Online Updates on Data Warehouses via Judicious Use of Solid-State Storage

MANOS ATHANASSOULIS, École Polytechnique Fédérale de Lausanne
SHIMIN CHEN, Chinese Academy of Sciences
ANASTASIA AILAMAKI, École Polytechnique Fédérale de Lausanne
PHILIP B. GIBBONS, Intel Labs, Pittsburgh
RADU STOICA, École Polytechnique Fédérale de Lausanne

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 article 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. We present detailed modeling of the proposed approach, and provide proofs regarding the fundamental properties of the *MaSM* algorithms. Our experimentation shows that MaSM incurs only up to 7% overhead both on synthetic range scans (varying range size from 4KB to 100GB) 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

Additional Key Words and Phrases: Materialized sort merge, online updates, data warehouses, SSD

An earlier version of this article appeared in the 2011 *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
M. Athanassoulis is currently affiliated with Harvard University. R. Stoica is currently affiliated with IBM Research, Zurich.
This work was partially supported by an ESF EurYI award (FN 511.261), NSF funds, an EU-funded FP7 project (grant no. 317858), and the Intel Science & Technology Center for Cloud Computing. S. Chen is supported by the CAS Hundred Talents program and by NSFC Innovation Research Group no. 61221062.
Authors' addresses: M. Athanassoulis (corresponding author), School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138; email: manos@seas.harvard.edu; S. Chen (corresponding author), State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, no. 6 Kexueyuan South Road, Zhongguancun, Haidian District Beijing, China; A. Ailamaki, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland; P. B. Gibbons, Intel Labs Pittsburgh and Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; R. Stoica, IBM Research Zurich, Saumerstrasse 4, 8803 Ruschlikon, Switzerland.

DOI: http://dx.doi.org/10.1145/2699484

ACM Reference Format:
Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Philip B. Gibbons, and Radu Stoica. 2015. Online updates on data warehouses via judicious use of solid-state storage. ACM Trans. Datab. Syst. 40, 1, Article 6 (March 2015), 42 pages.
DOI: http://dx.doi.org/10.1145/2699484

1. INTRODUCTION

Data warehouses (DWs) 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 [Inmon et al. 2003; White 2002]. 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 [Becla and Lim 2008]. The large influx of data is recognized as one of the key characteristics of modern workloads, often referred to as *velocity* of data [Zikopoulos et al. 2012]. Thus, active (or real-time) data warehousing has emerged as both a research topic [Polyzotis et al. 2008; Athanassoulis et al. 2011] and a business objective [Oracle 2013; White 2002; IBM 2013; Russom 2012] aiming to meet the increasing demands of applications for the latest version of data. Unfortunately, state-of-the-art data warehouse 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 for DW: Limitations of Prior Approaches

While updates can proceed concurrently with analysis queries using concurrency control schemes such as snapshot isolation [Berenson et al. 1995], 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 Double Query Time. A traditional approach used in OLTP systems is to update in place, that is, to store the new value in the same physical location as the previous one. However, as shown in Section 2.2, in-place updates can dramatically slow down data warehousing queries. Mixing random in-place updates with TPC-H queries increases the execution time, on average, by 2.2x on a commercial row-store data warehouse and by 2.6x on a commercial column-store data warehouse. In the worst case, the execution time is 4x longer. Besides having to service a second workload (i.e., the updates), the I/O subsystem 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 or HDD Performance. Recently, *differential updates* have been proposed as a means to enable efficient online updates in column-store data warehouses [Héman et al. 2010; Stonebraker et al. 2005], following the principle of differential files [Severance and Lohman 1976]. The basic idea is to: (i) cache incoming updates in an in-memory buffer; (ii) take the cached updates

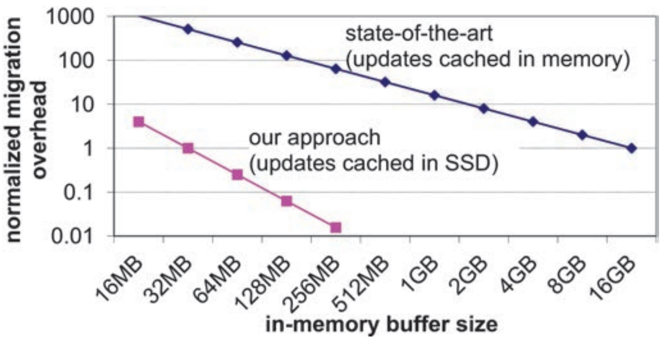

Fig. 1. An analysis of migration overheads for differential updates as a function of the memory buffer size. Overhead is normalized to the prior state-of-the-art using 16GB memory.

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 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. Because the updates are typically distributed across the entire file, a large buffer would cache updates touching virtually every physical page of the data warehouse. Each migration is expensive, incurring the cost of scanning the entire data warehouse, applying the updates, and writing back the results [Stonebraker et al. 2005; Héman et al. 2010]. 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., indexes) and storing intermediate results (e.g., in sorting, hash-joins). Moreover, in case of a crash, the large buffer of updates in memory is lost, prolonging crash recovery.

To overcome the preceding limitations, one can use HDD to cache updates. Fundamentally, DW updates can be maintained on HDD using several approaches based on differentials file and log-structure organizations [Severance and Lohman 1976; Jagadish et al. 1997, O'Neil et al. 1996; Graefe 2003, 2006; Graefe and Kuno 2010]. Any approach to store updates on HDD will suffer the performance limitations of HDDs. A key factor is the performance of the disk caching the updates compared with that of the disk storing the main data. If main data and cached updates reside on the same physical device, we will observe workload interference similar to what happens for the in-place updates in Section 2.2. If updates are cached on a separate HDD, its capabilities cannot support both high update rate and good query response times, since the incoming updates will interfere with the updates to be read for answering the queries. A third design is to use a separate high-end storage solution, such as an expensive disk array, for caching updates. Such a storage setup is able to offer very good read and write performance, but is also quite expensive. The introduction of flash-based *solid-state drive* (SSD) presents a new design point because it is much less expensive than HDD-based storage for achieving the same level of random read performance, as discussed next.

1.2. Our Solution: Cache Updates in SSDs for Online Updates in DWs

We exploit the recent trend towards including a small amount of SSDs in mainly HDD-based computer systems [Barroso 2010]. Our approach follows the idea of differential updates discussed previously, but, instead of being limited to an in-memory buffer, it

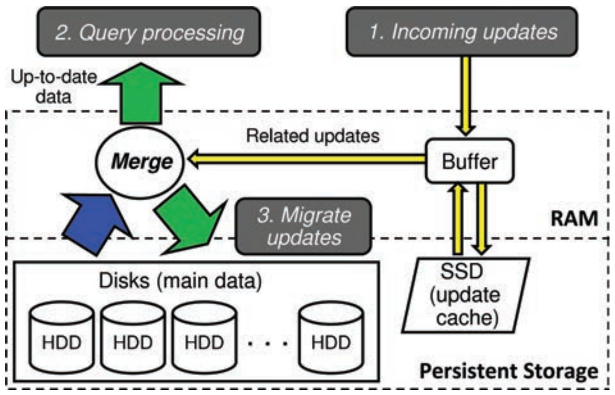

Fig. 2. Framework for SSD-based differential updates.

makes judicious use of 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.

(1) *Low query overhead with small memory footprint*. This addresses the main limitations of prior approaches.
(2) *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 [Bouganim et al. 2009]. Moreover, frequent random writes can transition an SSD into suboptimal states where even the well-supported operations suffer degraded performance [Bouganim et al. 2009; Stoica et al. 2009].
(3) *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.
(4) *Efficient migration*. Migrations should occur infrequently while supporting high sustained update rate. A naive approach is to migrate updates to a new copy of the data warehouse 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.
(5) *Correct ACID support*. We must guarantee that traditional concurrency control and crash recovery techniques still work.

Our approach draws parallels from the *stepped-merge* (SM) algorithm [Jagadish et al. 1997] which buffers updates, in sorted runs on disk. Using traditional hard disks to buffer updates, we can achieve low overhead on query response time for large-range queries. On the other hand, merging updates and main data becomes the main bottleneck for small-range queries and dominates the response time (Section 5.2). Our approach addresses this bottleneck by employing SSDs to cache the incoming updates in sorted runs. The SM algorithm on SSDs, however, cannot achieve the second and third design goals because, first, it reorganizes data on the secondary storage medium—hence causing random writes—and, second, it merges sorted runs iteratively—hence increasing the total number of writes per update. Prior differential update approaches [Stonebraker et al. 2005; Héman et al. 2010] maintain indexes on the cached updates in memory, which we call *indexed updates* (IUs). We find that naïvely extending IUs to SSDs incurs up to 3.8x query slowdowns (Sections 2.3 and 5.2). While employing *log-structured merge*-trees (LSM) [O'Neil et al. 1996] can address many of IUs' 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 many popular key-value store implementations (e.g., Bigtable [Chang et al. 2006], HBase [2013], and Cassandra [Lakshman and Malik 2010]) handle incoming updates by caching them in HDDs and merging related updates into query responses. In fact, their design follows the principle of LSM. The design of these key-value stores, however, is focused neither on low overhead for data warehousing queries with small memory footprint, using SSDs and minimizing SSD writes, nor on correct ACID support for multirow 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.

Using SSDs to Support Online Updates: MaSM. A key component of our solution is its use of *materialized sort-merge* (MaSM) algorithms to achieve our five design goals. 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. MaSM exploits 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. The number of passes is limited to two because one of the design goals is to minimize the amount of writes our algorithms perform on the SSDs. Compared with differential updates limited to an in-memory update cache, the MaSM 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 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. That is, there is a timestamp per update when it is cached on SSDs, a timestamp per data page in the main data, and a timestamp for every query. Using timestamps, MaSM can determine whether 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 those updates in the memory buffer (updates on SSDs survive the crash).

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 article makes the following contributions.

—First, we extend the work of Athanassoulis et al. [2011] which, to our knowledge, is the first paper that exploits SSDs for efficient online updates in data warehouses. We propose a high-level framework and identify five design goals for a good SSD-based solution.
—Second, we propose the family of MaSM algorithms that exploits 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 study the trade-off between memory footprint and short-range query performance for all MaSM algorithms. We provide a detailed model to predict MaSM query performance for a variety of different system setups.
—Finally, we present an experimental study of MaSM's performance. Our results show that MaSM incurs only up to 7% overhead both on range scans over synthetic data (varying range size from 4KB to 100GB) and in a TPC-H query replay study, while also increasing the sustained update throughput by orders of magnitude.

In addition, we discuss 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 the proposed MaSM design. Section 4 presents the proofs of the performance guarantees and analytical modeling of the proposed techniques. Section 5 presents the experimental evaluation. Section 6 discusses related work and Section 7 discusses DW-related issues. Section 8 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 data warehouses such as those characterized in the XLDB'07 report [Becla and Lim 2008]. There is typically a front-end operational system (e.g., OLTP) that produces the updates for the back-end analytical data warehouse. The data warehouse can be very large (e.g., petabytes), and does not fit in main memory.

Query Pattern. Large analytical data warehouses often observe “highly unpredictable query loads” as described by Becla and Lim [2008]. 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 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 uses the TPC-H benchmark, a decision support benchmark with emphasis on ad hoc queries.

