Materialized Views for Eventually Consistent Record Stores

Changjiu Jin, Rui Liu, Kenneth Salem

Technical Report CS-2012-26
Cheriton School of Computer Science, University of Waterloo
Waterloo, Ontario, Canada
December 2012

A shorter version of this paper [1] will appear in the Proceedings of the 2nd International Workshop on Data Management in the Cloud (DMC 2013), Brisbane, Australia, April, 2013.

***Abstract*—Distributed, replicated keyed-record stores are often used by applications that place a premium on high availability and scalability. Such systems provide fast access to stored records given a primary key value, but access without the primary key may be very slow and expensive. This problem can be addressed using materialized views. Materialized views redundantly store records, or parts of records, and the redundant copies can be organized and distributed differently than the originals, e.g, according to the value of a secondary key. In this paper, we consider the problem of supporting materialized views in multi-master, eventually consistent keyed-record stores. Incremental maintenance of materialized views is challenging in such systems because there no single master server responsible for serializing the updates to each record. We present a decentralized technique for incrementally maintaining materialized views in multi-master systems. We have implemented a prototype of our technique using Cassandra, a widely used system of this type. Using the prototype, we show that secondary-key-based access is much faster using materialized views than using Cassandra's native secondary indexes, but maintaining the views in the face of updates may be more expensive than maintaining indexes.**

I. Introduction

Keyed-record stores are often used by applications that place a premium on high availability and scalability. In these systems, each stored record is associated with a primary key value. Records are replicated and distributed across multiple servers. Client applications can access a stored record quickly by providing its primary key value. Examples of such systems include BigTable [2], Cassandra [3], and Amazon's SimpleDB.

One of the principal limitations of these systems is that the only way to access stored data efficiently is with the primary key. For example, if the stored data consist of customer records keyed by a customer identifier, it may be difficult or impossible to access records by, for example, customer name. Some systems, such as Cassandra, provide secondary indexes to address this problem. However, to help ensure consistency between the secondary index and and the records, such indexes are themselves typically partitioned and distributed according to the *primary key*. To access a record using such an index, the system must broadcast the request to multiple servers, each of which can then check for the requested record using its fragment of the index. As a result, accessing records through such an index is typically slower and much more expensive than accessing the data by primary key.

Applications that use keyed-record stores often address this problem by storing the same data multiple times, with different keys. For example, an application might store two customer tables, one keyed by customer ID, and the other by customer name. When customer records are updated, inserted and deleted, the burden of ensuring that these tables remain synchronized must be borne by the application.

Materialized views are a mechanism that allows this burden to be shifted from the client applications to the record storage system. Materialized views are tables that redundantly store records, or parts of records, from another table, called the base table. The records in the view can be organized and distributed differently than the originals in the base table. In particular, they can be distributed according to the value of a secondary key. Since the system is aware of the relationship between a materialized view and its base table, it can assume responsibility for maintaining (updating) the view when the base table changes.

Materialized views are widely implemented in relational database systems. In relational systems, incremental maintenance of (simple) materialized views is conceptually simple. Base table updates are typically propagated to the views in transaction serialization order, which is obtained from the database system's transaction log. A similar approach can be use to implement incremental view maintenance in distributed keyed-record stores, provided that there is some mechanism analogous to the transaction log for determining the order in which to propagate changes. For example, PNUTS [4] uses this approach to implement view maintenance. Each record stored in PNUTS has a single master copy which serializes all updates to that record. Thus, the master server for a record can be responsible for propagating the record's changes to the view(s).

In this paper, we focus on the problem of supporting ma-

terialized views in *multi-master, eventually consistent keyed-record stores*, such as Cassandra, SimpleDB, Project Voldemort, and Riak. In these systems, there is no master server that serializes updates to a given record. Furthermore, these systems may provide only an *eventual consistency* guarantee to applications. This means that, depending on how an application reads a record, it may not be guaranteed to see that record's most-recent version. As we will show, this complicates the problem of maintaining materialized views. In particular, the view maintenance approach used by PNUTS is not directly applicable to such systems.

This paper makes two primary research contributions. First, we present a technique for incrementally maintaining materialized views in multi-master, eventually consistent keyed-record stores. Our technique is decentralized, like multi-master systems. It is also asynchronous, which means that an application's updates to a base table are not guaranteed to be reflected in the view immediately. In Section III we present our rationale for this design choice. It is possible to complement our technique with an additional mechanism that will provide a *session consistency* guarantee for each application client. Although view update is still fundamentally asynchronous, session guarantees ensure that if a client updates a base table and then reads a view defined on that table, it can be sure that the view will reflect the effects of that client's own preceding updates.

Our second contribution is an empirical analysis of the performance of materialized views. We prototyped our incremental view maintenance technique within Cassandra, a widely used open-source multi-master system. In our analysis, we consider both the cost of accessing the view and the overhead of maintaining the view in response to base table updates. Since materialized views provide an alternative to native secondary indexes, we provide a comparison of the performance of Cassandra's native secondary indexes to that of materialized views.

II. System Model

We begin by presenting generic model of a multi-master, keyed-record system. We will use the model to present our view maintenance algorithm. Our generic system is similar to Cassandra, although we have eliminated many Cassandra-specific details that are not relevant to view maintenance.

The system allows applications to define sets of records, which we will refer to as *tables*. Each record has a key and one or more named attributes, or columns. Different records in the same table may have different attributes. The combination of key and a column name identifies a *cell* in the table. Each cell may have an associated value, and each cell with a value also has an associated timestamp. We will use the notation $T[k, c]$ to refer to the cell corresponding to key k and column c in table T.

Applications can use two operations on tables: `Put` and `Get`. A `Put` operation takes as parameters a table name (T), a key value (k), a set of column names ($[c_1, c_2, \ldots, c_n]$), a set of values ($[d_1, d_2, \ldots, d_n]$), a set of timestamps ($[t_1, t_2, \ldots, t_n]$), and a *write quorum* (W), which we will describe shortly.

For each column c_i, the `Put` operation sets the value and timestamp of $T[k, c_i]$ to d_i and t_i, respectively, unless the cell's timestamp is already larger than t_i, in which case the `Put` has no effect on that cell. A `Get` operation takes a table name (T), a key value (k), a set of column names ($[c_1, c_2, \ldots, c_n]$), and a read quorum (R) as parameters. It returns the current value and timestamp for each cell $T[k, c_i]$. The value in each cell will be the value written by the preceding `Put` operation with the largest timestamp. If no value has ever been `Put` into a cell, we assume that a read of the cell will obtain a `NULL` value and timestamp. (A `NULL` timestamp is assumed to be smaller than all non-`NULL` timestamps.)

To delete the value in a cell, an application can `Put` a `NULL` value into the cell. Internally, the system places a *tombstone* value in such a cell, along with the timestamp from the `Put` operation, to record when the value was deleted. Subsequent `Get` operations will read `NULL` from the cell until a non-`NULL` value (with a larger timestamp than the tombstone's) is put there.

The system has multiple servers. Each record is stored N times, on N different servers. (N is a configurable parameter.) The placement of records onto servers is typically determined by hashing the record key [5], [3], but the placement policy is orthogonal to our work. We assume only that placement of a record's copies is determined by its key value. To perform a `Get` or `Put`, an application client connects to any server in the system. That system acts as the *coordinator* for the request. The coordinator first uses the supplied table name and record key to determine which servers hold copies of the target record. In the case of a `Put` request, the coordinator sends the request to all N replicas of the record, and waits for responses from at least W ($1 \leq W \leq N$) of the replicas before acknowledging the `Put` request to the application. Each replica performs the `Put` operation on its local copy before sending an acknowledgment to the coordinator. In the case of a `Get` operation, the coordinator again forwards the request to all N replicas, and waits for the first R ($1 \leq R \leq N$) responses. Each replica server performs the `Get` operation on its local copy of the record and returns a list of cell values and timestamps to the coordinator. For each cell identified in the `Get` request, the coordinator chooses the value with the largest timestamp from among the first R responses, and returns that value, along with its associated timestamp, to the client application. The local `Put` and `Get` operations performed by each individual server are atomic.

