Industrial and Applications Paper

L-Store: A Real-time OLTP and OLAP System*

Mohammad Sadoghi[†], Souvik Bhattacherjee[‡], Bishwaranjan Bhattacharjee[#], Mustafa Canim[#]

[†]Exploratory Systems Lab
[†]University of California, Davis
[‡]University of Maryland, College Park
[#]IBM T.J. Watson Research Center

ABSTRACT

To derive real-time actionable insights from the data, it is important to bridge the gap between managing the data that is being updated at a high velocity (i.e., OLTP) and analyzing a large volume of data (i.e., OLAP). However, there has been a divide where specialized solutions were often deployed to support either OLTP or OLAP workloads but not both; thus, limiting the analysis to stale and possibly irrelevant data. In this paper, we present Lineage-based Data Store (L-Store) that combines the real-time processing of transactional and analytical workloads within a single unified engine by introducing a novel update-friendly lineage-based storage architecture. By exploiting the lineage, we develop a contention-free and lazy staging of columnar data from a write-optimized form (suitable for OLTP) into a read-optimized form (suitable for OLAP) in a transactionally consistent approach that supports querying and retaining the current and historic data.

1 INTRODUCTION

We are witnessing an architectural shift and divide in database community. The first school of thought emerged from an academic conjecture that "*one size does not fit all*" [37] (i.e., *advocating specialized solutions*), which has lead to manifolds of innovations over the last decade in creating specialized and subspecialized database engines geared toward various niche workloads and application scenarios [5, 9, 12, 22, 28, 29, 37, 38]. This school has motivated major commercial database vendors such as Microsoft to focus on building novel specialized engines offered as loosely integrated engines, namely, Hekaton in-memory engine [9] and Apollo column store engine [19], within a single umbrella of database portfolio. Notably, recent efforts are focused on a tighter real-time integration of Hekaton and Apollo engines [17]. It has inspired Oracle to push the boundary of the basic premise that "one size does not fit all" as far as data representation is concerned and has led Oracle to develop a dual-format technique [15] that maintains two tightly integrated representation of data (i.e., two copies of the data) in a transactionally consistent manner.

However, the second school of thought, supported by both academia (e.g., [2, 6, 7, 13, 16, 24]) and industry (e.g., SAP [10], IBM DB2 BLU [29], and IBM Wildfire [4]) have revisited the aforementioned fundamental premise and advocates a generalized solution. Proponents of this idea, rightly in our view, make the following arguments. First, there is a tremendous cost in building and maintaining multiple engines from both the perspective of database vendors and users of the systems (e.g., application development and deployment costs). Second, there is a compelling case to support real-time decision making on the latest version of the data [27] (likewise supported by [15, 17]), which may not be feasible across loosely integrated engines that are connected through the extract-transform load (ETL) process. Closing this gap may be possible, but its elimination may not be feasible without solving the original problem of unifying OLTP and OLAP capabilities or without being forced to rely on ad-hoc approaches to bridge the gap in hindsight. We argue that the separation of OLTP and OLAP capabilities defers solving the actual challenge of real-time analytics. Third, combining real-time OLTP and OLAP functionalities remains as an important basic research question, which demands deeper investigation even if it is purely from the theoretical standpoint.

In this dilemma, we support the latter school of thought (i.e., *advocating a generalized solution*) with the goal of undertaking an important step to study the entire landscape of single engine architectures and to support both transactional and analytical workloads holistically (i.e., "*one size fits all*"). In this paper, we present Lineage-based Data Store (L-Store) with a novel update-friendly lineage-based storage architecture to address the conflicts between row- and column-major representation. This is achieved by developing a contention-free and lazy staging of columnar data from write optimized into read optimized form in a transactionally consistent manner without the need to replicate data, to maintain multiple representation of data, or to develop multiple loosely integrated engines that sacrifices real-time capabilities.

To further disambiguate our notion of "*one size fits all*", in this paper, we restrict our focus to real-time relational OLTP and OLAP capabilities. We define a set of architectural characteristics for distinguishing the differences between existing techniques. First, there could be a single product consisting of multiple loosely integrated engines that can be deployed and configured to support either OLTP or OLAP. Second, there could be a single engine as opposed to having multiple specialized engines packaged in a single product. Third, even if we have a single engine, then we could have multiple instances running over a single engine, where one instance is dedicated and configured for OLTP workloads while another instance is optimized for OLAP workloads, in which these instances are assumed to be connected using an ETL process. Finally, even when using the same engine running a single instance, there could be multiple copies or representations (e.g., row vs. columnar layout) of the data, where one copy (or representation) of the data is read optimized while the second copy (or representation) is write optimized.

In short, we develop L-Store, an important first step towards supporting real-time OLTP and OLAP processing that faithfully satisfies our definition of *generalized solution*, and, in particular, we make the following contributions:

- Introducing an update-friendly lineage-based storage architecture that enables a contention-free update mechanism over a native multi-version, columnar storage model in order to lazily and independently stage stable data from a write-optimized columnar layout (i.e., OLTP) into a read-optimized columnar layout (i.e., OLAP)
- Achieving (at most) 2-hop away access to the latest version of any record (preventing read performance deterioration for point queries)

*Work by S. Bhattacherjee was performed as part of a summer internship at IBM T.J. Watson Research Center under M. Sadoghi's mentorship.

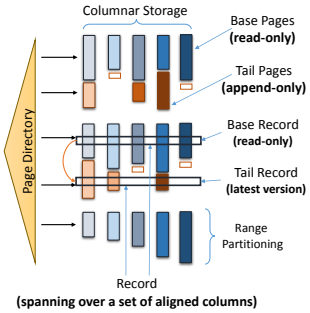


Figure 1: Overview of the lineage-based storage architecture.

- Contention-free merging of only stable data, namely, merging of the read-only base data with recently committed updates (both in columnar representation) without the need to block ongoing or new transactions by relying on the lineage
- Contention-free page de-allocation (upon the completion of the merge process) using an epoch-based approach without the need to drain the ongoing transactions
- A first of its kind comprehensive evaluation to study the leading architectural storage design for concurrently supporting short update transactions and analytical queries (e.g., an in-place update with a history table architecture and the commonly employed main and delta stores architecture)

2 UNIFIED ARCHITECTURE

The divide in the big data community is partly attributed to the storage conflict pertaining to the representation of transactional and analytical data. In particular, transactional data requires write--optimized storage, namely the row-based layout, in which all columns are co-located (and preferably uncompressed for in-place updates). This layout improves point update mechanisms, since accessing all columns of a record can be achieved by a single I/O (or few cache misses for memory-resident data). In contrast, to optimize the analytical workloads (i.e., reading many records), it is important to have read-optimized storage, i.e., columnar layout in highly compressed form. The intuition behind having columnar layout is due to the observation that most analytical queries tend to access only a small subset of all columns [1]. Thus, by storing data column-wise, we can avoid reading irrelevant columns (i.e., reducing the raw amount of data read) and avoid polluting processor's cache with irrelevant data, which substantially improve both disk and memory bandwidth, respectively. Furthermore, storing data in columnar form improves the data homogeneity within each page, which results in an overall better compression ratio.

2.1 L-Store Storage Overview

To address the dilemma between write- and read-optimized layouts, we develop L-Store. As demonstrated in Figure 1, the high-level architecture of L-Store is based on a native multi-version, columnar layout (i.e., data across columns are aligned to allow implicit re-construction), where records are (virtually) partitioned into disjoint ranges (also referred to as update range). Records within each range span a set of read-only, compressed pages, which we refer to them as the *base pages*. More importantly, for every range of records, and for each updated column within the range, we maintain a set of append-only pages to store the latest updates, which we refer to them as the *tail pages*. Anytime a record is updated in base pages, a new record is appended to its corresponding tail pages, where there are explicit values only for the updated columns (non-updated columns are preassigned a special null value when a page is first allocated). We refer to the records in base pages as the *base records* and the records in tail pages as the *tail records*. Each record (whether falls in base or tail pages) spans over a set of aligned columns (i.e., no join is necessary to pull together all columns of the same record).[1]

A unique feature of our lineage-based architecture is that tail pages are strictly append-only and follow a write-once policy. In other words, once a value is written to tail pages, it will not be over-written even if the writing transaction aborts. The append-only design together with retaining all versions of the record substantially simplifies low-level synchronization and recovery protocol and enables efficient realization of multi-version concurrency control. Another important property of our lineage-based storage is that all data are represented in a common unified form; there are no ad-hoc corner cases. Records in both base and tail pages are assigned record-identifiers (RIDs) from the same key space. Therefore, both base and tail pages are referenced through the database page directory using RIDs and persisted identically. Therefore, at the lower-level of the database stack, there is absolutely no difference between base vs. tail pages or base vs. tail records; they are presented and maintained identically.

To speed up query processing, there is also an explicit linkage (forward and backward pointers) among records. From a base record, there is a forward pointer to the latest version of the record in tail pages. The different versions of the same records in tail pages are chained together to enable fast access to an earlier version of the record. The linkage is established by introducing a table-embedded indirection column that stores forward pointers for base records and backward pointers for tail records (i.e., RIDs).

