MaSM: Efficient Online Updates in Data Warehouses

Manos Athanassoulis[†] Shimin Chen[⋆] Anastasia Ailamaki[†]
Phillip B. Gibbons[⋆] Radu Stoica[†]

[†]École Polytechnique Fédérale de Lausanne
manos.athanassoulis|natassa|radu.stoica@epfl.ch

[⋆]Intel Labs
shimin.chen|phillip.b.gibbons@intel.com

ABSTRACT

Data warehouses have been traditionally optimized for read-only query performance, allowing only offline updates at night, essentially trading off data freshness for performance. The need for 24x7 operations in global markets and the rise of online and other quickly-reacting businesses make concurrent online updates increasingly desirable. Unfortunately, state-of-the-art approaches fall short of supporting fast analysis queries over fresh data. The conventional approach of performing updates in place can dramatically slow down query performance, while prior proposals using differential updates either require large in-memory buffers or may incur significant update migration cost.

This paper presents a novel approach for supporting online updates in data warehouses that overcomes the limitations of prior approaches, by making judicious use of available SSDs to cache incoming updates. We model the problem of query processing with differential updates as a type of outer join between the data residing on disks and the updates residing on SSDs. We present *MaSM* algorithms for performing such joins and periodic migrations, with small memory footprints, low query overhead, low SSD writes, efficient in-place migration of updates, and correct ACID support. Our experiments show that MaSM incurs only up to 7% overhead both on synthetic range scans (varying range size from 100GB to 4KB) and in a TPC-H query replay study, while also increasing the update throughput by orders of magnitude.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Query Processing*; H.2.7 [**DATABASE MANAGEMENT**]: Database Administration—*Data Warehouse and Repository*

General Terms

Algorithms, Design, Performance

Keywords

Materialized Sort Merge, Online Updates, Data Warehouses, SSDs

1. INTRODUCTION

Data warehouses (DW) are typically designed for efficient processing of *read-only* analysis queries over large data. Historically, updates to the data were performed using bulk insert/update features that executed offline—mainly during extensive idle-times (e.g., at night). Two important trends lead to a need for a tighter interleaving of analysis queries and updates. First, the globalization of business enterprises means that analysis queries are executed round-the-clock, eliminating any idle-time window that could be dedicated to updates. Second, the rise of online and other quickly-reacting businesses means that it is no longer acceptable to delay updates for hours, as older systems did: the business value of the answer often drops precipitously as the underlying data becomes more out of date [12, 24]. In response to these trends, data warehouses must now support a much tighter interleaving of analysis queries and updates, so that analysis queries can occur 24/7 and take into account very recent data updates [2]. Thus, Active (or Real-Time) Data Warehousing has emerged as a business objective [16, 24] aiming to meet the increasing demands of applications for the latest version of data. Unfortunately, state-of-the-art DW management systems *fall short of the business goal of fast analysis queries over fresh data*. A key unsolved problem is how to efficiently execute analysis queries in the presence of *online* updates that are needed to preserve data freshness.

1.1 Efficient Online Updates: Limitations of Prior Approaches

While updates can proceed concurrently with analysis queries using concurrency control schemes such as snapshot isolation [3], the main limiting factor is the physical interference between concurrent queries and updates. We consider the two known approaches for supporting online updates, in-place updates and differential updates, and discuss their limitations.

In-Place Updates Dramatically Increase Query Time. A traditional approach, used in OLTP systems, is to update in place. However, as shown in Section 2.2, in-place updates can dramatically slow down DW queries. Mixing random in-place updates with TPC-H queries increases the execution time, on average, by 2.2X on a commercial row-store DW and by 2.6X on a commercial column-store DW. In the worst case, the execution time is 4X longer! Besides having to service a second workload (i.e., the updates), the I/O sub-system suffers from the *interference* between the two workloads: the disk-friendly sequential scan patterns of the queries are disrupted by the online random updates. This factor alone accounts for 1.6X slowdown on average in the row-store DW.

Differential Updates Limited by In-Memory Buffer. Recently, *differential updates* have been proposed as a means to enable efficient online updates in column-store DW [22, 11]. The basic idea is to (i) cache incoming updates in an in-memory buffer, (ii) take the cached updates into account on-the-fly during query processing, so that queries see fresh data, and (iii) migrate the cached updates to the main data whenever the buffer is full. While these proposals significantly improve query and update performance, their reliance

SIGMOD'11, June 12–16, 2011, Athens, Greece.

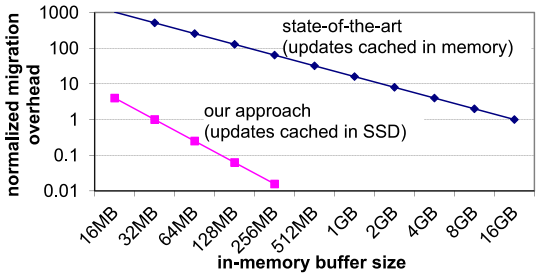


Figure 1: An analysis of migration overheads for differential updates as a function of the memory buffer size. Overhead is normalized to the prior state-of-the-art using 16GB memory.

on an *in-memory* buffer for the cached updates poses a fundamental trade-off between migration overhead and memory footprint, as illustrated by the "state-of-the-art" curve in Figure 1 (note: log-log scale, the lower the better). In order to halve the migration costs, one must double the in-memory buffer size so that migrations occur (roughly) half as frequently. Each migration is expensive, incurring the cost of scanning the entire DW, applying the updates and writing back the results [22, 11]. However, dedicating a significant fraction of the system memory solely to buffering updates degrades query operator performance as less memory is available for caching frequently accessed data structures (e.g., indices) and storing intermediary results (e.g., in sorting, hash-joins). Moreover, in case of a crash, the large buffer of updates in memory will be lost, prolonging crash recovery.

1.2 Our Solution: Cache Updates in SSDs

We exploit the recent trend towards including a small amount of flash storage (SSDs) in mainly HDD-based computer systems [1]. Our approach follows the differential updates idea discussed above, but instead of being limited to an in-memory buffer, makes judicious use of available SSDs to cache incoming updates. Figure 2 presents the high-level framework. Updates are stored in an SSD-based update cache, which is 1%–10% of the main data size. When a query reads data, the relevant updates on SSDs are located, read, and merged with the bulk of the data coming from disks. A small in-memory buffer is used as a staging area for the efficient processing of queries and incoming updates. The updates are migrated to disks only when the system load is low or when updates reach a certain threshold (e.g., 90%) of the SSD size.

Design Goals. We aim to achieve the following five design goals:

- *Low query overhead with small memory footprint:* This addresses the main limitations of prior approaches.
- *No random SSD writes:* While SSDs have excellent sequential read/write and random read performance, random writes perform poorly because they often incur expensive erase and wear-leveling operations [4]. Moreover, frequent random writes can transition an SSD into sub-optimal states where even the well-supported operations suffer from degraded performance [4, 21].
- *Low total SSD writes per update:* A NAND flash cell can endure only a limited number of writes (e.g., 10^5 writes for enterprise SSDs). Therefore, the SSDs' lifetime is maximized if we minimize the amount of SSD writes per incoming update.
- *Efficient in-place migration:* Migrations should occur infrequently while supporting high sustained update rate. Moreover, prior approaches [11, 22] migrate updates to a new copy of the DW and swap it in after migration completes, essentially doubling the disk capacity requirement. We want to remove such requirement by migrating to the main data in place.
- *Correct ACID support:* We must guarantee that traditional concurrency control and crash recovery techniques still work.

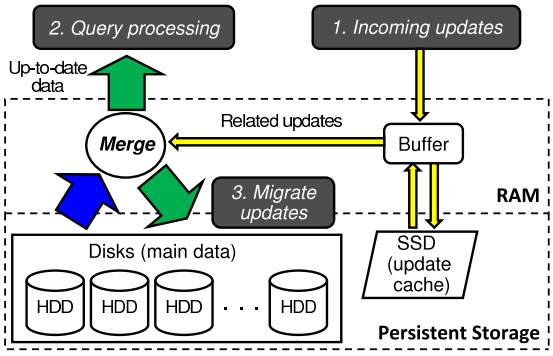


Figure 2: Framework for SSD-based differential updates.

Prior differential update approaches [22, 11] maintain indexes on the cached updates in memory, which we call Indexed Updates (IU). We find that naively extending IU to SSDs incurs up to 3.8X query slowdowns (Section 2.3 and 4.2). While employing log structured merge-trees (LSM) [15] can address many of IU's performance problems, LSM incurs a large number of writes per update, significantly reducing the SSDs' lifetime (Section 2.3).

At a high level, our framework is similar to the way Bigtable [7] handles incoming updates by caching them in HDDs and merging related updates into query responses. However, Bigtable's design is focused on neither low overhead for DW queries with small memory footprint, using SSDs and minimizing SSD writes, nor correct ACID support for multi-row transactions. Using SSDs instead of HDDs for the update cache is crucial to our design, as it reduces range scan query overhead by orders of magnitude for small ranges.