Applications can control a consistency/performance tradeoff by varying W and R in `Put` and `Get` operations. In particular, if $W + R > N$, the system implements classical quorum consensus [6] and each `Get` operation is guaranteed to return cell values written by the preceding `Put` with the largest timestamp. If $W + R < N$, `Get` operations may return stale values but `Gets` or `Puts` (or both) may finish more quickly because the coordinator need not wait for as many acknowledgments from replicas. The system includes

TICKET (base table)

Id	Status	AssignedTo	Description
1	open	rliu	...
2	open	kmsalem	...
3	open	kmsalem	...
4	resolved	rliu	...
5	open	cjin	...
6	new		...
7	resolved	cjin	...

ASSIGNEDTO (view)

AssignedTo	Ticket	Status
rliu	1	open
rliu	4	resolved
kmsalem	2	open
kmsalem	3	open
cjin	5	open
cjin	7	resolved

Fig. 1. Example of a Base Table and a View

mechanisms (not described here) that ensure that all updates to a cell eventually reach every replica of that cell's record, despite failures. All updates are totally ordered according to the application-specified timestamps supplied with `Put` operations, so all servers will agree on the ordering of updates to each cell.

III. VIEWS

A view is a table whose contents are determined by another table, called the base table. In our system, all views are materialized, replicated and stored like regular base tables. Thus, we will use the terms "view" and "materialized view" interchangeably.

Definition 1 (View)
A view V is defined by a base table name (B), a view-key column name *(c_V), and zero or more* view-materialized column names *($c_1, c_2, \ldots$). For each base key k_B such that $B[k_B, c_V]$ is not NULL, there is a row in the view, with key k_V equal to the value of $B[k_B, c_V]$. In that row, the following cells are defined:*

- *$V[k_V, B]$ has value k_B and timestamp equal to that of $B[k_B, c_V]$*
- *for each view-materialized column c_i, $V[k_V, c_i]$ has the same value and timestamp as $B[k_B, c_i]$.*

Figure 1 shows a simple example of base table and view. The TICKET base table tracks requests for a help desk application. Column `Id` is the key. ASSIGNEDTO is a view defined on TICKETS. The `AssignedTo` column is the view key and the `Tickets` column indicates the primary key of the base table row that corresponds to each view row. `Status` is a view-materialized column. The ticket `Description` field from the base table is not materialized in the view.

In relational database terms, the views we consider in this paper are single-table views that involve only relational projection. That is, each view includes a subset of the columns of a single base table. It would be easy to incorporate relational selection, so that a view would include only those rows that satisfy a selection condition. Furthermore, our approach could be extended to support equi-join views in much the same way as is done in PNUTS [4]. However, in this paper we will restrict ourselves to the single-table projection views of Definition 1.

Views are similar to base tables, and once a view has been defined, it can be used by applications in much the same way as a table. However, there are two differences between views and base tables. First, views are not updateable: applications are permitted to perform `Get` operations on views, but not `Put` operations. Second, according to our definition, it is possible for a view to have multiple rows with the same view key. For example, in Figure 1, each view key occurs twice in the view. Such rows will always be distinguishable by the value of the base key column, e.g., the `Ticket` column in the ASSIGNEDTO view. Since views can have multiple records with the same view key, a `Get` on a view returns a set of results, one per view record that matches the specified view key. For example, a `Get` of the `Ticket` and `Status` columns for key `rliu` in the view shown in Figure 1 will return {`[1,open],[4,resolved]`}. In contrast, a `Get` operation on a base table returns a single value for each requested column.

IV. VIEW MAINTENANCE AND CONSISTENCY

Since each view depends on a base table, each update to a base table may require corresponding changes in any views that depend on it. Since our views are materialized, we require a means of updating, or maintaining, views in response to base table updates. View maintenance can be synchronous or asynchronous. With synchronous maintenance, a base table update and the corresponding view update(s) occur as a single, atomic operation, so that each view and its base table remain mutually consistent at all times. In the case of asynchronous update, the base table is updated first, and dependent views are updated sometime later. In general, asynchronously updated views will be *stale* with respect to their base tables.

Unfortunately, even if we were to provide synchronous view maintenance, applications in our system cannot take advantage of mutual consistency between a base table and view. For example, consider an application that wants to retrieve the descriptions of open tickets assigned to `rliu` using the example database from Figure 1. The application must first `Get` from the ASSIGNEDTO view, using key `rliu`, to learn that the `Id` of `rliu`'s only open ticket is `1`. The application can then `Get` from the TICKET table, using key `1`, to get the task's `Description`. The problem is that the application must perform two `Get` operations to do this. Even if every TICKET update propagates synchronously to the ASSIGNEDTO view, the application cannot rule out the possibility that TICKET will be updated in between its two `Get` operations. Such an update could, for example, delete ticket 1, or change its assignment. Thus, the application cannot

be assured of mutual consistency between views and base tables.

Synchronous view maintenance adds latency to `Put` operations on base tables, since view maintenance must be completed before the `Put` completes. Since clients cannot take advantage of the mutual consistency that this extra latency buys, our system instead implements asynchronous view maintenance. Base table updates are propagated eventually to views, and thus views are normally slightly stale. Applications must be prepared for the possibility that a view will be inconsistent with its base table. However, applications can choose to reduce the impact of this problem by including *view-materialized columns* in their view definitions. View-materialized columns (such as the `Status` column in the TICKET view) cause additional information from the base table to be mirrored in the view, thus potentially allowing an application to access *only* the view and avoid accessing the base table. In our previous example, if the `Description` column had been included in as a view-materialized column in the TICKETS view, the application could have avoided its second `Get` operation, and thus could have avoided being exposed to potential inconsistencies between the view and the base table. Of course, the price of view-materialized columns is additional space overhead for the views, and additional view maintenance overhead when the value of the view-materialized column is updated in the base table.

A. Incremental View Maintenance

To illustrate the challenges associated with incremental view maintenance in our system, we will use two examples.

Example 1 (Propogating a Single Update)
Suppose that the base table and view from Figure 1 are as shown, and a client application uses a `Put` *operation to change the assignment of ticket 2 in the* TICKETS *table to* `rliu`*. This ticket corresponds to a single row, with view key* `kmsalem`*, in the* ASSIGNEDTO *view. To maintain the view in response to this update, the key of this row in the view must be changed from* `kmsalem` *to* `rliu`*. This can be done by deleting the existing* `kmsalem` *row from the view and creating a new row with key* `rliu` *and the same attributes and values as the original row.*

Now, consider a second example involving concurrent updates to the base table:

Example 2 (Propagating Concurrent Updates)
Suppose that the base table and view are as shown in Figure 1 and that two clients attempt to update TICKETS *concurrently. The first client performs the update described in Example 1, setting the assignment of ticket 2 to* `rliu`*. The second client concurrently attempts to set the assignment of ticket 2 to* `cjin`*. Furthermore, suppose that the second client's update has a larger timestamp. Thus, it is clear that both the base table and the view should eventually agree that ticket 2 is assigned to* `cjin`*. However, the correct actions to take when propagating these two updates to the view depends on the order in which they propagate. If the second client's update propagates second, the correct view maintenance action will be to delete the* `rliu` *record from the view and insert a* `cjin` *record. If the second client's update propagates first, the correct view maintenance action will be to delete the* `kmsalem` *record from the view and insert a* `cjin` *record. The situation for propagating the first client's update is similar.*

This second example illustrates the fundamental challenge of view maintenance in our system: to maintain the view in response to some change to a record in the base table, it is necessary to know the view key of the corresponding record in the view. However, it is difficult to determine the correct view key, since that depends on which updates have already been propagated.