The final aspect of our lineage-based architecture is a periodic, contention-free merging of a set of base pages with its corresponding tail pages. This is performed to consolidate base pages with the recent updates and to bring base pages forward in time (i.e., creating a set of merged pages). Each merged page independently maintains its lineage information, i.e., keeping track of all tail records that are consolidated onto the page thus far. By maintaining explicit in-page lineage information, the current state of each page can be determined independently, and the base page can be brought up to any desired snapshot. Tail pages that are already merged and fall outside the snapshot boundaries of all active queries are called historic tail-pages. These pages are re-organized, so that different versions of a record are stored contiguously inlined. Delta-compression is applied across different versions of tail records, and tail records are ordered based on the RIDs of their corresponding base records. Below, we describe the unique design and algorithmic features of L-Store that enables efficient transactional processing without performance deterioration of analytical processing; thereby, achieving a real-time OLTP and OLAP.

2.2 Lineage-based Storage Architecture

In L-Store, the storage layout is natively columnar and applies equally to both base and tail pages. A detailed view of our lineage-based storage architecture is presented in Figure 2. In general, one can perceive tail pages as directly mirroring the structure and the schema of base pages. As we pointed out earlier, conceptually for every record, we distinguish between base vs. tail records, where each record is assigned a unique RID. But it is important to note that the RID assigned to a base record is stable and remains constant throughout the entire life-cycle of a record, and all indexes only reference base records (base RIDs); consequently, eliminating index maintenance problem associated when update operation results in creation of a new version of the record [33, 34]. When a reader performing index lookup, it always lands at a base record, and from the base record it can reach any desired version of the record by following the table-embedded indirection to access the latest (if the base record is out-of-date) or an earlier version of the record. However, when a record is updated, a new version is

[1] Fundamentally, there is no difference between base vs. tail record, the distinction is made only to ease the exposition.

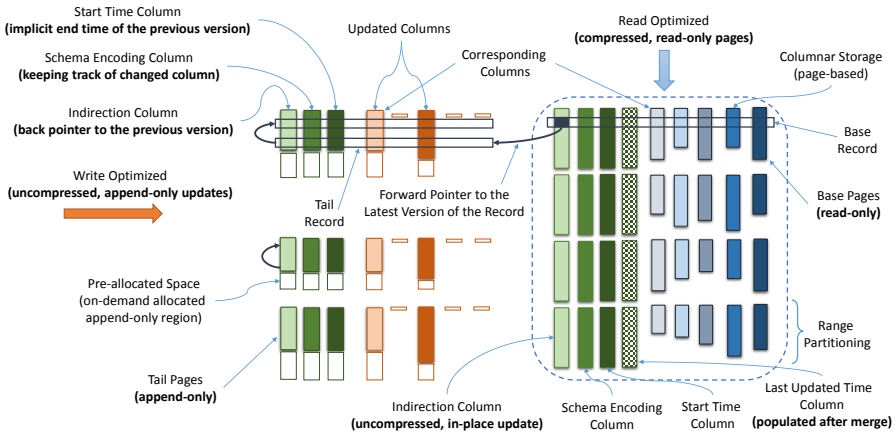


Figure 2: Detailed, unfolded view of lineage-based storage architecture (a multi-version, columnar storage model).

created. Thus, a new tail record is created to hold the new version, and the new tail record is assigned a new tail RID that is referenced by the base record (as demonstrated in Figure 2).

Each table in addition to having the standard data columns has several meta-data columns. These meta-data columns include the *Indirection* column, the *Schema Encoding* column, the *Start Time* column, and the *Last Updated Time* column. An example of table schema is shown in Table 1.

The *Indirection* column exists in both the base and tail records. For base records, the *Indirection* column is interpreted as a forward pointer to the latest version of a record residing in tail pages, essentially storing the RID of the latest version of a record. If a record has never been updated, then the *Indirection* column will hold a null value. In contrast, for tail records, the *Indirection* column is used to store a backward pointer to the last updated version of a record in tail pages. If no earlier version exists, then the *Indirection* column will point to the RID of the base record.

The *Schema Encoding* column stores the bitmap representation of the state of the data columns for each record, where there is one bit assigned for every column in the schema (excluding the meta-data columns), and if a column is updated, its corresponding bit in the *Schema Encoding* column is set to 1, otherwise is set to 0. The schema encoding enables to quickly determine if a column has ever been updated or not for base records. In tail records, the encoding tracks which columns have been updated and have explicit values as opposed to those columns that have not been updated and have an implicit special null values (denoted by ∅). An example of *Schema Encoding* column is provided in Table 1.

The *Start Time* column stores the time at which a base record was first installed in base pages (the original insertion time), and for a tail record, the *Start Time* column holds the time at which the record was updated, which is also the implicit end time of the previous version of the record. The *Start Time* column is essential for distinguishing between different version of the record. In addition, to the *Start Time* column, for base records, we maintain an optional *Last Updated Time* column, which is only populated after the merge process is taken place and reflects the *Start Time* of those tail records included in merged pages. Also note that the initial *Start Time* column for base records is always preserved even after the merge process for faster pruning of those records that are not visible to readers because they fall outside the reader's snapshot. Lastly, we may add the *Base RID* column optionally to tail records to store the RIDs of their corresponding base records; this is utilized to improve the merge process. *Base RID* is a highly compressible column that would require at most two bytes when restricting the range partitioning of records to 2^{16} records.

3 FINE-GRAINED MANIPULATION

The transaction processing can be viewed as two major challenges: (1) how data is physically manipulated at the storage layer and how changes are propagated to indexes and (2) how multiple

RID	*Indirection*	*Schema Encoding*	*Start Time*	*Key*	*A*	*B*	*C*
Partitioned base records for the key range of k_1 to k_3							
b_1	t_8	0000	10:02	k_1	a_1	b_1	c_1
b_2	t_5	0101	13:04	k_2	a_2	b_2	c_2
b_3	t_7	0001	15:05	k_3	a_3	b_3	c_3
Partitioned base records for the key range of k_4 to k_6							
b_4	⊥	0000	16:20	k_4	a_4	b_4	c_4
b_5	⊥	0000	17:21	k_5	a_5	b_5	c_5
b_6	⊥	0000	18:02	k_6	a_6	b_6	c_6
Partitioned tail records for the key range of k_1 to k_3							
t_1	b_2	0100*	13:04	∅	a_2	∅	∅
t_2	t_1	0100	19:21	∅	a_{21}	∅	∅
t_3	t_2	0100	19:24	∅	a_{22}	∅	∅
t_4	t_3	0001*	13:04	∅	∅	∅	c_2
t_5	t_4	0101	19:25	∅	a_{22}	∅	c_{21}
t_6	b_3	0001*	15:05	∅	∅	∅	c_3
t_7	t_6	0001	19:45	∅	∅	∅	c_{31}
t_8	b_1	0000	20:15	∅	∅	∅	∅

Table 1: An example of the update and delete procedures (conceptual tabular representation).

transactions (where each transaction consists of many statements) can concurrently coordinate reading and writing of the shared data. The focus of this paper is on the former challenge, and we defer the latter to our discussion on the employed low-level synchronization and concurrency control in Section 5.

Without loss of generality, from the perspective of the storage layer, we focus on how to handle a single point update or delete in L-Store (but note that we support multi-statement transactions through L-Store's transaction layer as demonstrated by our evaluation). Furthermore, in our technical report [31], we discuss how our model can easily be extended to deal with insertion as well. Each update may affect a single or multiple records. Since records are (virtually) partitioned into a set of disjoint ranges as shown in Table 1, each updated record naturally falls within only one range. Now for each range of records, upon the first update to that range, a set of tail pages are created (and persisted on disk optionally) for the updated columns and are added to the page directory, i.e., lazy tail-page allocation. Consequently, updates for each record range are appended to their corresponding tail pages of the updated columns only; thereby, retraining all versions of the record, avoiding in-place updates of modified data columns, and clustering updates for a range of records within their corresponding tail pages.

To describe the update procedure in L-Store, we rely on our running example shown in Table 1. When a transaction updates any column of a record for the first time, two new tail records (each tail record is assigned a unique RID) are created and appended to the corresponding tail pages. For example, consider updating the column A of the record with the key k_2 (referenced by the RID b_2) in Table 1. The first tail record, referenced by the RID t_1, contains the original value of the updated column, i.e., a_2, whereas implicit null values (∅) are preassigned for remaining unchanged columns. Taking a snapshot of the original changed values becomes essential in order to ensure contention-free merging as discussed in Section 4.1. The second tail record contains the newly updated value for column A, namely, a_{21}, and again

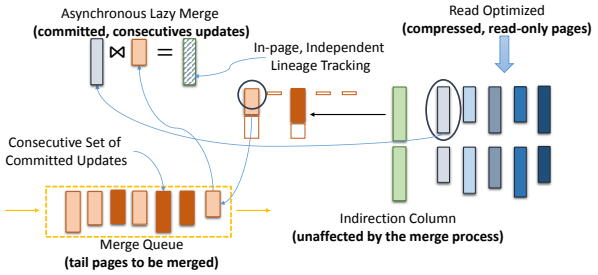


Figure 3: Lazily, independently merging of tail & base pages.

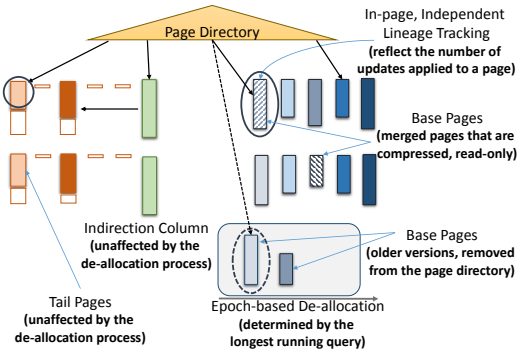


Figure 4: Epoch-based, contention-free page de-allocation.

implicit special null values for the rest of the columns; a column that has never been updated does not even have to be materialized with special null values. However, for any subsequent updates, only one tail record is created, e.g., the tail record t_3 is appended as a result of updating the column A from a_{21} to a_{22} for the record b_2.

