Rethinking Concurrency Control for In-Memory OLAP DBMSs

Pedro Pedreira, Yinghai Lu, Sergey Pershin, Amit Dutta, Chris Croswhite

Facebook Inc.
1 Hacker Way
Menlo Park, CA, USA
{pedroerp,yinghai,spershin,adutta,chrisc}@fb.com

***Abstract*—Although OLTP and OLAP database systems have fundamentally disparate architectures, most research work on concurrency control is geared towards transactional systems and simply adopted by OLAP DBMSs. In this paper we describe a new concurrency control protocol specifically designed for analytical DBMSs that can provide Snapshot Isolation for distributed in-memory OLAP database systems, called *Append-Only Snapshot Isolation* (AOSI). Unlike previous work, which are either based on multiversion concurrency control (MVCC) or *Two Phase Locking* (2PL), AOSI is completely lock-free and always maintains a single version of each data item. In addition, it removes the need for per-record timestamps of traditional MVCC implementations and thus considerably reduces the memory overhead incurred by concurrency control. In order to support these characteristics, the protocol sacrifices flexibility and removes support for a few operations, particularly record updates and single record deletions; however, we argue that even though these operations are essential in a pure transactional system, they are not strictly required in most analytic pipelines and OLAP systems. We also present an experimental evaluation of AOSI's current implementation within the Cubrick in-memory OLAP DBMS at Facebook, and show that lock-free single-version Snapshot Isolation can be achieved with low memory overhead and minor impact in query latency.**

I. Introduction

Traditional concurrency control protocols can be organized into two broad categories. The first comprises protocols that use locking to preserve mutual exclusion between conflicting transactions, leveraging *Two Phase Locking* (2PL) or one of its variations. The second, protocols that assign a timestamp to each transaction and begin execution immediately, checking either at execution or commit time for conflicting operations. Most timestamp-based implementations use Multiversion Concurrency Control (MVCC), in a way to generate new versions of data items for each update, thus allowing concurrent *readers* and *writers* to proceed in parallel. Today, the vast majority of commercial DBMSs are either based on MVCC with timestamps, 2PL or some combination of both. [1] [2] [3] [4].

Although widely used in practice, MVCC incurs significant memory overhead in order to store multiple versions of data items, in addition to the required timestamps per record version to control which transactions are allowed to see which versions. This overhead is sometimes tolerable for transactional systems where the working sets are smaller, but for in-memory OLAP DBMSs where the system tries to fit in memory as much of the dataset as possible, this overhead can be prohibitive. Moreover, despite having fundamentally different characteristics, most research work on concurrency control is geared towards transactional systems and simply adopted by OLAP DBMSs without considering that OLAP transactions might have different characteristics and requirements.

We argue in this paper that not all low-level operations supported by OLTP are strictly required by OLAP systems, and if one is willing to sacrifice flexibility, a more efficient concurrency control protocol can be designed while still providing the same level of guarantees and adequate functionality. Particularly, we show that if the system drops support for record updates and single record deletes, it is possible to provide Snapshot Isolation (SI) in a timestamp based protocol in a completely lock-free manner, without using MVCC or maintaining per record timestamps.

In this paper we also present a new concurrency control protocol specially designed for in-memory distributed OLAP databases called *Append-Only Snapshot Isolation* (AOSI). The presented protocol is compatible to any column-oriented DBMS implementation based on vectors, in addition to supporting systems where data is further horizontally partitioned. AOSI is based on timestamps and maintains only one version of each data item at all times since there is no support for record updates. Query performance is also improved since it is guaranteed that records are always stored contiguously in memory, thus removing many branches and cache misses caused by having to find the correct data item version.

AOSI enforces isolation by maintaining an auxiliary vector per partition to control the ranges of records inserted by each transaction, while deletes are implemented by inserting a special marker on this vector. Since it is common that only a small fraction of OLAP datasets change on a regular basis, the size of this vector is usually small as compared to the full dataset; in addition, these entries can be frequently recycled. AOSI also leverages Logical Clocks to locally assign timestamps to globally consistent distributed transactions, without requiring additional synchronization other than the communication needed for the execution of operations.

Finally, we present an experimental evaluation of AOSI

in the context of the Cubrick in-memory distributed OLAP DBMS developed at Facebook, described in a previous work [5].

This paper makes the following contributions:

- We present a discussion about the differences between transaction requirements in both analytical and transactional systems, and how some operations commonly found in OLTP can be relaxed in an OLAP system without sacrificing its ability to support most analytical data pipelines.
- We describe AOSI, a new concurrency control protocol able to provide Snapshot Isolation (SI) in column-oriented OLAP systems without leveraging locking or MVCC, in addition to eliminating the need to store timestamps per record.
- We detail how AOSI leverages Logical Clocks in order to locally assign timestamps to globally consistent distributed transactions without requiring extra synchronization.
- We present an experimental evaluation of our AOSI's implementation in the context of the Cubrick OLAP DBMS running on production workloads at Facebook.

The remaining of this paper is organized as follows. Section II discusses requirements for analytical transactions and the minimum set of operations needed to build a full-featured and scalable OLAP system. Section III presents the AOSI protocol, describing how timestamps are assigned and exemplifying each supported operation. Section IV discusses how timestamps are handled in the context of a distributed cluster and how logical clocks are leveraged. Section V outlines Cubrick's data model and internal data structures, while Section VI presents an experimental evaluation of our current implementation. Finally, Section VII points to related work and Section VIII concludes the paper.

II. OLAP Transaction Requirements

Database transaction processing and concurrency control is a topic being studied since the early days of database systems, where DBMSs were uniquely focused on online transaction processing (OLTP) and concurrency control was mostly done through locking and the use protocols such as 2PL. About two decades ago, a new movement led by R. Kimball [6] emphasized the need for systems focused on the execution of less complex queries over a larger volume of data, mostly focused on analytics, Business Intelligence (BI) workloads and reporting. In the following years, several commercial offerings started being released [4] [7] due to a large portion of the database market adopting the data warehouse model.

From the orientation in which data is stored (row-wise vs. column-wise) to data encoding, compression and ultimately low level query optimization details, the architecture of these two types of systems is drastically different, even though most OLAP's concurrency control subsystems were inherited from OLTP systems and left unchanged. In order to decide whether to reuse or propose a new concurrency control mechanism, one must first understand the requirements for transactions in an OLAP system.

Beyond the well understood differences between transaction life time, amount of data scanned and transactional throughput, we advocate in this paper that a few operations supported by OLTP systems are not strictly required in OLAP systems, namely *record updates* and *single record deletion*, and could be sacrificed for performance. These operations are discussed in the following subsections.

A. No Record Updates

Handling conflicting updates is a challenging task in concurrency control protocols. Traditional pessimistic approaches lock the data item being updated in such a way to stall and serialize all subsequent accesses, thus sacrificing performance and causing data contention. An alternative protocol based on multiversioning creates a new version of a data item once it is updated, hence preventing writes from blocking readers. However, the system needs to keep track of the data items updated by each transaction in order to detect write set conflicts, and rollback one of the conflicting transactions (optimistic approach), as well as implement some form of garbage collection for older versions. Another approach is to use multiversion concurrency control in conjunction with locks so that writers only block other writers, but never readers [8].