One way to solve this problem is to ensure that updates propagate to the view sequentially and in timestamp order. Sequential, in-order propagation simplifies the task of deciding how to propagate each update, since it is known exactly which updates have already propagated and which have not. Asynchronous incremental view maintenance in PNUTS [7] follows this approach. Each record has a designated master copy, which serializes updates to that record and propagates updates to views in serialization order. We could implement this approach in our system by, for example, designating one copy of each base row as master, and making it responsible for propagating all updates to that row. The designated master would propagate updates sequentially in the order in which they are applied at that master copy. Although such an approach would work, we have chosen to avoid it as it runs contrary to the decentralized, multi-master behavior of the rest of the system. Having a master copy for each row means that we must also have some mechanism for choosing new master in the event of master failure. While this is certainly possible, such a mechanism is not needed anywhere else in our multi-master system.

Instead, we propose an approach in which each update coordinator is responsible for propagating the updates that it coordinates. Since any server can coordinate updates, any server can also propagate updates to views, and all such servers propagate their updates independently and concurrently. With this approach, there is no need for a designated master for each row.

To illustrate the general idea behind our approach, we consider Example 2 again. Suppose that the first client's update propagates first. To determine the view key of the record for ticket 2, the first client's coordinator will read the value of the view key in the base table row before updating it, finding the value `kmsalem`. It will then look for the corresponding row in the ASSIGNEDTO using the key `kmsalem` and find the corresponding record. The coordinator will change the view key for this record to `rliu` as required to reflect its base table update. However, in addition to this, the coordinator will also insert a new record into the view with view key `kmsalem` and one field which contains the new view key (`rliu`) which replaced `kmsalem`. This extra row is called a *stale row*.

When the second client updates ticket 2's assignment to `cjin` in the base table, it will first read value of the view key from the base table record, just as the first client did. It will read either `kmsalem` (the original value of the view key) or `rliu`, depending on whether its base table update occurs before or after the first client's. It will then use the view key that it reads to look for the corresponding row to be updated in the view. If it reads `rliu`, it will immediately find the correct record in the view. If it reads `kmsalem`, it will instead find the stale row that was inserted by the first client. The stale row, which contains the new view key (`cjin`), will allow the second coordinator to locate the correct view record.

Before giving a full description of incremental view maintenance technique, we should define what it means to perform incremental view maintenance correctly in system in which updates propagate asynchronously, and not necessarily in the (timestamp) order in which they will be serialized at the base table. We define this as follows. Suppose that V is a view defined on base table B. Let B_0 be the initial state of the base table and V_0 the corresponding initial state of the view, according to Definition 1. Let $\mathcal{U}_n$ represent the first n base table updates to be propagated to the view, and let V_n represent the state of the view after the updates in $\mathcal{U}_n$ have been propagated.

Definition 2 (Incremental View Maintenance)
The correct view state (V_n) after the first n update propagations is the view state that would be obtained by applying the updates in $\mathcal{U}_n$ to the base table in timestamp order, starting from base table state B_0 and resulting in state B_n, and then determining the view state corresponding to B_n using Definition 1.

Note that when the view is in state V_n, the base table may not be state B_n (in fact, it may never enter state B_n at all) since the base table will likely have experienced updates in addition to the ones that have propagated to the view. Instead, Definition 2 says that an incrementally and asynchronously maintained view is correct if it is the view that would be obtained by applying only the propagated updates to the base table and then calculating the view.

B. Versioned Views

In order to support incremental view maintenance, our system stores *versioned views*. Versioned views contain stale rows in addition to the current records of the view. Stale rows are used during view maintenance to ensure that the proper view records can be located and updated, as was illustrated in Example 2.

A versioned view contains all of the records defined for a plain non-versioned views (Definition 1) - we refer to these as the versioned view's *live rows*. In addition, for each live row, the versioned view contains zero or more stale rows. The live rows represent the current state of the view. If the view key of a row has been updated, there will be stale rows for each of the old view keys. Each stale row contains a pointer (a view key value) referring to a more recent view key for the row. Each stale row's pointer leads, directly or indirectly, to its corresponding live row.

ASSIGNEDTO (view)

AssignedTo	Id	Status	Next
rliu	1	open	-
rliu	4	resolved	-
rliu	*2*	-	*cjin*
kmsalem	*2*	-	*rliu*
kmsalem	3	open	-
cjin	2	open	-
cjin	5	open	-
cjin	7	resolved	-

Fig. 2. Example of a Versioned View. Stale rows are shown in italics.

Figure 2 shows an example of a versioned view that could result if the two updates described in Example 2 were applied to the sample database shown in Figure 1. In versioned views, the additional `Next` column is used in stale rows to hold the pointer. In Figure 2, there are three rows that correspond to the record for ticket 2 in the TICKETS base table. Two of these rows are stale, and the third is the live row representing the current state of the view after the propagation of the two updates. In the example, the two `Next` pointers form a path that links the stale rows to the live rows. Stale rows in versioned views are used only to support view maintenance. They are not visible to applications, which see views as defined in Definition 1.

Definition 2 defines the correct state of a view after each incremental view update. Here, we extend this definition to specify the correct state of a *versioned* view after each update is propagated. Consider a base table B and view V with initial states B_0 and V_0, and let $\mathcal{U}_n = \{u_1, u_2, \ldots, u_n\}$ represent the first n updates to have been propagated to the view, listed in the order in which they finish propagating. Let $V_1, V_2, \ldots, V_n$ represent the non-versioned view states that result from these propagations - these are the states defined by Definition 2. Consider a specific base table key, k_B. The view key associated with base row k_B may change over time because of updates to the view key column in the base table. Let $k_{v1}, k_{v2}, \ldots, k_{vm}$ represent the set of distinct view keys associated with base key k_B in view states $V_1, V_2, \ldots, V_n$. Finally, let t_{vi} represent the timestamp associated with view key k_{vi}. t_{vi} is the largest timestamp among all updates in $\mathcal{U}$ that set the value of the view key column in row k_B to k_{vi}. Assume that the view keys and timestamps are numbered in increasing timestamp order, so that $t_{vi} < t_{vj}$ if $i < j$.

Let $\hat{V}_1, \hat{V}_2, \ldots, \hat{V}_n$ represent the *versioned* view states corresponding to the non-versioned states $V_1, V_2, \ldots, V_n$. These versioned states are defined in Definition 3.

Definition 3 (Versioned View)
For such base key k_B, the versioned view state $\hat{V}_n$ after the updates $\mathcal{U}$ have propagated will include the following rows:

- *A* live row *with view key k_{vm} (i.e., the most recent view key for k_B) containing the view-materialized cells defined*

in Definition 2. This row will be identical to k_B's row in the non-versioned view (V_n) except for the presence of one additional cell, which we will refer to as `Next`*. The value of the* `Next` *cell is k_{vm}, i.e., it is a self-pointer, and its timestamp is t_{vm}. The self-pointer identifies this as a live row.*

- *For each other key $k_{vj} (1 \leq j < m)$, the versioned view will contain a stale row with key k_{vj} and two columns, B and* `Next`*. The value of column B is k_B. The value of column* `Next` *will be $k_{vj'}$, for some j' such that $j < j' \leq m$. That is,* `Next` *will point to a more recent view key for base row k_B. Furthermore, the* `Next` *pointer in each stale row will lead, either directly or indirectly, through a chain of k_B's stale rows to the live row for k_B.*

C. Update Propagation

Algorithm 1 shows how a coordinator performs an update on a base table on which a view is defined. Before performing the update, the coordinator reads the current value of the view key for the row being updated (line 2 in Algorithm 1). This `Get` operation differs from the normal `Get` (described in Section II) in that it returns all of the distinct versions of the view key that it finds in the base rows replicas, not just the latest version. Although the `Get` and `Put` at lines 2 and 3 are shown as separate operations in Algorithm 1, in practice they can be combined into a single combined `Get`-then-`Put` request that the coordinator sends to all replicas of row k_B. The base table update operation returns to the client (line 4) as soon as the coordinator has received responses from the client-specified quorum (W) of base row replicas. After returning to the client, the coordinator continues to collect view keys from the remaining replicas (if any) and then initiates update propagation to the view. To propagate the update, the coordinator chooses one of the returned view keys (line 6) as its guess of the corresponding row's view key and passes that view key to `PropagateUpdate`. The coordinator is free to try the keys in any order, and if necessary may even try the same key more than once. In general, multiple attempts to propagate the update may be necessary because `PropagateUpdate` may fail. Such failures occur if the view key chosen by the coordinator at line 6 was written by another base table update that has not yet been propagated to the view. Unless there is a high rate of updates to the view key of a single row, we expect that failures of `PropagateUpdate` will be rare, so that the coordinator need not make multiple propagation attempts.