In general, updates could either be cumulative or non-cumulative. The cumulative property implies that when creating a new tail record, the new record will contain the latest values for all of the updated columns thus far. For example, consider updating the column C for the record b_2. Since the column C of the record b_2 is being updated for the first time, we first take a snapshot of its old value as captured by the tail record t_4. Now for the cumulative update, a new tail record is appended that repeats the previously updated column A, as demonstrated by the tail record t_5. If non-cumulative update approach was employed, then the tail record would consists of only the changed value for column C and not A. It is important to note that cumulation of updates can be reset at anytime. In the absence of cumulation, readers are simply forced to walk back the chain of recent versions to retrieve the latest values of all desired columns. Thus, cumulative update is an optimization that is intended to improve the read performance.

As part of the update routine, the embedded *Indirection* column (forward pointers) for base records is also updated to point to the newly created tail record. In our running example, the *Indirection* column of the record b_2 points to the tail record t_5. Also after updating the column C of the record b_3, the *Indirection* column points to the latest version of b_3, which is given by t_7. Likewise, the *Indirection* column in the tail records point to the previous version of the record. It is important to note that the *Indirection* column of base records is the only column that requires an in-place update in our architecture. However, as discussed in our low-level synchronization protocol (cf. Section 5), this is a special column that lends itself to latch-free synchronization.

Furthermore, indexes always point to base records (i.e., base RIDs), and they are never directly point to any tail records (i.e., tail RIDs) in order to avoid the index maintenance cost that arise in the absence of in-place update mechanism [33]. Therefore, when a new version of a record is created (i.e., a new tail record), first, all indexes defined on unaffected columns do not have to be modified and, second, only the affected indexes are modified with the updated values, but they continue to point to base records and not the newly created tail records. Suppose there is an index defined on the column C (cf. Table 1). Now after modifying the record b_2 from c_2 to c_{21}, we add the new entry (c_{21}, b_2) to the index on the column C.[2] Subsequently, when a reader looks up the value c_{21} from the index, it always arrives at the base record b_2 initially, then the reader must determine the visible version of b_2 (by following the indirection if necessary) and must check if the visible version has the value c_{21} for the column C, essentially re-evaluating the query predicates.

There are two other meta-data columns that are affected by the update procedure. The *Start Time* column for tail records simply holds the time at which the record was updated (an implicit end of the previous version). For example, the record t_7 has a start time of 19:45, which is also the implied end time of the first version of the record b_3. The *Schema Encoding* column is a concise representation that shows which data columns have been updated thus far. For example, the *Schema Encoding* of the tail record t_7 is set to "0001", which implies that only the column C has been changed. To distinguish between whether a tail record is holding new values or it is the snapshot of old values, we add a flag to the *Schema Encoding* column, which is shown as an asterisk. For example, the tail record t_6 stores the old value of the column C, which is why its *Schema Encoding* is set to "0001*". The *Schema Encoding* can also be maintained optionally for base records as part of the update process or it could be populated only during the merge process.

Notably, when there are multiple individual updates to the same record by the same transaction, each update is written as a separate entry to tail pages. Each update results in a creation of a new tail record and only the final update becomes visible to other transactions. The prior entries are implicitly invalidated and skipped by readers. Also delete operation is simply translated into an update operation, in which all data columns are implicitly set to $\varnothing$, e.g., deleting the record b_1 results in creating the tail record t_8. An alternative design for delete is to create a tail record that holds a complete snapshot of the latest version of the deleted record.

4 REAL-TIME STORAGE ADAPTION

To ensure a near optimal storage layout, outdated base pages are merged lazily with their corresponding tail pages in order to preserve the efficiency of analytical query processing. Recall that the base pages are read-only and compressed (read optimized) while the tail pages are uncompressed[3] that grow using a strictly append-only technique (write optimized). Therefore, it is necessary to transform the recent committed updates accumulated in tail pages that are write optimized into read optimized form. A distinguishing feature of our lineage-based architecture is to introduce a contention-free merging process that is carried out completely in the background without interfering with foreground transactions. Furthermore, the contention-free merging procedure is applied only to the updated columns of the affected update ranges. There is even no dependency among columns during the merge; thus, the different columns of the same record can be merged completely independent of each other at different points in time. This is achieved by independently maintaining in-page lineage information for each merged page. The merge process is conceptually depicted in Figure 3, in which writer threads (i.e., update transactions) place candidate tail pages to be merged into

[2] Optionally the old value (c_2, b_2) could be removed from the index; however, its removal may affect those queries that are using indexes to compute answers under snapshot semantics. Therefore, we advocate deferring the removal of changed values from indexes until the changed entries fall outside the snapshot of all relevant active queries.

[3] Even though compression techniques such as local and global dictionaries can be employed in tail pages, but these directions are outside the scope of the current work.

the merge queue while the merge thread continuously takes pages from the queue and processes them.

4.1 Contention-free, Relaxed Merge

In L-Store, we abide to one *main design principle* for ensuring contention-free processing that is *"always operating on stable data"*. The inputs to the merge process are (1) a set of base pages (committed base records) that are read-only,[4] thus, stable data and (2) a set of consecutive committed tail records in tail pages,[5] thus, also stable data. The output of the merge process (that is also relaxed) is a set of newly consolidated base pages (also referred to as merged pages) with in-page lineage information that are read-only, compressed, and almost up-to-date, thus, stable data. To decouple users' transactions (writers) from the merge process, we also ensure that the write path of the ongoing transactions does not overlap with the write path of the merge process. Writers append new uncommitted tail records to tail pages, but as stated before uncommitted records do not participate in the merge. Writers also perform in-place update of the *Indirection* column within base records to point to the latest version of the updated records in tail pages, but the *Indirection* column is not modified by the merge process. In contrast, the write path of the merge process consists of creating only a new set of read-only base pages.

Merge Algorithm The details of the merge algorithm, conceptually resembling the standard left-outer join, consists of (1) identifying a set of committed tail records in tail pages; (2) loading the corresponding outdated base pages; (3) consolidating the base and tail pages while maintaining the in-page lineage; (4) updating the page directory; and (5) de-allocating the outdated base pages. The pseudo code for the merge is shown in Algorithm 1, where each of the five mentioned steps are also highlighted.

Step 1: Identify committed tail records in tail pages: *Select a set of consecutive fully committed tail records (or pages) since the last merge within each update range.*

Step 2: Load the corresponding outdated base pages: *For a selected set of committed tail records, load the corresponding outdated base pages for the given update range (limit the load to only outdated columns). This step can further be optimized by avoiding to load sub-ranges of records that have not yet changed since the last merge. No latching is required when loading the base pages.*

Step 3: Consolidate the base and tail pages: *For every updated column, the merge process will read n outdated base pages and applies a set of recent committed updates from the tail pages and writes out m new pages.*[6] *First the Base RID column of the committed tail pages (from Step 1) are scanned in reverse order to find the list of the latest version of every updated record since the last merge (a temporary hashtable may be used to keep track whether the latest version of a record is seen or not). Subsequently, applying the latest tail records in a reverse order to the base records until an update to every record in the base range is seen or the list is exhausted, skipping any intermediate versions for which a newer update exists in the selected tail records. If a latest tail record indicates the deletion of the record, then the deleted record will be included in the consolidated records. The merged pages will keep track of the lineage information in-page, i.e., tracking how many tail records have been consolidated thus far. Any compression algorithm (e.g., dictionary encoding) can be applied on the consolidated pages (on column basis) followed by writing the compressed pages into newly created pages. Moreover, the old Start Time column is remained intact during the merge process because this column is needed to hold the original insertion time of the record.*[7] *Therefore, to keep track of the time for the consolidated records, the Last Updated Time column is populated to store the Start Time of the applied tail records. The Schema Encoding column may also be populated during the merge to reflect all the columns that have been changed for each record.*

Step 4: Update the page directory: *The pointers in the page directory are updated to point to the newly created merged pages. Essentially this is the only foreground action taken by the merge process, which is simply to swap and update pointers in the page directory – an index structure that is updated rarely only when new pages are allocated.*

Step 5: De-allocate the outdated base pages: *The outdated base pages are de-allocated once the current readers are drained naturally via an epoch-based approach. The epoch is defined as a time window, in which the outdated base pages must be kept around as long as there is an active query that started before the merge process. Pointers to the outdated base pages are kept in a queue to be re-claimed at the end of the query-driven epoch-window. The pointer swapping and the page de-allocation are illustrated in Figure 4.* ■

Algorithm 1: Merge Algorithm

```
   Input  : Queue of unmerged committed tail pages (mergeQ)
   Output : Queue of outdated and consolidated base pages to be deallocated
            (deallocateQ)
1  while true do
2      // Step 1
3      // wait until the the concurrent merge queue is not empty
4      if mergeQ is not empty then
5          // Step 2
6          // fetch references to a set of committed tail pages
7          batchTailPage <-- mergeQ.dequeue()
8          // create a copy of corresponding base pages
9          batchConsPage ← batchTailPage.getBasePageCopy()
10         decompress(batchConsPage)
11         // track if it has seen the latest update of every record
12         HashMap seenUpdatesH
13         //reading a set of tail pages in reverse order
14         // Step 3
15         for i = 0; i <batchTailPage.size; i ← i + 1 do
16             tailPage ← batchTailPages[i]
17             for j = k − 1; j ≥ tailPage.size; j ← j − 1 do
18                 record[j] ← j^th record in the tailPage
19                 RID ← record[j].RID
20                 if seenUpdatesH does not contain RID then
21                     seenUpdatesH.add(RID)
22                     // copy the latest version of record into consolidated pages
23                     batchConsPage.update(RID, record[j])
24                 end
25                 if if all RIDs OR all tail pages are seen then
26                     compress(batchConsPage)
27                     persist(batchConsPage)
28                     stop examining remaining tail pages
29                 end
30             end
31         end
32         // Step 4
33         // fetch references to the corresponding base pages
34         batchBasePage <-- batchTailPage.getBasePageRef()
35         // update page directory to point to the consolidated base pages
36         PageDirect.swap(batchBasePage, batchConsPage)
37         // Step 5
38         // queue outdated pages for deallocation once readers prior merge are drained
39         deallocateQ.enqueue(batchBasePage)
40     end
41 end
```

[4] The *Indirection* column is the only column that undergoes in-place update that also never participates in the merge process.