Record Order. In row stores, records are often stored in primary key order (with clustered indexes). In addition, in DW applications, records are often stored ordered based on the primary key or a sort key. Typically, in DWs, data is either generated ordered [Moerkotte 1998] or partitioned and consequently ordered [Muth et al. 2000; Silberstein et al. 2008]. The clustering of data, either explicit or implicit[1], enables building access methods with specialized optimizations [Jagadish et al. 1997; Moerkotte 1998]. In column stores (that support online updates), the attributes of a record are aligned in separate columns allowing retrieval using a position value (RID) [Héman et al. 2010].[2] 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 [Héman et al. 2010], 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.[3,4] We call these updates *well-formed* updates. Data in large analytical data warehouses are often “write-once, read-many” [Becla and Lim 2008], a special case of well-formed updates. Note that well-formed updates do not require reading existing DW data. This type of updates is typical for key-value stores as well, where data are organized based on the key, and the knowledge about the values is moved to the application. In such applications, most of the updates are well-formed and MaSM can offer significant benefits. In contrast to well-formed updates, general transactions can require an arbitrary number of reads and writes. This distinction is necessary 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 discuss in Section 5.2.

2.2. Conventional Approach: In-Place Updates

In order to clarify the impact of online random updates in analytic workloads, we execute TPC-H queries on both a commercial row-store *DBMS R* and a commercial

[1]*Implicit clustering* is the property of natural clustering of values of attributes which are generated based on the creation time [Moerkotte 1998].

[2]Following prior work [Héman et al. 2010], we focus on a single sort order for the columns of a table. We discuss how to support multiple sort orders in Section 7.

[3]We follow prior work on column stores to assume that the RID of an update is provided [Héman et al. 2010]. For example, if updates contain sort keys, RIDs may be obtained by searching the (in-memory) index on sort keys.

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

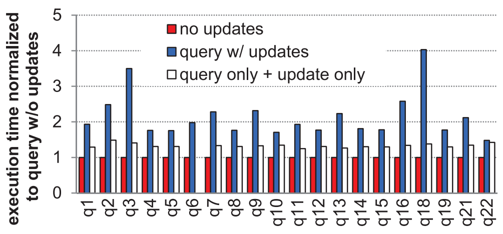

Fig. 3. TPC-H queries with randomly scattered updates on a row store have 1.5x–4.1x slowdowns when compared with query-only time.

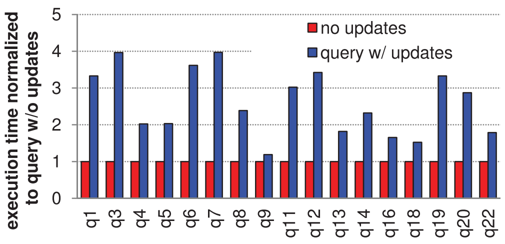

Fig. 4. TPC-H queries with randomly scattered emulated updates on a column store have 1.2x–4.0x slowdowns when compared with query-only time.

column-store *DBMS C* while running online in-place updates.[5] 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, that is, 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 not only of 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 [Héman et al. 2010; Stonebraker et al. 2005; Jagadish et al. 1997]. While the basic idea is straightforward (as described in Section 1), the efforts of prior work and this article 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 [Héman et al. 2010; Stonebraker et al. 2005], which we call *indexed updates* (IUs). Figure 5(a) shows the state-of-the-art IU proposal called *positional delta tree* (PDT), designed for column stores [Héman et al. 2010]. 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

[5]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.

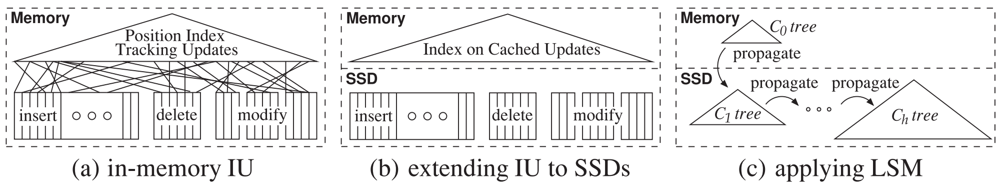

Fig. 5. Extending prior proposals of indexed updates (IUs) to SSDs: (a) In-memory indexed updates (PDT [Héman et al. 2010]); (b) extending IUs to SSDs; (c) applying LSM [O'Neil et al. 1996] to IU.

updates. Therefore, the PDT tables are 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.

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 preceding considerations (ignoring any memory footprint requirement). However, real machine experiments show up to 3.8x query slowdowns even for this ideal-case IU (Section 5.2). 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 [O'Neil et al. 1996]. 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 [Héman et al. 2010]. 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 $C_1, \ldots, C_h$ 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 prior ideal-case IU.

Unfortunately, LSM incurs a relatively 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 O'Neil et al. [1996], 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 aforesaid 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 SSD space and 16MB memory (which is our experimental setting in Section 5.1), we can compute that a two-level ($h = 1$) LSM writes every update entry $\approx$128 times.

The optimal LSM that minimizes total writes has $h = 4$ and writes every update entry $\approx$17 times! In both cases, in order to calculate the average number of physical writes per logical update, we have to take into account the inherent batching of LSM. In every propagation step, U updates are grouped together and written in one go, hence the number if physical writes is calculated by dividing by the factor U. This behavior is similar to buffering U updates before storing them with base data. In Section 4.5 we extend this analysis to compare with MaSM and we show that MaSM incurs significantly fewer physical writes. Compared to a scheme that writes every update entry once in batches of size U, applying LSM on an SSD reduces its lifetime $\frac{17/U}{1/U} = 17$-fold. For a more detailed comparison of the amount of writes performed by LSM and MaSM, see Section 4.5.

Applying Updates Using SM. Next, we consider applying updates using the *stepped-merge* (SM) algorithm [Jagadish et al. 1997]. SM is based on the principle of storing updates on secondary storage, organizing them in N progressively larger sorted runs, and merging them iteratively into larger B$^+$-Trees until they are merged with the main data. Limiting N to 1 yields a flash-friendly algorithm incurring only one physical write per update, which is the starting point of our approach. On the other hand, if we allow N to get higher values, then the physical writes per update increase drastically since each update is merged multiple times before it reaches the main relation. In particular, in the general case of $N > 1$, the SM algorithm buffers updates in memory and, once the buffer is full, it is sorted and written on the SSD as a packed B$^+$-Tree. Such trees are considered to be first-level sorted runs. When the number of first-level trees reaches a limit K, the SM algorithm merges all first-level trees and creates a new packed B$^+$-Tree with the result of the merging, which is a second-level tree. This process continues iteratively until a level N, at which point the trees are merged back to the main data.

In the next section we present a family of materialized sort-merge algorithms which is inspired by SM. The first variation is virtually the same as SM with $N = 1$, which guarantees that every update causes only one physical write, but requires a fixed number of on-the-fly merges in order to give the final result. Our approach does not need to store data as a B$^+$-Tree because the fast random access of flash storage allows to simply store sorted runs. We also present variations of the initial algorithm that allow a form of iterative merging by selectively storing the result of on-the-fly merging. We carefully design our algorithms to incur no more than two physical writes per update in order to respect the limitations of flash storage.

3. MaSM DESIGN

In this section, we propose MaSM (*materialized sort-merge*) algorithms 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*. A run index is a compact tree-based index which records the smallest key (primary key/RID) for every fixed number of SSD pages in a sorted run. It is created after the sorted runs are formed, and hence does not need to support insertion/deletion. 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 preceding ideas the *materialized sort-merge (MaSM)* algorithm.

The picture of the aforesaid 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 we 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 are at most M materialized sorted runs since the capacity of the SSD is M^2 (Section 3.1). 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 [Graefe 1994] operator tree to merge main data and updates, replacing the original `Table_range_scan` operator in query processing. During query execution, the updates buffered in memory are sorted, and then the $M + 1$ sorted runs (M sorted runs from SSD and the in-memory buffer) are merged by the `Merge_updates` operator and pipelined to the `Merge_data_updates` along with the output of the `Table_range_scan` operator. The `Merge_data_updates` operator produces the output, which includes the main data and the latest updates cached on SSD. When a new update arrives, unless the buffer is full, it is appended to the in-memory buffer. In the case the buffer is full, the buffer contents are sorted in key order and written out as a materialized sorted run on the SSD. The algorithms are detailed in Figure 7.

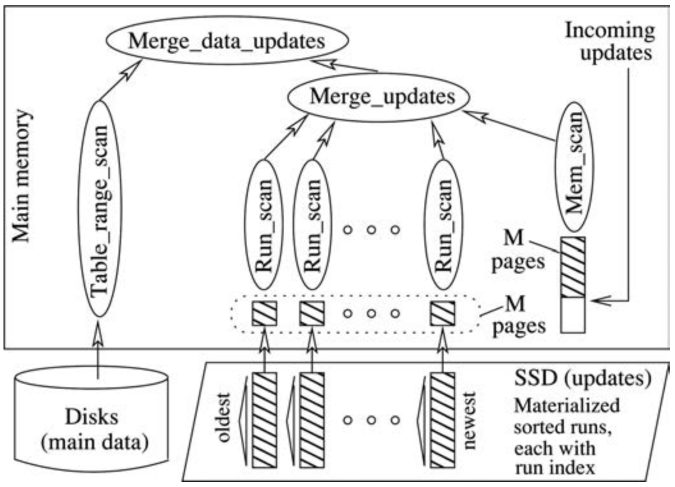

Fig. 6. Illustrating the MaSM algorithm using 2M memory.

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;
```

Fig. 7. MaSM-2M algorithm.

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 was 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 incurs getnext on operators in the subtree. The 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 changes the type field to replace. The operator Merge_data_updates performs the desired outer-join-like merging operation of the data records and the update records.

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 instantiates 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 load, or when the size of cached updates is above a prespecified threshold. When one of the

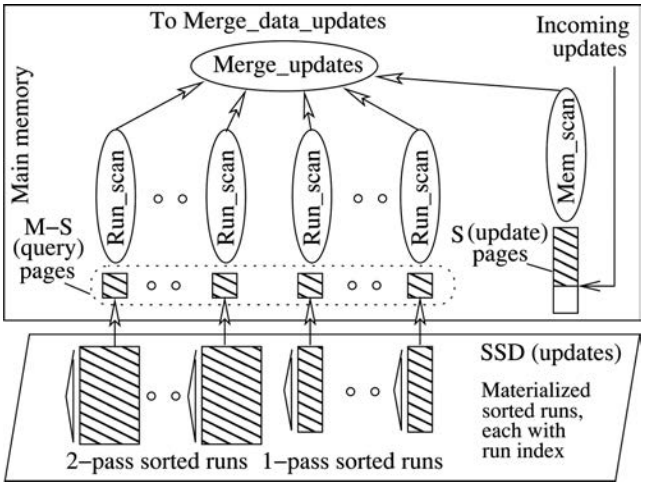

Fig. 8. Illustrating MaSM algorithm using M memory.

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 any queries that arrive during migration are evaluated correctly without additional delays. The table scan performed by the migration thread serves them correctly by delivering the data with the appropriate updates. On the other hand, the start of the migration process is delayed until after all queries with starting time earlier than the selected migration timestamp t have finished.

3.3. MaSM-M

Here we present the MaSM-M algorithm which reduces the memory requirements from $2M$ to M (where $M = \sqrt{\|SSD\|}$). The MaSM-M variation introduces three changes to the algorithm. First, MaSM-M reduces the memory footprint by managing memory more carefully. The overall number of pages is now M, out of which S are used for the in-memory buffer comprising the update pages. The remaining $M - S$ pages, called *query pages*, facilitate query processing. The core idea of the algorithm is the same as MaSM-2M (illustrated in Figure 8). During query execution, the query pages are used to merge the sorted runs (which now can be up to $M - S$), the in-memory buffer, and the main data. When a new update is submitted, it is appended in the in-memory buffer causing, in case where the buffer is full, the creation of a new sorted run. The second difference is related to the size and number of sorted runs. MaSM-M has $M - S$ query pages, hence the maximum number of sorted runs is $M - S$ as well. In order to guarantee that we always have enough query pages, we have to merge multiple smaller materialized sorted runs into larger materialized sorted runs. We call those sorted runs that are directly generated from the in-memory buffer *1-pass* runs, and those resulting from merging 1-pass runs are called *2-pass* runs. The third change is an impact of the different memory management as well. We introduce the number of updates pages S as a new parameter of the design. The optimal value of this parameter is calculated in the following as the value that minimizes the number of SSD writes.

All *1-pass* sorted runs are comprised of S pages each. Since 1-pass sorted runs have the same size, when generating 2-pass runs we merge N contiguous 1-pass sorted runs.

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;
```

Fig. 9. MaSM-M algorithm.