Note that the coordinator need not wait for view keys from *all* of the view replicas before invoking `PropagateUpdate` (line 5). It may do so at any time after acknowledging the base table `Put` operation to the client, choosing from among the view keys that it has collected so far. Once `PropagateUpdate` has succeeded, there is no need for the coordinator to wait for view keys from any base row replicas that have not yet responded to the original `Get`-then-`Put` request.

Algorithm 2 shows the implementation of the `PropagateUpdate` function that is invoked by Algorithm 1. `PropagateUpdate` is responsible for performing incremental view maintenance in response to a base table update. In our presentation of `PropagateUpdate`, we have made several simplifying assumptions. First, we have assumed that the base table `Put` operation that triggers incremental view maintenance updates a single base table column, which may be the view key or a view-materialized column. If a base table `Put` operation modifies more than one column in a base table row, it is easy to modify `PropagateUpdate` to propagate those changes together, within a single invocation of `PropagateUpdate`. Second, we have ignored the handling of deletions (`Puts` of NULL values) in the base table. NULL `Puts` to view materialized cells can be propagated in exactly the same way as non-NULL Puts, with tombstones being used to represent deletions in the view in the same way they are used to represent deletions

Algorithm 1: Base Table `Put` with Update Propagation

Input: B: the base table name
Input: k_B: the base key value
Input: c: base column to be updated
Input: v_c: value to be written
Input: t_c: update timestamp
Input: W: write quorum
Output: R: return status (success/failure)

```
// V is a view defined on B
```
1 **if** *c is the view key or a view-materialized column of V* **then**
```
    // Get current view key(s) for row
    // k_B and then perform the update.
    // This Get returns all view key
    // versions it finds, not just the
    // latest one.
```
2 oldkeys ← Get(B, k_B, c_V, W);
```
    // Perform base table update.
    // Can be combined with Get at
    // line 2.
```
3 $R \leftarrow$ Put(B, k_B, c, v_c, t_c, W);
4 return R to the client;
```
    // update propagation happens
    // asynchronously, after Put has
    // returned to the client, and
    // after Get has received view keys
    // from all copies of base row
```
5 **repeat**
```
      // try keys from oldkeys until
      // propagation succeeds
```
6 choose k_v from oldkeys;
7 **until** *PropagateUpdate(V, c, k_B, k_v, v_c, t_c) succeeds*;
8 **else**
9 $R \leftarrow$ Put(B, k_B, c, v_c, t_c, W);
10 return R to the client

in the base table. However, deletion of the view key value in the base table is more complicated, since it should cause the corresponding row to be eliminated from the view. We can handle view key deletions by leaving the corresponding row in the view but marking it as deleted, however, this special case is not handled by Algorithm 2 as shown.

Algorithm 2: PropagateUpdate

```
Input: V: the name of the view on base table B
Input: c: base table column that was updated
Input: k_B: the key of the updated row in the base table
Input: k_v: the view key guess
Input: k_new: the new value in column c
Input: t_new: the new value's timestamp
   // Note: write quorum for all Puts
   // is a majority of the view replicas.
   // get the current live key
1  if [k_live, t_live] ← GetLiveKey(k_v) fails then
2      return failure
3  if c is the view key column of V then
       // write the new row
4      Put(V, k_new, [B, Next], [k_B, k_new], [t_new, t_new]);
5      if k_new ≠ k_live then
6          if t_new > t_live then
               // copy view-materialized cells
               // to the new row
7              CopyData(k_live, k_new);
               // make the old live row stale
8              Put(V, k_live, [Next], [k_new], [t_new]);
9          else
               // make the new row stale
10             Put(V, k_new, [Next], [k_live], [t_new]);
11 else    // c is a view-materialized column
12     Put(V, k_live, [c], [v_c], [t_c]);
```

Finally, and most importantly, Algorithm 2 assumes that update propagations to a given view happen *sequentially*, although they may occur in any order. In practice, this may not be true: even if individual base table update coordinators propagate changes sequentially, different coordinators might attempt to propagate changes to the same view concurrently. Assuming sequential propagation simplifies the presentation of Algorithm 2, allowing us to focus on correct maintenance of versioning in the views. We defer a discussion of concurrent update propagation to Section IV-F, which describes the issues that can arise when updates propagate concurrently, and how they can be resolved.

The first task for `PropagateUpdate` is to find the view row that corresponds to the updated base row. `PropagateUpdate` starts with a guess (k_v) as to the corresponding row's current key, but this guess may be inaccurate, either because the guess is stale or the view is stale. To find the row, `PropagateUpdate` starts with its guess (k_v) and follows pointers in stale rows in the versioned view until it finds the current live row, its target. This is accomplished by the call to `GetLiveKey` (Algorithm 3) at line 1 in Algorithm 2. `GetLiveKey` starts with `PropagateUpdate`'s guess. If the guessed view key identifies the live view row corresponding to the base table row that was updated, `GetLiveKey` is done. If, instead, the guessed view key identifies a stale row, `GetLiveKey` iteratively follows pointers in the versioned view's stale rows until it has identified the live row. It is also possible that `GetLiveKey` will not find the specified key in the view at all. This can happen if the update that wrote that view key in the base table has not yet propagated. In this case, both `GetLiveKey` and `PropagateUpdate` will fail, requiring the coordinator to attempt the propagation again with a different view key guess.

Once `PropagateUpdate` has identified the live row, its behavior depends on whether the view key or a view-materialized cell is being updated. In the latter case, `PropagateUpdate` simply `Put`s the new value into the appropriate cell in the live row (line 12). In the former case, `PropagateUpdate` needs to create a new row in the versioned view. If the newly-propagated view key is older than the key in the current live row, then the new row will be a stale row pointing to the current live row (line 10). Otherwise, the new row becomes the live row. In this case, `PropagateUpdate` needs to copy the values of all view-materialized cells from the old live row to the new one (line 7), and mark the old live row as stale (line 8).

Algorithm 3: GetLiveKey

```
Input: V: the name of the view
Input: k_v: the initial view key
Output: The live row key for the given view key k_v, and
        its timestamp
1  done ← false;
2  repeat
       // Get's quorum is majority
       // of view replicas.
3      [nextkey, nextTS] ← Get(V, k_v, [Next]);
4      if (nextkey ≠ NULL) then   // key k_v exists
5          if nextkey = k_v then   // found live row
6              done ← true
7          else
8              k_v ← nextkey
9      else                  // key k_v does not exist
10         return failure
11 until done;
12 return nextkey,nextTS
```

D. Correctness of Update Propagation

Theorem 1 states that algorithms 1, 2, and 3 together correctly propagate updates to versioned views.

Theorem 1

At the successful completion of each update propagation, the versioned view state will be as described in Definition 3.