Regardless of the approach, the database system must either (a) lock the data item or (b) keep additional metadata per record informing which transaction last updated a data item. Neither approach is ideal in an in-memory OLAP database: (a) introduces unnecessary contention and is particularly inadequate for in-memory systems that are supposed to deliver high transactional throughput and very low latency; (b) can be implemented in a lock-less manner, but increases the memory footprint since the system needs to maintain one transaction id or timestamp per data item. In fact, in some cases the timestamp overhead can even double the memory requirements of a dataset (if the table has only a few columns, for example).

In the next two subsections, we argue that record updates are not strictly necessary for most OLAP ETL workflows and can be sacrificed by a database system in favor of performance.

1) Changing Facts: Data warehouses are commonly modeled following a dimensional data model and hence composed of *fact* and *dimension* tables. ETL pipelines are the periodic workflows responsible for scrapping data out of transactional (OLTP) systems, transform, clean and load it into the data warehouse, thereby updating both fact and dimension tables. Facts, by their very nature, are *immutable*. Still, we acknowledge a few cases where users would like to be able to update a particular fact stored in the data warehouse. They fall into one of the two categories: (a) the fact itself changed, such as a post previously made by a user being removed or an order being canceled; (b) measurement or ETL pipeline error, such as data collection errors, data cleaning issues, business logic bugs or data corruption.

In the first situation (a), an update to a fact is in reality a new fact, and should be modeled and recorded as such. Treating a

fact update as a new fact instead of updating records in-place gives the data warehouse model more context and makes it able to answer a broader type of queries, still not requiring the database system to support individual record updates. In the second category (b), if an error on a specific fact was detected, the entire ETL workflow must be executed again after the root cause was fixed. In practice, updating individual records in-place in a data warehouse is an inefficient and error prone process. Maintaining a single idempotent ETL workflow that can be re-run whenever errors are found is a better engineering practice overall, since it reduces the amount of code to be written, tested and maintained, as well as being easier to monitor than having specific ad-hoc workflows to fix data issues.

2) Changing Dimensions: Another possible requirement for record updates is changing dimension tables. Changes to dimension tables are required, for instance, whenever a user's marital status changes. In [6], Kimball defines eight different strategies to deal with changes in dimension tables in a data warehouse, the so-called *slowly changing dimensions*. From adding a new row or column that holds the updated value to further normalizing the history of values in a new *mini-dimension* table, they all require a burdensome workflow of comparing values item by item and resolving updates. This is an onerous operation and impractical at scale, considering that dimension tables alone collected by nowadays companies may contain over a few billion records and many updates per day.

One approach that trades ETL workflow complexity for storage space is taking a full snapshot of the dimensions tables every time the ETL workflow is executed, instead of comparing each single record and running in place updates. This strategy is similar to *Type 1 - Overwrite* as defined by Kimball, which basically overwrites the previous value, but with the difference that a whole new copy (a new partition) of the table is created and that older partitions are kept for history. On the downside, this approach requires more logic when querying in order to join the correct partition, and can only keep history up to the dimension table's snapshot retention policy. However, this approach makes the ETL significantly simpler and more efficient and eliminates the needs for burdensome in-place updates.

B. No Record Deletes

Handling deletion in a concurrency control protocol is analogous to handling updates. A concurrent read transaction must always see the previous (the record value before deletion) or the updated value (the fact that the record has been deleted), as well as concurrent write transactions with overlapping write sets (two transactions deleting the same records) may conflict.

In order to support this operation, one timestamp recording the transaction in which a particular record has been deleted must be associated to each data record (*deleted_at*). Furthermore, based on this timestamp it is possible to control, for instance, which read transactions are supposed to see this data item, as well as rollback a transaction trying to delete an item that has been deleted by a previous transaction. In practice, many modern database systems associate two timestamps to each record, controlling both record creation and deletion timestamps, and based on this range control which records each transaction is supposed to see.

In its simpler form, this strategy requires additional 16 bytes per record if no special packing is used, assuming two 64 bit timestamps. Considering a dataset containing 10 billion records (which is commonly found nowadays), this approach incurs on 160 GB of memory footprint overhead. In addition, the fact that the write set of an OLAP database is usually small if compared to the amount of historical data stored (usually only the newest partition being processed and the one in the tail of the retention window being deleted) makes a great part of these timestamps unnecessary.

The same arguments presented in Section II-A for in-place updates are valid for single record delete: fact deletions must be recorded as new facts, and deleted dimension items will be skipped on the next snapshot. In our view, the only valid use cases for deletes in an OLAP system is deleting (or dropping) an entire partition, for instance, when a data quality problem was found or to enforce retention policies. In these cases, one would only need one timestamp per partition, instead of one timestamp per record, to record the transaction in which the partition was deleted.

III. AOSI

Append-Only Snapshot Isolation (AOSI) is a distributed concurrency control protocol specially designed for in-memory OLAP DBMSs. AOSI provides Snapshot Isolation (SI) guarantees to column-oriented DBMSs on a completely lock-free manner and without using MVCC — or always keeping a single version of each data item. In order to achieve this, AOSI drops support for two operations, namely *record updates* and *single record deletions*.

AOSI is an optimistic concurrency control protocol based on timestamps, and thus assigns monotonically increasing timestamps to transactions. Transactions are always free to execute without any type of database locks. If record updates were to be supported on a lock-free system, the DBMS would be required to store multiple versions of updated data items (MVCC), and select the correct version to read based on the transaction's timestamp. Similarly, if single record deletes were to be supported, one *deleted_at* timestamp would have to be maintained per deleted record in order to control which transactions are supposed to see the delete. AOSI makes the design choice of dropping support for these two operations and keep the protocol simple and memory efficient. We argue that although these are valid concerns on transactional systems, they are not strictly required on most OLAP workflows, as discussed in Section II.

The protocol assumes that the underlying database engine is column-oriented and that each attribute of a record is stored on a separate vector-like structure. Moreover, AOSI also assumes that records are appended to these vectors in an unordered

and append-only manner, and that records can be materialized[1] by using the implicit ids on these vectors. The protocol also supports further horizontal partitioning of these records by the use of some function $f(x)$ that can map records to partitions, where f can be based on any criteria such as ranges, lists or hashes, for instance.

In the remaining of this section we describe the different transaction types (Subsection III-A), the timestamps maintained and how they are assigned to transactions (Subsection III-B) and how low-level database operations are executed (Subsection III-C).

A. Transactions

AOSI transactions are timestamp based. Each transaction receives a timestamp at initialization time generated by the local node's transaction manager, which we refer to as *epochs*. Epochs advance in a cadence of num_nodes epochs at a time shifted by $node\_id$ on each node, in such a way that epochs assigned by different nodes never conflict. In addition, Lamport logical clocks [9] are used to maintain epoch counters *mostly* synchronized between cluster nodes. Distributed transaction synchronization is further discussed in Section IV.

Transactions can be composed by three basic operations: *read*, *append* and *delete*. An implicit transaction is created whenever a user executes one of these operations outside the scope of an open transaction, and is finalized as soon as the operation finishes. Alternatively, users can explicitly initialize a transaction and use it to execute successive operations in the context of the same transaction. Naturally, users must also actively commit or rollback explicit transactions.