Our Proposal: MaSM. We propose MaSM (*materialized sort-merge*) algorithms that achieve the five design goals with the following techniques. First, we observe that the "Merge" component in Figure 2 is essentially an outer join between the main data on disks and the updates cached on SSDs. Among various join algorithms, we find that sort-merge joins fit the current context well: cached updates are sorted according to the layout order of the main data and then merged with the main data. We exploit external sorting algorithms both to achieve *small memory footprint* and to *avoid random writes* to SSDs. To sort $\|SSD\|$ pages of cached updates on SSD, two-pass external sorting requires $M = \sqrt{\|SSD\|}$ pages of memory. Compared with differential updates limited to an in-memory update cache, our approach can effectively use a small memory footprint, and exploits the larger on-SSD cache to greatly reduce migration frequency, as shown in the "our approach" curve in Figure 1.

Second, we optimize the two passes of the "Merge" operation: generating sorted runs and merging sorted runs. For the former, because a query should see all the updates that an earlier query has seen, we materialize and reuse sorted runs, amortizing run generation costs across many queries. For the latter, we build simple read-only indexes on materialized runs in order to reduce the SSD read I/Os for a particular query. Combined with the excellent sequential/random read performance of SSDs, this technique successfully achieves *low query overhead* (at most only 7% slowdowns in our experimental evaluation).

Third, we consider the *trade-off between memory footprint and SSD writes*. The problem is complicated because allocated memory is used for processing both incoming updates and queries. We first present a MaSM-2M algorithm, which achieves the minimal SSD writes per update, but allocates M memory for incoming updates and M memory for query processing. Then, we present a more sophisticated MaSM-M algorithm that reduces memory footprint

to M but incurs extra SSD writes. We select optimal algorithm parameters to minimize SSD writes for MaSM-M. After that, we generalize the two algorithms into a MaSM-αM algorithm. By varying α, we can obtain a spectrum of algorithms with different trade-offs between memory footprint and SSD writes.

Fourth, in order to support *in-place migration* and *ACID properties*, we propose to attach timestamps to updates, data pages, and queries. Using timestamps, MaSM can determine whether or not a particular update has been applied to a data page, thereby enabling concurrent queries during in-place migration. Moreover, MaSM guarantees serializability in the timestamp order among individual queries and updates. This can be easily extended to support two-phase locking and snapshot isolation for general transactions involving both queries and updates. Furthermore, crash recovery can use the timestamps to determine and recover only the updates in the memory buffer, but not those on (non-volatile) SSDs.

Finally, we minimize the impact of MaSM on the DBMS code in order to reduce the development effort to adopt the solution. Specifically, MaSM can be implemented in the storage manager (with minor changes to the transaction manager if general transactions are to be supported). It does not require modification to the buffer manager, query processor or query optimizer.

1.3 Contributions

This paper makes the following contributions. First, to our knowledge, this is the first paper that exploits SSDs for efficient online updates in DWs. We propose a high-level framework and identify five design goals for a good SSD-based solution. Second, we propose MaSM algorithms that exploit a set of techniques to successfully achieve the five design goals. Third, we study the trade-off between memory footprint and SSD writes with MaSM-2M, MaSM-M, and MaSM-αM. Fourth, we present a real-machine experimental study. Our results show that MaSM incurs only up to 7% overhead both on synthetic range scans (varying range size from 100GB to 4KB) and in a TPC-H query replay study, while also increasing the sustained update throughput by orders of magnitude. Finally, we discuss considerations for various DW-related aspects, including shared nothing architectures, Extract-Transform-Load (ETL) processes, secondary indexes, and materialized views.

Outline. Section 2 sets the stage for our study. Section 3 presents MaSM design for achieving the five design goals. Section 4 presents the experimental evaluation. Section 5 discusses related work and considerations for various DW-related issues. Section 6 concludes.

2. EFFICIENT ONLINE UPDATES AND RANGE SCANS IN DATA WAREHOUSES

In this section, we first describe the basic concepts and clarify the focus of our study. As in most optimization problems, we would like to achieve good performance for the frequent use cases, while providing correct functional support in general. After that, we analyze limitations of prior approaches for handling online updates.

2.1 Basic Concepts and Focus of the Study

Data Warehouse. Our study is motivated by large analytical DWs such as those characterized in the XLDB'07 report [2]. There is typically a front-end operational (e.g., OLTP) system that produces the updates for the back-end analytical DW. The DW can be very large (e.g., petabytes), and does not fit in main memory.

Query Pattern. Large analytical DWs often observe "highly unpredictable query loads", as described in [2]. Most queries involve "summary or aggregative queries spanning large fractions of the database". As a result, table range scans are frequently used in the queries. Therefore, we will focus on table range scans as the optimization target: preserving the nice sequential data access patterns of table range scans in the face of online updates. The evaluation will use the TPC-H benchmark, which is a decision support benchmark with emphasis on ad-hoc queries.

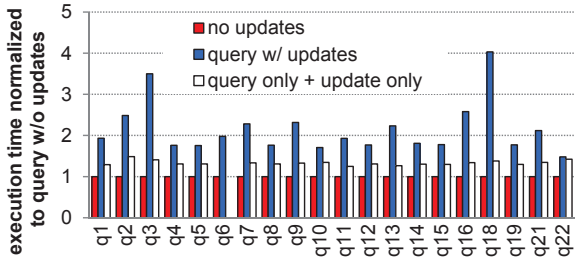


Figure 3: TPC-H queries with random updates on a row store.

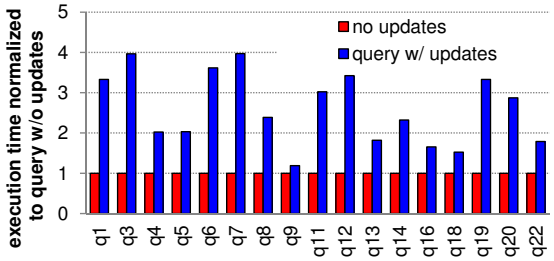


Figure 4: TPC-H queries with emulated random updates on a column store.

Record Order. In row stores, records are often stored in primary key order (with clustered indices). In column stores (that support online updates), the attributes of a record are aligned in separate columns allowing retrieval using a position value (RID) [11].[1] We assume that range scans provide data in the order of primary keys in row stores, and in the order of RIDs in column stores. Whenever primary keys in row stores and RIDs in column stores can be handled similarly, we use "key" to mean both.

Updates. Following prior work on differential updates [11], we optimize for incoming updates of the following forms: (i) inserting a record given its key; (ii) deleting a record given its key; or (iii) modifying the field(s) of a record to specified new value(s) given its key.[2,3] We call these updates *well-formed* updates. Data in large analytical DWs are often "write-once, read-many" [2], a special case of well-formed updates. Note that well-formed updates do not require reading existing DW data. In contrast, general transactions can require an arbitrary number of reads and writes. This distinction is important because the reads in general transactions may inherently require I/O reads to the main data and thus interfere with the sequential data access patterns in table range scans. For well-formed updates, our goal is to preserve range scan performance as if there were no online updates. For general transactions involving both reads and updates, we provide correct functionality, while achieving comparable or better performance than conventional online update handling, which we will discuss in Section 4.2.

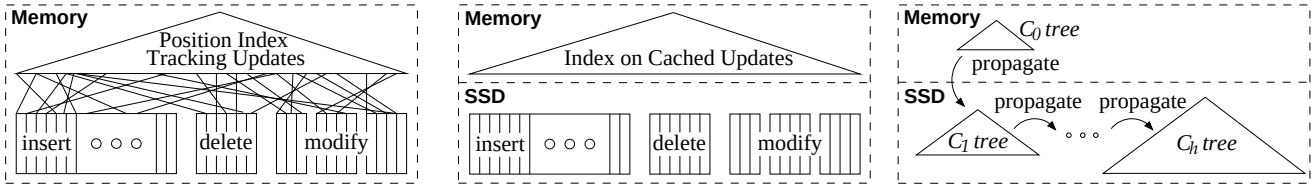
2.2 Conventional Approach: In-Place Updates

In order to clarify the impact of online random updates in ana-

[1] Following prior work [11], we focus on a single sort order for the columns of a table. We discuss how to support multiple sort orders in Section 5.

[2] We follow prior work on column stores to assume that the RID of an update is provided [11]. For example, if updates contain sort keys, RIDs may be obtained by searching the (in-memory) index on sort keys.

[3] A modification that changes the key is treated as a deletion given the old key followed by an insertion given the new key.



(a) In-memory IU (Positional Delta Tree [11]) (b) Directly extending IU to SSDs (c) Applying LSM [15] to IU

Figure 5: Extending prior proposals of Indexed Updates (IU) to SSDs.

lytic workloads, we execute TPC-H queries on both a commercial row-store *DBMS R* and a commercial column-store *DBMS C* while running online in-place updates.[4] The TPC-H scale factor is 30. We make sure that the database on disk is much larger than the allocated memory buffer size. We were able to perform concurrent updates and queries on the row store. However, the column store supports only offline updates, i.e., without concurrent queries. We recorded the I/O traces of offline updates, and when running queries on the column store, we use a separate program to replay the I/O traces outside of the DBMS to emulate online updates. During replay, we convert all I/O writes to I/O reads so that we can replay the disk head movements without corrupting the database.

Figure 3 compares the performance of TPC-H queries with no updates (first bar) and queries with online updates (second bar) on *DBMS R*. The third bar shows the sum of the first bar and the time for applying the same updates offline. Each cluster is normalized to the execution time of the first bar. Disk traces show sequential disk scans in all queries. As shown in Figure 3, queries with online updates see 1.5–4.1X slowdowns (2.2X on average), indicating significant performance degradation because of the random accesses of online updates. Moreover, the second bar is significantly higher than the third bar in most queries (with an average 1.6X extra slowdown). This shows that the increase in query response time is a result of not only having two workloads executing concurrently, but also the interference between the two workloads. Figure 4 shows a similar comparison for the column store *DBMS C*. Compared with queries with no updates, running in-place updates online slows down the queries by 1.2–4.0X (2.6X on average).