Proof: By induction on the number of propagated updates. View assumed to be initialized correctly, so that view state $\hat{V}_0$, which contains no stale rows, is correct. Assume view is correct after $i-1$ updates have propagated, and consider view state $\hat{V}_i$ that results from the propagation of the ith update.

Since $\hat{V}_{i-1}$ is correct and u_i has finished propagating, `GetLiveKey` (line 1 in Figure 2) must have returned, and must have reported the current view key (the live row's key) for the base table row updated by u_i. There are two cases to consider:

Case 1: u_i is an update to a view-materialized column. According to Definition 3, $\hat{V}_i$ should be identical to $\hat{V}_{i-i}$ except possibly for the value of the view-materialized column in the view row corresponding to the base row that was updated by u_i. That cell's value should change only if u_i has a larger timestamp than the cell has in $\hat{V}_{i-1}$.

Since $\hat{V}_{i-1}$ is correct, the live row contains the latest value of each view-materialized cell as of update u_{i-1}. The `Put` operation at line 12 (Figure 2) will update the view cell iff the update's timestamp is larger than the current timestamp in the cell, as required. `PropagateUpdate` makes no other changes to the view in this case, thus $\hat{V}_i$ is correct.

Case 2: u_i is an update to the view key column. Here there are several sub-cases to be considered:

Case 2a: k_{new} does not exist as a view key for this row in $\hat{V}_{i-1}$. In this case, the view key for this row is being set to k_{new} for the first time. Definition 3 requires that $\hat{V}_i$ contain a new row with key k_{new}. If u_i's timestamp is larger than the timestamp of the current live row in $\hat{V}_{i-1}$, then this new row must be the live row, must contain the current values of all view-materialized cells for this row, and must be reachable from all stale rows via their Next pointers. If u_i's timestamp is smaller than the current live row, then the new row must be stale, and its Next pointer must lead (directly or indirectly) to the existing live row.

Since $\hat{V}_{i-1}$ is correct, `GetLiveKey` (line 1 in Figure 2) correctly identifies the key of the current live row. Suppose first that u_i's timestamp is larger than that of the current live row. The `Put` operation (line 4) creates the new live row with key k_{new}. `CopyData` (line 7 ensures that the new live row contains the latest values of all view-materialized cells. Finally, the `Put` operation at line 8 marks the old live row as stale and sets its Next pointer to refer to the newly-created live row. Since the old live row in $\hat{V}_{i-1}$ was reachable by all stale rows, the new live row in $\hat{V}_i$ is reachable by all stale rows by way of the old live (and now stale) row. Thus, $\hat{V}_i$ is correct. If, instead, u_i's timestamp is smaller than that of the current live row, the `Put` operation at line 10 changes the newly-inserted row from live to stale, and makes its Next pointer refer directly to the current live row, which remains unchanged. All other stale rows reach the live row in $\hat{V}_i$ the same way they did in $\hat{V}_{i-1}$. Thus, $\hat{V}_i$ is correct.

Case 2b: k_{new} exists as a stale view key for this row in $\hat{V}_{i-1}$.
As in Case 2a, u_i's timestamp may be smaller than or larger than the timestamp of the view's current live row. If the update has the smaller timestamp, Definition 3 requires that k_{new} remain as a stale view key, with a timestamp equal to the larger of its existing timestamp and that of u_i. This is accomplished by the `Put` operation at line 4. If u_i has the smaller timestamp, this `Put` operation will have no effect, as required. If u_i has a larger timestamp than the existing row, the `Put` will write the larger timestamp, and will make the existing row's Next pointer point directly to the live row. Any stale rows that reached the live row through k_{new} in $\hat{V}_{i-1}$ will continue to do so in $\hat{V}_i$, using the new pointer to go directly to the live row from k_{new}. Stale rows that were not ancestors of k_{new} in $\hat{V}_{i-1}$ will reach the live row in $\hat{V}_i$ exactly as they did in $\hat{V}_{i-1}$. Thus, $\hat{V}_i$ is correct.

If the u_i's timestamp is larger than the live row's, then Definition 3 requires that k_{new}'s row becomes the live row. Line 4 makes it live (overwriting the old Next pointer value in k_{new}'s row), line 7 copies the materialized cells from the old live row, and line 8 marks the old live row stale by pointing it to k_{new}. After these updates, any stale ancestors of k_{new} in $\hat{V}_{i-1}$ still reach k_{new} in $\hat{V}_i$. All other stale nodes reached the old live row in $\hat{V}_{i-1}$ without going through k_{new}. They now reach the new live row (k_{new}) on a path through the old live row. Thus, $\hat{V}_i$ is correct.

Case 2c: k_{new} is the live view key for this row in $\hat{V}_{i-1}$. In this case, Definition 3 requires that $\hat{V}_i$ should be identical to $\hat{V}_{i-1}$, except that the timestamp in the live row should be equal to the larger of its existing timestamp and that of u_i. This is accomplished by the `Put` operation at line 4, which will change the live row's timestamp only if u_i's timestamp is larger. `PropagateUpdate` makes no other changes to the view in this case, thus $\hat{V}_i$ is correct. ■

Theorem 1 shows that `PropagateUpdate` leaves the versioned view in the correct state, assuming that it terminates successfully. However, we have not shown that every relevant base table update will eventually propagate successfully to the view. Since the number of stale rows in a versioned view is finite, and since the stale rows form a tree leading to the live row (there are no loops), any call to `GetLiveKey`, and hence any call to `PropagateUpdate`, must eventually terminate, either in success or in failure. However, one potential concerns is that `PropagateUpdate` could fail repeatedly as it is invoked by the coordinator of a base table update (line 5 of Algorithm 1). Since a versioned view includes rows (stale or live) for *all* view keys that have already propagated, `PropagateUpdate` will eventually succeed as long as at least one of the view key "guesses" used by the update coordinator was written by an update that has already propagated. However, it is conceivable that none of the view key guesses obtained by the coordinator have propagated. Fortunately, if there are several unpropagated view key updates for a given base table row, we can show that at for at least

one of those updates, the coordinator must have a view key "guess" that is *not* from the set up unpropagated changes, i.e., a view key for which `PropagateUpdate` will succeed. This is true because at each replica of the base row, one of those unpropagated view key updates must have been applied first, and therefore must have seen an existing view key that was not written by one of the other unpropagated updates. Since each coordinator retains all of the existing view keys that it sees, the coordinator for at update must have a view key "guess" that will allow its update propagation to succeed. Once it has succeeded, some other unpropagated update must be able to succeed, by a similar argument. Thus, although a base table update coordinator may have to make multiple attempts to propagate an update, it should eventually succeed.

E. Reading from Versioned Views

Stale rows in versioned views are used only during view propagation, to ensure that the correct live row can be found. View reads involve only the live rows, and thus always show the current state of the view. Algorithm 4 shows the algorithm used to `Get` from a view. To simplify the presentation, the algorithm in Figure 4 assumes that only a single view column is being read. However, it is easy to extend Algorithm 4 to allow multiple columns to be read with a single `Get`. For simplicity, Algorithm 4 also assumes that there are no view key updates propagating to the view concurrently with the `Get` operation. We discuss concurrency issues further in Section IV-F.

Algorithm 4: View `Get`

```
Input: V: the view name
Input: k_V: the view key value
Input: c: view column to be read
Output: R: set of values of column c
1  R ← ∅;
   // Get columns c and Next from V.
   // Result is a set of pairs of values,
   // one pair for each view row with
   // view key k_v.
   // Get also returns a timestamp (not
   // shown) for each value.
2  D ← Get(V, k_V, [c, Next]);
3  foreach [cvalue, next] ∈ D do
       // Return only live rows
4      if next = k_V then
5          R ← R ∪ {cvalue}
```

Reading from a versioned materialized view differs from reading from a base table in two ways. First, the view may contain stale rows with the given view key. The view reading algorithm simply ignores such rows, since view versioning is supposed to be transparent to the client applications. Second, as was noted in Section III, a view may contain multiple live rows with the same view key. This occurs when there are multiple rows in the base table with the same view key. Thus, the view reading algorithm returns a set of results, one per view record that matches the specified view key.