[5] Note that not every committed update has to be applied as the merge process is relaxed, and the merge eventually process all committed tail records.

[6] At most up to one merged page per column could be left underutilized for a range of records after the merge process. To further reduce the underutilized merged pages, one may define finer range partitioning for updates (e.g., 2^{12} records), but operate merges at coarser granularity (e.g., 2^{16} records). This will provide the benefit of locality of access for readers given smaller range size of 2^{12}, yet it provides a better space utilization and compression for newly created merge pages when larger ranges are chosen (cf. Section 4.3).

[7] The *Start Time* column is also highly compressible column with a negligible space overhead to maintain it.

RID	Indirection	Schema Encoding	Start Time	Last Updated Time	Key	A	B	C
Partitioned base records for the key range of k_1 to k_3; Tail-page Sequence Number (TPS) = 0								
b_1	t_8	0000	10:02		k_1	a_1	b_1	c_1
b_2	t_5	0101	13:04		k_2	a_2	b_2	c_2
b_3	t_7	0001	15:05		k_3	a_3	b_3	c_3
Relevant tail records (below TPS $\leq t_7$ high-watermark) for the key range of k_1 to k_3								
t_5	t_4	0101	19:25		∅	a_{22}	∅	c_{21}
t_7	t_6	0001	19:45		∅	∅	∅	c_{31}
Resulting merged records for the key range of k_1 to k_3; TPS = t_7								
b_1	t_8	0000	10:02	10:02	k_1	a_1	b_1	c_1
b_2	t_5	0101	13:04	19:25	k_2	a_{22}	b_2	c_{21}
b_3	t_7	0001	15:05	19:45	k_3	a_3	b_3	c_{31}

Table 2: An example of the relaxed and almost up-to-date merge procedure (conceptual tabular representation).

An example of our merge process is shown in Table 2 based on our earlier update example, in which we consolidate the first seven tail records (denoted by t_1 to t_7) with their corresponding base pages. The resulting merged pages are shown, where the affected records are highlighted. Note that only the updated columns are affected by the merge process (and the *Indirection* column is not affected). Furthermore, not all updates are needed to be applied, only the latest version of every updated record needs to be consolidated while the other entries are simply discarded. In our example, only the tail records t_5 and t_7 participated in the merge, and the rest were discarded.

Merge Correctness Analysis A key distinguishing feature of our lineage-based storage architecture is to allow contention-free merging of tail and base pages without interfering with concurrent transactions. To formalize our merge process, we prove that merge operates only on stable data while maintaining in-page lineage without any information loss and that the merge does not limit users' transactions to access and/or modify the data that is being merged.

LEMMA 4.1. *Merge operates strictly on stable data.*

PROOF. By construction, we enforced that merge *"always operate on stable data"*. The inputs to the merge process are (1) a set of base pages consisting of committed base records that are read-only, i.e., stable data and (2) a set of consecutive committed tail records in tail pages, thus, also stable data. The output of the merge process is a set of newly merged pages that are read-only, i.e., stable data as well. Hence, the merge process strictly takes as inputs stable data and produces stable data as well. □

LEMMA 4.2. *Merge safely discards outdated base pages without violating any query's snapshot.*

PROOF. In order to support snapshot isolation semantics and time travel queries, we need to ensure that earlier versions of records that participate in the merge process are retained. Since we never perform in-place updates and each update is transformed into appending a new version of the record to tail pages, then as long as tail pages are not removed, we can ensure that we have access to every updated version. But recall that outdated base pages are de-allocated using our proposed epoch-based approach after being merged. Also note that base pages contain the original values of when a record was first created. Therefore, any original values that later were updated must be stored before discarding outdated base pages after a merge is taken place. In another words, we must ensure that outdated base pages are discarded safely.

As a result, the two fundamental criteria, namely, relaxing the merge (i.e. constructing an almost up-to-date snapshot) and operating on stable data, are not sufficient to ensure the *safety property* of the merge. The last missing piece that enables safety of the merge is accomplished by taking a snapshot of the original values when a column is being updated for the first time (as described in Section 3). In other words, we have further strengthened our *data stability* criterion by ensuring even *stability in the committed history*. Hence, outdated base pages can be safely discarded without any information loss, namely, the merge process is safe. □

RID	Indirection	Schema Encoding	Start Time	Last Updated Time	Key	A	B	C
Recently merged records for the key range of k_1 to k_3; TPS = t_7								
b_1	t_8	0000	10:02	10:02	k_1	a_1	b_1	c_1
b_2	t_{12}	0101	13:04	19:25	k_2	a_{22}	b_2	c_{21}
b_3	t_{11}	0001	15:05	19:45	k_3	a_3	b_3	c_{31}
Partitioned tail records for the key range of k_1 to k_3								
t_1	b_2	0100*	13:04		∅	a_2	∅	∅
t_2	t_1	0100	19:21		∅	a_{21}	∅	∅
t_3	t_2	0100	19:24		∅	a_{22}	∅	∅
t_4	t_3	0001*	13:04		∅	∅	∅	c_2
t_5	t_4	0101	19:25		∅	a_{22}	∅	c_{21}
t_6	b_3	0001*	15:05		∅	∅	∅	c_3
t_7	t_6	0001	19:45		∅	∅	∅	c_{31}
t_8	b_1	0000	20:15		∅	∅	∅	∅
t_9	t_5	0010*	13:04		∅	∅	b_2	∅
t_{10}	t_9	0010	21:25		∅	∅	b_{21}	∅
t_{11}	t_7	0001	21:30		∅	∅	∅	c_{32}
t_{12}	t_{10}	0110	21:55		∅	a_{23}	b_{21}	∅

Table 3: An example of the indirection interpretation and lineage tracking (conceptual tabular representation).

THEOREM 4.3. *The merge process and users' transactions do not contend for base and tail pages or the resulting merged pages, namely, the merge process is contention-free.*

PROOF. As part of ensuring contention-free merge, we have already shown that merge operates on stable data (proven by Lemma 4.1) and that there is no information loss as a result of the merge process (proven by Lemma 4.2). Next we prove that the write path of the merge process does not overlap with the write path of users' transactions (i.e., writers). Recall that writers append new uncommitted tail records to tail pages, but as stated before, uncommitted records do not participate in the merge. Writers also perform in-place update of the *Indirection* column within base records to point to the latest version of the updated records in tail pages, but the *Indirection* column is not modified by the merge process. In contrast, the write path of the merge process consists of creating only a new set of read-only merged pages and eventually discarding the outdated base pages safely.

Therefore, we must show that safely discarding base pages does not interfere with users' transactions. In particular, as explained in Lemma 4.2, if the original values were not written to tail records at the time of the update, then during the merge process, we were forced to store them somewhere or encounter information loss. It is not even clear where would be the optimal location for storing the original values. A simple minded approach of just adding them to tail pages would have broken the linear order of changes to records such that the older values would have appeared after the newer values, and it would have interfered with the ongoing update transactions. But, more importantly, the need to store the old values at any location would have implied that during the merge process multiple coordinated actions were required to ensure consistency across modification to isolated locations; hence, breaking the contention-free property of the merge. Therefore, by storing the original updated values at the time of update, we trivially eliminate all the potential contention during the merge process in order to safely discarding outdated base pages.

As a result, users' transactions are completely decoupled from the merge process, and users' transactions and the merge process do not contend over base, tail, or merged pages. □

4.2 Maintaining In-Page Lineage

The lineage of each base page and consequently merged pages is maintained within each page independently as a result of the merge process. In-page lineage information is instrumental to decouple the merge and update operations and to allow independent merging of the different columns of the same record at different points in time. In-page lineage information is captured using a rather simple and elegant concept, which we refer to as *tail-page sequence number (TPS)* in order to keep track of how many updated entries (i.e., tail records) from tail pages have been applied to their corresponding base pages after a completion of a merge.

Original base pages always start with TPS set to 0, a value that is monotonically increasing after every merge. Again to ensure this monotonicty property, as stressed earlier, always a consecutive set of committed tail records are used in the merge process.