2.3 Prior Proposals: Indexed Updates (IU)

Differential updates is the state-of-the-art technique for reducing the impact of online updates [11, 22]. While the basic idea is straightforward (as described in Section 1), the efforts of prior work and this paper are on the data structures and algorithms for efficiently implementing differential updates.

In-Memory Indexed Updates. Prior proposals maintain the cache for updates in main memory and build indexes on the cached updates [11, 22], which we call Indexed Updates (IU). Figure 5(a) shows the state-of-the-art IU proposal, Positional Delta Tree (PDT) designed for column stores [11]. PDT caches updates in an insert table, a delete table, and a modify table per database attribute. It builds a positional index on the cached updates using RID as the index key. Incoming updates are appended to the relevant insert/delete/modify tables. During query processing, PDT looks up the index with RIDs to retrieve relevant cached updates. Therefore, the PDT tables will be accessed in a random fashion during a range scan on the main data. Migration of the updates is handled by creating a separate copy of the main data, then making the new copy available when migration completes. Note that this requires twice as much data storage capacity as the main data size.

[4]We were able to run 20 TPC-H queries on *DBMS R* and 17 TPC-H queries on *DBMS C*. The rest of the queries either do not finish in 24 hours, or are not supported by the DBMS.

Problems of Directly Extending IU to SSDs. As discussed in Section 1, we aim to develop an SSD-based differential update solution that achieves the five design goals. To start, we consider directly extending IU to SSDs. As shown in Figure 5(b), the cached updates in insert/delete/modify tables are on SSDs. In order to *avoid random SSD writes*, incoming updates should be appended to these tables. For the same reason, ideally, the index is placed in memory because it sees a lot of random writes to accommodate incoming updates. Note that the index may consume a large amount of main memory, reducing the SSDs' benefit of saving memory footprint. We implemented this ideal-case IU following the above considerations (ignoring any memory footprint requirement). However, real-machine experiments show up to 3.8X query slowdowns even for this ideal-case IU (Section 4.2). We find that the slowdown is because the insert/delete/modify tables are randomly read during range scan operations. This is wasteful as an entire SSD page has to be read and discarded for retrieving a single update entry.

Problems of Applying LSM to IU. The log-structured merge-tree (LSM) is a disk-based index structure designed to support a high rate of insertions [15]. An LSM consists of multiple levels of trees of increasing sizes. PDT employs the idea of multiple levels of trees to support snapshot isolation in memory [11]. Here, we consider the feasibility of combining LSM and IU as an SSD-based solution.

As shown in Figure 5(c), LSM keeps the smallest C_0 tree in memory, and C_1, ..., C_h trees on SSDs, where $h \geq 1$. Incoming updates are first inserted into C_0, then gradually propagate to the SSD-resident trees. There are asynchronous rolling propagation processes between every adjacent pair (C_i, C_{i+1}) that (repeatedly) sequentially visit the leaf nodes of C_i and C_{i+1}, and move entries from C_i to C_{i+1}. This scheme avoids many of IU's performance problems. Random writes can be avoided by using large sequential I/Os during propagation. For a range scan query, it performs corresponding index range scans on every level of LSM, thereby avoiding wasteful random I/Os as in the above ideal-case IU.

Unfortunately, *LSM incurs a large amount of writes per update entry, violating the third design goal*. The additional writes arise in two cases: (i) An update entry is copied h times from C_0 to C_h; and (ii) the propagation process from C_i to C_{i+1} rewrites the old entries in C_{i+1} to SSDs once per round of propagation. According to [15], in an optimal LSM, the sizes of the trees form a geometric progression. That is, $size(C_{i+1})/size(C_i) = r$, where r is a constant parameter. It can be shown that in LSM the above two cases introduce about $r + 1$ writes per update for levels $1, \ldots, h-1$ and $(r + 1)/2$ writes per update for level h. As an example, with 4GB flash space and 16MB memory (which is our experimental setting in Section 4.1), we can compute that a 2-level ($h = 1$) LSM writes every update entry ≈ 128 times. The optimal LSM that minimizes total writes has $h = 4$ and it writes every update entry ≈ 17 times! In other words, compared to a scheme that writes every update entry once, applying LSM on an SSD reduces its lifetime 17 fold (e.g., from 3 years to 2 months).

3. MaSM DESIGN

In this section, we propose MaSM (*materialized sort-merge*) al-

gorithms for achieving the five design goals. We start by describing the basic ideas in Section 3.1. Then we present two MaSM algorithms: MaSM-2M in Section 3.2 and MaSM-M in Section 3.3. MaSM-M halves the memory footprint of MaSM-2M but incurs extra SSD writes. In Section 3.4, we generalize these two algorithms into a MaSM-αM algorithm, allowing a range of trade-offs between memory footprint and SSD writes by varying α. After that, we discuss a set of optimizations in Section 3.5, and describe transaction support in Section 3.6. Finally, we analyze the MaSM algorithms in terms of the five design goals in Section 3.7.

3.1 Basic Ideas

Consider the operation of merging a table range scan and the cached updates. For every record retrieved by the scan, it finds and applies any matching cached updates. Records without updates or new insertions must be returned too. Essentially, this is an outer join operation on the record key (primary key/RID).

Among various join algorithms, we choose to employ a sort-based join that sorts the cached updates and merges the sorted updates with the table range scan. This is because the most efficient joins are typically sort-based or hash-based, but hash-based joins have to perform costly I/O partitioning for the main data. The sort-based join also preserves the record order in table range scans, which allows hiding the implementation details of the merging operation from the query optimizer and upper-level query operators.

To reduce memory footprint, we keep the cached updates on SSDs and perform external sorting of the updates; two-pass external sorting requires $M = \sqrt{\|SSD\|}$ pages in memory to sort $\|SSD\|$ pages of cached updates on SSDs. However, external sorting may incur significant overhead for generating sorted runs and merging them. We exploit the following two ideas to reduce the overhead. First, we observe that a query should see all the cached updates that a previous query has seen. Thus, we materialize sorted runs of updates, deleting the generated runs only after update migration. This amortizes the cost of sorted run generation across many queries. Second, we would like to prune as many irrelevant updates to the current range scan query as possible. Because materialized runs are read-only, we can create a simple, read-only index, called *a run index*, to record the smallest key (primary key/RID) for every fixed number of SSD pages in a sorted run. Then we can search the query's key range in the run index to retrieve only those SSD pages that fall in the range. We call the algorithm combining the above ideas the *Materialized Sort-Merge (MaSM)* algorithm.

The picture of the above design is significantly complicated by the interactions among incoming updates, range scans, and update migrations. For example, sharing the memory buffer between updates and queries makes it difficult to achieve a memory footprint of M. In-place migrations may conflict with ongoing queries. Concurrency control and crash recovery must be re-examined. In the following, we first present a simple MaSM algorithm that requires $2M$ memory and a more sophisticated algorithm that reduces the memory requirement to M, then generalize them into an algorithm requiring αM memory. We propose a timestamp-based approach to support in-place migrations and ACID properties.

3.2 MaSM-2M

Figure 6 illustrates MaSM-2M, which allocates M pages to cache recent updates in memory and (up to) M pages for supporting a table range scan. Incoming (well-formed) updates are inserted into the in-memory buffer. When the buffer is full, MaSM-2M flushes the in-memory updates and creates a materialized sorted run of size M on the SSD. There will be at most M materialized sorted runs since the capacity of the SSD is M^2. For a table range scan, MaSM-2M allocates one page in memory for scanning every materialized sorted run—up to M pages. It builds a Volcano-style [9] operator tree to merge main data and updates, replacing the original `Table_range_scan` operator in query processing. We describe the detailed algorithm in the following.

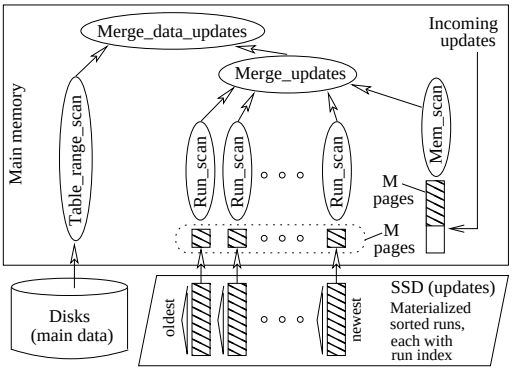


Figure 6: Illustrating the MaSM algorithm using $2M$ memory.

Timestamps. We associate every incoming update with a timestamp, which represents the commit time of the update. Every query is also assigned a timestamp. We ensure that a query can only see earlier updates with smaller timestamps. Moreover, we store in every database page the timestamp of the last update applied to the page, for supporting in-place migration. To do this, we reuse the Log Sequence Number (LSN) field in the database page. This field is originally used in recovery processing to decide whether to perform a logged redo operation on the page. However, in the case of MaSM, the LSN field is not necessary because recovery processing does not access the main data, rather it recovers the in-memory buffer for recent updates.