Transactions can also be *read-only* (RO) or *read-write* (RW), depending on the type of operations involved. Implicit transactions initialized by a read operation (query) are always RO, as well as explicitly declared RO transactions. RO transaction are always assigned to the latest committed epoch, whereas RW transactions generate a new uncommitted epoch and advance the system's clock.

B. Timestamps

Each cluster node must maintain three local atomic global counters to represent timestamps:

- *Epoch Clock (EC)*. Ever-increasing counter that maintains the timestamp to be assigned to the next transaction in the local node. EC is initialized to $node\_id$ and incremented by $node\_num$ at a time to avoid conflicts between timestamps assigned by different cluster nodes. On initialization, RW transactions atomically fetch and advance this counter.
- *Latest Committed Epoch (LCE)*. Latest epoch committed by a RW transaction in the system. LCE only advances on a transaction commit event if all prior RW transactions are also finished.
- *Latest Safe Epoch (LSE)*. Latest epoch for which (a) all prior transactions are finished, (b) no read transaction is being executed against an older epoch and (c) all data is safely flushed to disk on all replicas. Essentially, LSE controls which transaction logs must be kept and which logs can be purged.

The following invariant is maintained on each node at all times:

$$EC > LCE >= LSE$$

It has been pointed out in previous work [10] that maintaining shared atomic counters to track transaction timestamps can become a bottleneck in a multi-threaded environment once the transactional throughput scales. However, loads are usually batched into larger transactions on in-memory OLAP system (so that other transactions cannot see partial loads), and the amount of queries in the system is in the order of tens to hundreds of queries per second. This makes even worst case predictions of transactional throughput to be comfortably served with shared atomic counters.

Pending Transactions. Each node also maintains an additional set containing the epochs of all pending RW transactions seen so far, called $pendingTxs$. Whenever a RW transaction is created, the transaction epoch is inserted on this set and when transactions commit or rollback, the corresponding epoch is removed. In addition, each transaction i holds a similar set denoted by $T_i.deps$ that holds the epochs of all pending RW transactions prior to i, created based on $pendingTxs$ when T_i is initialized. $T_i.deps$ prevents i from seeing operations made by any uncommitted transaction, in such a way that T_i is only allowed to see operations made by all transactions j, such that $j < i$ and $j \notin T_i.deps$. Hence, the changes made by all transactions j form the *snapshot* of the database that transaction i is supposed to see.

action	EC	LCE	pendingTxs	T_1	T_2	T_3
—	1	0	{}			
start T_1	2	0	{1}	{}		
start T_2	3	0	{1,2}	{}	{1}	
start T_3	4	0	{1,2,3}	{}	{1}	{1,2}
commit T_1	4	1	{2,3}		{1}	{1,2}
commit T_3	4	1	{2,3}		{1}	
commit T_2	4	3	{}			

TABLE I
HISTORY OF EXECUTION OF THREE TRANSACTIONS. THE THREE LAST COLUMNS DENOTE THE SETS OF DEPENDENT EPOCHS FOR EACH TRANSACTION.

Table I describes how timestamps and pending sets are updated within a single node when concurrent RW transactions are executed. Initially, EC is 1 (denoting that 1 is the epoch to assign to the next RW transaction), LCE is zero and $pendingTxs$ is empty. T_1 is the first transaction to start, followed by T_2 and T_3. They advance EC and insert their epochs to the $pendingTxs$ set. $T_2.deps$ is set to $\{1\}$ since T_1 had already started when T_2 was initialized, and similarly $T_3.deps$ is set to $\{1, 2\}$. When T_1 decides to commit, 1 is removed from $pendingTxs$ and LCE advances since all prior transactions are also finished. However, LCE cannot be update

[1]Record materialization is the process of converting the column-wise representation of a record into a more natural row-wise format.

when T_3 commits, since one of its dependent transactions, T_2, is still running. In this case, T_3 is committed but it is still not visible for subsequent read transactions until T_2 finishes and LCE can finally advance to 3.

C. Operations

The following subsections describes how AOSI executes low-level database operations such as insertions, deletes, reads (queries), garbage collection and rollbacks.

1) Insertion: Column-oriented DBMSs partition data vertically by storing different record attributes in different vectors (or *partitions*). Therefore, appending a new record to the database involves pushing one new element to each vector and records can be materialized by using the implicit index on these vectors. We assume in this paper that records are pushed to the back of these vectors without following any specific order. Without loss of generality, in the next few subsections we illustrate records as being composed by a single attribute and hence stored on a single vector.

In addition to the data vectors, within each partition AOSI maintains an auxiliary vector called *epochs* that keeps track of the association between records and the transactions that inserted them. Each element on this vector is comprised by the pair $\langle epoch, idx \rangle$, referencing a transaction's epoch (timestamp), and the implicit record id of the last record inserted by that transaction on the current partition, respectively.

Figure 1 illustrates the basic flow of an insertion. Before initializing the operation, a new unique epoch greater than all current transactions is generated and assigned to the new transaction. T_1 is the first transaction to initialize and in (a) inserts 3 records to the illustrated partition. Consequently, the pair $\langle T_1, 2 \rangle$ is appended to the *epochs* vector to denote that the first three records belong to T_1. Later, in (b), 2 more records are appended by T_1. Since T_1 is the transaction in *epochs*' back, the last element's idx is simply incremented to point to the new last position. In (c), a new transaction (T_2) inserts 4 new records to the current partition while T_1 is still running. Since T_2 is not the transaction at the end of the $epochs$ vector, a new $\langle T_2, 8 \rangle$ pair is appended to the vector. Finally, in (d), T_1 inserts 4 more records and a new entry in added to the $epochs$ vector.

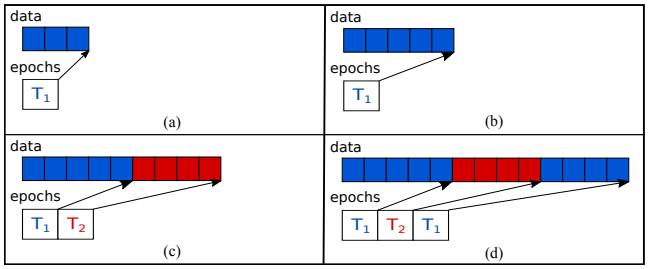

Fig. 1. Two transactions T_1 and T_2 running in parallel and appending data to the same partition.

Note that, even though the auxiliary $epochs$ vector must be held in memory to keep track of transactional history, its structure is relatively lightweight and only requires a pair of integers per transaction, differently from current approaches that store one or a couple of timestamps per record. Moreover, the *epochs* vector is only required to keep history from LSE onwards, while history of older transactions can be safely purged.

2) Delete: As presented in Section II-B, deletes are restricted to removing entire partitions, whereas single record deletes are not supported. We have decided to drop support for single record deletes since it would increase the metadata overhead (in order to control which records have been deleted from a partition), and the fact that none of the use cases evaluated truly required it.

Deleting a partition is done by appending a special tuple to the $epochs$ vector. In order to support deletions and still keep low memory overhead, we reserve one bit from one of the integers on the tuple to use as the *is_delete* flag, that dictates whether the tuple represents a delete event. Similarly to insertions, the delete tuple contains the current transaction's epoch and a pointer to the back of the data vector.

