Materialized Views for Eventually Consistent Record Stores

Changjiu Jin [#1], Rui Liu [#2], Kenneth Salem [#3]

[#] *Cheriton School of Computer Science, University of Waterloo*
Waterloo, Ontario, Canada
[1] changjiu.jin@gmail.com
[2] r46liu@cs.uwaterloo.ca
[3] ken.salem@uwaterloo.ca

***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 [1], Cassandra [2], 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 [3] 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 materialized 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. Because of space constraints, the session consistency mechanism is not presented in this paper, but it is described in an extended version of this paper, which is available as a technical report [4].

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], [2], 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 those values, along with their associated timestamps, 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 `get`s or `put`s (or both) may finish more quickly because the coordinator need not wait for as many acknowledgments from replicas. The system includes 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.

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

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 [3]. 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 *multi-set* of values for each requested column, one per view record that matches the specified view key. For example, a `get` of the `Status` column for key `rliu` in the view shown in Figure 1 will return {`open`,`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 as the original row.*

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

Example 2 (Concurrent Update Propagation)
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 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

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.

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.

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 the preceding example.

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.

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. When a client performs a `get` operation on a view, any stale rows with view keys that match the `get` operation's search key are filtered out before the result of the `get` is returned to the application.

C. Update Propagation

In this section we describe how base table updates are propagated to versioned views. Because of space limitations, we give only a high-level description of the approach. A more detailed description of the propagation algorithms, along with a proof of their correctness, can be found in the companion technical report [4].

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
// c_V is the view key column for V
1 **if** *c is the view key or a view-materialized column of V* **then**
    // Get the view key for row k_B
2 $[k_V, t_{k_V}] \leftarrow$ get$(B[k, c_V], W)$;
    // Perform the update of B
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
5 PropagateUpdate$(V, c, k_B, k_v, v_c, t_c)$;
6 **else**
7 $R \leftarrow$ Put$(B[k_B, c], v_c, t_c, W)$;

Each server in the system is responsible for propagating the base table updates for which it acts as coordinator. Algorithm 1 shows how a coordinator performs a base table update. Before performing the actual update, the coordinator obtains the current value of the view key in the row being updated (line 2). This will be used as the coordinator's "guess" as to the row's current key in the view. The coordinator then updates the base table. Although the `get` and `put` are shown as separate operations in Algorithm 1, in practice they can be combined into a single operation that the coordinator requests of all replicas of row k_B. Once the update has been acknowledged to the client, the coordinator then performs view maintenance (Line 5).

To propagate a base table update to the view (method PropagateUpdate in Figure 1), the coordinator must first find the live row in the the view that corresponds to row k_B in the base table. To do this, it first accesses row k_V in the view. If it finds the live row corresponding to k_B, it proceeds to update the view. This will involve either changing the value of one attribute in the live row, or, if the view key has been updated, inserting a new row into the view (with view key v_c), copying attribute values from the old row to the new one, and marking the old row as stale by setting its `Next` pointer to v_c. If the coordinator does not find the live row at k_V, it will instead find a stale row. In this case, it uses a series of `get` operations to follow the `Next` pointer from the stale row to

the live row corresponding to k_B. Once it has found the live row, it updates the view as just described.

In practice, there are several issues that complicate the update propagation technique described in the previous paragraph. One issue is that conflicting updates may not propagate in their correct serialization order, which is determined by the updates' timestamps. However, the `PropagateUpdate` can detect out-of-order updates the same way that they are detected at the base table: each cell in the view contains the timestamp of the latest update to that cell. An update with an earlier timestamp than its target cell in the view is simply ignored. A second issue is that multiple view updates may propagate concurrently. In our decentralized system, in which server propagates the base table updates that it coordinates, there is no mechanism to ensure that view updates happen sequentially. The detailed propagation algorithms presented in the companion technical report are designed to work properly even in the presence of concurrent updates.