Update Record. For an incoming update, we construct a record of the format `(timestamp, key, type, content)`. As discussed in Section 2.1, table range scans output records in key order, either the primary key in a row store or the RID in a column store, and well-formed updates contain the key information. The `type` field is one of `insert/delete/modify/replace`; `replace` represents a deletion merged with a later insertion with the same key. The `content` field contains the rest of the update information: for `insert/replace`, it contains the new record except for the key; for `delete`, it is null; for `modify`, it specifies the attribute(s) to modify and the new value(s). During query processing, a `getnext` call on `Merge_data_updates` will incur `getnext` on operators in the subtree. `Run_scan` and `Mem_scan` scan the associated materialized sorted runs and the in-memory buffer, respectively. The `(begin key, end key)` of the range scan is used to narrow down the update records to scan. `Merge_updates` merges multiple streams of sorted updates. For updates with the same key, it merges the updates correctly. For example, the `content` fields of multiple modifications are merged. A deletion followed by an insertion will change the `type` field to `replace`. `Merge_data_updates` performs the desired outer-join like merging operation of the data records and the update records.

Incoming Update. The steps to process an incoming update are:

1: **if** In-memory buffer is full **then**
2: Sort update records in the in-memory buffer in key order;
3: Build a run index recording `(begin key, SSD page)`;
4: Create a new materialized sorted run with run index on SSD;
5: Reset the in-memory buffer;
6: **end if**
7: Append the incoming update record to the in-memory buffer;

Table Range Scan. MaSM-2M constructs a Volcano-style query operator tree to replace the `Table_range_scan` operator:

1: Instantiate a `Run_scan` operator per materialized sorted run using the run index to narrow down the SSD pages to retrieve;
2: Sort the in-memory buffer for recent update records;
3: Instantiate a `Mem_scan` operator on the in-memory buffer and locate the begin and end update records for the range scan;
4: Instantiate a `Merge_updates` operator as the parent of all the `Run_scan` operators and the `Mem_scan` operator;
5: Instantiate a `Merge_data_updates` operator as the parent of the `Table_range_scan` and the `Merge_updates`;
6: return `Merge_data_updates`;

Online Updates and Range Scan. Usually, incoming updates are appended to the end of the update buffer, and therefore do not interfere with ongoing `Mem_scan`. However, flushing the update buffer must be handled specially. MaSM records a flush timestamp with the update buffer therefore `Mem_scan` can discover the flushing. When this happens, `Mem_scan` will instantiate a `Run_scan` operator for the new materialized sorted run and replaces itself with the `Run_scan` in the operator tree. The update buffer must be protected by latches (mutexes). To reduce latching overhead, `Mem_scan` retrieves multiple update records at a time.

Multiple Concurrent Range Scans. When multiple range scans enter the system concurrently, each one builds its own query operator tree with distinct operator instances, and separate read buffers for the materialized sorted runs. `Run_scan` performs correctly because materialized sorted runs are read-only. On the other hand, the in-memory update buffer is shared by all `Mem_scan` operators, which sort the update buffer then read sequentially from it. For reading the buffer sequentially, each `Mem_scan` tracks `(key, pointer)` pairs for the `next` position and the `end_range` position in the update buffer. To handle sorting, MaSM records a sort timestamp with the update buffer whenever it is sorted. In this way, `Mem_scan` can detect a recent sorting. Upon detection, `Mem_scan` adjusts the pointers of the `next` and `end_range` positions by searching for the corresponding keys. There may be new update records that fall between the two pointers after sorting. `Mem_scan` correctly filters out the new update records based on the query timestamp.

In-Place Migration of Updates Back to Main Data. Migration is implemented by performing a full table scan where the scan output is written back to disks. Compared to a normal table range scan, there are two main differences. First, the `(begin key, end key)` is set to cover the entire range. Second, `Table_range_scan` returns a sequence of data pages rather than a sequence of records. `Merge_data_updates` applies the updates to the data pages in the database buffer pool, then issues (large sequential) I/O writes to write back the data pages. For compressed data chunks in a column store, the data is uncompressed in memory, modified, recompressed, and written back to disks. The primary key index or the RID position index is updated along with every data page. Here, by in-place migration, we mean two cases: (i) new data pages overwrite the old pages with the same key ranges; or (ii) after writing a new chunk of data, the corresponding old data chunk is deleted so that its space can be used for writing other new data chunks.

The migration operation is performed only when the system is under low loads or when the size of cached updates is above a prespecified threshold. When one of the conditions is true, MaSM logs the current timestamp t and the IDs of the set R of current materialized sorted runs in the redo log, and spawns a migration thread. The thread waits until all ongoing queries earlier than t complete, then migrates the set R of materialized runs. When migration completes, it logs a migration completion record in the redo log then deletes the set R of materialized runs. Note that new queries arriving after t are handled correctly. They must see all the updates being migrated. A new range scan can compare the timestamp of a data page and the timestamp of an update in `Merge_data_updates` to determine whether the update has already been applied to the data page, thereby correctly handling both the before-migration and the after-migration data pages.

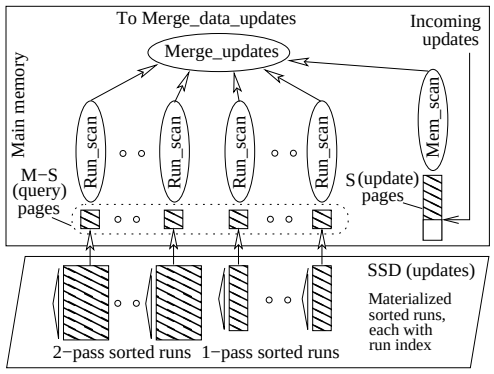


Figure 7: Illustrating MaSM algorithm using M memory.

3.3 MaSM-M

Figure 7 illustrates MaSM-M, the MaSM algorithm using M memory. There are two main differences between MaSM-M and MaSM-2M. First, compared to MaSM-2M, MaSM-M manages memory more carefully to reduce its memory footprint to M pages. S of the M pages, called update pages, are dedicated to buffering incoming updates in memory. The rest of the pages, called query pages, are mainly used to support query processing. Second, materialized sorted runs have different sizes in MaSM-M. Since the query pages can support up to only $M-S$ materialized sorted runs, the algorithm has to merge multiple smaller materialized sorted runs into larger materialized sorted runs. We call the sorted runs that are directly generated from the in-memory buffer *1-pass* runs, and those resulted from merging 1-pass runs *2-pass* runs.

Figure 8 presents the pseudo code of MaSM-M. Algorithm parameters are summarized in Table 1. Incoming updates are cached in the in-memory buffer until the buffer is full. At this moment, the algorithm tries to steal query pages for caching updates if they are not in use (Lines 2–3). The purpose is to create *1-pass* runs as large as possible for reducing the need for merging multiple runs. When query pages are all in use, the algorithm creates a 1-pass materialized sorted run with run index on SSDs (Line 5).

The table range scan algorithm consists of three parts. First, Lines 1–4 create a *1-pass* run if the in-memory buffer contains at least S pages of updates. Second, Lines 5–8 guarantee that the number of materialized sorted runs is at most $M-S$, by (repeatedly) merging the N earliest 1-pass sorted runs into a single 2-pass sorted run until $K_1 + K_2 \leq M - S$. Note that the N earliest 1-pass runs are adjacent in time order, and thus merging them is meaningful. Since the size of a 1-pass run is at least S pages, the size of a 2-pass sorted run is at least NS pages. Finally, Lines 9–14 construct the query operator tree. This part is similar to MaSM-2M.

Like MaSM-2M, MaSM-M handles concurrent range scans, updates, and in-place migration using the timestamp approach.

Minimizing SSD Writes for MaSM-M. We choose the S and N parameters to minimize SSD writes for MaSM-M.

Lemma 3.1. *The size of a 1-pass materialized sorted run is at least S. The size of a 2-pass materialized sorted run is at least NS.*

PROOF. These are clear from the algorithm in Figure 8. □

Incoming Updates:

1: **if** In-memory buffer is full **then**
2: **if** At least one of the query pages is not used **then**
3: Steal a query page to extend the in-memory buffer;
4: **else**
5: Create a 1-pass materialized sorted run with run index from the updates in the in-memory buffer;
6: Reset the in-memory buffer to have S empty pages;
7: **end if**
8: **end if**
9: Append the incoming update record to the in-memory buffer;

Table Range Scan Setup:

1: **if** In-memory buffer contains at least S pages of updates **then**
2: Create a 1-pass materialized sorted run with run index from the updates in the in-memory buffer;
3: Reset the in-memory buffer to have S empty update pages;
4: **end if**
5: **while** $K_1 + K_2 > M - S$ **do** {*merge 1-pass runs*}
6: Merge N earliest adjacent 1-pass runs into a 2-pass run;
7: $K_1 = K_1 - N$; K_2++;
8: **end while**
9: Instantiate a `Run_scan` operator per materialized sorted run using the run index to narrow down SSD pages to retrieve;
10: **if** In-memory buffer is not empty **then**
11: Sort the in-memory buffer and create a `Mem_scan` operator;
12: **end if**
13: Instantiate a `Merge_updates` operator as the parent of all the `Run_scan` operators and the `Mem_scan` operator;
14: Instantiate a `Merge_data_updates` operator as the parent of the `Table_range_scan` and the `Merge_updates`;
15: return `Merge_data_updates`;

Figure 8: MaSM-M algorithm

Theorem 3.2. *The MaSM-M algorithm minimizes the number of SSD writes in the worst case when $S_{opt} = 0.5M$ and $N_{opt} = 0.375M + 1$. The average number of times that MaSM-M writes every update record to SSD is $1.75 + \frac{2}{M}$.*

PROOF. Every update record is written at least once to a 1-pass run. Extra SSD writes occur when 1-pass runs are merged into a 2-pass run. We choose S and N to minimize the extra writes.