Any other approach would incur a higher number of writes on the device, leading to undesired increase in the write amplification of the MaSM algorithm. Replacement selection [Knuth 1998] offers fewer writes in total and better merge performance, but breaks the correct operation of the migration process discussed in Section 3.2. The 1-pass sorted runs are created sequentially and hence have disjoint timestamp sets. This property is important during migration and crash recovery in order to guarantee that there is a timestamp t before which all updates are taken into account when the migration is finished. Using replacement selection would generate sorted runs with timestamp overlaps, making it impossible for the migration process to complete correctly.

Figure 9 presents the pseudocode of MaSM-M. Algorithm parameters are summarized in Table I. 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 materializes a 1-pass 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

Table I. 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, $N \leq M - S$

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 simplifies the overall MaSM operation when merging updates with main data. 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. Theorem 3.1 computes the bound of the number of SSD writes that MaSM incurs. In fact, it is a corollary of a more general result shown in the next section (Theorem 3.2).

THEOREM 3.1. *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* $1.75 + \frac{2}{M}$.

3.4. MaSM-αM

We generalize the MaSM-M algorithm to a MaSM-αM algorithm that uses variable size of memory αM, introducing the parameter α, where $\alpha \in [\frac{2}{M^{\frac{1}{3}}}, 2]$. The lower bound on α ensures that the memory is sufficiently large to make 3-pass sorted runs unnecessary. The MaSM-algorithms do not need more than 2M pages of memory, hence the upper bound for α is 2. MaSM-M is a special case of MaSM-αM when $\alpha = 1$, and MaSM-2M is a special case of MaSM-αM when $\alpha = 2$. MaSM-αM is identical to MaSM-M, only with different memory budget. The total allocated memory size is αM pages, instead of M pages. MaSM-αM has S update pages and $\alpha M - S$ query pages. Similarly to Theorem 3.1, the following theorem gives the optimal value of S for minimizing SSD writes as a function of α.

THEOREM 3.2. *Let* $\alpha \in [\frac{2}{M^{\frac{1}{3}}}, 2]$. *The MaSM-$\alpha$M 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-αM 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 3.2 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 ($2M^{\frac{2}{3}}$) 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 purposes, 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: $\approx 0.1\%$ and $\approx 4 \cdot 10^{-4}\%$, respectively. Note that there is no reason to have finer granularity than one value (4 bytes in our setup) every 4KB of updates because this is the access granularity as well. Hence, the run index overhead is never going to be more than 0.1%.

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 several shared scans techniques (coordinated scans [Fernandez 1994], circular scans [Harizopoulos et al. 2005], cooperative scans [Zukowski et al. 2007]), we can combine the migration with a query requesting a table scan in order to avoid the cost of performing a table scan for migration purposes only. Hence, any queries that arrive during the migration process can be attached to the migration table scan and use the newly merged data directly as their output. In case they did not attach at the very beginning of the migration process, the queries spawn a new range scan from the beginning of the fact table until the point where they actually started. Since the data residing on disk is the same in the two parts of the query, that is, the version of the data after the latest migration, a simple row ID is enough to signify the ending of the second range scan. 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, shortening any delays for queries arriving during an ongoing migration. To do this, each materialized sorted run records those ranges that have already been migrated and those that are still active.

3.6. Transaction Support

The Lifetime of an Update. A new update U arriving in the system is initially logged on the transactional log and subsequently stored in the in-memory buffer. The transactional log guarantees that, in the event of a crash, the updates stored in memory will not be lost. When the buffer gets full it is sorted and sequentially written as a 1-pass materialized sorted run on the SSD. Once this step is completed, the record for U on the transactional log is not needed because MaSM guarantees the durability of

U on the SSD. Hence, the transactional log can be truncated as often as the in-memory buffer is flushed to the SSD. For MaSM-αM and $\alpha < 2$ (and hence MaSM-M as well), 1-pass sorted runs are merged and written a second time on the SSD as 2-pass materialized sorted runs. For MaSM-2M this step is skipped. When the SSD's free capacity is smaller than a given threshold, the migration process is initiated. During this process, all updates are applied on the main data during a full table scan. Each data page has its timestamp updated to be equal to the timestamp of the last added update. The migration process is protected with log records that make sure that, in the event of a crash, the process will start over. When the migration process completes correctly, the sorted runs including migrated updates are removed from the SSD and the relevant updates are now solely part of the main data.

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 [Berenson et al. 1995], 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 data warehouses. While snapshot isolation removes the logical dependencies between updates and queries so that they may proceed concurrently, the physical interference between updates and queries presents major performance problems. This interference is the target of MaSM algorithms.

Similar to prior work [Héman et al. 2010], 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 has the timestamp of the transaction start time so that it sees only the snapshot of data at 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 Figures 6 and 8. 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 that B correctly see these updates.

The In-Memory Update Buffer. The incoming updates are first stored in an in-memory buffer before they are sorted and written on the SSD. During query execution (one or more concurrent queries), this buffer can be read by every active query without any protection required. In fact, even in the presence of updates, only minimal protection is needed because no query will ever need to read updates made by subsequent transactions. Hence, the buffer is protected for consistency with a simple latch. However, when updates are issued concurrently, the in-memory buffer is bound to get full. In this case the buffer is locked, sorted, and flushed to the SSD. During the in-memory buffer flush, MaSM delays the writes. A different approach is to have a second buffer and to switch the pointers between the two buffers when one gets full, sorted, and flushed to the SSD.

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 comparing the per-data-page timestamps with the per-update timestamps, MaSM naturally determines whether updates should be applied to the data pages. Cached updates are removed only after the migration succeeds. Therefore, this process is idempotent, so in the event of multiple crashes it can still handle correctly the updates. 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. A detailed analysis of the impact of the SSD I/Os is presented both through an analytical model in Section 4.4 and through experimentation in Section 5.2.

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 5.2 see 128 sorted runs for 100GB data.) Note that SSDs can support 100x–1000x random 4KB reads per second compared to disks [Intel 2009]. 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.

Bounded Memory Footprint. Theorem 3.2 shows that our MaSM algorithms use $[2M^{\frac{2}{3}}, 2M]$ pages of memory, where $M = \sqrt{\|SSD\|}$. In fact, the memory utilization of the MaSM algorithms is only a very small fraction of the memory that a server is equipped with today. For example, when we use 1GB of SSD, the base memory requirement is 8MB (for 4GB SSD it is 16MB, for 16GB SSD it is 32MB, for 64GB SSD it is 64MB, for 256GB SSD it is 128MB, and so on). MaSM requires very small amount of memory for its operation by design. The exact memory size, however, plays an important role as it has a strong correlation with the sustainable update rate. Using larger SSD sizes—and, as a result, using more memory—results in a linear increase in sustained update rate (see Section 5.2, Figure 21 for more details). Orthogonally to that, our design does not keep many updates in memory in order to exploit the durability of flash storage. Having the updates moved to flash quickly gives the additional benefit that the system recovers fast in an event of a crash, since only those updates in the memory buffer would be lost, after the crash redone, and, eventually some of them undone).

Bounded Number of Total Writes on SSD. MaSM has a limited effect in the device lifetime by bounding the number of writes per update according to Theorem 3.2. All MaSM algorithms use fewer than 2 SSD writes per update, with MaSM-2M using only 1 write per update.

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 nontrivial (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 shown in Section 5.2. 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 capacity 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 $P = 64$KB, 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 $F^2/(64\text{KB}) = 16$GB.

4. ANALYSIS AND MODELING OF MaSM

In this section, we present the proofs of the theorems in Section 3 and provide an in-depth analysis of the behavior of the MaSM algorithm.

4.1. Theorems about MaSM Behavior

We start by proving Theorem 3.2 for the general case and then show how to deduce Theorem 3.1.

THEOREM 3.2. *Let* $\alpha \in [\frac{2}{M^{\frac{1}{3}}}, 2]$. *The MaSM-*$\alpha$*M 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-*α*M writes every update record to SSD is roughly* $2 - 0.25\alpha^2$.

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 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-αM can still merge all the sorted runs. Therefore

$$K_1 S + K_2 NS = M^2 \qquad (1)$$

and

$$K_1 + K_2 \leq \alpha M - S. \tag{2}$$

Compute K_1 from (1) and plugging it into (2) yields

$$K_2 \geq \frac{1}{N-1}\left(S - \alpha M + \frac{M^2}{S}\right). \tag{3}$$

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

$$\mathit{ExtraWrites} = K_2 N S \geq \frac{N}{N-1}(S^2 - \alpha M S + M^2). \tag{4}$$

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 $S_{opt} = 0.5\alpha M$. Plug it into (3) and (4):

$$K_2 \geq \frac{1}{N-1}\left(\frac{2}{\alpha} - 0.5\alpha\right)M, \tag{5}$$

$$\mathit{ExtraWrites} = 0.5\alpha M K_2 N \geq \frac{N}{N-1}(1 - 0.25\alpha^2)M^2. \tag{6}$$

Note that the equality signs in the previous 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 are held. We can rewrite $min(\mathit{ExtraWrites})$ as a function of K_2:

$$min(\mathit{ExtraWrites}) = \left(0.5\alpha\frac{K_2}{M} + 1 - 0.25\alpha^2\right)M^2. \tag{7}$$

The global minimum is achieved with the smallest nonnegative integer K_2. Since $N \leq \alpha M - S_{opt} = 0.5\alpha M$, we know $K_2 > \frac{4}{\alpha^2} - 1$ according to (5).

Therefore *ExtraWrites* achieves the minimum when $S_{opt} = 0.5\alpha M$, $K_{2,opt} = \lfloor\frac{4}{\alpha^2}\rfloor$, and $N_{opt} = \frac{1}{K_{2,opt}}(\frac{2}{\alpha} - 0.5\alpha)M + 1$. We can compute the minimum $\mathit{ExtraWrites} \simeq (1 - 0.25\alpha^2)M^2$. In this setting, the average number of times that MaSM-αM writes every update record to SSD is roughly $2 - 0.25\alpha^2$. Moreover, since $K_2 \leq \alpha M - S_{opt}$ iff $\lfloor\frac{4}{\alpha^2}\rfloor \leq 0.5\alpha M$, we know that $\alpha \geq \frac{2}{M^{\frac{1}{3}}}$. □

THEOREM 3.1. *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. When $\alpha = 1$, Theorem 3.2 yields $S_{opt} = 0.5M$ and $N_{opt} = \frac{1}{4}(2 - 0.5)M + 1 = \frac{3}{8}M + 1$. Moreover, Eq. (7) with $K_2 = \lfloor\frac{4}{\alpha^2}\rfloor = 4$ yields $\mathit{ExtraWrites} = (\frac{2}{M} + 1 - 0.25)M^2 = 0.75M^2 + 2M$.