TPS is also used to interpret the indirection pointer (also a monotonically increasing value) by readers after the merge is taken place. Consider our running example in Table 2. After the first merge process, the newly merged pages have TPS set to 7, which implies that the first seven updates (tail records t_1 to t_7) in the tail pages have been applied to the merged pages. Consider the record b_2 in the base pages that has an indirection value pointing to t_5 (cf. Table 2), there are two possible interpretations. If the transaction is reading the base pages with TPS set to 0, then the 5^{th} update has not yet reflected on the base page. Otherwise if the transaction is reading the base pages with TPS 7, then the update referenced by indirection value t_5 has already been applied to the base pages as seen in Table 2. Notably, the *Indirection* column is updated only in-place (also a monotonically increasing value) by writers, while merging tail pages does not affect the indirection value.

More importantly, we can leverage the TPS concept to ensure read consistency of users' transactions when the merge is performed lazily and independently for the different columns of the same records. Therefore, when the merge of columns is decoupled, each merge occurs independently and at different points in time. Consequently, not all base pages are brought forward in time simultaneously. Additionally, even if the merge occurs for all columns simultaneously, it is still possible that a reader reads base pages for the column *A* before the merge (or during the merge before the page directory is updated) while the same reader reads the column *C* after the merge; thus, reading a set of inconsistent base and merged pages.

LEMMA 4.4. *An inconsistent read with concurrent merge is always detectable.*

PROOF. Since each base page independently tracks its lineage, i.e., its TPS counter; therefore, TPS can be used to verify the read consistency. In particular, for a range of records, all read base pages must have an identical TPS counter; otherwise, the read will be inconsistent. Hence, an inconsistent read across different columns of the same record is always detectable. □

THEOREM 4.5. *Constructing consistent snapshots with concurrent merge is always possible.*

PROOF. As proved in Lemma 4.4, the read inconsistency is always detectable. Furthermore, once a read inconsistency is encountered, then each page is simply brought to the desired query snapshot independently by examining its TPS and the indirection value and consulting the corresponding tail pages using the logic outlined earlier. Hence, consistent reads by constructing consistent snapshots across different columns of the same record is always possible. □

TPS, or an alternative but similar counter conceptually, could be used as a high-water mark for resetting the cumulative updates as well. Continuing with our running scenario, in which we have the original base pages with the TPS 0 (as shown in Table 2), the merged pages the with TPS 7 (as shown in Table 3). For simplicity, we assume the cumulation was also reset after the 7^{th} tail record. For the record b_2, we see that the indirection pointer is t_{12}, for which we know that the cumulative update has been reset after the 7^{th} update. This means that the tail record t_{12} does not carry updates that were accumulated between tail records 1 to 7. Suppose that the record was updated four times, where the update entries in the tail pages are 3^{rd}, 5^{th}, 10^{th}, and 12^{th} tail records. The tail record t_5 is a cumulative and carries the updated values from the tail record t_3. However, the tail record t_{10} is not cumulative (reset occurred at the 8^{th} update), whereas the tail record t_{12} is cumulative, but carries updates only from the tail record t_{10} and not from t_5 and t_3. Suppose that a transaction is reading the base pages with the TPS 0, then to reconstruct the full version of the record b_2, it must read both the tail records t_5 and t_{12} (while skipping 3^{rd} and 10^{th}). But if a transaction is reading from the merged pages with the TPS 7, then it is sufficient to only read the tail record t_{12} to fully reconstruct the record because the 3^{rd} and 5^{th} updates have already been applied to the merged pages.

4.3 Record Partitioning Trade-offs

When choosing the range of records for partitioning (i.e., update range) there are several dimensions that needs to be examined. An important observation is that regardless of the range size, recent updates to tail pages will be memory resident and no random disk I/O is required. This trend is supported by continued increase in the size of main memory and the fact that the entire OLTP database is expected to fit in the main memory [9, 17].

In our evaluation, we did an in-depth study of the impact of the range size, and we observed that the key deciding factor is the frequency at which the merges are processed. How frequent a merge is initiated is proportional to how many tail records are accumulated before the merge process is triggered. We further experimentally observed that the update range sizes in the order of 2^{12} to 2^{16} exhibit a superior overall performance vs. data fragmentation depending on the workload update distribution. Because for a smaller update range size, we may have many corresponding half-filled tail pages, but as the range size increases, the cost of half-filled tail pages are amortized over a much larger set of records.[8] Furthermore, the range size affects the clustering of updates in tail pages. For larger the range size, it is more likely that cache misses occur when scanning the recent update that are not merged yet. Again, considering that recent cache sizes are in order of tens of megabytes, the choice of any range value between 2^{12} to 2^{16} is further supported. As noted before, one may choose a finer range partitioning for handling updates (i.e., update range), e.g., 2^{12}, to improve locality of access while choosing coarser virtual range sizes when performing merges, essentially forcing the merge to take-in as input a set of consecutive update ranges that have been updated, e.g., choosing 2^4 consecutive 2^{12} ranges in order to merge $2^{12} \times 2^4 = 2^{16}$ records.

For example, suppose the scan operation (even if there are concurrent scans) may access 2 columns, assume each column is 2^3 bytes long. We further assume that the merge can keep up, namely, even for 2^{16} update range size, the number of tail records yet to be merged is less than 2^{16} (as shown in Section 6, such merging rate can be achieved while executing up to 16 concurrent update transactions). The overall scan footprint (combining both base pages and tail pages) is approximately $2^{16} \times 2^3 \times 2 \times 2 = 2^{21}$ (2 MB), which certainly fits in today's processor cache (in our evaluation, we used Intel Xeon E5-2430 processor, which has 15 MB cache size). Thus, even as scanning base records, if one is forced to perform random lookup within a range of 2^{16} tail records, the number of cache misses are limited compared to when the range size was beyond the cache capacity.

Another criteria for selecting an effective update range size is the need for RID allocation. In L-Store, upon the first update to a range of records (e.g., 2^{12} to 2^{16} range), we pre-allocate 2^{12} to 2^{16} unused RIDs for referencing its corresponding tail pages. Tail RIDs are special in a sense they are not added to indexes and no unique constraint is applied on them. Once the tail RID range is

[8] To reduce space under-utilization, tail pages could be smaller than base pages, for instance, tail pages could be 4 KB while base pages are 32 KB or larger.

fully used, then either a new unused RID range is allocated or an existing underutilized tail RID range can be re-assigned (partially used RID range must satisfy TPS monotonicity requirement). Furthermore, in order to avoid overlapping the base and tail RIDs, one could assign tail RIDs in the reverse order starting from 2^{64}; therefore, tail RIDs will be monotonically decreasing, and the TPS logic must be reversed accordingly. The benefit of reverse assignment is that while scanning page directory for base pages, there is no need to first read and later skip tail page entries (read optimization).

5 FAST TRANSACTIONAL CAPABILITIES

In order to support concurrent transactions where each transaction may consists of many statements, any database engine must provide necessary functionalities to ensure the correctness of concurrent reads and writes of the shared data. Furthermore, transaction logging is required in order to recover the system from crash and media failure. In this section, we focus on low-level synchronization protocol and logging requirements. In terms of concurrency protocol for transaction processing, any existing protocols can be leveraged because L-Store primarily focuses on the storage architecture. In particular, we relied on our recently proposed optimistic concurrency model in [32] that supports full ACID properties for multi-statement transactions, and we also employed the speculative reads proposed in [18]. The details of the concurrency protocol is presented in our technical report [31].

Low-level Synchronization Protocol In terms of low-level latching, our lineage-based storage has a set of unique benefits, namely, readers do not have to latch the read-only base pages or fully committed tail pages. Also there is no need to latch partially committed tail pages when accessing committed records. More importantly, writers never modify base pages (except the *Indirection* column) nor the fully committed tail pages, so no latching is required for stable pages. The *Indirection* column is at most 8-byte long; therefore, writers can simply rely on atomic compare-and-swap (CAS) operators to avoid latching the page.

As part of the merge process, no latching of tail and base pages are required because they are not modified. The only latching requirement for the merge is updating the page directory to point to the newly created merged pages. Therefore, every affected page in the page directory are latched one at a time to perform the pointer swap or alternatively atomic CAS operator is employed for each entry (pointer swap) in the page directory. Alternatively, the page directory can be implemented using latch-free index structures such as Bw-Tree [20].

Recovery and Logging Protocol Our lineage-based storage architecture consists of read-only base pages (that are not modified) and append-only updates to tail pages (which are not modified once written). When a record is updated, no logging is required for base pages (because they are read-only), but the modified tail pages requires redo logging. Again, since we eliminate any in-place update for tail pages, no undo log is required. Upon a crash, the redo log for tail pages are replayed, and for any uncommitted transactions (or partial rollback), the tail record is marked as invalid (e.g., tombstone), but the space is not reclaimed until the compression phase.

The one exception to above rule for logging and recovery is the *Indirection* column, which is updated in-place. There are two possible recovery options: (1) one can rely on standard undo-redo log for the *Indirection* column only or (2) one can simply rebuild the *Indirection* column upon crash. The former option can further be optimized based on the realization that tail pages undergo strictly redo policy and aborted transactions do not physically remove the aborted tail records as they are only marked as tombstones. Therefore, it is acceptable for the *Indirection* column to continue pointing to tombstones, and from the tombstones finding the latest committed values. As a result, even for the *Indirection* column only the redo log is necessary. For the latter recovery option, as discussed earlier, to speedup the merge process, we materialize the *Base RID* column in tail records that can be used to populate the *Indirection* column after the crash. Alternatively, even without materializing an additional RID column, one can follow back-pointers in the *Indirection* column of tail records to fetch the base RID because the very first tail record always points back to the original base record.