To this extent, delete operations do not actually delete data but simply *mark* data as deleted. In fact, data cannot be deleted immediately since a prior read transaction might still need to scan it. The proper data removal is conducted by a background procedure (*purge*) at a later time when all prior transactions have already finished.

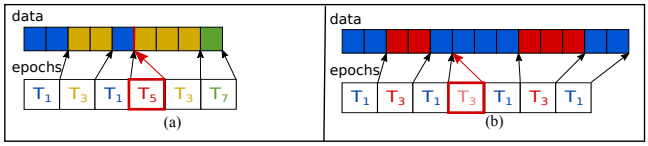

Fig. 2. State of the $epochs$ vector after the execution of two sequence of operations containing deletes. The red outline represents a delete operation.

Figure 2 illustrates the resulting $epochs$ vector after the operations shown in Table II have been executed.

(a)	(b)
T_1: loads 2 records	T_1: loads 2 records
T_3: loads 2 records	T_3: loads 2 records
T_1: loads 1 record	T_1: loads 1 record
T_5: deletes partition	T_3: deletes partition
T_3: loads 3 records	T_1: loads 3 records
T_7: loads 1 record	T_3: loads 3 records
	T_1: loads 2 records

TABLE II
SEQUENCE OF OPERATIONS EXECUTED OVER THE PARTITIONS ILLUSTRATED BY FIGURE 2.

3) Query: In order to properly exploit the benefits of column-wise data organization, most column-oriented DBMSs delay record materialization whenever possible. Therefore, column-wise scans usually carry a bitmap containing one bit per row, dictating whether a particular value should be considered by the scan or skipped. These bitmaps are commonly used for filtering and joining, but they can also be integrated

with concurrency control and leveraged to ensure transactional isolation with low friction on the query execution engine.

Prior to scan execution, a per-partition bitmap is generated for T_i based on the *epochs* vector by setting bits to one whenever a record was inserted by j, such that $j \leqslant i$ and $j \notin T_i.deps$. The remaining bits are set to zero. Additional bits may be set to zero in order to apply further filtering logic, but records skipped due to concurrency control may never be reintroduced.

Every time a delete on T_k is found by T_i, such that $k < i$ and $k \notin T_i.deps$, T_i must do another pass and clean up all bits related to transactions smaller than k, as well as records from k up to the delete point. If $i < k$ no special processing is needed since i is not supposed to see k's delete.

Table III illustrates the bitmaps generated by different read transactions when executed against the partitions depicted in Figure 2 (a) and (b). The secondary cleanup pass mentioned above is necessary in column (a) for transactions 6 and 8, and in column (b) for transactions 4, 6, and 8.

Read Tx	(a)	(b)
2	110010000	1100111100011
4	111111110	0000000011100
6	000000000	0000000011100
8	000000001	0000000011100

TABLE III
BITMAPS GENERATED BY DIFFERENT READ TRANSACTION SCANS OVER FIGURE 2'S DATA.

4) Garbage Collection: Garbage collection is implemented by a procedure called *purge*. Purge always operates over LSE, since it is guaranteed that all data prior to it is safely stored on disk (so that recovery won't be needed), and that there are no pending read transactions over an epoch prior to LSE. Naturally, LSE only advances if all prior transactions are finished.

The purge procedure has two main responsibilities: (a) removing old transactional history and (b) applying deletes. In both cases, if a cleanup is needed, a brand new partition is created by copying only the entries on $epochs$ vector newer or equal than LSE as well as applying deletes older than LSE, and finally swapping new and old partitions atomically for future queries. If there are no entries in the $epochs$ vector older than LSE and no pending delete operations, the purge procedure skips the current evaluated partition.

Figure 3 illustrates the state of the partition shown in Figure 2 (a) after the purge procedure is executed, when LSE is 3 and 5, in (a) and (b), respectively. Purging when LSE = 3 allows (a) to merge all pointers on $epochs$ prior to LSE into a single entry (when contiguous). However, the pending delete still cannot be applied since it comes from a transaction later than LSE, and therefore there might still be a read transaction in flight reading data prior to it. In (b), however, when LSE = 5, all data prior to 5 can be safely deleted, even if it was inserted after the delete operation chronologically. Hence, the only record and $epoch$ entry required is the one inserted by T_7.

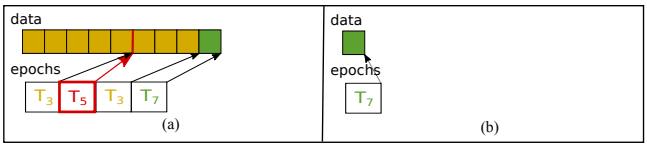

Fig. 3. State of the partition shown in Figure 2 (a) after purging it when LSE is 3 (a) and 5 (b).

5) Rollbacks: Most trasactional conflicts observed in traditional concurrency control protocols come from some sort of conflicting record update. A common situation that requires rollback is a transaction T_i trying to update a record that was read (or modified) by T_j, such that $j > i$. An optimistic protocol would execute both transactions, detect the conflict at i's commit time and ultimately rollback i [11].

Since AOSI drops support for record updates and single record deletes, all *deterministic* reasons why a transaction might have to be aborted due to isolation violations are removed. However, there are still a few situations where rollback support is required, such as database consistency violations, non-deterministic failures or explicit user aborts. AOSI assumes that rollbacks are unlikely events, being optimistic and largely optimized for commits.

Rollbacks are intensive operations that need to scan the $epochs$ vector in every single partition in the system and remove all data related to the transaction being rolledback. The deletion of these records is done by creating a new in-memory object containing all records but the ones inserted by the target transaction and swapping new and old partitions atomically.

One alternative to make rollbacks more efficient operations and avoid scanning the *epochs* vector of every single partition is to keep an auxiliary global hash map to associate transactions to the partitions in which they appended or deleted data. We do not recognize this as a good trade-off though, considering that rollbacks are uncommon operations in OLAP workflows and this hash map would increase the in-memory footprint.

D. Persistency and Durability

In-memory OLAP databases maintain persistency and ensure durability by using two basic mechanisms: (a) disk flushes and (b) replication. AOSI assumes that disk flushes are constantly being executed in the background in order to reduce the amount of data lost in the event of a crash. Every time a disk flush round is initialized, a new candidate LSE (LSE′) is selected and data between LSE and LSE′ is flushed on every single partition. Data on this range can be identified by analyzing the $epochs$ vectors. After the flush procedure finishes, LSE is eventually updated to LSE′. Since transactional history prior to LSE can be safely purged (by definition all transactions prior to that are already finished) no transactional history needs to be flushed to disk, and only a single timestamp containing the current LSE is saved on disk.

On the event of a crash, data should be recovered up to the last complete execution of a flush, ignoring any subsequent

partial flush executions that might be found on disk. Lastly, data from LSE onwards can be retrieved from the replica nodes. Important to notice that LSE needs to be prevented from advancing if data is not safely stored on all replicas or if any replica is offline.

IV. Distributed Transactions

In a distributed DBMS, the execution of operations such as reads, appends and deletes require communication with all nodes that store data for the target table. Read and delete operations must test the user's predicates against each partition on every node, while append requests need to forward the input records to the correct host. Hence, a message between the node initiating the operation and every other node must be sent even without any transactional synchronization. AOSI leverages these messages in order to synchronize clocks and get the list of pending transactions from remote nodes, and avoid introducing additional network overhead.