In this setting, the average number of times that MaSM-M writes every update record to SSD is $1 + (0.75M^2 + 2M)/M^2 = 1.75 + \frac{2}{M} \simeq 1.75$. □

4.2. SSD Wear vs. Memory Footprint

MaSM-αM specifies a spectrum of MaSM algorithms trading off SSD writes for memory footprint. By varying α from 2 to $\frac{2}{M^{\frac{1}{3}}}$, the memory footprint reduces from $2M$ pages

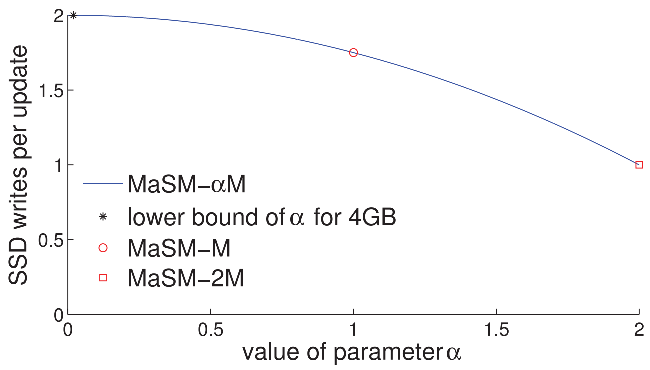

Fig. 10. The trade-off between SSD wear and memory footprint.

to $2M^{\frac{2}{3}}$ pages, while the average number of times that an update record is written to SSDs increases from the (minimal) 1 to close to 2. In all MaSM algorithms, we achieve small memory footprint and low total SSD writes.

In Figure 10 we vary the value of α on the x-axis and depict the resulting number of writes per updates of the corresponding MaSM-αM algorithm. The figure shows the operating point of the MaSM-2M algorithm (on the rightmost part of the graph), the operating point of the MaSM-M (in the middle), and the point corresponding to the lower bound of the value of the parameter α, which is the smallest possible memory footprint allowing MaSM to avoid 3-pass external sorting. We see that the minimum value of α for 4GB SSD space is $\alpha \approx 0.02$, leading to 100x less memory for twice the number of writes per update compared to MaSM-2M. Therefore, one can choose the desired parameter α based on the requirements of the application and the capacities of the underlying SSD.

The write endurance of the enterprise-grade SLC (*single-level cell*) NAND Flash SSD, 32GB Intel X25E SSD, used in our experimentation is 10^5. Therefore the Intel X25E SSD can support 3.2-petabyte writes, or 33.8MB/s for 3 years. MaSM writes on flash using large chunks equal to an erase block for several SSDs, allowing the device to operate with minimal write amplification close to 1. 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.

Newer enterprise-grade NAND Flash SSDs, however, have switched to MLC (*multilevel cell*). The endurance of an MLC cell is typically one order of magnitude lower than SLC, 10^4, and the capacity of MLC NAND Flash SSDs are about one order of magnitude higher, 200GB to 320GB. The lack of endurance in erase cycles is balanced out by the larger capacity and, as a result, the lifetime of an MLC SSD is in fact similar to an SLC SSD.

4.3. Memory Footprint vs. Performance

As discussed in Section 3.7, the I/O costs of small-range scans are determined by the number of materialized sorted runs. For the extreme case of a point query where a single page of data is retrieved, the merging process has to fetch one page of updates from every sorted run. In this case the response time is dominated by the time of reading from the SSD. In order to mitigate this bottleneck, one has to reduce the number of sorted runs stored on the SSD.

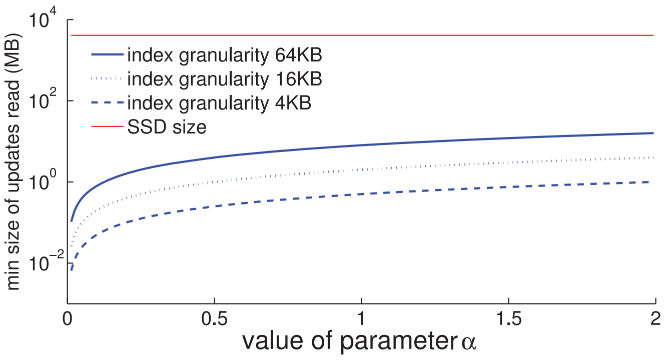

Fig. 11. Memory footprint vs. short-range performance.

For the family of MaSM-αM algorithms, the number of sorted runs stored on the SSD can be calculated using Eq. (2). If we assume the optimal distribution of buffer pages as described in Theorem 3.2, that is, $S_{opt} = \alpha M/2$, the equality holds for Eq. (2) and we get the maximum number of sorted runs, N.

$$N = K_1 + K_2 = \alpha M - S_{opt} = \alpha M/2 \tag{8}$$

A new trade-off is formed: by increasing the allocated memory for the algorithm (i.e., by increasing α)[6], one increases the lower bound of the cost of reading updates from the SSD in the case of sort range scans. In fact, the lower bound of updates read in MB from the SSD is given by the product of the number of sorted runs N, the SSD page size P, and the index granularity fraction g, defined as the granularity of the run index divided by the page size. Finally, $M = \sqrt{\|SSD\|}$ and α is the parameter showing the available memory to the MaSM-αM algorithm.

$$UpdatesRead = NgP = \alpha MgP/2 \tag{9}$$

Eq. (9) shows the relationship between the size of updates to be read and the value of the parameter α. Figure 11 shows how the lower bound of the size of fetched updates (y-axis in MB) changes as α changes (x-axis). Three blue lines are presented, each corresponding to index granularity 64KB, 16KB, and 4KB, respectively. The straight red line on the top of the graph corresponds to the overall size of the SSD, which in our experimentation and analysis is 4GB, and the SSD page size P is equal to 64KB. Higher values of α lead to higher I/O cost for fetching the updates because a bigger percentage of the updates stored on SSD should be read. The prior analysis is extended in the following section in order to fully understand how the overhead of reading the updates affects the performance of the MaSM algorithms.

4.4. Modeling the I/O Time of Range Queries

In this section we present an analytical model for range queries which considers the I/O time of reading updates from SSDs and the I/O time of reading the main data. We model various aspects of the range queries, including relation size, disk capacity, SSD capacity, range size, and disk and SSD performance characteristics. The parameters used in the performance model, including the ones detailed in the discussion about the MaSM algorithms in Section 3.4 (see Table I), are summarized in Table II.

In Sections 3.7 and 4.3 it is argued that the lower bound for the cost of retrieving updates from SSD is given by the number of sorted runs. The cost of retrieving updates

[6]Increasing α helps with SSD wear (Section 4.2) but has adverse effects regarding short-range query performance. Hence there is a trade-off between SSD wear, allocated memory, and short-range query performance.

Table II. Parameters of MaSM-M's Performance Model and their Value

T	The total size of the table	100 GB
s	The seek time of the main storage hard disk	4.17 ms, 3 ms, 2 ms
DBW	The sequential bandwidth offered by the main storage hard disk	77 MB/s, 200 MB/s, 800 MB/s
r	The percentage of the relation queried sequentially	$2^{-26} \ldots 2^{-1}, 1$
$\|SSD\|$	SSD capacity (in pages)	65536 pages
M	MaSM-M memory, $M = \sqrt{\|SSD\|}$	256 pages
P	SSD page size	64 KB
α	The α parameter, giving the amount of memory available to MaSM-αM	0.5, 1.0, 2.0
N	The number of sorted runs	64, 128, 256
g	The granularity fraction of the run index	1, 1/16
$ReadBW$	The nominal read bandwidth offered by the SSD	250 MB/s, 1500 MB/s
FBW	The random read bandwidth offered by the SSD when reading P-sized pages	193 MB/s, 578 MB/s

for larger ranges, however, is a function of the range of the relation we retrieve. More specifically, if we assume that the updates are uniformly distributed over the main data then the number of SSD pages retrieved is proportional to the range size of the query. Hence the time needed to fetch the updates is given by Eq. (10).

$$FetchUpdates = max(\|SSD\| \cdot r, N \cdot g) \cdot \frac{P}{FBW} \tag{10}$$

The size of the range $r\%$ is expressed as a percentage of the total size of the relation (T). $\|SSD\|$ is the number of SSD pages devoted to the MaSM algorithm, g is the granularity of the run index as a fraction of SSD pages, which in turn have size P. The bandwidth that updates are fetched from the SSD, FBW, defines how fast MaSM can read updates. In Eq. (10) we first calculate the number of pages containing updates to be read, which is equal to $max(\|SSD\|r, Ng)$, and then take the worst-case scenario regarding the I/O time per page.

MaSM overlaps SSD I/Os for retrieving updates with HDD I/Os for retrieving the main data. In order to model this behavior, we devise a simple HDD model. For a query that requires a range scan of $r\%$ of the main data (with total size T), the time needed to retrieve the data is given by Eq. (11).

$$ReadMainData = s + \frac{r \cdot T}{DBW} \tag{11}$$

The seek time s plays an important role since it dominates the response time for short ranges. For large ranges, however, the most important factor is the disk bandwidth DBW.

Achieving Perfect I/O Overlapping Using MaSM. Using Eqs. (10) and (11), we compute under which conditions the I/O for fetching updates can be entirely overlapped by the I/O needed to read the main data, that is, the formal condition for perfect I/O overlapping:

$$ReadMainData \geq FetchUpdates \Rightarrow s + \frac{r \cdot T}{DBW} \geq max(\|SSD\| \cdot r, N \cdot g) \cdot \left(\frac{P}{FBW}\right) \Rightarrow$$

$$s \geq max(\|SSD\| \cdot r, N \cdot g) \cdot \frac{P}{FBW} - \frac{r \cdot T}{DBW}. \tag{12}$$

Eq. (12) has two branches, depending on whether $\|SSD\| \cdot r$ has higher value than $N \cdot g$. When the query range is large enough to bring at least one page per sorted run (i.e., $\|SSD\| \cdot r \geq N \cdot g$) the condition becomes

$$s \geq \frac{\|SSD\| \cdot r \cdot P}{FBW} - \frac{r \cdot T}{DBW} \Rightarrow s \geq r \cdot \left(\frac{P \cdot \|SSD\|}{FBW} - \frac{T}{DBW} \right). \tag{13}$$

Hence, for large enough ranges, MaSM can perfectly overlap the I/O needed to fetch the updates with I/O for the main data if the time needed to read randomly all of the updates, $T_U = P \cdot \|SSD\|/FBW$, is less than the time needed to read sequentially the entire relation, $T_R = T/DBW$. If this condition does not hold, perfect I/O overlap is still possible provided that the difference $T_U - T_R$ multiplied by the fraction of the range query is less than the time needed to initiate a main data I/O, that is, the seek time of the HDD storing the main data. Regarding the second branch of Eq. (12), that is, for short ranges that $\|SSD\| \cdot r < N \cdot g$ the condition is.

$$s \geq \frac{N \cdot g \cdot P}{FBW} - \frac{r \cdot T}{DBW}. \tag{14}$$

Eq. (14) implies that, for short ranges, MaSM can perfectly overlap the I/O needed to fetch the updates with I/O for the main data if the time needed to read randomly the minimal number of pages per sorted run, $T_{mU} = N \cdot g \cdot P/FBW$, is less than the time needed to read sequentially the queried range of the relation, $T_{rR} = (r \cdot T)/DBW$, or, if this does not hold, the difference $T_{mU} - T_{rR}$ is less than the seek time of the HDD storing the main data.

Analysis of MaSM Behavior. Next, we use the analytical model given before to predict the anticipated behavior of the MaSM algorithms for our experimental setup in Section 5, as well as for different setups with varying performance capabilities of the disk subsystem storing the main data and varying capabilities of the SSD storing the updates.

Table II shows the parameter values considered in our analysis[7]. In particular, for the disk subsystem we use seek time $s = 4.17$ms and disk bandwidth $DBW = 77$MB/s which are either directly measured or taken from device manuals for the experimental machine in the next section. The SSD used shows 75μs read latency and offers 250MB/s read bandwidth. Please note that FBW is not equal to read bandwidth, but calculated as the effective bandwidth to read a P-sized page.