F. Concurrency

Our presentations of view update propagation and view reading assumed that these operations occurred sequentially. In practice, this may not be true. In this section, we discuss that additional challenges imposed by concurrency.

Our first observation is that propagation of updates to different rows in a base table can proceed concurrently, without restriction and without the need for any changes to the propagation algorithms. Each base table row is associated with a set of rows in the view, and the sets of rows associated with different base table rows are completely disjoint from one another in the view. Since propagation of an update to a base table row only reads from and changes view rows associated with that base row, propagations of updates to different rows can proceed concurrently without interference.

Similarly, propagations of updates to *view-materialized columns* can proceed concurrently, even if there are several such updates to the same base table row. Such updates do not change the structure of the versioned view, i.e., they do not add rows, change pointers, or change the live row. When propagated, each such update changes at most the value of a single cell in a live row in the view, after finding the live row. In the event that two updates to the *same* view-materialized cell propagate concurrently, those propagations will attempt to update the same cell in the view, and may conflict when they do so. However, such conflicts are already handled by the `Put` operation with which they perform the update. i.e., the updates are serialized in the same way that concurrent updates to a base table cell are serialized. For similar reasons, view `Get` operations can also proceed concurrently with the propagation of updates to view-materialized cells.

This leaves updates to *view key* cells in the base table. These are more challenging to execute concurrently. Since view `Get` operations to not depend on the versioned view structure, we can allow `Get` operations and view key update propagations to happen concurrently by making a small change to our algorithms. When `PropagateUpdate` creates a new live row there is a period of time during which the new row has been created but not initialized, e.g., the values of the view-materialized cells have not been copied from the old live row. To allow concurrent `Get` operations on the view, we must ensure that they do not see such incomplete rows, and that at all times there is at most one accessible live row in the view for each base table row. This can be achieved by making `PropagateUpdate` mark newly-created live rows as inaccessible until they have been fully initialized, unmarking them only after the old live row has been marked as stale. In addition, the view `Get` operation must be modified to wait (spin) in the event that it encounters such an inaccessible row. With those changes, view `Get` operations can proceed concurrently with any update propagation.

Unfortunately, such changes are not enough to allow multiple view key updates (on the same base table row) to proceed concurrently, nor are they sufficient to allow view key updates to propagate concurrently with view-materialized cell updates (on the same base table row). As one illustration of the problems that can arise, consider what happens if there are two updates to the view key in a base row, and those updates propagate concurrently to the view. Assuming that both updates have larger timestamp than that of the current live row, both updates will create new rows in the view and will make them live. Both will attempt to make the former live row stale by pointing it to a newly created live row, but only the propagation with the larger timestamp will succeed. The result will be two live rows, only one of which is reachable from the stale rows. This is clearly inconsistent with the definition of a versioned view.

One way to handle this problem is to introduce a locking mechanism to block unsupported concurrency. Since each base row corresponds to a distinct set of view rows, it is sufficient for propagation operations to lock the key of the base row for which they are propagating an updates. Propagations of view key updates must obtain an exclusive lock, while propagations of view-materialized cell updates can proceed with a shared lock. Locks could be implemented by a separate lock service. Note that these locks only affect update propagation. They do not affect `Get` or `Put` operations on the base table, nor do they affect `Get` operations on views.

An alternative to locking is to take responsibility for update propagation from the base table update coordinators and transfer it to a set of dedicated update propagators, such that a single propagator would be responsible for propagating all of the view updates associated with any given base table row. (This is not the case when update propagation is driven by the update coordinators, since any update coordinator can handle updates for any base table row.) For example, the update coordinators can use consistent hashing of the base row key to assign responsibility for that row's update propagations to one of a set of propagators. The propagator for each row can then easily prevent view key updates from propagating concurrently with other updates to the row.

V. Session Guarantees

Because view maintenance is asynchronous, updates made to a base table are not reflected in the view immediately. If a client updates a base table and then reads from that base table, it can be sure (by choosing appropriate read and write quorums) that its read operation is "seeing" the base table after its update has been applied. However, if there is a view defined on that table and the client wishes to read from the view, it has no way to ensure that its view read sees the view after its update has been propagated.

It is possible to add a mechanism to our system to provide clients with such an assurance by enforcing a *session consistency guarantee*. To do this, we must extend our system with some notion of sessions, which can be created and ended by clients. Specifically, a client can establish a session, issue a set of `Put` and `Get` requests within the context of that session, and then terminate the session when it is finished. Abstractly, a session consists of a sequence of `Put` and `Get` requests issued by a single client.

Suppose that B is a base table and V is a view defined on B. Let o_i be a `Put` operation on B that affects either the view key of V or a view-materialized column of V, and let o_j be a a `Get` operation on V. In this paper we consider the following simple session guarantee:

Definition 4 (Session Guarantee)
For any o_i and o_j (as described above) such that both o_i and o_j are in the same session and o_i precedes o_j in the session, o_j will see a view state at least as late as the view state created by the propagation of o_i.

This guarantee ensures that when a client reads a view, its own updates will be reflected in the view. However, the client still has no way of ensuring that base table updates from other sessions have been reflected in the view.

Our system enforces session guarantees with a very simple mechanism, which assumes that all requests in a session are directed by the client to the same coordinator server. The coordinator associates each pending or incomplete view update propagation with the session of the base table update that triggered the propagation. When the coordinator receives a `Get` operation on a view, it checks whether there any pending or incomplete propagations for that view that are associated with the `Get`'s session. If there are, it blocks the `Get` operation until all such propagations are complete.

VI. Evaluation

The techniques proposed in this paper were prototyped in Cassandra, an open-source multi-master replicated keyed-record storage system originally contributed by Facebook. Using the prototype, we conducted some simple experiments intended to answer several questions:

1) Materialized views provide a means for applications to access data using a secondary key. How does the performance of secondary-key data access using a materialized view compare to that of secondary-key access using Cassandra's native secondary indexing mechanism?
2) View maintenance introduces overhead when base tables are updated. How does the cost of maintaining materialized views compare to the cost of maintaining Cassandra's native secondary indexes?

To address these questions, we ran experiments using a small, 4 node instance of our Cassandra-based prototype. Each node ran on a dedicated physical server with a 2.4GHz dual-core AMD Opteron Processor, 8GB memory and a single 60GB disk, attached through a private 1 Gb network. An additional, separate server was used for clients.

A. Read Performance

To measure read performance, we created a single column family (table) in Cassandra and populated it with 1 million rows, with a total size of about 1 GB - small enough to

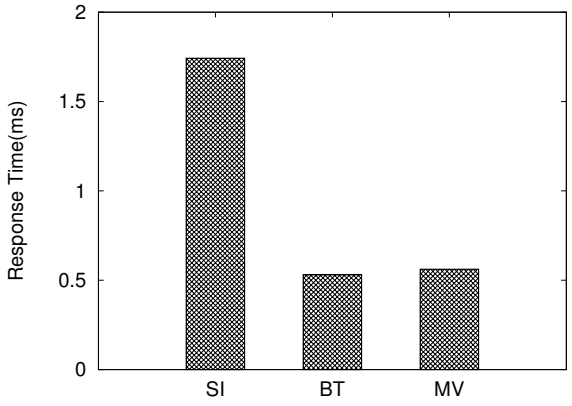

Fig. 3. Read Latency

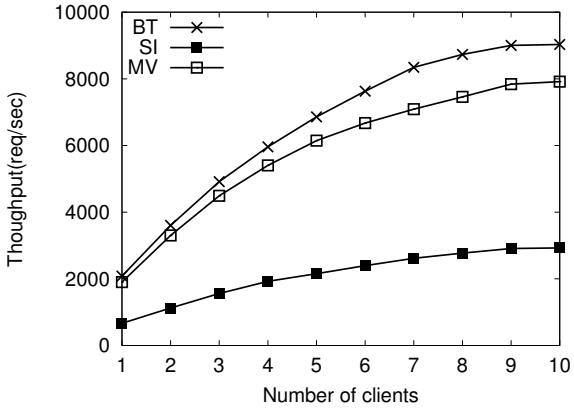

Fig. 4. Read Throughput

fit entirely in memory in our servers. We also defined a materialized view on a secondary key attribute in this table. Secondary key values were unique across the million rows of the table. Both the table and the view were replicated 3 times in our 4-server cluster, i.e., $N = 3$.