Since there is no deterministic reason why a transaction could fail once it starts execution (no possible isolation conflicts), there is no need to use any consensus protocol and the commit message can be implemented using a single roundtrip to each node. The single message commit technique can only be leveraged since our current database implementation does not support any types of data consistency checks or triggers that could cause transaction aborts and violate this assumption. In addition, non-deterministic failures are handled using replication in order to ensure durability.

The following subsections discuss how timestamps are maintained and assigned to transactions by different cluster nodes, and how regular node communication is leveraged in order to guarantee that each transaction operates over a consistent snapshot of the database (SI).

A. Logical Clocks

Timestamps are assigned locally to transactions without requiring any network communication by the use of Lamport Logical Clocks [9]. Lamport Logical Clocks provide a simple algorithm to specify a *partial order* of events on a distributed system with minimum overhead and without requiring extra synchronization. In its simpler form, each node maintains a local clock (atomic counter) that is incremented before each event and attached to every message sent to other nodes. Whenever a remote node receives a message, it first updates its current clock in case the received clock is greater, and optionally returns its current clock so that the source node can execute the same procedure. Using this simple schema, events can be ordered by the assigned clock timestamps, and in case different servers assign the same clock to concurrent events, some deterministic node ordering can be used as a tie breaker.

In AOSI, we set each node's local clock (EC) to $node\_idx$ on initialization and increment it num_nodes units at a time to prevent ties, thus avoiding that concurrent events (transactions) starting on different nodes have conflicting timestamps. Naturally, all messages between cluster nodes (both request and response) are piggybacked with its local EC in order to maximize the synchrony between nodes.

event	$n1$	$n2$	$n3$
—	1	2	3
$create(n1) \rightarrow T_1$	4	2	3
$append(T_1)$	4	5	6
$create(n3) \rightarrow T_6$	4	5	9
$create(n2) \rightarrow T_5$	4	8	9
$commit(T_1)$	10	8	9

TABLE IV
Epoch Clocks advancing on a 3 node cluster.

Table IV illustrates how ECs advance on a cluster comprised by 3 nodes ($n1$, $n2$ and $n3$). Initially, each node's EC is set to its own $node\_idx$. When the first transaction is created, on node $n1$, it gets the current value of EC (1) and increments it by num_nodes. In the following event, when an append operation is executed in T_1, the input records need to be forwarded to each of the remote nodes, thus carrying the local node's EC and updating EC on $n2$ and $n3$ to 5 and 6, respectively. Next, two transactions are started in nodes $n3$ and $n2$ and are assigned to their local ECs. Note that in this case the logical order does not reflect the chronological order of events since transaction T_6 was actually started before T_5. In the last event, T_1 is committed and sends a message to inform the other nodes carrying $n1$'s EC; in the same way, $n2$ and $n3$ include their local EC's on the response, making $n1$ finally update its EC to 10.

Despite having low overhead and being relatively simple to reason about, logical clocks do not always reflect the chronological order of events, as shown in the example above. It is possible that two consecutive chronological events a and b, where a happened before b, end up in a situation where $timestamp(b) < timestamp(a)$. In the context of a concurrency control where transaction sequencing is solely dictated by timestamp ordering, T_b might have to be rolledback, since a could depend on data supposed to be generated by b. In fact, a non-trivial number of transactions might need to be restarted due to this type of conflicts in case transactional throughput is considerably high, hurting the overall system performance. In the following subsection we describe how the presented protocol deal with this situation without having to rollback or restart transactions.

B. Isolation Level

In a traditional timestamp based concurrency control protocol, a transaction T_i is supposed to see all operations from transactions T_j, such that $j < i$. In fact, timestamp based systems usually rollback delayed or out-of-order transactions that could violate this invariant, such as a transaction in j updating a value read by T_i.

The presented protocol guarantees to never rollback transactions by excluding all pending transactions j from i's snapshot of the database. However, this strategy allows for a situation where given two concurrent transactions, T_k and T_l where $k < l$, neither T_k sees T_l because of regular timestamp order-

ing restrictions, nor T_l sees T_k, since T_k was pending when T_l executed. Even though this approach violates *Serializability*, since there is no serial transaction order in which this schedule could be possible, it does not violate SI since each transaction ran on a consistent snapshot of the database containing only committed transactions. This situation is a particular case of the well-known *write-skew* anomaly that differentiates SI from *Serializable* isolation levels [12].

C. Distributed Flow

Whenever a RW transaction T_i starts, a broadcast message is sent to all nodes in order to retrieve their list of pending RW transactions. This operation is usually piggybacked in the message sent to execute the actual operation, such as forward a delete request or an insertion buffer, removing the need for an extra roundtrip [2]. Once the message is received, $T_i.deps$ is set to the union of all remote $pendingTxs$, representing the set of all pending transactions in the system at that specific point in time. In addition, this message carries the local node's EC (which by definition is guaranteed to be larger than i) and updates every EC in the cluster following the Logical Clocks methodology. Hence, after the initial broadcast of i every transaction j in the system must fall into one of these categories:

- If j is committed in at least one node and $j > i$, j is not visible to i due to timestamp ordering.
- If j is committed in at least one node and $j < i$, j is visible to i due to timestamp ordering. Moreover, j is guaranteed to be finished since it is already committed in at least one node and the fact that commits are deterministic.
- If j is pending and $j > i$, j is not visible because of timestamp ordering.
- If j is pending and $j < i$ then at least one node will have j in its $pendingTxs$ set, and therefore $T_i.deps$ will contain j after the initial broadcast.
- If j is yet to be initialized, then it is guaranteed that $j > i$, since all nodes' EC were updated to a number larger than i.

RO transactions do not require this step considering they simply run on a committed snapshot of the database, usually on LCE.

When committing transaction i, the coordinator node must send a new broadcast containing the local EC (to update logical clocks) and $T_i.deps$, which contains the list of pending transactions gathered when i initialized. Next, each node marks i as committed, but can only update their local LCEs when all transactions in $T_i.deps$ are finished.

Delaying updating LCE when committing i and prior transactions are pending is a design choice. On the one hand, it creates the invariant that all transactions prior to LCE must be finished, so that RO transactions can safely run on LCE without keeping track of pending transactions, hence speeding up queries. On the other hand, the system loses *read-your-writes* consistency between transactions, since in two consecutive transactions from the same client, k and l, k might not be visible to l even after k is committed, if there is still any pending transaction p in the systems such that $p < k$. We decided to delay LCE to make query execution simpler, and assume that if a client needs *read-your-writes* consistency, the operations much be done in the context of the same transaction.

[2]However, if the first operation in a RW transaction is a read, then the list of pending transaction must be materialized before the read (query) execution.

V. CUBRICK

This section presents an overview of the Cubrick OLAP DBMS and the details about its data model and architecture required to understand the current AOSI implementation and the conducted experiments in Section VI. A complete description of the system can be found in [5].