The merge process is idempotent because it operates strictly on committed data and repeated executions of the merge always produce the exact same results given a set of base pages, their corresponding tail pages, and a merge threshold that dictates how many consecutive committed tail records to be used in the merge process. Therefore, only operational logging is required for the merge process. Also updating the entries in the page directory upon completion of the merge process simply requires standard index logging (both undo-redo logs). If crash occurs during the merge, simply the partial merge results can be ignored and the merge can be restarted.

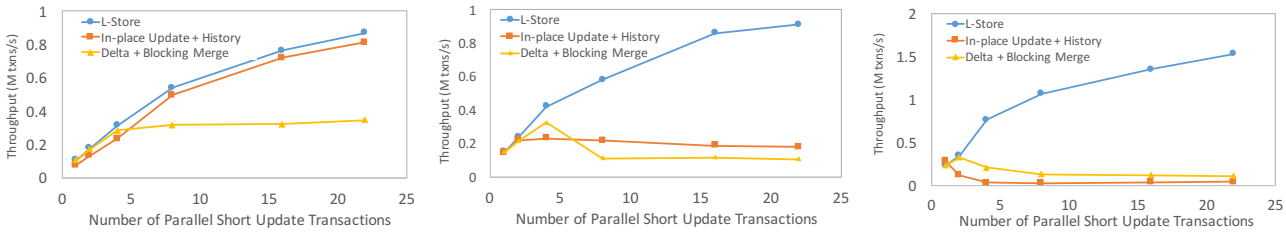
6 EXPERIMENTAL EVALUATION

In order to study the impact of high-throughput transaction processing in the presence of long-running analytical queries, we carried out a comprehensive set of experiments. These experiments were performed using an existing micro benchmark proposed in [18, 32], i.e., a comprehensive transactional YCSB-like benchmark [8], for the sake of a fair comparison and evaluation. This benchmark allows us to study different storage architectures by narrowing down the impact of concurrency with respect to the active data set by adjusting the degree of contention between readers and writers.

6.1 Experimental Setting

We evaluate the performance of various aspects of our real-time OLTP and OLAP system. Our experiments were conducted on a two-socket Intel Xeon E5-2430 @ 2.20 GHz server that has 6 cores per socket with hyper-threading enabled (providing a total of 24 hardware threads). The system has 64 GB of memory and 15 MB of L3 cache per socket. We implemented a complete working prototype of L-Store and compared it against two different techniques, (i) In-place Update + History and (ii) Delta + Blocking Merge, which are described subsequently. The prototype was implemented in Java (using JDK 1.7). Our primary focus here is to simultaneously evaluate read and write throughputs of these systems under various transactional workloads concurrently executed with long-running analytical queries, which is the key characteristic of any real-time OLTP and OLAP system.

Our employed micro benchmark defined in [18, 32] consists of three key types of workloads: (1) low contention, where the active set is 10M records; (2) medium contention, where the active set is 100K records; and (3) high contention, where the active set is 10K records. It is important to note that the data size is not limited to the active set and can be much larger (millions or billions of records). Similar to [18, 32], we consider two classes of transactions. (1) *Read-only transactions* executed under snapshot isolation semantics that scan up to 10% of the data to model TPC-H style analytical queries. (2) *Short update transactions* executed under committed read semantics to model TPC-C and TPC-E transactions, in which each *short update transaction* consists of 8 read and 2 write statements over a table schema with 10 columns. In addition, we vary the ratio of read/writes in these update transactions to model different customer scenarios with different read/write degrees. By default, transactional throughput of these schemes are evaluated while running (at least) one scan thread and one merge thread to create the real-time OLTP



(a) Low Contention. (b) Medium Contention (c) High Contention

Figure 5: Scalability under varying contention level.

and OLAP scenario. Unless stated explicitly, the percentage of reads and writes in the transactional workload is fixed at 80% and 20%, respectively. On average 40% of all columns are updated by the writers. Lastly, the page size is set to 32 KB for both base and tail pages because a larger page size often results in a higher compression ratio suitable for analytical workloads [13].

Next we describe the two techniques that are compared with L-Store. We point out the primary features of these techniques and describe it with respect to L-Store. For fairness, across all techniques, we have maintained columnar storage, maintained a single primary index for fast point lookup, and employed the embedded-indirection column to efficiently access the older/newer versions of the records. Additionally, logging has been turned off for all systems as logging could easily become the main bottleneck (unless sophisticated logging mechanisms such as group commits and/or enterprise-grade SSDs are employed). In the In-place Update + History technique, we are required to write both redo-undo logs for all updates while for L-Store and Delta + Blocking Merge only redo log is needed due to their append-only scheme.

In-place Update + History (IUH): A prominent storage organization is to append old versions of records to a history table and only retain the most recent version in the main table, updating it in-place. Such table organization is motivated by commercial systems such as [26]; thus, our In-place Update + History is inspired by such table organization that avoids having multiple copies and representations of the data. However, due to the nature of the in-place update approach, each page requires standard shared and exclusive latches that are often found in major commercial big data systems. In addition to the page latching requirement, if a transaction aborts, then the update to the page in the main table is undone, and the previous record is restored. Scans are performed by constructing a consistent snapshots, namely, if records in the main table are invisible with respect to query's read time, then the older versions of the records are fetched from the history table by following the indirection column. In our implementation of In-place Update + History, we also ignored other major costs of in-place update over the compressed data, in which the new value may not fit in-place due to compression and requires costly page splits or shifting data within the page as part of update transactions. We further optimized the history table to include only the updated columns as opposed to inserting all columns naively.

Delta + Blocking Merge (DBM): This technique is inspired by [14], where it consists of a main store and a delta store, and undergoes a periodic merging and consolidation of the main and delta stores. However, the periodic merging requires the draining of all active transactions before the merge begins and after the merge ends. Although the resulting contention of the merge appears to be limited to only the boundary of the merge for a short duration, the number of merges and the frequency at which this merge occurs has a substantial impact on the overall performance. We optimized the delta store implementation to be columnar and included only the updated columns [27]. Additionally, we applied our range partitioning scheme to the delta store by dedicating a separate delta store for each range of records to further reduce the cost of merge operation in presence of data skew. The partitioning allow us to avoid reading and writing the unchanged portion of the main store.

	L-Store	IUH	DBM
Scan Performance (in secs.)	0.24	0.28	0.38

Table 4: Scan performance for different systems.

6.2 Experimental Results

In what follows, we present our comprehensive evaluation results in order to compare and study our proposed L-Store with respect to state-of-the-art approaches.

Scalability under contention: In this experiment, we show how transaction throughput scales as we increase the number of update transactions, in which each update transaction is assigned to one thread. For the scalability experiment, we fix the number of reads to 8 and writes to 2 for each transaction against a table with active set of $N = 10$ million rows. Figure 5(a) plots the transaction throughput (y-axis) and the number of update threads (x-axis). Under low contention, the throughput for L-Store and In-place Update + History scales almost linearly before data is spread across the two NUMA nodes. The Delta + Blocking Merge approach however does not scale beyond a small number of threads due to the draining of active transaction before/after of each merge process, which brings down the transaction throughput noticeably. With increasing number of threads, the number of merges and the draining of active transactions become more frequent, which reduces the transaction throughput significantly. The In-place Update + History approach has lower throughput compared to L-Store due to the exclusive latches held for data pages that block the readers attempting to read from the same pages. The presence of a single history table also results in reduced locality for reads and more cache misses.

In addition, we study the impact of increasing the degree of contention by varying the size of the active set. For a fixed degree of contention, we vary the number of parallel update transactions from 1 to 22. For both medium contention (Figure 5(b)) and high contention (Figure 5(c)), we observe that L-Store consistently outperforms the In-place Update + History and Delta + Blocking Merge techniques as the number of parallel transactions is increased. For medium contention, we observed a speedup of up to 5.09× compared to the In-place Update + History technique and up to 8.54× compared to the Delta + Blocking Merge technique. Similarly for high contention, we observed up to 40.56× and 14.51× speedup with respect to the In-place Update + History and Delta + Blocking Merge techniques, respectively. The greater performance gap is attributed to the fact that in In-place Update + History, latching contention on the page is increased that is altogether eliminated in L-Store. In Delta + Blocking Merge, since the active set is smaller, and all updates are concentrated to smaller regions, the merging frequency is increased, which proportionally reduces the overall throughput due to the constant draining of all active transactions. Finally, due to the smaller active set sizes in the medium- and high-contention workloads, the cache misses are also reduced as the cache-hit ratio increases. As a consequence, the transaction throughput also increases proportionately.

Scan Scalability: Scan performance is an important metric for real-time OLTP and OLAP systems because it is the basic building block for assembling complex ad-hoc queries. We measure the

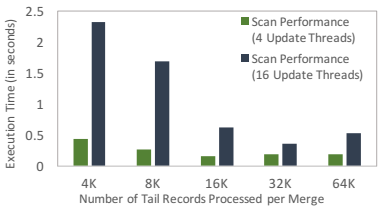
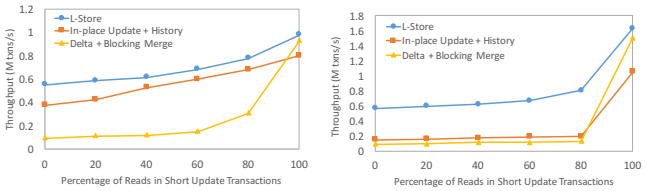


Figure 6: Scan performance.