We wrote a simple client application that sequentially accesses randomly chosen records from the table, as quickly as possible. The client can be configured to access data in one of three ways

- BT: The client accesses data from the base table, specifying a primary key value for each record accessed.
- SI: The client accesses data from the base table using Cassandra's native secondary indexing mechanism, specifying a secondary key value for each record accessed.
- MV: The client accesses data from the materialized view, specifying a secondary key value (a view key) for each record accessed.

To measure read throughput, we varied the number of concurrent clients. The clients were run for a fixed amount of time (5 minutes), and we measured the aggregate read request rate across all of the clients during the run. To measure read latency, we ran a single client until it had completed 100,000 requests and measuring the total time required.

Figure 3 shows the average latency for `Get` (read) requests under each of the three scenarios. Latencies for base table (BT) and materialized view (MV) access were similar, and about 3.5 times less than the latency of accessing the base table through a secondary index (SI). Figure 4 presents the aggregate read throughput we measured for each of the three types of clients, as a function of the number of concurrent clients of that type. Read throughput for materialized view access (MV) is slightly lower than our baseline, which is base table read throughput (BT). This is because view reading involves reading and filtering out stale rows, in addition to retrieving the desired live row. However, both BT and MV are much less costly than secondary access using Cassandra's native secondary indexing mechanism. This is because Cassandra's secondary indexes are replicated and distributed by *primary*

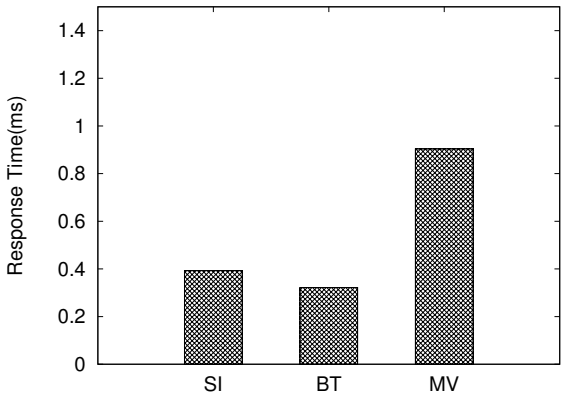

Fig. 5. Write Latency

key, rather than secondary key. This makes it possible for Cassandra to update the index synchronously when the base table is updated. However, reads are relatively slow because the target secondary key must be broadcast to all servers, each of which must check for the record using its part of the index. In summary, materialized views provide a lower latency, higher throughput alternative to native secondary indexes for secondary-key-based read access, although the data may be stale.

B. Write Performance

To measure write performance, we ran similar experiments except that the clients performed base table updates using the record's primary keys. We compared the performance of these updates under three conditions:

- BT: The base table has no materialized views or native secondary indexes.
- SI: The base table has a native secondary index defined on the column updated by the client.
- MV: The base table has a materialized view whose key is the column updated by the client

In the SI and MV cases, each base table update requires view or index maintenance.

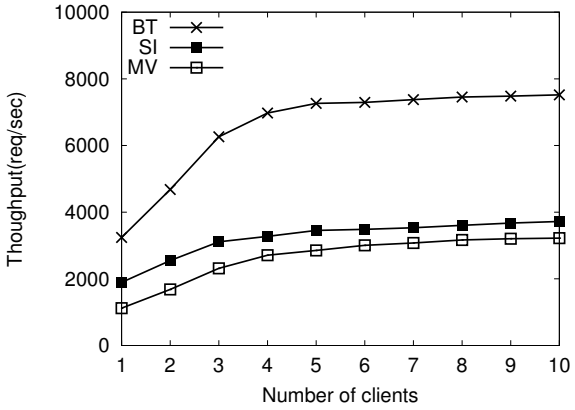

Fig. 6. Write Throughput

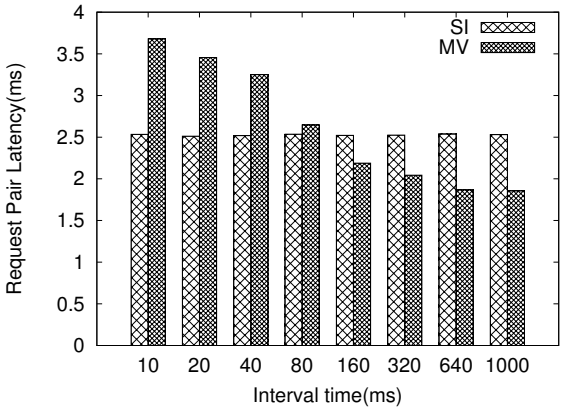

Fig. 7. Average Total Latency of `Put`/`Get` Pairs with Session Guarantees

Figure 5 shows the average latency of `Put` requests in each of our three scenarios. Write latencies in the BT and SI scenarios were similar. Native secondary indexes can be updated quickly because they are partitioned and distributed by primary key. Thus, each server that updates a copy of the base table can also update its copy of index. Write latency in the MV case was about 2.5 times higher. Although most view maintenance activity is asynchronous and does not increase write latency, our update propagation algorithm requires that the updating server read the old value of the view key when a base table record is updated. This accounts for the additional write latency. As noted in Section IV-C, it may be possible to eliminate some or all of this additional latency by combining the `Put` and `Get` operations of Algorithm 2, but our prototype does not do so.

Figure 6 shows the aggregate write throughput we measured in each of the three scenarios, as a function of the number of concurrent clients of that type. Both SI and MV have lower throughput than BT because of the additional costs imposed by view or index maintenance. This experiment represents a best case for the update throughput of MV, because updates were randomly and uniformly distributed over the base table records. As a result, the stale record chains that need to be traversed to find a live view record to update are usually short. However, update chains can grow longer, and update performance can grow corresponding worse for MV, if the update pattern is highly skewed, so that some records are updated very frequently. The extended version of this paper [8] includes and experiment that illustrates the effect of highly skewed updates on write performance.

C. Session Guarantees

In this experiment, we measure the cost of session guarantees on materialized views. We use one single-threaded client to issue 100,000 pairs of `Put` and `Get` requests. There are two versions of the experiment, which we refer to as SI and MV:

SI In this version, there is a secondary index defined on the base table, and each `Put` is followed by a `Get` of the updated row through the secondary index.

MV In this version, there is a view defined on the base table, and the column updated by the `Put` operation is a view-materialized column in the view. The view key is the column on which the secondary index is built in the SI version of the experiment. Each `Put` is followed by a `Get` of the view cell that corresponds to the one updated by the preceding `Put`. In this experiment, we configure the system to enforce a session guarantee - all `Put` and `Get` operations are part of a single session.

These two scenarios represent two alternatives for accessing the table using a secondary key. With the session guarantee, the client can be assured that its `Get` operation will see the effect of the preceding `Put` in the MV experiment, as it is in the SI experiment. Our goal is to compare the costs of these alternatives.