Cubrick is a distributed in-memory OLAP DBMS written from the ground-up at Facebook, targeting interactive analytics over highly dynamic datasets. Cubrick deployments are usually focused on smaller data marts of curated datasets that can either be periodically generated and loaded in batches from the larger data warehouse, or ingested directly from realtime streams.

All data is organized according to an indexing technique called *Granular Partitioning*, that provides amortized constant time lookups and insertions, constant space complexity and indexed access through any combination of columns.

A. Data Organization

The data model of Cubrick is composed by cubes (the equivalent of relational database tables), where each column is either a *dimension* or a *metric*. Each dimension must have its cardinality estimated and specified beforehand at cube creation time, as well as a *range size* that dictates the size of each partition. The overlap of one range on each dimension forms one partition, also called *brick*, which stores the set of records falling into these ranges.

All bricks (partitions) are sparse and only materialized when required for record ingestion. Each brick is identified by one id (*bid*) that dictates the spatial position in the conceptual d-dimensional space where this brick occupies and is composed by the bitwise concatenation of the range indexes on each dimension. *Bids* are also used to assigning bricks to cluster nodes through the use of consistency hashing. Internally, bricks store data column-wise in an unordered and append-only fashion.

An auxiliary map is associated to each string column (dimension or metric) in order to dictionary encode all string values into a more compact representation. Dictionary encoding all strings makes the core aggregation engine more concise and simpler, considering that it only needs to cope with numeric values.

Example. Figure 4 illustrates Cubrick's internal data organization for the example dataset shown in Figure 4 (a). The following DDL is executed in Cubrick in order to create the respective cube:

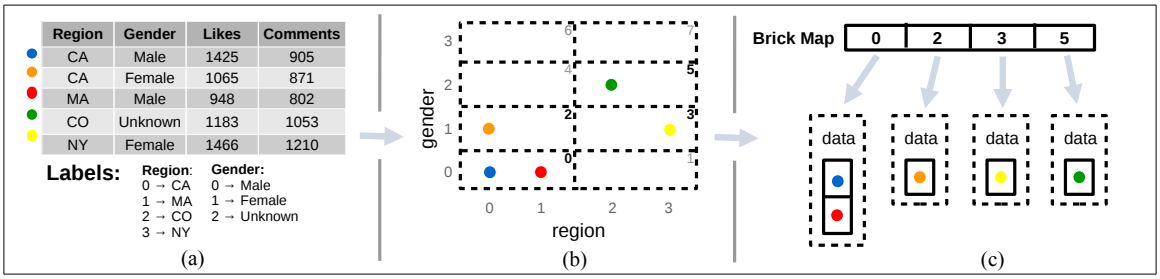

Fig. 4. Data layout of Cubrick using an example dataset.

```
cql=> create cube test
  [region 4:2 string, gender 4:1 string]
  (likes int, comments int);
```

This statement creates a cube containing two dimensions, *region* and *gender*, both with cardinality of 4 and *range size* of 2 and 1, respectively. Since both dimensions are created as strings, two auxiliary encoding maps are maintained in order to encode string values to a monotonically increasing counter. In addition, two integer metrics are defined, *likes* and *comments*.

Figure 4(b) illustrates the data model by placing *region* on the x-axis and *gender* on the y-axis. For *region*, cardinality was set to 4 and range size to 2, meaning that two ranges are available. In a similar way, there are 4 ranges available for *gender*. Since 4 ranges are available for *gender* (2 bits) and 2 for *region* (1 bit), 3 bits are required to represent *bid*, resulting in at most 8 bricks to represent the dataset.

Figure 4(c) depicts the inner-brick layout. A *brick map* maintains the collection of bricks materialized in memory indexing them by *bid*. Within each brick, data is stored column-wise using one vector per column and implicit record ids.[3].

B. Ingestion Pipeline

Cubrick data ingestion pipeline can be organized into three main steps: (a) parsing, (b) validation and forwarding and (c) flushing.

Parsing. Parsing is the first step executed when a buffer is received for ingestion. This is a CPU only procedure that can be executed on any node of the cluster and is always executed by the node that receives the buffer to avoid traffic redirection. During the parsing phase, input records are extracted and validated regarding number of columns, metric data types, dimensional cardinality and string to id encoding. Records that do not comply to these criteria are rejected and skipped. After all valid input records are extracted, based on each input record's coordinates the target *bid* and responsible node are computed — the latter by placing *bid* in the consistent hashing ring.

Validation and Forwarding. Each load request takes an additional *max_rejected* parameter that dictates the maximum amount of records that can be rejected before the system discards the entire batch. If the amount of rejected records is below *max_rejected* a transaction is created, following the rules defined in Subsection III-B, and the parsed records are forwarded to their target cluster nodes. It is important to note that at this point, all *deterministic* reasons why a transaction could fail are already discarded; therefore, all remote nodes are required to commit the transaction and no consensus protocol is required (also emphasizing that commit order is not important). Replication is used to achieve durability in face of non-deterministic failures such as hardware issues or remote node out of memory. A strategy with similar trade-offs has been described in the past in [13].

Flushing. The flushing procedure is triggered once parsed records are received by a target node. In order to avoid synchronization when multiple parallel transactions are required to append records to the same bricks, all bricks are sharded based on *bid* — where number of shards is equal to the number of available CPU cores. Each shard has an input queue where all brick operations should be placed, such as queries, insertions, deletions and purges, and a single thread consumes and applies the operations to the in-memory objects. Furthermore, since all operations on a brick (shard) are applied by a single thread, no low-level locking is required. This strategy is similar to the one described in [14] and [15], given the context of transactional database systems.

This sharding technique is specially interesting for OLAP systems where most queries are composed of large scans, since they can be naturally parallelized. In addition, this allows the system to exploit low-level optimizations, such as pinning shard threads to particular CPU cores and leverage NUMA architectures to store bricks closer to the CPU in charge of executing operations on a specific shard. Finally, this sharding technique helps providing transactional guarantees, since the system can assume that operations will be applied in the exact same order an seen by the transaction manager — assuming there is no race condition between transaction manager and inserting operations on the flush queues.

Figure 5 depicts the typical latency distribution observed in load requests on a Cubrick production cluster continuously ingesting about 1 million records per second. Parse and flush latency are usually small and the total time is dominated by network latency incurred in order to forward records to remote nodes.

[3]In reality all dimension columns are packed together and encoded in a single vector called *bess*. Details can be found in [5].

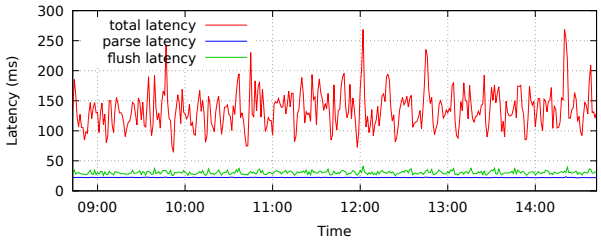

Fig. 5. Load request latency distribution on a typical production cluster.

VI. Experimental Evaluation

In this Section, we present an experimental evaluation of our current implementation of AOSI within the Cubrick distributed OLAP DBMSs. For all experiments, the servers had 32 logical cores and 256 GB of memory, although each experiment may leverage a different cluster size. The experiments are organized as follows. The first experiment measures the memory overhead caused by AOSI in order to maintain the *epochs* vector under different usage scenarios and compare to the expected overhead of traditional MVCC approaches. Further, we compare latency results of queries implementing AOSI against queries running on *read uncommitted* isolation mode in order to measure the CPU overhead of controlling pending transactions and enforcing Snapshot Isolation. Lastly, we illustrate the scale that can be currently achieved by Cubrick's ingestion pipeline.