Finally, there is the issue of garbage collection. Propagating a base table update results in the creation of a new stale row in the versioned view. A stale row remains potentially useful only as long as its (stale) view key exists in at least one replica of the corresponding row in the base table. If useless stale rows are not deleted, the size of the versioned views will increase without bound. Currently, we use a simple, time-based scheme for garbage collection. It removes stale rows that are older than a specified garbage collection threshold. By setting the threshold conservatively, we can avoid garbage collecting stale rows that are still potentially useful.

V. 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 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., $N = 3$.

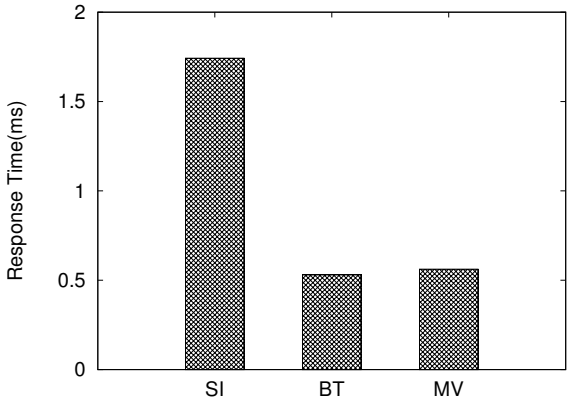

Fig. 3. Read Latency

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*

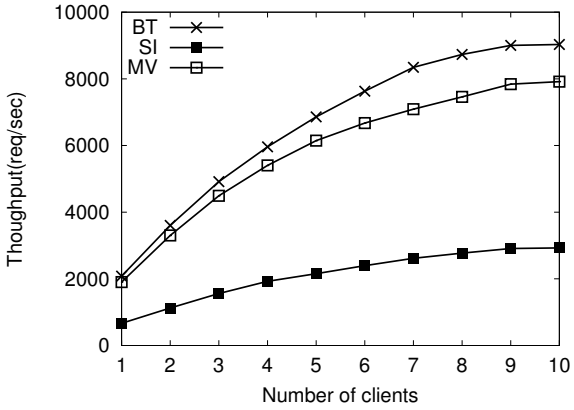

Fig. 4. Read Throughput

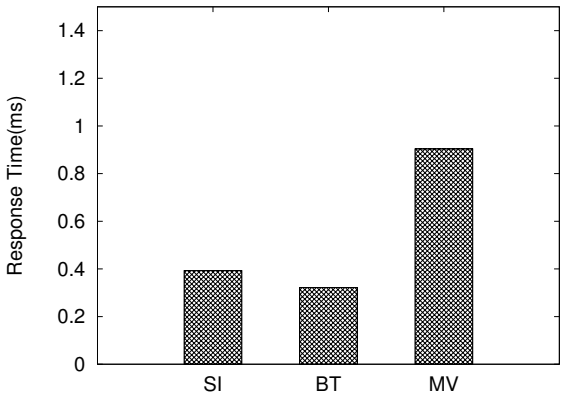

Fig. 5. Write Latency

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.

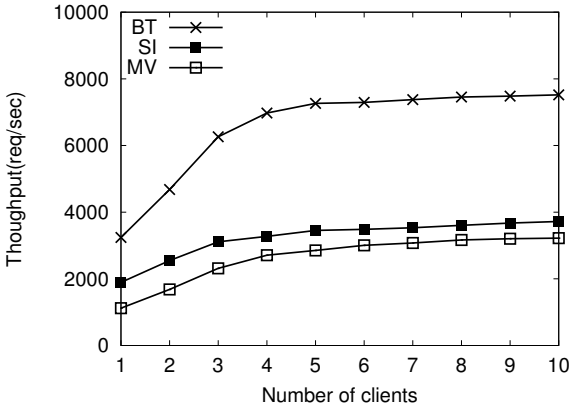

Fig. 6. Write Throughput

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 1, 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 [4] includes and experiment that illustrates the effect of highly skewed updates on write performance.

VI. Related Work

Materialized views for relational database systems have received considerable attention in the database research community [8], [9], [10], [11], [12] and they are widely implemented [13], [14], [15]. 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 [11], [16], [12]. 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 [17], [18], [19]. Such situations do not arise in our work because of the simplicity of our self-maintainable [20] 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.

VII. 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] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.