We expect that the cost (in terms of additional blocking latency) imposed by session guarantee enforcement will depend how quickly a client issues a view read after updating the base table. Thus, in our experiment, we introduce a configurable amount of client-introduced delay between the `Put` and `Get` operations in each pair. For each pair, we measure the total time from the start of the `Put` operation to the completion of the `Get` operation and subtract the client-introduced inter-request latency to obtain the total latency of the pair of operations. We report the average total latency over all such pairs.

Figure 7 shows the average total latency of `Get`/`Put` pairs as a function of the amount of client-introduced latency. In the MV experiments, the total latency drops as the amount of client-introduced latency increases. As gap between the `Put` and `Get` increases, it becomes more likely the propagation of the base table update will be complete before the `Get` operation occurs. When this occurs, the coordinator does not need to block the `Get` request when it arrives. The total latency levels off after 640 ms, which indicates that almost all update propagations completed in less time than that.

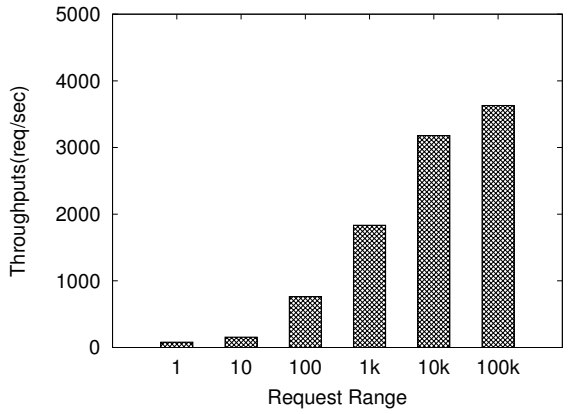

Fig. 8. Effect of Write Skew on Write Throughput

D. Update Skew

One potential concern with our approach to incremental view maintenance is that the cost of maintenance will depend on the intensity of updates to a given base table row. The more updates to a row, the larger then number of corresponding stale rows in the view, and potentially the longer it will take to find the live row when propagating updates.

To measure this effect, we ran experiments in which a materialized view was defined on the base table, and 10 clients concurrently updated the base table for 5 minutes. For each run of the experiment, we chose a specific range of base table keys that would be updated - all clients updated base rows in the same key range. Each client randomly selected keys from the specified range and updated the view key column in the selected base row. In each experiment, we varied the width of the key range from which the clients randomly selected base keys. The range width varied from 100,000 keys down to a single key (meaning all updates from all clients would be directed to a single row it the base table). For each run of the experiment, we report the average base table update throughput (of all clients) over the 5 minute update period. Figure 8 shows the write throughput as a function of the update key range width. As the range narrows, the total write throughput decreased significantly, which suggests that the cost of propagation increases significantly as the updates become more skewed.

VII. Related Work

Materialized views for relational database systems have received considerable attention in the database research community [9], [10], [11], [12], [13] and they are widely implemented [14], [15], [16]. In relational systems, views can be defined using relational queries - a much richer class of views than the simple single-table views we consider in this paper. This gives rise to a variety of view maintenance issues that do not arise in our work, or arise in a very simple form. In relational systems, views may be maintained synchronously or asynchronously [12], [17], [13]. However, in either case updates are normally applied to the views in transaction serialization order. In contrast, we consider a scenario in which updates may be propagated concurrently and out of order, but for a much simpler class of views.

Materialized views also play a role in data warehousing, where a view may be materialized in a different database system than its the base relations. Several algorithms have been proposed to reduce the cost of updating such views in situations in which the base relations must be queried in order to update the view [18], [19], [20]. Such situations do not arise in our work because of the simplicity of our self-maintainable [21] single-table views.

Materialized views are also widely used, in an ad hoc, application-managed manner, by applications running on keyed-record stores. However, most keyed-record stores do not yet support materialized views. One exception is PNUTS, a replicated key-value record store. PNUTS implements a more general class of materialized views than that used in this work, and it can perform asynchronous incremental view maintenance [7]. The views that we consider in this paper correspond to what PNUTS calls Remote View Tables (RVTs), since view records may be located on different servers than the base records on which they depend. However, there is a single master copy of each record in PNUTS, and PNUTS relies on this to serialize updates and to ensure that updates are propagated sequentially and in the correct order when it maintains RVTs.

VIII. Conclusion

In this paper we have considered the problem of providing simple, single-table materialized views in a multi-master keyed-record storage system. Such views are useful because they provide applications with a means of accessing stored records, or parts of stored records, using a secondary key rather than the primary key.

We have presented a technique for asynchronous, incremental maintenance of such single-table views. Our technique is decentralized, meaning that many servers can propagate updates concurrently, and they need not be propagated in serialization order. We prototyped this technique in Cassandra and used the prototype to evaluate its performance. Our experiments show that materialized views can be used to provide fast access to data by secondary key - almost as fast as access using a primary key, and significantly faster than secondary key access using Cassandra's native secondary indexing. However, views may be stale because they are updated asynchronously, and view maintenance introduces a significant overhead when the base table is updated. Thus, our technique is probably best-suited to views for which the underlying base data (especially the view keys) are updated infrequently.

Acknowledgment

The authors wish to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for its support of this work.

REFERENCES

[1] C. Jin, R. Liu, and K. Salem, "Materialized views for eventually consistent record stores," in *Proc. Int'l Conf. on Data Engineering 2013 Workshop Proceedings - Data Management in the Cloud (DMC'13)*, Apr. 2013.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proc. USENIX Symposium on OSDI*, 2006.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *Proc. ACM SIGOPS Int'l Workshop on Large Scale Distributed Systems and Middleware (LADIS'09)*, Oct. 2009.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein *et al.*, "Pnuts: Yahoo!'s hosted data serving platform," *The Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, and A. Lakshman, "Dynamo: Amazon's highly available key-value store," in *Proc. ACM SOSP*, 2007, pp. 205–220.

[6] D. K. Gifford, "Weighted voting for replicated data," in *SOSP*, 1979.

[7] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for vlsd databases," *Proc. ACM SIGMOD*, pp. 179–192, 2009.

[8] C. Jin, R. Liu, and K. Salem, "Materialized views for eventually consistent record stores," Cheriton School of Computer Science, University of Waterloo, Tech. Rep. Technical Report CS-2012-26, Dec. 2012.

[9] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *Proc. ACM SIGMOD*, 1986, pp. 61–71.

[10] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *Proc. ACM SIGMOD*, 1993, pp. 157–167.

[11] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 18, no. 2, pp. 3–19, 1995.

[12] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey, "Algorithms for deferred view maintenance," *Proc. ACM SIGMOD*, pp. 469–480, 1996.

[13] J. Zhou, P.-Å. Larson, and H. G. Elmongui, "Lazy maintenance of materialized views," *Proc. VLDB*, pp. 231–242, 2007.

[14] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin, "Materialized views in oracle," *Proc. VLDB*, pp. 659–664, 1998.

[15] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in SQL databases," in *Proc. VLDB*, 2000, pp. 496–505.

[16] D. C. Zilio, C. Zuzarte, S. Lightstone *et al.*, "Recommending materialized views and indexes with IBM DB2 design advisor," in *Proc. IEEE ICAC*, 2004, pp. 180–188.

[17] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay, "How to roll a join: Asynchronous incremental view maintenance," *Proc. ACM SIGMOD*, pp. 129–140, 2000.

[18] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View maintenance in a warehousing environment," *Proc. ACM SIGMOD*, pp. 316–327, 1995.

[19] Y. Zhuge, H. Garcia-Molina, and J. Wiener, "The strobe algorithms for multi-source warehouse consistency," in *Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[20] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek, "Efficient view maintenance at data warehouses," *Proc. ACM SIGMOD*, pp. 417–427, 1997.

[21] D. Quass, A. Gupta, I. S. Mumick, and J. Widom, "Making views self-maintainable for data warehousing," in *Conference on Parallel and Distributed Information Systems (PDIS)*, 1996, pp. 158–169.