes to 1.

VIII. Conclusions

Motivated by a concrete use case of the airline industry, this paper showed how column stores and shared scans can be efficiently combined. The two techniques have been shown to be useful in isolation, but they would seem to contradict each other in practice. Column stores reduce the amount of data needed to answer a query while shared scans require access to much more data in a single scan than a single query would. We resolve this contradiction through two novel techniques. The first is an efficient implementation of position lists based on Direct Dynamic Arrays. The second is a Zig-Zag approach of scanning the columns that avoids scanning irrelevant data. This paper presented experimental results that evaluate both techniques and show that they perform better than alternative designs. Furthermore, the performance experiments indicate that, if done right, shared scans on column stores outperform any traditional storage management architecture such as row stores, B-tree indexes, or storage managers that process a request at a time.

Acknowledgment

We would like to thank Dietmar Fauser and Jeremy Meyer from Amadeus for providing the data sets and the query and update traces. This work was funded in part by the Enterprise Computing Center, a research collaboration of the ETH Systems Group with Amadeus, Credit Suisse, and SAP; and by the Swiss National Science Foundation as part of the ProDoc program on Enterprise Computing.

References

[1] M. Stonebraker and U. Çetintemel, "One Size Fits All: An Idea Whose Time Has Come and Gone," in *ICDE*, 2005.
[2] J. R. Hamilton, "Internet scale storage," in *SIGMOD Conference*, 2011, pp. 1047–1048.
[3] G. P. Copeland and S. N. Khoshafian, "A Decomposition Storage Model," in *SIGMOD*, 1985.
[4] T. K. Sellis, "Multiple-Query Optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, 1988.
[5] P. M. Fernandez, "Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms," in *SIGMOD*, 1994.
[6] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl, "Parallelizing Query Optimization," *Proc. VLDB Endow.*, vol. 1, no. 1, 2008.
[7] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable Performance for Unpredictable Workloads," *PVLDB*, vol. 2, no. 1, 2009.
[8] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS," in *VLDB*, 2007.
[9] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-Stores vs. Row-Stores: How Different Are They Really?" in *SIGMOD*, 2008.
[10] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column Oriented Database Systems," in *VLDB*, 2009.
[11] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-Store: A Column-Oriented DBMS," in *VLDB*, 2005.
[12] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Commun. ACM*, vol. 54, no. 6, pp. 114–123, 2011.
[13] "MonetDB http://monetdb.cwi.nl/testing/projects/monetdb/Current."
[14] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006.
[15] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves, "An Architecture for Recycling Intermediates in a Column-Store," in *SIGMOD*, 2009.
[16] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden, "Materialization Strategies in a Column-Oriented DBMS," in *ICDE*, 2007.
[17] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores," in *SIGMOD*, 2009.
[18] S. Idreos, M. L. Kersten, and S. Manegold, "Self-Organizing Tuple Reconstruction in Column-Stores," in *SIGMOD*, 2009.
[19] H. Min and H. Franke, "Improving In-memory Column-Store Database Predicate Evaluation Performance on Multi-core Systems," *Computer Architecture and High Performance Computing, Symposium on*, vol. 0, pp. 63–70, 2010.
[20] A. Halverson, J. L. Beckmann, J. F. Naughton, and D. J. Dewitt, "A Comparison of C-Store and Row-Store in a Common Framework," University of Wisconsin-Madison, Tech. Rep. TR1570, 2006.
[21] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman, "Main-Memory Scan Sharing for Multi-Core CPUs," *PVLDB*, vol. 1, no. 1, 2008.
[22] G. Candea, N. Polyzotis, and R. Vingralek, "A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses," *PVLDB*, vol. 2, no. 1, 2009.
[23] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman, "MonetDB/X100 - A DBMS In The CPU Cache," *IEEE Data Eng. Bull.*, vol. 28, no. 2, pp. 17–22, 2005.
[24] S. Harizopoulos and A. Ailamaki, "StagedDB: Designing Database Servers for Modern Hardware," *IEEE Data Eng. Bull.*, vol. 28, no. 2, pp. 11–16, 2005.
[25] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.
[26] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe," in *SIGMOD Conference*, 2001, pp. 115–126.
[27] M. Nelson, *C++ Program Guide to Standard Template Library*. IDG Books Worldwide, Inc., 1995.
[28] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.
[29] S. B. Yao, "Approximating Block Accesses in Database Organizations," *Commun. ACM*, vol. 20, no. 4, 1977.