The worst case scenario happens when all the 1-pass sorted runs are of the minimal size S. In this case, the pressure for generating 2-pass runs is the greatest. Suppose all the 1-pass runs have size S and all the 2-pass runs have size NS. We must make sure that when the allocated SSD space is full, MaSM-M can still merge all the sorted runs. Therefore:

$$K_1 S + K_2 NS = M^2 \quad \text{and} \quad K_1 + K_2 \leq M - S$$

Compute K_1 from the first equation and plug it into the inequality:

$$K_2 \geq \frac{1}{N-1}(S - M + \frac{M^2}{S}) \tag{1}$$

When the SSD is full, the total extra writes is equal to the total size of all the 2-pass sorted runs:

$$ExtraWrites = K_2 NS \geq \frac{N}{N-1}(S^2 - MS + M^2) \tag{2}$$

To minimize $ExtraWrites$, we would like to minimize the right hand side and achieve the equality sign as closely as possible. The right hand side achieves the minimum when N takes the largest possible value and $S_{opt} = 0.5M$. Plug it into (1) and (2):

$$K_2 \geq \frac{1.5M}{N-1} \text{ iff } N \geq \frac{1.5M}{K_2} + 1 \tag{3}$$

$$ExtraWrites = 0.5MK_2N \geq 0.75M^2 \frac{N}{N-1} \tag{4}$$

Note that the equality signs in the above two inequalities are achieved at the same time. Given a fixed K_2, the smaller the N, the lower the $ExtraWrites$. Therefore, $ExtraWrites$ is minimized when the equality signs hold. We can rewrite $min(ExtraWrites)$ as a function of K_2:

$$min(ExtraWrites) = 0.75M^2(1 + \frac{K_2}{1.5M}) \tag{5}$$

The global minimum is achieved with the smallest non-negative integer K_2. Since $N \leq M - S_{opt} = 0.5M$, we know $K_2 > 3$ according to (3).

Therefore, $ExtraWrites$ achieves the minimum when $S_{opt} = 0.5M$, $K_{2,opt} = 4$, and $N_{opt} = 0.375M + 1$. We can compute the minimum $ExtraWrites = 0.75M^2 + 2M$. In this setting, the average number of times that MaSM-M writes every update record to SSD is $1 + (0.75M^2 + 2M)/M^2 = 1.75 + \frac{2}{M} \simeq 1.75$. □

Table 1: Parameters used in the MaSM-M algorithm.

$\|SSD\|$	SSD capacity (in pages), $\|SSD\| = M^2$
M	memory size (in pages) allocated for MaSM-M
S	memory buffer (in pages) allocated for incoming updates
K_1	number of 1-pass sorted runs on SSDs
K_2	number of 2-pass sorted runs on SSDs
N	merge N 1-pass runs into a 2-pass run, $N \leq M - S$

3.4 MaSM-αM

We generalize the MaSM-M algorithm to a MaSM-αM algorithm that uses αM memory, where $\alpha \in [\frac{2}{\sqrt[3]{M}}, 2]$. The algorithm is the same as MaSM-M except that the total allocated memory size is αM pages. The lower bound on α ensures that the memory is sufficiently large to make 3-pass sorted runs unnecessary. MaSM-M is a special case of MaSM-αM when $\alpha = 1$, and MaSM-2M is a special case of MaSM-αM when $\alpha = 2$. Similar to the proof of Theorem 3.2, we can obtain the following theorem for minimizing SSD writes for MaSM-αM.

Theorem 3.3. *The MaSM-αM algorithm minimizes the number of SSD writes in the worst case when $S_{opt} = 0.5\alpha M$ and $N_{opt} = \frac{1}{\lfloor \frac{4}{\alpha^2} \rfloor}(\frac{2}{\alpha} - 0.5\alpha)M + 1$. The average number of times that MaSM-M writes every update record to SSD is roughly $2 - 0.25\alpha^2$.*

We can verify that MaSM-M incurs roughly $2 - 0.25 * 1^2 = 1.75$ writes per update record, while MaSM-2M writes every update record once ($2 - 0.25 * 2^2 = 1$).

Theorem 3.3 shows the trade-off between memory footprint and SSD writes for a spectrum of MaSM algorithms. At one end of the spectrum, MaSM-2M achieves minimal SSD writes with 2M memory. At the other end of the spectrum, one can achieve a MaSM algorithm with very small memory footprint ($2\sqrt[3]{M^2}$) while writing every update record at most twice.

3.5 Further Optimizations

Granularity of Run Index. Because run indexes are read-only, their granularity can be chosen flexibly. For example, suppose the page size of the sorted runs on SSDs is 64KB. Then we can keep the begin key for a coarser granularity, such as one key per 1MB, or a finer granularity, such as one key per 4KB. The run index should be cached in memory for efficient accesses (especially if there are a

lot of small range scans). Coarser granularity saves memory space, while finer granularity makes range scans on the sorted run more precise. The decision should be made based on the query workload. The former is a good choice if very large ranges are typical, while the latter should be used if narrower ranges are frequent. For indexing purpose, one can keep a 4-byte prefix of the actual key in the run index. Therefore, the fine-grain indexes that keep 4 bytes for every 4KB updates are $\|SSD\|/1024$ large. If keeping 4 bytes for every 1MB updates, the total run index size is $\|SSD\|/256K$ large. In both cases the space overhead on SSD is very small.

Handling Skews in Incoming Updates. When updates are highly skewed, there may be many duplicate updates (i.e., updates to the same record). The MaSM algorithms naturally handle skews: When generating a materialized sorted run, the duplicates can be merged in memory as long as the merged update records do not affect the correctness of concurrent range scans. That is, if two update records with timestamp $t1$ and $t2$ are to be merged, there should not be any concurrent range scans with timestamp t, such that $t1 < t \leq t2$. In order to further reduce duplicates, one can compute statistics about duplicates at the range scan processing time. If the benefits of removing duplicates outweigh the cost of SSD writes, one can remove all duplicates by generating a single materialized sorted run from all existing runs.

Improving Migration. There are several ways to improve the migration operation. First, similar to coordinated scans [8], we can combine the migration with a table scan query in order to avoid the cost of performing a table scan for migration purposes only. Second, one can migrate a portion (e.g., every $1/10$ of table range) of updates at a time to distribute the cost across multiple operations. To do this, each materialized sorted run will record the ranges that have already been migrated and the ranges that are still active.

3.6 Transaction Support

Serializability among Individual Queries and Updates. By using timestamps, MaSM algorithms guarantee that queries see only earlier updates. In essence, the timestamp order defines a total serial order, and thus MaSM algorithms guarantee serializability among individual queries and updates.

Supporting Snapshot Isolation for General Transactions. In snapshot isolation [3], a transaction works on the snapshot of data as seen at the beginning of the transaction. If multiple transactions modify the same data item, the first committer wins while the other transactions abort and roll back. Note that snapshot isolation alone does not solve the online updates problem in DWs. While snapshot isolation removes the logical dependencies between updates and queries so that they may proceed concurrently, the physical interferences between updates and queries present major performance problems. Such interferences are the target of MaSM algorithms.

Similar to prior work [11], MaSM can support snapshot isolation by maintaining for every ongoing transaction a small private buffer for the updates performed by the transaction. (Note that such private buffers may already exist in the implementation of snapshot isolation.) A query in the transaction will have the timestamp of the transaction start time so that it sees only the snapshot of data at the the beginning of the transaction. To incorporate the transaction's own updates, we can instantiate a `Mem_scan` operator on the private update buffer, and insert this operator in the query operator tree in Figure 6 and 7. At commit time, if the transaction succeeds, we assign the commit timestamp to the private updates and copy them into the global in-memory update buffer.

Supporting Locking Schemes for General Transactions. Shared (exclusive) locks are used to protect reads (writes) in many database systems. MaSM can support locking schemes as follows. First, for an update, we ensure that it is globally visible only after the associated exclusive lock is released. To do this, we allocate a small private update buffer per transaction (similar to snapshot isolation), and cache the update in the private buffer. Upon releasing the exclusive lock that protects the update, we assign the current timestamp to the update record and append it to MaSM's global in-memory update buffer. Second, we assign the normal start timestamp to a query so that it can see all the earlier updates.

For example, two phase locking is correctly supported. In two phase locking, two conflicting transactions A and B are serialized by the locking scheme. Suppose A happens before B. Our scheme makes sure that A's updates are made globally visible at A's lock releasing phase, and B correctly see these updates.

Crash Recovery. Typically, MaSM needs to recover only the in-memory update buffer for crash recovery. This can be easily handled by reading the database redo log for the update records. It is easy to use update timestamps to distinguish updates in memory and updates on SSDs. In the rare case, the system crashes in the middle of an update migration operation. To detect such cases, MaSM records the start and the end of an update migration in the log. Note that we do not log the changes to data pages in the redo log during migration, because MaSM can simply redo the update migration during recovery processing: By using the timestamps in data pages, MaSM naturally determines whether updates should be applied to the data pages. The primary key index or RID position index is examined and updated accordingly.