A. Memory Overhead

In this experiment we measure the amount of memory required by AOSI in order to maintain the *epochs* vectors. Figures 6 and 7 show the execution of a Cubrick load task ingesting data from Hive, using 4 clients in parallel issuing batches of 5000 rows at a time and creating one implicit transaction per request. The target Cubrick cluster is composed by a single node. In both experiments, we show the total number of records ingested over time, the dataset size in bytes and the amount of memory being used to store the *epochs* vectors — the AOSI overhead. In addition, for the sake of comparison we also show a *baseline overhead* representing the amount of memory required to store two timestamps per record, a common practice in MVCC implementations [1]. *Baseline overhead* is set to $16 * num\_records$, representing two 8-byte timestamps. For clarity, y-axis is shown on a log scale in both charts.

Figure 6 shows the results of a load task of a single column dataset containing about 100 million rows. A single column dataset is the worst case scenario when evaluation memory overhead of concurrency protocols, since most metadata is stored per record. As shown in the chart, after about 10 minutes of execution AOSI's overhead reaches its peak, at 35 MB but still only about 5% if compared to the current dataset size. The baseline overhead at that point reached 1 GB or about 130% of the dataset size. Shortly after, a purge procedure is triggered by LSE being advanced, therefore recycling old

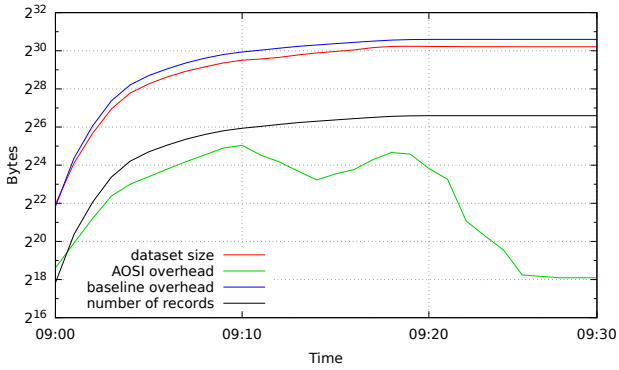

Fig. 6. Memory overhead of 4 clients loading in parallel 100 million records of a 1 column dataset to a single node cluster.

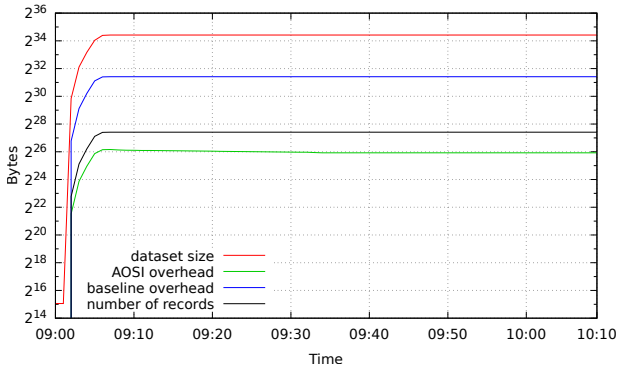

Fig. 7. Memory overhead of 4 clients loading in parallel 176 million records of a 40 column dataset to a single node cluster.

epochs entries and bringing memory overhead down to 10 MB, or about 1% of the dataset. Eventually, after the jobs finishes LSE advances again and more *epochs* entries are recycled, dropping AOSI's overhead to 300 KB (0.02% of the full dataset size), while the baseline maintains steady at 1.6 GB, roughly 4 orders of magnitude larger than AOSI.

Figure 7 shows the results of a similar experiment on a typical 40 column dataset. After about 5 minutes of execution, all 176 million records of the dataset are loaded, consuming roughly 22 GB of memory. In that instant, the baseline overhead is approximately 2.8 GB (13% of the full size) whilst AOSI's overhead is 74 MB. Moreover, after LSE advances and some *epochs* pointers are recycled, AOSI's overhead drops to about 60 MB, or 0.2% of the dataset.

B. Query Performance

This experiment presents a comparison of queries running on SI and best-effort queries that simply read all available data — or *read uncommitted* (RU) mode. We started a single thread of execution running the same query successively, alternating between SI and RU in order to evaluate the overhead and subsequent latency increase observed when controlling which records each transaction in supposed to see using the *epochs* vector, *pendingTxs* set and bitmap generation.

We ran two experiments in a single node cluster: (a) queries

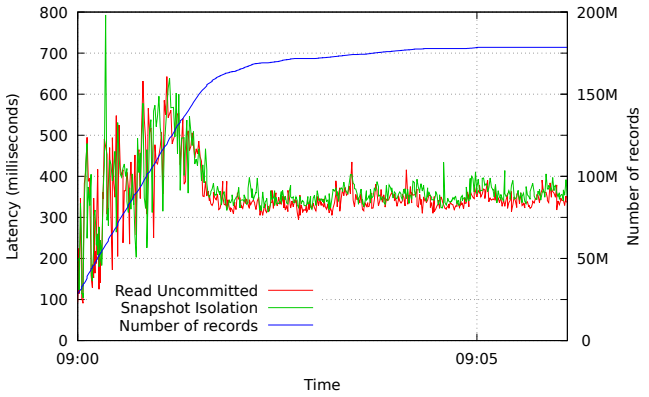

Fig. 8. Query latency of the same query running in SI and RU mode during a batch load.

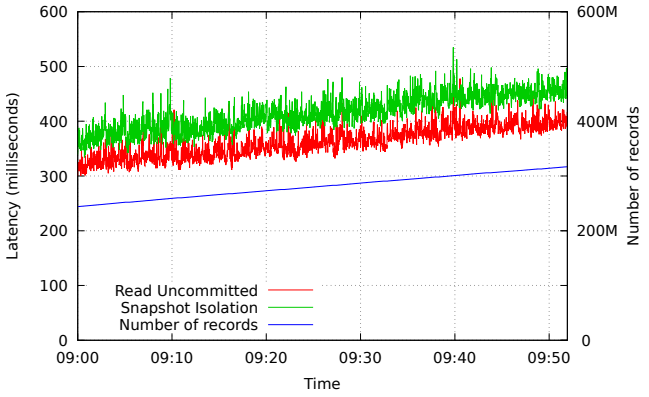

Fig. 9. Query latency of the same query running in SI and RU mode while the cube is continuously loaded in realtime.

during a batch load of a cube of about 170 million records from 4 consecutive clients sending 5000 records per batch (one transaction per batch) running in Hive, and (b) queries on a cube being continuously loaded from a realtime stream by 32 parallel threads executing one load transaction for each 5000 records batch.

Figure 8 shows the results for (a). During the first 2 minutes of execution when most records were loaded, there was an increase on the overall query latency due to ingestion and queries competing for CPU. However, the ratio between query latency on SI and RU remained constant during the entire experiment, close to 1%.