(a) Low Contention (b) Medium Contention

Figure 7: Impact of varying the read/write ratio of short update transactions.

scan performance of L-Store by computing the SUM aggregation on a column that is continuously been updated by the concurrent update transactions. Thus, the goal of this experiment is to determine whether the merge can keep up with high-throughput OLTP workloads. As such, this scenario captures the worst-case scan performance because it may be necessary for the scan thread to search for the latest values in the merged page or tail pages when the merge cannot cope with the update throughput. For columns which do not get updated, the latest values are available in the base page itself, as described before. In this experiment (Figure 6), we study the single-threaded scan performance with one dedicated merge thread. We vary the number of tail records (M) that are processed per merge (x-axis) and observe the corresponding scan execution time (y-axis) while keeping the range partitioning fixed at $64K$ records. We repeat this experiment by fixing the number of update threads to 4 and 16, respectively. In general, we observe that as we increase M, the scan execution time decreases. The main reasoning behind this observation is that the scan thread visits tail pages for the latest values less often because the merge is able to keep up. However, for the smaller values of M, the merge is triggered more frequently and cannot be sustained. Additionally, the overall cost of the merge is increased because the cost of merge is amortized over fewer tail records while still reading the entire range of $64K$ base records. Notably, if we delay the merge by accumulating too many tail records, then there is slight deterioration in performance. Therefore, it is important to balance the merge frequency vs. the amortization cost of the merge for the optimal performance, which based on our evaluation, it is when M is set to around 50% of the range size.

We also compare the single-threaded scan performance (for low contention and $4K$ range size) of L-Store with the other two techniques in the presence of 16 concurrent update threads (as shown in Table 4). Our technique outperforms the In-place Update + History and Delta + Blocking Merge techniques by 14.28% and 36.84%, respectively. It is important to note that smaller update range sizes, namely, assigning separate tail pages for each $4K$ base records instead of $64K$ base records, increases the overall scan performance by improving the locality of access within tail pages. Therefore, as elaborated previously in Section 4.3, it is beneficial to apply (virtual) fine-grained partitioning over base records (e.g., $4K$ records) to handle updates in order to improve locality of access within tail pages while applying (virtual) coarser-grained partitioning (e.g., $64K$ records) when performing the merge in order to reduce the space fragmentation in the resulting merged pages.

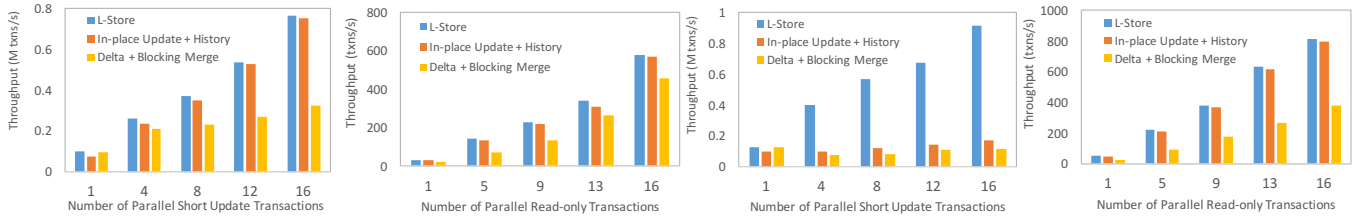
Impact of varying the workload read/write ratio: Short update transactions update only a few records. A typical transactional workload comprises of 80% read statements and 20% writes [18]. However, our goal is to explore the entire spectrum from a read-intensive workload (read/write ratio 10:0) to a write-intensive workload (read/write ratio 0:10) while fixing the number of update threads to 16. Figure 7(a) shows transaction throughput (y-axis) as the ratio of read-only transactions varies in the workload (x-axis) with low contention. As expected, the performance of all the schemes increases as we increase the ratio of reads in the transactions because contention is a function of writes. As we have more writes in the workload, In-place Update + History technique suffers from increased contention as acquiring read latches conflict with the exclusive latches resulting in an extended wait time. The performance of the Delta + Blocking Merge technique also exacerbates since increasing the number of writes increases the number of merges performed. This brings down the performance further due to frequent halt of the system while draining active transactions. However, note that the gap between all of the schemes is the least when the workload consists of 100% reads. In summary, the speedup obtained with respect to In-place Update + History is up to 1.45× and up to 5.78× with respect to Delta + Blocking Merge technique. Note, even for 100% read, In-place Update + History continues to pay the cost of acquiring read latches on each page.

We repeat the same experiment but restrict the active set size to 100K rows (Figure 7(b)). L-Store significantly outperforms the other techniques across all workloads while varying the read/write ratio. But the performance gap is similar with respect to the low contention scenario when there are no update statements in the workload. The speedup obtained compared to In-place Update + History and Delta + Blocking Merge techniques is up to 4.19× and up to 6.34× respectively.

Impacts of long-read transactions: As mentioned previously, it is not uncommon to have long-running read-only transactions in real-time OLTP and OLAP systems. These analytical queries touch a substantial part of the data compared to the short update transactions, and the main goal is to reduce the interference between OLTP and OLAP workloads. In this experiment, we investigate the performance of the different schemes in the presence of these long-running read-only transactions, which on an average touch 10% of the base table. We fix the number of concurrent active transactions to 17 while increasing the number of concurrent read-only transactions from 1 to 16 (the short transactions simultaneously vary from 16 to 1). We also allocated a single merge thread for L-Store and Delta + Blocking Merge. Figures 8(a)-8(b) represent the scenario for a low contention workload, while Figures 8(c)-8(d) represent the scenario for medium contention.

We observe that for both low and medium contention, there is an increase in throughput for both long-read transactions and short update transactions when the number of threads are increased. Moreover, the performance of read-only transaction increases for the medium contention scenario for all the techniques as the updates are restricted to a small portion of the data resulting in a higher read throughput. In other words, majority of the read-only transactions touch portions of the data in which updates do not take place resulting in higher throughput. For read-only transactions, our technique outperforms Delta + Blocking Merge up to 1.97× and 2.37× for low and medium contention workloads, respectively. For short update transactions, we outperform In-place Update + History and Delta + Blocking Merge by at most 5.37× and 7.91×, respectively, for medium-contention workload. In the earlier experiments, we had demonstrated that L-Store outperforms other leading approaches for update-intensive workloads, and in this experiment, we further strengthen our claim that L-Store substantially outperforms the leading approaches in the mixed OLTP and OLAP workload as well, the latter is due to our novel contention-free merging that does not interfere with the OLTP portion of the workload.

Impacts of comparing row vs. columnar layouts We revisit the scan and point query performance while considering both row



(a) Update throughput with long-read transactions (Low Cont.).

(b) Read throughput with short update transactions (Low Cont.).

(c) Update throughput with long-read transactions (Med Cont.).

(d) Read throughput with short update transactions (Med Cont.).

Figure 8: Impact of varying the number of short update vs. long read-only transactions.

	L-Store (Column)	L-Store (Row)
Scan Performance without updates (in secs.)	0.043	0.196
Scan Performance with updates (in secs.)	0.24	0.66

Table 5: Scan performance based on row vs. columnar layouts.

	10% of Columns	20%	40%	80%	All Columns
L-Store (Column)	1.46	1.35	1.17	1.08	0.98
L-Store (Row)	1.45	1.45	1.45	1.45	1.45

Table 6: Point query performance vs. percentage of columns read (M txns/second).

	Lineage-based	Log-structured
Update throughput (16 threads) with 1 scan thread	0.76M txns/sec	0.15M txns/sec
Scan performance (in sec) with update txns (16 threads)	0.24	0.63

Table 7: Update/Scan performance of lineage-based vs. log-structured storage architecture.

and columnar storage layouts. To enable this comparison, we additionally developed a variation of our L-Store prototype using row-wise storage layout, which we refer to as L-Store (Row).[9] In particular, we compared the single-threaded scan performance (for low contention and $4K$ range size) of L-Store using both row and columnar layouts in the presence of when there is no updates or when there are 16 concurrent update threads (as shown in Table 5). As expected, the scan performance of L-Store (Column) is substantially higher than L-Store (Row) by a factor of 2.75× and 4.56×, with and without updates, respectively. Also note that we did not enabled column compression for L-Store (Column), otherwise even a higher performance gap would be observed because in column stores, an average of 10× compression is commonly expected [5, 36].

We further conducted an experiment with only point queries (on a table with 10 columns), where each transaction now consists of 10 read statements, and each read statement may read 10% to 100% of all columns (as shown in Table 6). As expected, the performance of any column store is deteriorated as more columns are fetched. When reading only 10-20% of columns, L-Store (Column) exhibit a comparable throughout as L-Store (Row); however, as we increase the number of fetched columns, the throughput is decreased. But, even in the worst case when all columns are fetched, the throughout only drops by 33%. However, the prevalent observation is that rarely all columns are read or updated in either OLTP or OLAP workloads [1, 5, 36]; thus, given the substantial performance benefit of columnar layout for predominant workloads, then it is justified to expect a slight throughput decrease in rare cases of when point queries are forced to access all columns.