In Figure 12 we see the I/O time of: (i) reading the main data and (ii) reading the updates from the SSD as a function of the query range. The x-axis shows the range in MB for a relation with total size $T = 100$GB. The vertical dashed lines correspond to the ranges we used for our experimentation in Section 5.2: 4KB, 1MB, 10MB, 100MB, 1GB, 10GB, 100GB. The blue *Scan-only* line shows the I/O time of accessing the main data for the given range, the red lines show the I/O of fetching the updates for different values of α (0.5, 1, 2) and index granularity 64KB. Finally, in the black lines the only change is the granularity, which is now 4KB. We observe that, for any combination of g and α, MaSM can hide the cost of reading updates if the range is more than 10MB. For $\alpha \leq 1$ and index granularity 4KB, the cost of reading the updates can be hidden for every possible range, even for the smallest possible range of 4KB which models the point query. To detail the comparison between the time to fetch the updates and the time to read the main data, we depict the ratio between these two values in Figure 13 for index granularity 64KB and in Figure 14 for index granularity 4KB.

[7]Some explanations: the total SSD capacity used is 4GB in 64KB pages: $64KB \cdot 65536 = 4GB$. $M = \sqrt{\|SSD\|} = 256$. $N = \alpha M/2$. The index granularity fraction g is 1 for granularity 64KB and 1/16 for granularity 4KB.

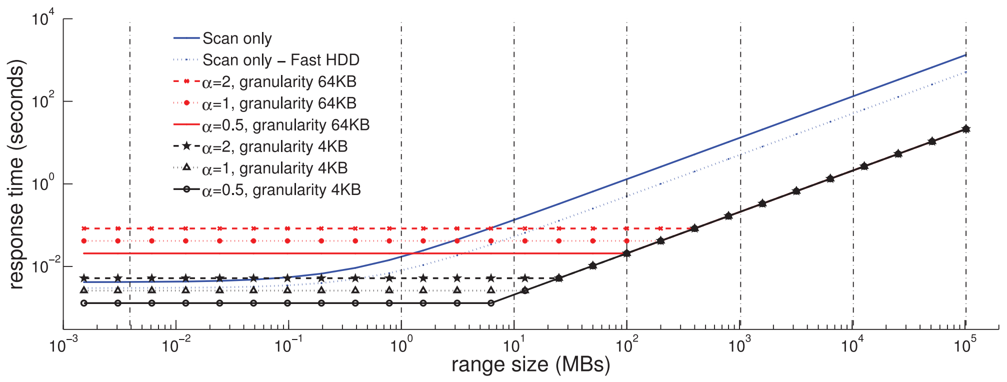

Fig. 12. Modeling MaSM performance for reading main data and fetching the updates using the characteristics of the storage used in our experimentation.

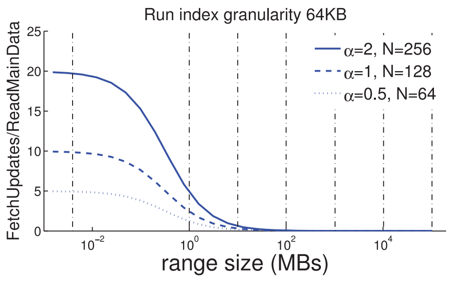

Fig. 13. The ratio of the time to fetch the updates from SSD to the time to read the main data for index granularity 64KB.

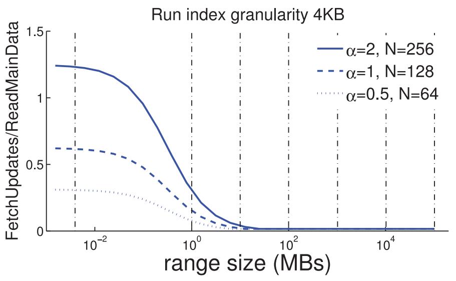

Fig. 14. The ratio of the time to fetch the updates from SSD to the time to read the main data for index granularity 4KB.

Figure 13 shows this ratio quickly rises when the range goes below 10MB: fetching the updates may need 5x to 20x times more time than reading the main data for short ranges. In these cases MaSM is unable to hide the overhead of reading and merging the updates. On the other hand, Figure 14 shows that, if we decrease the index granularity from 64KB to 4KB, MaSM with $\alpha = 1$ can read the updates somewhat faster compared with the time needed to read the data of a point query. This behavior is corroborated in Section 5.2 when merging updates from the SSD for the point queries leads to small performance penalties. Both Figures 13 and 14 show that, for large ranges starting from 1–10MB, the overhead of reading the updates can be easily hidden by the cost of reading the main data.

Next, we calculate the I/O cost of reading the main data from a faster disk, assuming that the seek time is now $s = 3$ms and the disk bandwidth is $DBW = 200$MB/s. The performance of this disk is depicted in Figure 12 in the blue dotted line named *Scan only - Fast HDD*. This line shows that if the current system is equipped with a faster disk, then the overhead of MaSM grows, particularly for short-range queries, and one is forced to use less memory to hide the overhead of fetching the updates from the SSD.

The previous analysis of a faster disk opens the question whether MaSM is relevant in a setup involving a read-optimized array of disks, which can deliver the optimal disk performance while having larger than before and cost-effective capacity. In fact, in such a setup a single enterprise state-of-the-art flash device suffices to hide the overhead of reading and merging the updates. We argue that if one is equipped with a high-end disk subsystem it is feasible to accompany it with a high-end flash device.

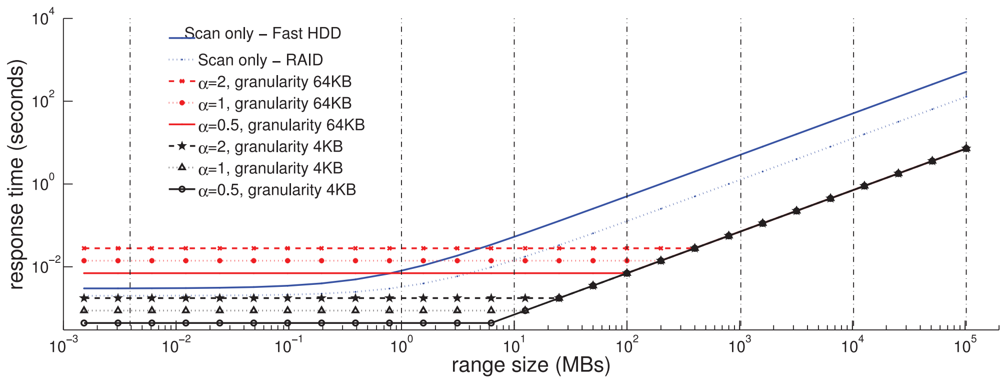

Fig. 15. Modeling MaSM performance for reading main data and fetching the updates using the characteristics of a high-end storage system.

Table III. Computing Optimal Height h for LSM Tree, Using Size Ratio between Adjacent Levels r and I/O Write Rate Normalized by Input Insertion Rate, $(h - 0.5)(1 + r)$

h	1	2	3	**4**	5	6	7	8	9	10
r	256.0	15.5	6.0	**3.7**	2.8	2.3	2.0	1.8	1.7	1.6
writes/update	128.5	24.8	17.5	**16.5**	17.0	18.1	19.5	21.1	22.7	24.5

The C_0 tree is in memory, and $C_1, \ldots, C_h$ trees are on SSDs. Suppose memory size allocated to the C_0 tree is 16MB, and the SSD space allocated to the LSM tree is 4GB.

Figure 15 shows the performance prediction of the previously described *Fast HDD* (s = 3ms and DBW = 200MB/s) and of a read-optimized *RAID* installation (s = 2ms and DBW = 800MB/s). The flash device is now replaced by a high-end flash device with specifications similar to the ioDrive2 FusionIO card [FusionIO 2013] (read latency 68μs, read bandwidth 1500MB/s, and FBW = 578MB/s). The blue dotted line named *Scan only - RAID* shows the I/O time needed for the RAID subsystem to deliver the corresponding range of pages. For the case of a high-end flash device with any possible value of α, the MaSM algorithms manage to entirely hide the overhead of fetching the updates when the index granularity is 4KB. Thus MaSM can be used as an enabler, both in a commodity system—as we demonstrate in our experimentation as well—and in a system with high-end storage for both the main data and for the updates.

4.5. LSM Analysis and Comparison with MaSM

In Section 2.3 we show that an LSM with $h = 4$ applies minimum writes per update. Next we give more details for this computation assuming the same storage setup as in our experimentation in Section 5.

LSM Optimal Height. We compute the optimal height for an LSM tree which uses 16MB memory and up to 4GB SSD space, as shown in Table III. According to O'Neil et al. [1996, Theorem 3.1], in an optimal LSM tree, the ratios between adjacent LSM levels are all equal to a common value r. Therefore $Size(C_i) = Size(C_0) \cdot r^i$. In Table III, we vary h, the number of LSM levels on SSD, from 1 to 10. For a specific h, we compute r in the second row in the table so that the total size of $C_1, \ldots, C_h$ (i.e., $C_{tot} = \sum_{i=1}^{h} C_0 \cdot r^i$) for C_0 = 16MB, is C_{tot} = 4GB. To compute the total I/O writes, we follow the same analysis as in the proof of O'Neil et al. [1996, Theorem 3.1]. Let us consider the rolling merge from C_{i-1} to C_i. Suppose the input insertion rate is R bytes per second. Then the rolling merge must migrate R bytes from C_{i-1} to C_i per second. This entails reading R bytes from C_{i-1}, reading $R * r$ bytes from C_i, and writing $R * (1 + r)$ bytes to C_i

per second. Therefore the LSM tree sees an $R * (1 + r)$-bytes-per-second I/O write rate at level $C_1, \ldots, C_{h-1}$. For the last level, we consider it to grow from 0 to the full size because when it is close to full, all the update entries in the LSM can be migrated to the main data. On average, the LSM would see $0.5R * (1 + r)$-bytes-per-second I/O write rate at C_h. Therefore we can compute the I/O write rate normalized by input insertion rate as $(h - 0.5)(1 + r)$. As shown in Table III, the minimal I/O write rate is achieved when $h = 4$. The optimal LSM tree writes every update entry 16.5 times on SSDs. In order to compute, however, the average number of physical writes per logical update, we need to consider one more factor. For the propagation of the updates, LSM uses buffers (disk blocks) able to hold a number of updates, say U. Hence the average number of physical writes per update is $\frac{16.5}{U}$.

When compared with MaSM, optimal-height LSM has much higher physical writes per updates. As the analysis of Section 4.2 shows, every update is written up to 2 times using the generic MaSM-αM algorithm. Similarly to the batching of U updates in LSM, any MaSM algorithm uses flash blocks holding U updates, effectively batching updates when writing. In a head-to-head comparison between the optimal-height LSM and the MaSM algorithm with maximum SSD wear (α is minimal), MaSM incurs $\frac{16.5/U}{2/U} = 8.25$x less physical writes per update.

MaSM vs. LSM. Next, we compare writes per update for LSM and MaSM when LSM, like MaSM, has two levels. For MaSM we assume the worst case, that is, every update is copied twice before merged with base data. We assume variable memory and SSD space, maintaining, however, the ratio between the two: M pages in memory and M^2 pages on the SSD have as a result a ratio M between C_0 and C_1.