3.7 Achieving The Five Design Goals

As described in Sections 3.2–3.6, it is clear that the MaSM algorithms perform *no random SSD writes* and provide *correct ACID support*. We analyze the other three design goals in the following.

Low Overhead for Table Range Scan Queries. Suppose that updates are uniformly distributed across the main data. If the main data size is $\|Disk\|$ pages, and the table range scan query accesses R disk pages, then MaSM-αM reads $max(R\frac{\|SSD\|}{\|Disk\|}, 0.5\alpha M)$ pages on SSDs. This formula has two parts. First, when R is large, run indexes can effectively narrow down the accesses to materialized sorted runs, and therefore MaSM performs $(R\frac{\|SSD\|}{\|Disk\|})$ SSD I/Os, proportional to the range size. Compared to reading the disk main data, MaSM reads fewer bytes from SSD, as $\frac{\|SSD\|}{\|Disk\|}$ is 1%–10%. Therefore, the SSD I/Os can be completely overlapped with the table range scan on main data, leading to very low overhead.

Second, when the range R is small, MaSM-αM performs at least one I/O per materialized sorted run: the I/O cost is bounded by the number of materialized sorted runs (up to $0.5\alpha M$). (Our experiments in Section 4.2 see 128 sorted runs for 100GB data.) Note that SSDs can support 100X–1000X random 4KB reads per second compared to disks [13]. Therefore, MaSM can overlap most latencies of the $0.5\alpha M$ random SSD reads with the small range scan on disks, achieving low overhead.

Memory Footprint vs. Total SSD Writes. MaSM-αM specifies a spectrum of MaSM algorithms trading off SSD writes for memory footprint. By varying α from 2 to $\frac{2}{\sqrt[3]{M}}$, the memory footprint reduces from $2M$ pages to $2\sqrt[3]{M^2}$ pages, while the average times that an update record is written to SSDs increases from the (minimal) 1 to close to 2. In all these algorithms, we achieve small memory footprint and low total SSD writes.

The write endurance of enterprise-grade SLC (Single Level Cell) NAND Flash is typically 10^5. Therefore, a 32GB Intel X25E SSD can support 3.2-petabyte writes, or 33.8MB/s for 3 years. MaSM-2M writes SSDs once for any update record, therefore a single SSD can sustain up to 33.8MB/s updates for 3 years. MaSM-M writes about

1.75 times for an update record. Therefore, for MaSM-M, a single SSD can sustain up to 19.3MB/s updates for 3 years, or 33.8MB/s for 1 year and 8 months. This limit can be improved by using larger total SSD capacity: doubling SSD capacity doubles this bound.

Efficient In-Place Migration. MaSM achieves in-place update migration by attaching timestamps to updates, queries, and data pages as described in Section 3.2.

We discuss two aspects of migration efficiency. First, MaSM performs efficient sequential I/O writes in a migration. Because update cache size is non-trivial (1%–10%) compared to main data size, it is likely that there exist update records for every data page. Compared to conventional random in-place updates, MaSM can achieve orders of magnitude higher sustained update rate, as will be shown in Section 4.2. Second, MaSM achieves low migration frequency with a small memory footprint. If SSD page size is P, then MaSM-αM uses $F = \alpha MP$ memory pages to support an SSD-based update cache of size $M^2P = \frac{F^2}{\alpha^2 P}$. Note that as the memory footprint doubles, the size of MaSM's update cache increases by a factor of 4, and the migration frequency decreases by a factor of 4, as compared to a factor of 2 with prior approaches that cache updates in memory (see Figure 1). For example, for MaSM-M, if $P = 64KB$, a 16GB in-memory update cache in prior approaches has the same migration overhead as just an $F = 32$MB in-memory buffer in our approach, because $F^2/(64\text{KB}) = 16\text{GB}$.

4. EXPERIMENTAL EVALUATION

We perform real-machine experiments to evaluate our proposed MaSM algorithm. We start by describing the experimental setup in Section 4.1. Then, we present experimental studies with synthetic data in Section 4.2 and perform experiments based on TPC-H traces recorded from a commercial database system in Section 4.3.

4.1 Experimental Setup

Machine Configuration. We perform all experiments on a Dell Precision 690 workstation equipped with a quad-core Intel Xeon 5345 CPU (2.33GHz, 8MB L2 cache, 1333MHz FSB) and 4GB DRAM running Ubuntu Linux with 2.6.24 kernel. We store the main table data on a dedicated SATA disk (200GB 7200rpm Seagate Barracuda with 77MB/s sequential read and write bandwidth). We cache updates on an Intel X25-E SSD [13] (with 250MB/s sequential read and 170MB/s sequential write bandwidth). All code is compiled with g++ 4.2.4 with "-O2".

Implementation. We implemented a prototype row-store DW, supporting range scans on tables. Tables are implemented as file system files with the slotted page structure. Records are clustered according to the primary key order. A range scan performs 1MB-sized disk I/O reads for high throughput unless the range size is less than 1MB. Incoming updates consist of insertions, deletions, and modifications to attributes in tables. We implemented three algorithms for online updates: (1) In-place updates; (2) IU (Indexed Updates); and (3) MaSM-M. In-place updates perform 4KB-sized read-modify-write I/Os to the main data on disk. The IU implementation caches updates on SSDs and maintains an index to the updates. We model the best performance for IU by keeping its index always in memory in order to avoid random SSD writes to the index. Note that this consumes much more memory than MaSM. Since the SSD has 4KB internal page size, IU uses 4KB-sized SSD I/Os. For MaSM, we experiment with the more sophisticated MaSM-M algorithm with smaller memory footprint. Note that MaSM-M provides performance lower bounds for MaSM-αM. By default, MaSM-M performs 64KB-sized I/Os to SSDs. Asynchronous I/Os (with libaio) are used to overlap disk and SSD I/Os, and to take advantage of the internal parallelism of the SSD.

Experiments with Synthetic Data. We generate a 100GB table with 100-byte sized records and 4-byte primary keys. The table is initially populated with even-numbered primary keys so that odd-numbered keys can be used to generate insertions. We generate updates randomly uniformly distributed across the entire table, with update types (insertion, deletion, or field modification) selected randomly. By default, we use 4GB flash space for caching updates, thus MaSM-M requires 16MB memory for 64KB SSD effective page size. We also study the impact of varying the flash space.

TPC-H replay experiments. We ran the TPC-H benchmark with scale factor $SF = 30$ (roughly 30GB database) on a commercial row-store DBMS and obtained the disk traces of the TPC-H queries using the Linux `blktrace` tool. We were able to obtain traces for 20 TPC-H queries except queries 17 and 20, which did not finish in 24 hours. By mapping the I/O addresses in the traces back to the disk ranges storing each TPC-H table, we see that all the 20 TPC-H queries perform (multiple) table range scans. Interestingly, the 4GB memory is large enough to hold the smaller relations in hash joins, therefore hash joins reduce to a single pass algorithm without generating temporary partitions. Note that MaSM aims to minimize memory footprint to preserve such good behaviors.

We replay the TPC-H traces using our prototype DW as follows. We create TPC-H tables with the same sizes as in the commercial database. We replay the query disk traces as the query workload on the real machine. We perform 1MB-sized asynchronous (prefetch) reads for the range scans for high throughput. Then we apply online updates to TPC-H tables using in-place updates and our MaSM-M algorithm. Although TPC-H provides a program to generate batches of updates, each generated update batch deletes records and inserts new records in a very narrow primary key range (i.e. 0.1%) of the `orders` and `lineitem` tables. To model the more general and challenging case, we generate updates to be randomly distributed across the `lineitem` and `orders` tables (which occupy over 80% of the total data size). We make sure that an `orders` record and its associated `lineitem` records are inserted or deleted together. For MaSM, we use 1GB flash space, 8MB memory, and 64KB sized SSD I/Os.

Measurement Methodology. We use the following steps to minimize OS and device caching effect: (i) opening all the (disk or SSD) files with `O_DIRECT|O_SYNC` flag to get around OS file cache; (ii) disabling the write caches in the disk and the SSD; (iii) reading an irrelevant large file from each disk and each SSD before every experiment so that the device read caches are cleared. In experiments on synthetic data, we randomly select 10 ranges for scans of 100MB or larger, and 100 ranges for smaller ranges. For the larger ranges, we run 5 experiments for every range. For the smaller ranges, we run 5 experiments, each performing all the 100 range scans back-to-back (to reduce the overhead of OS reading file inodes and other metadata). In TPC-H replay experiments, we perform 5 runs for each query. We report the averages of the runs. The standard deviations are within 5% of the averages.

4.2 Experiments with Synthetic Data

Comparing All Schemes for Handling Online Updates. Figure 9 compares the performance impact of online update schemes on range scan queries while varying the range size from 100GB (the entire table) to 4KB (a disk page). For IU and MaSM schemes, the cached updates occupy 50% of the allocated 4GB flash space (i.e. 2GB), which is the average amount of cached updates expected to be seen in practice. The coarse-grain index records one entry per 64KB cached updates on flash, while the fine-grain index records

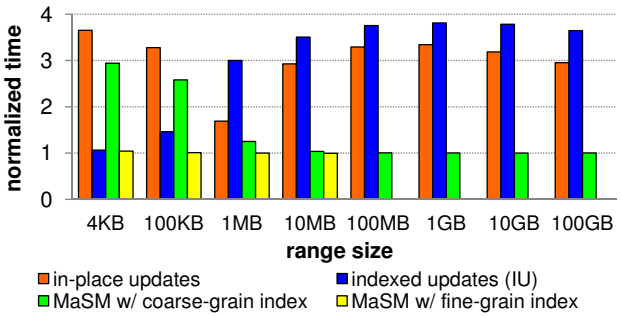


Figure 9: Comparing the impact of online update schemes on range scan performance (normalized to scans without updates).