Impacts of comparing lineage-based vs. log-structured storage architecture For completeness, in our prototype, we also implemented log-structured merge-tree (LSM) [25] storage architecture that is predominant in the distributed key-value stores. In particular, we have based our implementation on LevelDB [21]. In this experiment, we studied the single-threaded scan performance (for low contention) of L-Store (i.e., lineage-based storage architecture) and LSM while having 16 concurrent update threads (as shown in Table 7). As expected, due to the multi-layered structured of LSM, the fine-grained read/write access and scan performance of LSM are substantially lower than L-Store. As a result, L-Store outperforms LSM on update throughput and scan performance by a factor of 5× and 2.6×, respectively.

[9]Notably our proposed lineage-based storage architecture is not limited to any particular data layout; in fact, our technique can be employed even for non-relational data such as document or graph data.

7 RELATED WORK

In recent years, we have witnessed the development of many in-memory engines optimized for OLTP workloads either as research prototypes such as HyPer [13, 24], ES2 [6], and ExpoDB [11] or for commercial use such as Microsoft Hekaton [9], Oracle In-Memory [15], VoltDB [38], and HANA [14, 27]. Most of these systems are designed to keep the data in row format and in the main memory to increase the OLTP performance. In contrast, to optimize the OLAP workloads, columnar format is preferred. The early examples of these engines are C-Store [36] and MonetDB [5]. Recently, major big data vendors also started integrating columnar storage format into their existing engines. SAP HANA [10] is designed to handle both OLTP and OLAP workloads by supporting in-memory columnar format. IBM DB2 BLU [29] introduces a novel columnar OLAP engine that is memory-optimized and substantially improves the execution of complex analytical workloads by operating directly on compressed data. In what follows, we shift our focus to the recent developments that aim to bring both OLTP and OLAP capabilities into the same platform.

HyPer, a powerful main-memory system, guarantees the ACID properties of OLTP transactions and supports running OLAP queries on consistent snapshot [13]. The design of HyPer leverages a novel OS-processor-controlled lazy copy-on-write mechanism enabling to create a consistent virtual memory snapshot. Unlike L-Store, HyPer resorts to running transactions serially when the workload is not partitionable. Notably, HyPer recently employed multi-version concurrency to close this gap [24]. IBM Wildfire is a variant of DB2 BLU [29] that is integrated into Apache Spark to support fast ingest by adopting the relaxed last-writer-wins semantics and offers an efficient snapshot isolation on recent, but stale, data by relying on periodic shipment and writing of the logs onto a distributed file system [4]. In the same spirit, BatchDB is based on primary-secondary replication design to efficiently isolate OLTP and OLAP workloads by relying on batch migration of recent updates and executing OLAP queries over recent (but possibly stale) snapshots [23]. The unified storage architecture in L-Store eliminates the need for classical log shipment design and does not restrict reads to stale snapshots. Elastic power-aware data-intensive cloud computing platform (epiC) was designed to provide scalable big data services on cloud [6]. epiC is designed to handle both OLTP and OLAP workloads [7]. However, unlike L-Store, the OLTP queries in ES2 are limited to basic get, put, and delete requests (without multi-statements transactional support). Furthermore, in ES2, it is possible that snapshot consistency is violated and the user is notified subsequently [6].

Microsoft SQL Server currently consists of three unique engines: the classical SQL Server engine designed to process disk-based tables in row format, the Apollo engine designed to maintain the data in columnar format that offers significant performance gain for OLAP workloads [19], and the completely redesigned Hekaton in-memory engine designed to excel at OLTP workloads [9, 17]. Noteworthy, Microsoft has also recently announced

moving towards supporting real-time OLTP and OLAP capabilities [17], which further reinforces our position to support real-time analytics. To support OLTP and OLAP among loosely integrated engines, an intricate foreground routine is proposed to enable a continuous data migration from Hekaton (a row-based engine) into Apollo (a columnar engine) [17]. In contrast, in L-Store, we rely on a single unified columnar engine (without the need for maintaining multiple copies of the data) and, more importantly, our consolidation is based on a novel contention-free merge process that is performed asynchronously and completely in the background, and the only foreground task is pointer swaps in the page directory for pointing to the newly created merged pages.

Oracle offers a novel dual-format option to support real-time OLTP and OLAP, where data resides in both columnar and row formats [15]. To avoid maintaining two identical copies of data in both columnar and row format, an effective "layout transparency" abstraction was introduced that maps data into a set of disjoint tiles (driven by the query workload and the age of data), where a tile could be stored in either columnar or row format [3]. The key advantage of the layout-transparent mapping is that the query execution runtime operates on the abstract representation (layout independent) without the need to create two different sets of operators for processing the column- and row-oriented data. In the same spirit, SnappyData proposed a unified runtime engine to combine streaming, transaction, and analytical processing, but from the storage perspective, it maintains recent transactional data in row format while it ages data to a columnar format for analytical processing [30]. SnappyData employed data aging strategies similar to the original version of SAP HANA [35].

Contrary to the aforementioned efforts, in L-Store, we strictly keep only one copy and one representation of data; thus, fundamentally eliminating the need to maintain layout-independent mapping abstraction and storing data in both columnar and row formats. HANA [14, 27] also strives to achieve real-time OLTP and OLAP engine. Most notably, we share the same philosophy governing HANA that aims to develop a generalized solution for unifying OLTP and OLAP as opposed to building specialized engines. But what distinguishes our architecture from HANA is that we propose a unified columnar storage without the need to distinguishing between a main store and a delta store. We further propose a contention-free merge process, whereas in [14], the merge process is forced to drain all active transactions at the beginning and end of the merge process, a contention that results in a noticeable slow down as demonstrated in our evaluation.

8 CONCLUSIONS

We develop Lineage-based Data Store (L-Store) to realize real-time OLTP and OLAP processing within a single unified engine. The key features of L-Store can succinctly be summarized as follows. Recent updates for a range of records are strictly appended and clustered in its corresponding tail pages to eliminate read/write contention, which essentially transforms costly point updates into an amortized, fast analytical-like update query.s L-Store achieves (at most) 2-hop access to the latest version of any record through an effective embedded indirection layer. We introduce a novel contention-free and relaxed merging of only stable data in order to lazily and independently bring base pages (almost) up-to-date without blocking on-going and new transactions. Every base page relies on independently tracking the lineage information in order to eliminate all coordination and recovery even when merging different columns of the same record independently. Lastly, a novel contention-free page de-allocation using epoch-based approach is introduced without interfering with ongoing transactions. We demonstrate that L-Store outperforms In-place Update + History by factor of up to 5.37× for short update transactions while achieving slightly improved performance for scans. It also outperforms Delta + Blocking Merge by 7.91× for short update transactions and up to 2.37× for long-read analytical queries.

9 ACKNOWLEDGMENTS

We wish to thank C. Mohan, K. Ross, V. Raman, R. Barber, R. Sidle, A. Storm, X. Xue, I. Pandis, Y. Chang, and G. M. Lohman for many insightful discussions and invaluable feedback in the earlier stages of this work.

REFERENCES

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB '01*.
[2] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: a hands-free adaptive store. In SIGMOD'14.
[3] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD'16*.
[4] Ronald Barber, et al. Evolving Databases for New-Gen Big Data Applications. In *CIDR'17*.
[5] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR'05*.
[6] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. 2011. ES2: A Cloud Data Storage System for Supporting Both OLTP and OLAP. In *(ICDE'11)*.
[7] Chun Chen, Gang Chen, Dawei Jiang, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Providing scalable database services on the cloud. In *WISE'10*.
[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC'10*.
[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD'13*.
[10] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.'12*
[11] Suyash Gupta and Mohammad Sadoghi. EasyCommit: A Non-blocking Two-phase Commit Protocol. In *EDBT'18*.
[12] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB'08*.
[13] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *(ICDE'11)*.
[14] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-optimized Databases Using Multi-core CPUs. *PVLDB'11*
[15] T. Lahiri, et al. Oracle Database In-Memory: A dual format in-memory database. In *ICDE'15*.
[16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD'16*.
[17] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time Analytical Processing with SQL Server. *PVLDB'15*.
[18] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB'11*.
[19] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. SQL Server Column Store Indexes. In *SIGMOD'11*.
[20] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE'13*.
[21] LevelDB http://leveldb.org/
[22] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Eng. Bull.'13*.
[23] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD '17*.
[24] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD '15*.
[25] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.'96*.
[26] Oracle Total Recall/Flashback Data Archive.
[27] Hasso Plattner. The Impact of Columnar In-memory Databases on Enterprise Systems: Implications of Eliminating Transaction-maintained Aggregates. *PVLDB'14*
[28] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *PVLDB'12*.
[29] Vijayshankar Raman, et al. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB'13*
[30] Jags Ramnarayan, Sudhir Menon, Sumedh Wale, and Hemant Bhanawat. SnappyData: A Hybrid System for Transactions, Analytics, and Streaming: Demo. In *DEBS'16*.
[31] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-Store: A Real-time OLTP and OLAP System. *CoRR'16* abs/1601.04084.
[32] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. Reducing Database Locking Contention Through Multi-version Concurrency. *PVLDB'14*.
[33] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2013. Making Updates disk-I/O Friendly Using SSDs. *PVLDB'13*.
[34] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. Exploiting SSDs in operational multiversion databases. *VLDBJ'16*.
[35] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD'12*.
[36] Mike Stonebraker, et al. C-Store: A Column-oriented DBMS. In *VLDB'05*.
[37] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE'05*.
[38] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.'13*