Every time the in-memory buffer fills, its contents are merged with the existing data on the SSD. In the best-case scenario the SSD is initially empty, so the first time M pages are written on the SSD, the second time M pages are merged with M pages of sorted data (incurring in total $2 \cdot M$ writes), the third time M pages are merged with $2 \cdot M$ pages of sorted data (incurring in total $3 \cdot M$ writes), and so on until the final M^{th} time, during which $M \cdot M$ writes take place. In total, during this migration operation the number of writes is

$$\begin{aligned} LSMWrites &= M + 2 \cdot M + \cdots + M \cdot M \\ &= 0.5 \cdot M \cdot M \cdot (M + 1) = 0.5 \cdot M^3 + 0.5 \cdot M^2. \end{aligned}$$

On the other hand, the worst case for MaSM is that every page has been written 2 times on flash, hence leading to total writes equal to $MaSMWrites = 2 \cdot M^2$. Dividing the two quantities we get the ratio between the number of writes for LSM and the number of writes for the worst case of MaSM.

$$\frac{LSMWrites}{MaSMWrites} = \frac{0.5 \cdot M^3 + 0.5 \cdot M^2}{2 \cdot M^2} = 0.25 \cdot M + 0.25$$

The preceding ratio shows that LSM not only performs more writes than MaSM (since $0.25 \cdot M + 0.25 \geq 1$ for any $M \geq 3$) but, additionally, increases the number of writes for larger amount of resources available. On the other hand, as explained in Section 4.2, when more memory is available, MaSM manages to decrease the number of writes per update.

5. EXPERIMENTAL EVALUATION

We perform real machine experiments to evaluate our proposed MaSM algorithms. We start by describing the experimental setup in Section 5.1. Then, we present experimental studies with synthetic data in Section 5.2 and perform experiments based on TPC-H traces recorded from a commercial database system in Section 5.3.

5.1. 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 [Intel 2009] (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 data warehouse 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 MaSM-M algorithm, noting that this provides performance lower bounds for any MaSM-αM with $1 \leq \alpha \leq 2$. 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 (Section 5.2). 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 of flash-based SSD space for caching updates, thus MaSM-M requires 16MB memory for 64KB SSD effective page size. We also study the impact of varying the SSD space. In our experiments we vary the SSD space used for caching updates from 2GB to 8GB in order to increase the sustained update rate. Note that the devices used have higher SSD capacity—that is, 32GB and 64GB—but the MaSM algorithms need only a fraction of that space to offer high update rate with low or negligible impact on the query response time. Hence the rest of the SSD capacity can be used by the database system for other operations, or can be used as additional space, allowing the SSD device to optimize garbage collection and prolong its lifetime [Bux and Iliadis 2010; Stoica and Ailamaki 2013]. Similarly, the MaSM algorithms need a fraction of the available memory for its operation, leaving the better part of the memory available for memory-hungry operators like hash-joins.

TPC-H Replay Experiments (*Section 5.3*). 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 for 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, 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.

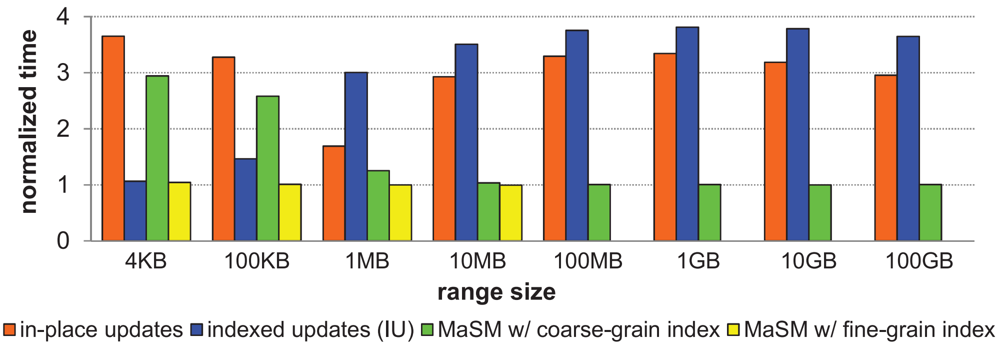

Fig. 16. Comparing the impact of online update schemes on the response time of primary key range scans (normalized to scans without updates).

We replay the TPC-H traces using our prototype data warehouse 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 `orders` and `lineitem` 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 SSD 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 the `O_DIRECT|O_SYNC` flag to get around the 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.

5.2. Experiments with Synthetic Data

Comparing All Schemes for Handling Online Updates. Figure 16 compares the performance impact of online update schemes on range scan queries on the primary key while varying the range size from 4KB (a disk page) to 100GB (the entire table). For IU and MaSM schemes, the cached updates occupy 50% of the allocated 4GB SSD 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 the SSD, while the fine-grain index records one entry per 4KB cached updates. All the bars are normalized to the execution time of range scans without updates. As shown in Figure 16, 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 4KB to 1MB, 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 those 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.

Observing the second bars in Figure 16 shows that 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 for this range size.

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 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 35000 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.

Comparing Experimental Results with Modeled MaSM's Behavior. The previous experimental results follow the behavior predicted by the performance model described in Section 4.4. MaSM with coarse-grained run indexes results in significant performance degradation for short-range scans, while MaSM with fine-grained run indexes hides the merging overhead and shows zero to negligible slowdown for all ranges. According to the model, the same pattern is observed when one employs a system with a high-end array of disks for storage of the main data and an enterprise flash-based SSD for buffering the updates.

MaSM Varying Cached Update Size. Figure 17 varies both the range size (from 4KB to 100GB) and the cached update size (from 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 with coarse-grain index for 100MB to 100GB ranges. From Figure 17 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 has similar performance to MaSM with a 50% full 2GB-sized update cache. Therefore Figure 17 also represents the performance varying SSD space from 2GB to 8GB with a 50% full update cache.

Varying Update Speed. Figure 18 shows the MaSM performance while varying concurrent update speed from 2000 updates per scan to 10000 updates per scan. The allocated SSD space is 50% full at the beginning of the experiments. We perform 10 MaSM range scans with concurrent updates. We see that 10000 updates roughly occupies an 8MB space on the SSD, which is the allocated memory size for MaSM. Therefore the increase of unsorted updates is between 20% to 100% of M. As shown in the figure, the normalized scan time of MaSM shows a slight upward trend as the update speed increases, especially for the smaller ranges. This is because MaSM must process more unsorted data, which cannot be overlapped with the scan operation. The significance of this overhead increases as range size decreases.

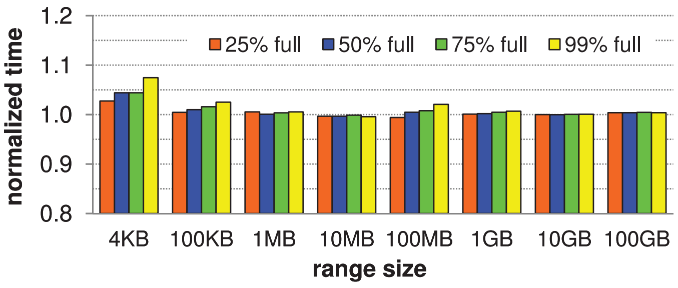

Fig. 17. MaSM range scans varying updates cached in SSD.

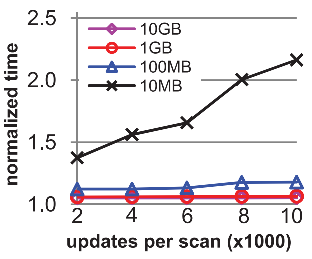

Fig. 18. Range scan with MaSM varying update speed.

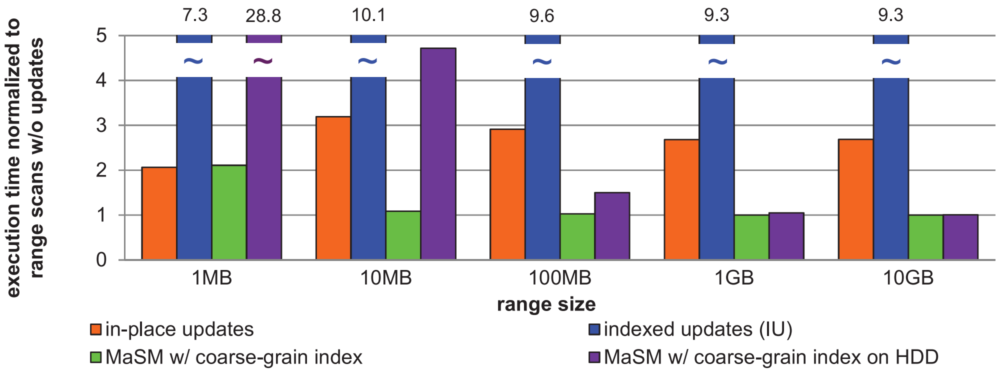

Fig. 19. Comparing the impact of online update schemes, with MaSM using both SSD and HDD, on the response time of primary key range scans (normalized to scans without updates).

HDD as Update Cache. We experiment using a separate SATA disk (identical to the main disk) as the update cache in MaSM. In this experiment we use a 10GB dataset and the size of the cache is 1GB. Figure 19 compares in-place updates, indexed updates (IU), and MaSM with coarse-grain index using either SSD or HDD to cache the updates. While the performance trends for the first three configurations are very similar to the respective configurations presented in Figure 16, when MaSM uses HDD to cache the updates, the overhead for short ranges is impractical. The poor random read performance of the disk-based update cache results in in 28.8x (4.7x) query slowdowns for 1MB-(10MB)-sized range scans. The overhead of using HDD to cache updates in MaSM diminishes for larger ranges where the cost of reading the cached updates is amortized with the cost to retrieve the main data. This experiment shows that, in order to accommodate modern analytical workloads which include both short and long range scans, it is essential to cache the updates in a storage medium offering good random read performance, making flash devices a good match.

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 20 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,

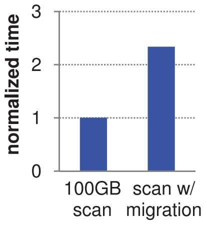

Fig. 20. MaSM update migration.

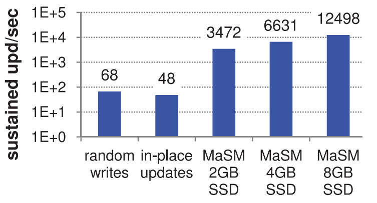

Fig. 21. Sustained updates per second varying SSD size in MaSM.

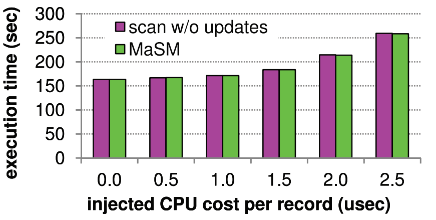

Fig. 22. Range scan vs. MaSM while emulating CPU cost of query processing (10GB ranges).

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 21 reports the sustained update throughput of in-place updates and three MaSM schemes with different SSD space. For in-place updates, we depict the best update rate by performing only updates, without concurrent queries. For 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 50% so that, in steady state, a table scan with migration is migrating updates in 50% of the SSD while the other 50% of the SSD is holding incoming updates. Figure 21 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 SSD space roughly doubles the sustained update rate.

Varying CPU Cost of Query Processing. Complex queries may perform a significant amount of in-memory processing after retrieving records from range scans. We study how MaSM behaves when the range scan becomes CPU bound. In Figure 22, 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 22, the performance is almost flat until the 1.5us point, indicating the range scan is I/O bound. From 1.5us to 2.5us, the execution time grows roughly linearly, indicating 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 the 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. Those sorted runs containing the cached updates are merged using heap-merge, hence the complexity is logarithmic in the number of sorted runs, that is, $O(log(\#runs))$.