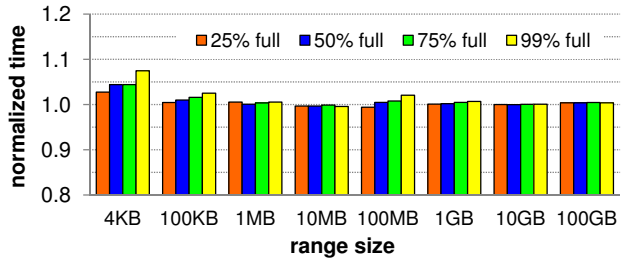


Figure 10: MaSM range scans varying updates cached in SSD.

one entry per 4KB cached updates. All the bars are normalized to the execution time of range scans without updates.

As shown in Figure 9, range scans with in-place updates (the leftmost bars) see 1.7–3.7X slowdowns. This is because the random updates significantly disturb the sequential disk accesses of the range scans. Interestingly, as the range size reduces from 1MB to 4KB, the slowdown increases from 1.7X to 3.7X. We find that elapsed times of pure range scans reduce from 29.8ms to 12.2ms, while elapsed times of range scans with in-place updates reduce from 50.3ms to only 44.7ms. Note that the queries perform a single disk I/O for data size of 1MB or smaller. The single I/O is significantly delayed because of the random in-place updates.

As shown by the second bars in Figure 9, range scans with IU see 1.1–3.8X slowdowns. This is because IU performs a large number of random I/O reads for retrieving cached updates from the SSD. When the range size is 4KB, the SSD reads in IU can be mostly overlapped with the disk access, leading to quite low overhead.

MaSM with coarse-grain index incurs little overhead for 100MB to 100GB ranges. This is because the total size of the cached updates (2GB) is only 1/50 of the total data size. Using the coarse-grain run index, MaSM retrieves roughly 1/50 SSD data (cached updates) compared to the disk data read in the range scans. As the sequential read performance of the SSD is higher than that of the disk, MaSM can always overlap the SSD accesses with disk accesses for the large ranges.

For the smaller ranges (4KB to 10MB), MaSM with coarse-grain index incurs up to 2.9X slowdowns. For example, at 4KB ranges, MaSM has to perform 128 SSD reads of 64KB each. This takes about 36ms (mainly bounded by SSD read bandwidth), incurring 2.9X slowdown. On the other hand, MaSM with fine-grain index can narrow the search range down to 4KB SSD pages. Therefore, it performs 128 SSD reads of 4KB each. The Intel X25-E SSD is capable of supporting over 35,000 4KB random reads per second. Therefore, the 128 reads can be well overlapped with the 12.2ms 4KB range scan. Overall, MaSM with fine-grain index incurs only 4% overhead even at 4KB ranges.

MaSM Varying Cached Update Size. Figure 10 varies both the range size (from 100GB to 4KB) and the cached update size (from

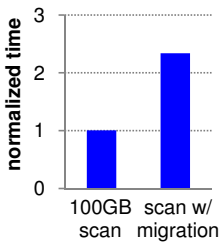


Figure 11: MaSM update migration.

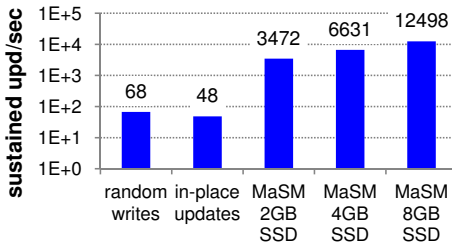


Figure 12: Sustained updates per second varying SSD size in MaSM.

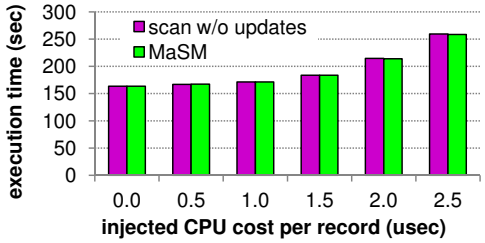


Figure 13: Range scan and MaSM performance while emulating CPU cost of query processing (10GB ranges).

25% full to 99% full) on the SSD. We disable update migration by setting the migration threshold to be 100%. We use MaSM with fine-grain index for 4KB to 10MB ranges, and MaSM with coarse-grain index for 100MB to 100GB ranges. From Figure 10, we see that in all cases, MaSM achieves performance comparable to range scans without updates. At 4KB ranges, MaSM incurs only 3%–7% overheads. The results can be viewed from another angle. MaSM with a 25% full 4GB-sized update cache will have similar performance to MaSM with a 50% full 2GB-sized update cache. Therefore, Figure 10 also represents the performance varying flash space from 2GB to 8GB with a 50% full update cache.

HDD as Update Cache. We experimented with using a separate SATA disk (identical to the main disk) as the update cache in MaSM. However, the poor random read performance of the disk-based update cache results in high query overhead for small range scans. Experiments see 28.8X (4.7X) query slowdowns for 1MB (10MB) sized range scans. This shows the significance of MaSM's use of SSDs for the update cache.

General Transactions with Read-Modify-Writes. Given the low overhead of MaSM even at 4KB ranges, we argue that MaSM can achieve good performance for general transactions. With MaSM, the reads in transactions achieve similar performance as if there were no online updates. On the other hand, writes are appended to the in-memory buffer, resulting in low overhead.

MaSM Migration Performance. Figure 11 shows the performance of migrating 4GB-sized cached updates while performing a table scan. Compared to a pure table scan, the migration performs sequential writes in addition to sequential reads on the disk, leading to 2.3X execution time. The benefits of the MaSM migration scheme are as follows: (i) multiple updates to the same data page are applied together, reducing the total disk I/O operations; (ii) disk-friendly sequential writes rather than random writes are performed; and (iii) it updates main data in place. Finally, note that MaSM incurs its migration overhead orders of magnitude less frequently than prior approaches—recall Figure 1.

Sustained Update Rate. Figure 12 reports the sustained update throughput of in-place updates, and three MaSM schemes with different flash space. For in-place updates, we obtain the best update rate by performing only updates, without concurrent queries. For

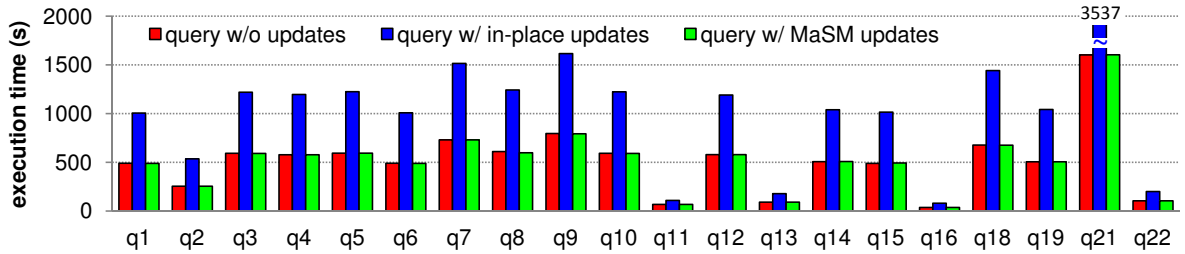


Figure 14: Replaying I/O traces of TPC-H queries on a real machine with online updates.

MaSM, we continuously perform table scans. The updates are sent as fast as possible so that every table scan incurs the migration of updates back to the disk. We set the migration threshold to be 50% so that in steady state, a table scan with migration is migrating updates in 50% of the flash while the other 50% of the flash is holding incoming updates. Figure 12 also shows the disk random write performance. We see that (i) compared to in-place updates, which perform random disk I/Os, MaSM schemes achieve orders of magnitude higher sustained update rates; and (ii) as expected, doubling the flash space will roughly double the sustained update rate.

Varying CPU Cost of Query Processing. Complex queries may perform a lot of in-memory processing after retrieving records from range scans. In Figure 13, we model query complexity by injecting CPU overhead. For every 1000 retrieved records, we inject a busy loop that takes 0.5ms, 1.0ms, 1.5ms, 2.0ms, or 2.5ms to execute. In other words, we inject 0.5us to 2.5us CPU cost per record. As shown in Figure 13, the performance is almost flat until the 1.5us point, indicating that the range scan is I/O bound. From 1.5us to 2.5us, the execution time grows roughly linearly, indicating that the range scan is CPU bound. Most importantly, we see that range scans with MaSM have indistinguishable performance compared with pure range scans for all cases. The CPU overhead for merging cached updates with main data is insignificant compared to (i) asynchronous I/Os when the query is I/O bound and (ii) in-memory query overhead when the query is CPU bound.

4.3 TPC-H Replay Experiments

Figure 14 shows the execution times of the TPC-H replay experiments (in 1000s seconds). The left bar is the query execution time without updates; the middle bar is the query execution time with concurrent in-place updates; the right bar is the query execution time with online updates using MaSM. For the MaSM algorithm, the flash space is 50% full at the start of the query. MaSM divides the flash space to maintain cached updates per table (for `orders` table and `lineitem` table in the TPC-H experiments).