For the second part of this experiment, we stressed AOSI by loading a dataset in realtime using 32 clients issuing load requests every 5000 records. In addition, we disabled the purge procedure in order to force the $epochs$' vector size to grow, and evaluate the worst case overhead for query latency. During the experiments the cube was comprised by about 240 to 320 million records of a dataset containing about 20 columns.

Figure 9 shows the results for (b). More processing is required in order to iterate over the large $epochs$ vectors when running in SI mode, but even in the worse case scenario the query latency increase is below 10%.

C. Ingestion Scale

The purpose of this experiment is to illustrate the scale in which Cubrick can ingest data. To that end, we measured the number of records as well as the amount of raw data in bytes ingested per second in a daily job that loads data from an HDFS warehouse running Hive [7] to a 200 node Cubrick cluster. Similarly to the experiment shown in Section VI-A, we present the results for a single-column dataset that maximizes the effect and overhead of concurrency control protocols.

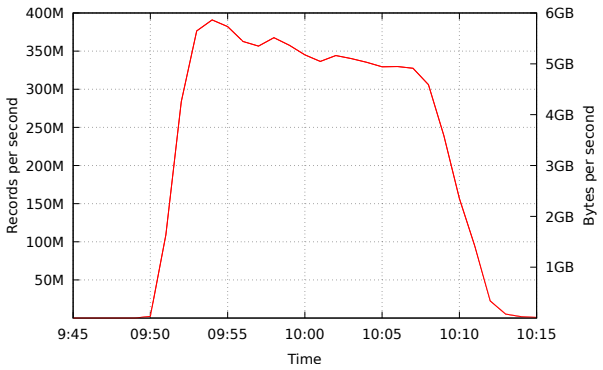

Fig. 10. Ingestion scale in both number of records (left y-axis) and amount of data (right y-axis) per second for a daily batch load on a 200 node cluster.

Figure 10 shows the results. About 4 minutes after the job initializes, the ingestion reaches its peak at about 390 million records per second, corresponding to roughly 6 GB/s of raw incoming data, and slowly decreases when Hive tasks start to finalize. In about 25 minutes the job is finished, having loaded about 400 billion records from the data warehouse.

VII. Related Work

This section describes the main commercial offerings for in-memory OLAP databases and their concurrency control protocols.

Hive [7] is a widely used OLAP query engine built on top of map-reduce that allows the execution of SQL-like queries over files stored on a distributed filesystem such as HDFS. Starting with version 0.13, Hive introduced support for ACID transactions (SI only). Since HDFS does not support in-place changes to files, Hive's concurrency control protocol works by creating a delta file per transaction containing updates and deletes, and merging them at query time to build the visible dataset [16]. Periodically, smaller deltas are merged together as well as deltas are merged into the main files. Hive relies on Zookeeper to control shared and exclusive distributed locks in a protocol similar to 2PL (Two Phase Locking).

SAP HANA is an in-memory DBMS that allows the execution of both OLTP and OLAP workloads in the same database system. It is comprised by a layered architecture, where incoming records are stored in an uncompressed row-oriented format and periodically merged to a compressed and encoded column-wise format [17]. HANA's concurrency control protocol is based on MVCC and maintains two timestamps per record

(*created_at* and *deleted_at*) that enforces the snapshot isolation model. Updates are modeled as a deletion plus reinsertion and record-level locks are used to prevent update conflicts [18].

Oracle's In-Memory Database [3] also offers a hybrid OLTP and OLAP architecture by selecting a few columns of the OLTP database to be materialized column-wise. Concurrency control is handled in the same multiversioned manner as for OLTP by using timestamps and locks, and changes on the materialized columns are stored as deltas and periodically merged - a process called *repopulation* [19].

HyPer [20] is another in-memory DBMS that can handle both OLTP and OLAP simultaneously by using hardware-assisted replication mechanisms to maintain consistent snapshots. All OLTP transactions are executed serially, since they are usually short-lived and there is no halt to await IO, whereas OLAP transactions are forked into a new child process and rely on the operating system's page sharing and copy-on-write techniques for isolation. HyPer supports serializable transactions by updating records in place and storing a list of older versions (undo log) associated with the transaction that created it, hence maintaining an extra pointer for each record [21]. Update conflicts are handled optimistically by rolling back one of the updates when a conflict is detected.

Hekaton [1] is a lock-free in-memory OLTP engine built on top of SQL Server in order to speed-up transactional workloads. Hekaton provides several levels of isolation by using MVCC and storing two timestamps per record version to determine visibility.

VIII. Conclusion

This paper presented a new concurrency control protocol called *Append-Only Snapshot Isolation* (AOSI), suited for in-memory OLAP systems. Unlike previous work, which are either based on locking or MVCC, the presented protocol is able to provide Snapshot Isolation (SI) in a completely lock-free manner and always maintain a single version of each data item. In addition, the protocol removes the need to store per record timestamps, common in MVCC implementations, and allows for more efficient memory usage.

The protocol supports these characteristics by sacrificing a few operations commonly found in transactional systems, but that we advocate are not strictly required in order to build flexible and scalable OLAP systems, namely *updates* and *single-record deletes*. Through an experimental evaluation, we have shown that the presented protocol is able to provide SI with considerably less memory overhead than traditional techniques. In addition, we have also shown that the performance overhead on query latency is minor, which is a crucial factor for in-memory OLAP systems.

References

[1] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1243–1254.

[2] T. P. G. D. Group, "Postgresql 9.1.9 documentation," http://www.postgresql.org/docs/9.1/static/, Jun. 2013.

[3] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, J. Loaiza, N. MacNaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zat, "Oracle database in-memory: A dual format in-memory database." in *ICDE*. IEEE Computer Society, 2015, pp. 1253–1258.

[4] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," in *International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2012, pp. 1790–1801.

[5] P. Pedreira, C. Croswhite, and L. Bona, "Cubrick: Indexing millions of records per second for interactive analytics," *Proceedings of the 42nd International Conference on Very Large Data Bases*, vol. 9, no. 13, pp. 1305–1316, Sep. 2016.

[6] R. Kimball, *, The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley & Sons, 1996.

[7] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*. IEEE, 2010, pp. 996–1005.

[8] D. R. K. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1850–1861, 2012.

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[10] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *Proceedings of the 41st International Conference on Very Large Data Bases*, no. 11, pp. 1190–1201, Jul. 2015.

[11] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '95. New York, NY, USA: ACM, 1995, pp. 1–10.

[13] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 1–12.

[14] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, 2008.

[15] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[16] Apache Hive, "Hive transactions," https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions, 2016.

[17] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in SAP HANA database: The end of a column store myth," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. ACM, 2012, pp. 731–742.

[18] A. Böhm, J. Dittrich, N. Mukherjee, I. Pandis, and R. Sen, "Operational analytics data management systems," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1601–1604, Sep. 2016.

[19] N. Mukherjee, S. Chavan, M. Colgan, D. Das, M. Gleeson, S. Hase, A. Holloway, H. Jin, J. Kamp, K. Kulkarni, T. Lahiri, J. Loaiza, N. Macnaughton, V. Marwah, A. Mullick, A. Witkowski, J. Yan, and M. Zait, "Distributed architecture of oracle database in-memory," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1630–1641, Aug. 2015.

[20] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 195–206.

[21] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 677–689.