Merging Overhead Analysis. We further investigate the merging overhead by running MaSM while storing both main data and updates on a ramdisk. This is a pure CPU-bound execution, hence any slowdowns are due to the merging overhead of MaSM. To run this experiment we use a server with more main memory: a Red-Hat Linux server with 2.6.32 64-bit kernel, equipped with 2 6-core 2.67GHz Intel Xeon CPU X5650 and 48GB of main memory. For this experiment, we generate a 10GB table with 100-byte-sized records and 4-byte primary keys. Similarly to the previous experiments, the table

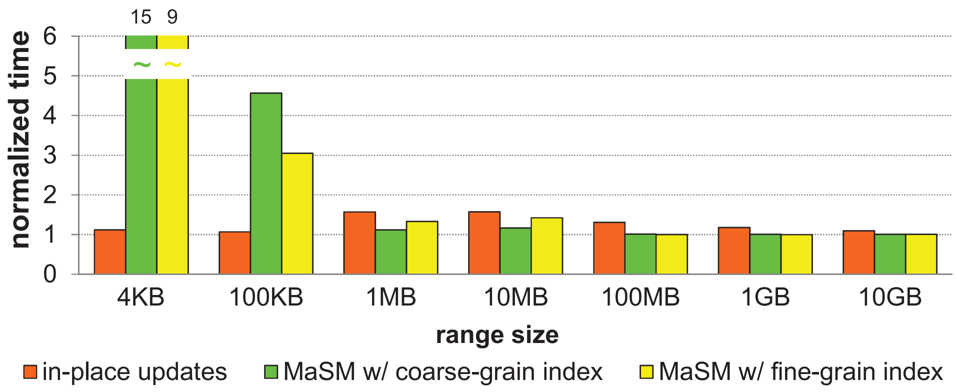

Fig. 23. Comparing in-place updates with MaSM performance (using both coarse-grain and fine-grain run indexes) when both data and updates are stored in memory (10GB table size).

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. We use 1GB of in-memory space for caching updates, thus MaSM-M requires 8MB memory for 64KB effective page size.

Figure 23 compares the performance of range scans using in-place updates and MaSM (both with coarse-grain and fine-grain indexes), varying the range size from 4KB (a disk page) to 10GB (the entire table). Similarly to previous experiments, coarse-grain indexes record one entry per 64KB cached updates, and fine-grain indexes record one entry per 4KB cached updates. When comparing coarse-grain with fine-grain indexes, note that coarse-grain indexes require reading bigger chunks of the sorted runs but perform fewer comparisons, while fine-grain indexes read more, but smaller chunks and, as a result, perform more comparisons. For both MaSM schemes, the cached updates occupy about 50% of the allocated 1GB space, which is the average amount of cached updates expected to be seen in practice. All bars are normalized to the execution time of range scans without updates.

As shown in Figure 23, contrary to the I/O-bound experiments, range scans with in-place updates see small performance penalty varying between 7% and 58%. On the other hand, both MaSM schemes show significant performance penalty for 4KB and 100KB ranges (varying from 3x to 15x). For 1MB and 10MB ranges, MaSM sees small performance degradation (12% to 42%) and for 100MB, 1GB, and 10GB ranges, MaSM sees no performance penalty. Similarly to the I/O-bound experiments, in CPU-bound experiments the MaSM overhead is hidden when a large portion of the main file is requested by the range scan. The CPU overhead of MaSM consists of two parts. The first part is the initial operation to search for the starting update record for a given range. The second part is the CPU overhead for merging update records with main data. When the range size is large, the second part dominates. From the figure, it is clear this overhead is negligible. This means the overhead for merging updates with main data is very small. When the range size is small, the first part becomes important, which is the reason for the large overhead for the 4KB and 100KB cases.

Overall, as expected, the merging overhead of MaSM for very short range scans cannot be amortized in the in-memory case. On the other hand, for range scans that are more than 0.01% of the file, MaSM offers better performance than in-place updates even when data are stored in the main memory. An observation that is entirely hidden in the I/O-bound experiments is that, for intermediate ranges (1MB and 10MB), the comparison overhead of the fine-grain run indexes exceeds the benefits of the reduced

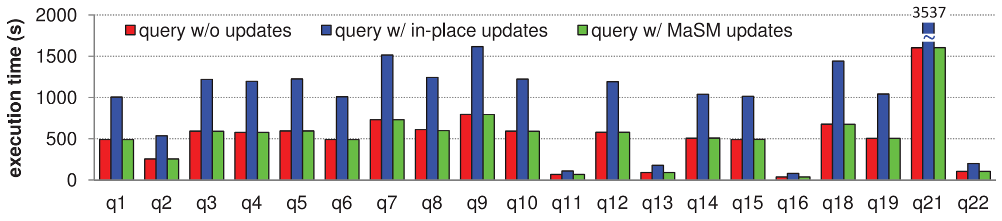

Fig. 24. Replaying I/O traces of TPC-H queries on a real machine with online updates.

amount of fetched data compared with coarse-grain run indexes, leading, nevertheless, to smaller slowdowns when compared to range scans with in-place updates.

5.3. TPC-H Replay Experiments

Figure 24 shows the execution times of the TPC-H replay experiments (in seconds). The left bar is the query execution time without updates, the middle bar is the query execution time with concurrent in-place updates, and the right bar is the query execution time with online updates using MaSM. For the MaSM algorithm, the SSD space is 50% full at the start of the query. MaSM divides the SSD space to maintain cached updates on a per-table basis in the TPC-H experiments.

From Figure 24, 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.

6. RELATED WORK

Extending MaSM Analysis. This work extends the analysis of the MaSM algorithms [Athanassoulis et al. 2011]. We extend MaSM analysis by providing guarantees (i.e., proofs) for the impact of MaSM algorithms regarding wearing the solid-state storage devices used (Section 4.1), analytical modeling of MaSM algorithms' performance trade-offs (Section 4.2 and Section 4.3), and analytical modeling of MaSM performance, using which we show that MaSM can be used for a wide variety of systems regarding their storage capabilities (Section 4.4). We provide a thorough comparison between MaSM and LSM regarding the impact in wearing the devices (Section 4.5). Moreover, we use the MaSM prototype system and extend the experimental evaluation of MaSM as well (Section 5) by showing the impact of varying the update rate. Finally, we discuss practical and economical implications of using MaSM algorithms regarding the cost of using SSDs in a storage setup and the applications of MaSM in the context of cloud data management.

In-Memory and External Data Structures. Our MaSM design extends prior work on in-memory differential updates [Héman et al. 2010; Stonebraker et al. 2005] to overcome the limitation on high migration costs versus large memory footprint. We assume I/O to be the main bottleneck for data warehousing 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.

Handling Updates with Stacked Tree Structures. In Section 2.3 we described two structures proposed to efficiently support updates in the context of data warehousing and high-performance transactional systems. Positional delta trees [Héman et al. 2010] provide efficient maintenance and ease in merging differential updates in column stores. They defer touching the read-optimized image of the data as long as possible and buffer differential updates in a separate write-store. The updates from the write-store are merged with the main data during column scans in order to present to the user an up-to-date version of the data. O'Neil et al. [1996] proposed *log-structured merge-tree* (LSM) which is organized as a collection of trees (two or more) stacked vertically, supporting high update rate of indexing structures of a database, like B-trees. The first level of LSM is always in memory and the last on disk. The updates are initially saved in the first level and consequently moved to the next levels (in batches) in order to reach the last level. Searching using LSM requires to traverse all stacked trees as each one is not inclusive of the subsequent trees. While LSM is capable of sustaining a high update rate and, if tuned appropriately, delivers good read performance, it performs more writes per update than MaSM, as detailed in Section 4.5.

LSM has inspired other approaches that maintain, index, and query data. FD-Tree [Li et al. 2010] is a tree structure that extends LSM by reducing the read overhead and by allowing flash-friendly writes. The key ideas are to limit random writes to a small part of the index and to transform random writes to sequential writes. FD-Tree assumes a fixed ratio between two consecutive levels in order to offer the best insertion cost amortization, which is the optimal case for LSM as well. As we discussed in Sections 2.3 and 4.5, this will incur an increased number of writes per update because every update is propagated a number of times.

The stepped-merge algorithm [Jagadish et al. 1997] stores updates lazily in a B^+-Tree organization by first maintaining updates in memory in sorted runs, and eventually forming a B^+-Tree of these updates using an external merge-sort. The stepped-merge approach aims at minimizing random I/O requests. On the other hand, MaSM focuses on minimizing main memory consumption and unnecessary writes on the flash device at the expense of more, yet efficient, random read I/O requests.

The quadtree-based storage algorithm [McCarthy and He 2011] was proposed concurrently and independently of the original conference paper [Athanassoulis et al. 2011]. It uses in-memory Δ-quadtrees to organize incoming updates for a data cube. The Δ-quadtrees are used to support the computation of specific operations like SUM, COUNT, and AVG over a predefined data cube. When the available memory is exhausted, the Δ-quadtrees are flushed to an SSD. Subsequent range queries are answered by merging the Δ-quadtrees in memory, the Δ-quadtrees on flash, and the main data in the data cube. Contrary to the quadtree-based storage algorithm, MaSM does not assume to be part of an aggregate calculation, and can handle incoming updates in a DW supporting a wide range of queries.

Database Cracking and Adaptive Indexing. There are many differential approaches for maintaining indexes and other data structures in order to enhance query performance. Database cracking [Idreos et al. 2007] reorganizes the data in memory to match how queries access data. Adaptive indexing [Graefe and Kuno 2010; Graefe et al. 2012] follows the same principle as database cracking by focusing the index optimization on key ranges used in actual queries. Their hybrids for column stores [Idreos et al. 2011] take the best of the two worlds. Contrary to MaSM, database cracking, adaptive index, and their hybrids focus on static data that are dynamically reorganized (or the corresponding indexes optimized) according to the incoming queries. The MaSM algorithms enable long-running queries to see almost no performance penalty in the presence of concurrent updates.

Partitioned B-Trees. Graefe [2003] proposed partitioned B-Trees in order to offer efficient resource allocation during index creation, or data loading into an already indexed database. The core idea is to maintain an artificial leading key column to a B-Tree index. If each value of this column has cardinality greater than one, the index entries are effectively partitioned. Such an indexing structure: (i) permits storing all runs of an external sort together, (ii) reduces substantially the wait time until a newly created index is available, and (iii) solves the dilemma whether one should drop the existing index or update the existing index one-record-at-a-time when a large amount of data is added to an already indexed large data warehouse. Partitioned B-Trees can work in parallel with MaSM in order to offer efficient updating of the main data as well as efficient indexing in data warehouses.

Extraction-Transformation Loading for Data Warehouses. We focus on supporting efficient query processing given online, well-formed updates. An orthogonal problem is an efficient ETL (*extraction-transformation-loading*) process for data warehouses [Oracle 2013; Polyzotis et al. 2008]. ETL is often performed at a data integration engine outside the data warehouse to incorporate changes from front-end operational data sources. Streamlining the ETL process has been both a research topic [Polyzotis et al. 2008; Simitsis et al. 2009] and a focus of a data warehouse product [Oracle 2013]. These ETL solutions can be employed to generate those well-formed updates to be applied to the data warehouse.

Sequential Reads and Random Writes in Storage Systems. Concurrent with our work, Schindler et al. [2011] proposed exploiting flash as a nonvolatile 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 data warehouses, 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 Data Warehouses. Orthogonal to our work, previous studies have investigated placing objects (such as data, updates, and indexes) on SSDs versus disks [Koltsidas and Viglas 2008; Agrawal et al. 2009; Canim et al. 2009; Ozmen et al. 2010; Li et al. 2010; Lim et al. 2011; Sadoghi et al. 2013; Athanassoulis and Ailamaki 2014], including SSDs as a caching layer between main memory and disks [Canim et al. 2010] and as temporary storage for enhancing performance of nonblocking joins [Chen et al. 2010].

7. DISCUSSION

Can SSDs Enhance Performance and Be Cost Efficient? SSDs have been studied as potential alternatives or enhancements to HDDs in many recent database studies [Athanassoulis et al. 2010; Bouganim et al. 2009; Canim et al. 2009, 2010; Chen 2009; Chen et al. 2010; Gao et al. 2011; Koltsidas and Viglas 2008; Lee et al. 2008; Ozmen et al. 2010; Stoica et al. 2009; Tsirogiannis et al. 2009]. In this article, we exploit SSDs to cache differential updates. Here, we perform a back-of-the-envelope computation to quantify the cost benefits of using SSDs.