From Figure 14, we see that in-place updates incur 1.6–2.2X slowdowns. In contrast, compared to pure queries without updates, MaSM achieves very similar performance (with up to 1% difference), providing fresh data with little I/O overhead. Note that the queries typically consist of multiple (concurrent) range scan operations on multiple tables. Therefore, the results also show that MaSM can handle multiple concurrent range scans well.

5. DISCUSSION AND RELATED WORK

In-Memory and External Data Structures. Our MaSM design extends prior work on in-memory differential updates [11, 22] to overcome the limitation on high migration costs vs. large memory footprint. We assume I/O to be the main bottleneck for DW queries and therefore the focus of our design is mainly on the I/O behaviors. On the other hand, prior differential update approaches propose efficient in-memory data structures, which is orthogonal to the MaSM design, and may be applied to MaSM to improve CPU performance for CPU-intensive workloads.

Moreover, MaSM may benefit from clever data structures for enhancing external sorting. For example, partitioned B-trees [10] store sorted runs as portions in a B-tree in order to support effective value-based prefetching during merging. MaSM can employ partitioned B-trees to organize materialized sorted runs. Note that MaSM has key features beyond data structures for sorted runs, including the insight for materializing and reusing sorted runs, the careful orchestration of updates, queries, and migrations, and the trade-off between memory footprint and SSD writes.

Shared-Nothing Architectures. Large analytical DWs often employ a shared-nothing architecture for achieving scalable performance [2]. The system consists of multiple machine nodes with local storage connected by a local area network. The main data is distributed across multiple machine nodes by using hash partitioning or range partitioning. Incoming updates are mapped and sent to individual machine nodes, and data analysis queries often are executed in parallel on many machine nodes. Because updates and queries are eventually decomposed into operations on individual machine nodes, we can apply MaSM algorithms on a per-machine-node basis. Note that recent data center discussions show that it is reasonable to enhance every machine node with SSDs [1].

Secondary Index. We discuss how to support index scans in MaSM. Given a secondary index on Y and a range $[Y_{begin}, Y_{end}]$, an index scan is often served in two steps in a database. In the first step, the secondary index is searched to retrieve all the record pointers within the range. In the second step, the record pointers are used to retrieve the records. An optimization for disk performance is to sort the record pointers according to the physical storage order of the records between the two steps.

For every retrieved record, MaSM can use the key (primary key or RID) of the record to look up corresponding cached updates and then merge them. However, we must deal with the special case where Y is modified in an incoming update: We build a *secondary update index* for all the update records that contain any Y value, comprised of a read-only index on every materialized sorted run and an in-memory index on the unsorted updates. The index scan searches this secondary update index to find any update records that fall into the desired range $[Y_{begin}, Y_{end}]$. In this way, MaSM can provide functionally correct support for secondary indexes.

Multiple Sort Orders. Heavily optimized for read accesses, column-store DWs can maintain multiple copies of the data in different sort orders (a.k.a. projections) [22, 23]. For example, in addition to a prevailing sort order of a table, one can optimize a specific query by storing the columns in an order that is most performance friendly for the query. However, multiple sort orders present a challenge for differential updates; prior work does not handle this case [11].

One way to support multiple sort orders would be to treat columns with different sort orders as different tables, and to build different update caches for them. This approach would require that every

update must contain the sort keys for all the sort orders so that the RIDs for individual sort orders could be obtained.

Alternatively, we could treat sort orders as secondary indexes. Suppose a copy of column X is stored in an order O_X different from the prevailing RID order. In this copy, we store the RID along with every X value so that when a query performs a range scan on this copy of X, we can use the RIDs to look up the cached updates. Note that adding RIDs to the copy of X reduces compression effectiveness, because the RIDs may be quite randomly ordered. Essentially, X with RID column looks like a secondary index, and can be supported similarly.

Materialized Views. Materialized views can speed up the processing of well-known query types. A recent study proposed lazy maintenance of materialized views in order to remove view maintenance from the critical path of incoming update handling [25]. Unlike eager view maintenance where the update statement or the update transaction eagerly maintains any affected views, lazy maintenance postpones the view maintenance until the DW has free cycles or a query references the view. It is straightforward to extend differential update schemes to support lazy view maintenance, by treating the view maintenance operations as normal queries.

Extraction-Transformation-Loading (ETL) for DWs. We focus on supporting efficient query processing given online, well-formed updates. An orthogonal problem is an efficient ETL (Extraction Transformation Loading) process for DWs [16, 18]. ETL is often performed at a data integration engine outside the DW to incorporate changes from front-end operational data sources. Streamlining the ETL process has been both a research topic [18, 20] and a focus of a DW product [16]. These ETL solutions can be employed to generate the well-formed updates to be applied to the DW.

Sequential Reads and Random Writes in Storage Systems. Concurrent with our work, Schindler et al. [19] proposed exploiting flash as a non-volatile write cache in storage systems for efficient servicing of I/O patterns that mix sequential reads and random writes. Compared to the online update problem in DWs, the settings in storage systems are significantly simplified: (i) an "update record" in storage systems is (a new version of) an entire I/O page and (ii) ACID is not supported for accessing multiple pages. As a result, the proposal employs a simple update management scheme: modifying the I/O mapping table to point to the latest version of pages. Interestingly, the proposal exploits a disk access pattern, called *Proximal I/O*, for migrating updates that write to only 1% of all disk pages. MaSM could employ this device-level technique to reduce large update migrations into a sequence of small migrations.

Orthogonal Uses of SSDs for DWs. Orthogonal to our work, previous studies have investigated placing objects (such as data and indexes) on SSDs vs. disks [5, 14, 17], and including SSDs as a caching layer between main memory and disks [6].

6. CONCLUSION

Efficient analytical processing in applications that require data freshness is challenging. The conventional approach of performing random updates in place degrades query performance significantly, because random accesses disturb the sequential disk access patterns of the typical analysis query. Recent studies follow the differential update approach, by caching updates separate from the main data and combining cached updates on-the-fly in query processing. However, these proposals all require large in-memory buffers or suffer from high update migration overheads.

In this paper, we propose to judiciously use flash storage (SSDs) to cache differential updates. Our work is based on the principle of using flash as a performance booster for databases stored primarily on magnetic disks, since for the foreseeable future, magnetic disks are still much cheaper but slower than SSDs. We present a high-level framework for SSD-based differential update approaches, and identify five design goals. We present an efficient algorithm, MaSM, that achieves low query overhead, small memory footprint, no random SSD writes, few SSD writes, efficient in-place migration, and correct ACID support. Experimental results using a prototype implementation show that, using MaSM, query response times remain nearly unaffected even if updates are running at the same time.

7. ACKNOWLEDGMENTS

This work was partially supported by an ESF EurYI award and SNF funds. We would like to thank, as well, the anonymous reviewers for their insightful comments during the review process.

8. REFERENCES

[1] L. Barroso. Warehouse scale computing. *SIGMOD*, 2010.
[2] J. Becla and K.-T. Lim. Report from the first workshop on extremely large databases (XLDB 2007). *Data Science Journal*, 2008.
[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD*, 1995.
[4] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. *CIDR*, 2009.
[5] M. Canim, B. Bhattacharjee, G. A. Mihaila, C. A. Lang, and K. A. Ross. An Object Placement Advisor for DB2 Using Solid State Storage. *PVLDB*, 2009.
[6] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD Bufferpool Extensions for Database Systems. *PVLDB*, 2010.
[7] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *OSDI*, 2006.
[8] P. M. Fernandez. Red brick warehouse: A read-mostly RDBMS for open SMP platforms. *SIGMOD*, 1994.
[9] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1), 1994.
[10] G. Graefe. Sorting and indexing with partitioned B-Trees. *CIDR*, 2003.
[11] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. *SIGMOD*, 2010.
[12] W. Inmon, R. Terdeman, J. Norris-Montanari, and D. Meers. *Data Warehousing for E-Business*. John Wiley & Sons, 2003.
[13] Intel Corp. Intel X25-E SATA Solid State Drive. http://download.intel.com/design/flash/nand/extreme/319984.pdf.
[14] I. Koltsidas and S. Viglas. Flashing up the storage layer. *VLDB*, 2008.
[15] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4), 1996.
[16] Oracle. On-time data warehousing with oracle10g - information at the speed of your business. Oracle White Paper, 2003.
[17] O. Ozmen, K. Salem, J. Schindler, and S. Daniel. Workload-aware storage layout for database systems. *SIGMOD*, 2010.
[18] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. Knowl. Data Eng.*, 2008.
[19] J. Schindler, S. Shete, and K. A. Smith. Improving throughput for small disk requests with proximal I/O. *FAST*, 2011.
[20] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. QoX-driven ETL design: reducing the cost of ETL consulting engagements. *SIGMOD*, 2009.
[21] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. *DaMoN*, 2009.
[22] M. Stonebraker et al. C-store: a column-oriented DBMS. *VLDB*, 2005.
[23] Vertica. Online reference. *http://www.vertica.com*, 2010.
[24] C. White. Intelligent business strategies: Real-time data warehousing heats up. DM Review, 2002.
[25] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy Maintenance of Materialized Views. *VLDB*, 2007.