Assuming we have a budget B for storage, we can either spend it on N disks or spend $X\%$ for disks and the rest for SSD. If $N \cdot X\%$ disks can provide the desired capacity then the performance gain of running a table scan when using SSD can be computed as follows. A table scan is linearly faster with the number of disks, while random updates slow performance down by a constant factor F_{HDD} which ranges between 1.5 and 4, as shown in Figure 3. When executing a table scan with concurrent random updates, the performance gain is N/F_{HDD}. If we buy SSD with $100\% - X\%$ of our budget, then, on the one hand, we decrease the performance gain due to disks to $N \cdot X\%$; on the other hand, the update cost is slowed down with a different constant factor F_{SSD} which is close to 1 in our experiments. The performance gain now is $(N \cdot X\%)/F_{SSD}$. Therefore the relative performance gain of using SSD is computed as follows:

$$RelativeGain = \frac{N \cdot X\%}{F_{SSD}} \cdot \frac{F_{HDD}}{N} = \frac{F_{HDD} \cdot X\%}{F_{SSD}}.$$

We can select the budget used for SSD to be low enough to ensure the desired overall capacity and high enough to lead to important performance gain. For example, if $X\% = 90\%$ of the budget is used for the disk storage, $RelativeGain$ of using SSD as in our proposal is between 1.35 and 3.6.

Shared-Nothing Architectures. Large analytical data warehouses often employ a shared-nothing architecture for achieving scalable performance [Becla and Lim 2008]. 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 datacenter discussions show it is reasonable to enhance every machine node with SSDs [Barroso 2010].

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. Similar to index-to-index navigation [Graefe 2010], in the first step, the secondary index is searched to retrieve all the record pointers within the range. In the second step, those record pointers are used to retrieve the records. Thus, any query retrieving a range of k records using a secondary index can be transformed to k point queries on the primary index. We show that point queries on the primary index can be handled efficiently, so that a query using a secondary index will be handled correctly and efficiently. An optimization for disk performance is to sort the record pointers according to the physical storage order of the records between the two steps. In this case, instead of k point queries, a query on a secondary index will be transformed to a number of range queries (less than or equal to k), having as a result better I/O performance.

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 those 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 data warehouses can maintain multiple copies of the data in different sort orders (a.k.a. projections) [Stonebraker et al. 2005; Lamb et al. 2012]. For example, in addition to a prevailing sort order of a table, one can optimize a specific query by storing 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 [Héman et al. 2010]. 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 an 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 [Zhou et al. 2007]. 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 data warehouse 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.

Applying MaSM to Cloud Data Management. Key-value stores are becoming a popular cloud data storage platform because of their scalability, availability, and simplicity. Many key-value store implementations (e.g., Bigtable, HBase, and Cassandra) follow the LSM principle to handle updates. In Section 1.2, we have discussed the difference between these schemes and MaSM. In the past several years, SSD capacity has been increasing dramatically and its price per capacity ratio is reducing steadily. This makes SSDs increasingly affordable; SSDs have already been considered as standard equipment for cloud machines [Barroso 2010]. When a key-value store node is equipped with an SSD, the presented MaSM algorithm can be applied to improve the performance, reduce memory footprint, and reduce SSD wear in key-value stores.

8. 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 high update migration overheads.

In this article, we proposed to judiciously use solid-state storage to cache differential updates. Our work is based on the principle of using SSD as a performance booster for databases stored primarily on magnetic disks since, for the foreseeable future, magnetic disks offer cheaper capacity (higher GB/$), while SSDs offer better but more expensive read performance (higher IOPS/$). We presented a high-level framework for SSD-based differential update approaches, and identified five design goals. We presented 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 so that, using MaSM, query response times remain nearly unaffected even if updates are running at the same time. Moreover, update throughput using the MaSM algorithms is several orders of magnitude higher than in-place updates and is, in fact, tunable by selecting different SSD buffer capacity.

ACKNOWLEDGMENTS

We are very grateful to the referees for their suggestions that helped improve the article; we are also grateful to the reviewers of the conference version whose comments, too, are reflected in this article.

REFERENCES

Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. 2009. Lazy adaptive tree: An optimized index structure for flash devices. *Proc. VLDB Endow.* 2, 1, 361–372.

Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-tree: Approximate tree indexing. *Proc. VLDB Endow.* 7, 14, 1881–1892.

Manos Athanassoulis, Anastasia Ailamaki, Shimin Chen, Phillip B. Gibbons, and Radu Stoica. 2010. Flash in a DBMS: Where and how? *IEEE Data Engin. Bull.* 33, 4, 28–34.

Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. 2011. MaSM: Efficient online updates in data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11).* 865–876.

Luiz Andre Barroso. 2010. Warehouse-scale computing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).*

Jacek Becla and Kian-Tat Lim. 2008. Report from the first workshop on extremely large databases (XLDB 2007). *Data Sci. J.* 7. http://www-conf.slac.stanford.edu/xldb07/xldb07_report.pdf.

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'neil, and Patrick E. O'neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Rec.* 24, 2, 1–10.

Luc Bouganim, Bjorn Thor Jonsson, and Philippe Bonnet. 2009. uFLIP: Understanding flash IO patterns. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'09).*

Werner Bux and Ilias Iliadis. 2010. Performance of greedy garbage collection in flash-based solid-state drives. *Perform. Eval.* 67, 11, 1172–1186.

Mustafa Canim, Bishwaranjan Bhattacharjee, George A. Mihaila, Christian A. Lang, and Kenneth A. Ross. 2009. An object placement advisor for DB2 using solid state storage. *Proc. VLDB Endow.* 2, 2, 1318–1329.

Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.* 3, 1–2, 1435–1446.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'06).*

Shimin Chen. 2009. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09).* 73–86.

Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2010. PR-join: A non-blocking join achieving higher early result rate with statistical guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 147–158.

Phillip M. Fernandez. 1994. Red brick warehouse: A read-mostly RDBMS for open SMP platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'94).* 492–492.

Fusionio. 2013. ioDrive2. http://www.fusionio.com/products/iodrive2/.

Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. 2011. PCMLogging: Reducing transaction logging overhead with PCM. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM'11).* 2401–2404.

Goetz Graefe. 1994. Volcano - An extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Engin.* 6, 1, 120–135.

Goetz Graefe. 2003. Sorting and indexing with partitioned B-trees. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'03).*

Goetz Graefe. 2006. B-tree indexes for high update rates. *ACM SIGMOD Rec.* 35, 1, 39–44.

Goetz Graefe. 2010. Modern B-Tree techniques. *Foundat. Trends Databases* 3, 4, 203–402.

Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, and Stefan Manegold. 2012. Concurrency control for adaptive indexing. *Proc. VLDB Endow.* 5, 7, 656–667.

Goetz Graefe and Harumi Kuno. 2010. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT'10).* 371–381.

Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastasia Ailamaki. 2005. QPipe: A simultaneously pipelined relational query engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. 383–394.

HBASE. 2013. Online reference. http://hbase.apache.org/.

Sandor Heman, Marcin Zukowski, and Niels J. Nes. 2010. Positional update handling in column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 543–554.

IBM. 2013. Keeping the data in your warehouse fresh. *IBM Data Mag.* http://ibmdatamag.com/2013/04/keeping-the-data-in-your-warehouse-fresh/.

Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR'07).*

Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.* 4, 9, 586–597.

William H. Inmon, R. H. Terdeman, Joyce Norris-Montanari, and Dan Meers. 2003. *Data Warehousing for E-Business*. John Wiley and Sons.

INTEL. 2009. Intel X25-E SSD. http://download.intel.com/design/flash/nand/extreme/319984.pdf.

H. V. Jagadish, P. P. S. Narayan, Sridhar Seshadri, S. Sudarshan, and Rama Kanneganti. 1997. Incremental organization for data recording and warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'97).* 16–25.

Donald E. Knuth. 1998. *The Art of Computer Programming:* Volume 3. Sorting and Searching. 2nd Ed. Addison-Wesley Longman.

Ioannis Koltsidas and Stratis D. Viglas. 2008. Flashing up the storage layer. *Proc. VLDB Endow.* 1, 1, 514–525.

Avinash Lakshman and Prashant Malik. 2010. Cassandra - A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* 44, 2, 35–40.

Andrew Lamb, Matt Fuller, and Ramakrishna Varadarajan. 2012. The Vertica analytic database: C-Store 7 years later. *Proc. VLDB Endow.* 5, 12, 1790–1801.

Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. 2008. A case for flash memory SSD in enterprise database applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08).* 1075–1086.

Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, Ke Yi, and Robin Jun Yang. 2010. Tree indexing on solid state drives. *Proc. VLDB Endow.* 3, 1–2, 1195–1206.

Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'11).* 1–13.

Mitzi Mccarthy and Zhen He. 2011. Efficient updates for OLAP range queries on flash memory. *Comput. J.* 54, 11, 1773–1789.

Guido Moerkotte. 1998. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'98).* 476–487.

Peter Muth, Patrick E. O'neil, Achim Pick, and Gerhard Weikum. 2000. The LHAM log-structured history data access method. *VLDB J.* 8, 3–4, 199–221.

Patrick E. O'neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4, 351–385.

Oracle. 2013. Oracle database 12c for data warehousing and big data. Oracle white paper. http://www.oracle.com/technetwork/database/bi-datawarehousing/data-warehousing-wp-12c-1896097.pdf.

Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. 2010. Workload-aware storage layout for database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 939–950.

Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils-Erik Frantzell. 2008. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. Knowl. Data Engin.* 20, 7, 976–991.

Philip Russom. 2012. High-performance data warehousing. TDWI best practices report. http://tdwi.org/webcasts/2012/10/high-performance-data-warehousing.aspx.

Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and, Bishwaranjan Bhattacharjee. 2013. Making updates disk-I/O friendly using SSDs. *Proc. VLDB Endow.* 6, 11, 997–1008.

Jiri Schindler, Sandip Shete, and Keith A. Smith. 2011. Improving throughput for small disk requests with proximal I/O. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'11)*. 133–147.

Dennis G. Severance and Guy M. Lohman. 1976. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.* 1, 3, 256–267.

Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee, Ramana Yerneni, and Raghu Ramakrishnan. 2008. Efficient bulk insertion into a distributed ordered table. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 765–778.

Alkis Simitsis, Kevin Wilkinson, Malu Castellanos, and Umeshwar Dayal. 2009. QoX-driven ETL design: Reducing the cost of ETL consulting engagements. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. 953–960.

Radu Stoica and Anastasia Ailamaki. 2013. Improving flash write performance by using update frequency. *Proc. VLDB Endow.* 6, 9, 733–744.

Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. 2009. Evaluating and repairing write performance on flash devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DA-MON'09)*. 9–14.

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sammuel R. Madden, Elizabeth J. O'neil, Patrick E. O'neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A column-oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'05)*. 553–564.

Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. 2009. Query processing techniques for solid state drives. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. 59–72.

Colin White. 2002. Intelligent business strategies: Real-time data warehousing heats up. In *DM Review*. http://www.information-management.com/issues/20020801/5570-1.html.

Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. 2007. Lazy maintenance of materialized views. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'07)*. 231–242.

Paul Zikopoulos, Dirk Deroos, Krishnan Parasuraman, Thomas Deutsch, David Corrigan, and James Giles. 2012. *Harness the Power of Big Data – The IBM Big Data Platform*. McGraw-Hill.

Marcin Zukowski, Sandor Heman, Niels J. Nes, and Peter Boncz. 2007. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'07)*. 723–734.

Received March 2013; revised August 2014; accepted November 2014