Asynchronous View Maintenance for VLSD Databases

Parag Agrawal
Stanford University
Stanford, CA, USA
paraga@cs.stanford.edu

Adam Silberstein, Brian F. Cooper,
Utkarsh Srivastava and Raghu Ramakrishnan
Yahoo! Research
Santa Clara, CA, USA
{silberst,cooperb,utkarsh,ramakris}@yahoo-inc.com

ABSTRACT

The query models of the recent generation of *very large scale distributed (VLSD)* shared-nothing data storage systems, including our own PNUTS and others (e.g. BigTable, Dynamo, Cassandra, etc.) are intentionally simple, focusing on simple lookups and scans and trading query expressiveness for massive scale. Indexes and views can expand the query expressiveness of such systems by materializing more complex access paths and query results. In this paper, we examine mechanisms to implement indexes and views in a massive scale distributed database. For web applications, minimizing update latencies is critical, so we advocate deferring the work of maintaining views and indexes as much as possible. We examine the design space, and conclude that two types of view implementations, called **remote view tables (RVTs)** and **local view tables (LVTs)**, provide a good tradeoff between system throughput and minimizing view staleness. We describe how to construct and maintain such view tables, and how they can be used to implement indexes, group-by-aggregate views, equijoin views and selection views. We also introduce and analyze a consistency model that makes it easier for application developers to cope with the impact of deferred view maintenance. An empirical evaluation quantifies the maintenance costs of our views, and shows that they can significantly improve the cost of evaluating complex queries.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*parallel databases*

General Terms

Performance

Keywords

indexes, views, distributed and parallel databases

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

1. INTRODUCTION

In order to support the enormous data sizes and query rates seen at web scale, companies have begun to develop massively distributed, shared-nothing databases. Examples of such *very large scale distributed (VLSD)* systems, designed to scale to tens of thousands of nodes, include our own PNUTS [10], as well as Google's BigTable [6], Amazon's Dynamo [11], and Facebook's Cassandra [15]. These systems typically trade away query expressiveness for scalability and performance, usually only supporting operations on single records identified through their primary key, and in some cases, range scans on primary key. Since data is partitioned over many servers, executing even moderately complex queries such as a group-by aggregation or a join would incur too much latency and be too detrimental to system throughput. Similarly, looking up a record by a secondary attribute other than the primary key requires executing a full table scan in these systems, thereby making it infeasible.

The absence of support for these common queries severely limits the usability of such a database for many web applications. For example, consider a database table of items for sale that is part of a comparison shopping website. The primary key of the table is the listing id of each item, and the website often looks up items by listing id in order to produce a product detail page for each item. However, other types of queries must also be supported. Examples include:

- Find items between $10 and $20.
- Join items with reviews, where reviews are stored in another table.
- Count the number of items for sale in each category.

A natural approach to supporting such queries is to create indexes and materialized views. Indexes and views are important in a traditional database to improve performance, but they are critical in a massively distributed database as they are often the only feasible means of answering join, aggregation and secondary-attribute queries. Note that an index can be viewed as one example of a materialized view (in particular, a projection view sorted by one or more secondary attributes). For this reason, we use the term "views" in this paper to mean both indexes and materialized views, and use the term "indexes" only when we mean the specific kind of materialized view that is used as an index.

Synchronous view maintenance can significantly increase the latency of writing to the database. In particular, in a distributed database, both base tables and views are partitioned across many servers. Unless the base and view tables

are partitioned identically, synchronously updating a given record on one server may require that several view records on other servers also be updated. Such cross-server communication adds significant latency to requests, especially if one of the involved servers is slow or experiencing a transient failure. If we define multiple views, as is often required in real applications, the latency impact is even higher.

Our approach is to *defer* expensive view maintenance work until after the base update completes. In particular, consider an update that is applied to server S_1. While we might perform some local (inexpensive) view maintenance on S_1 at the time of the update, we will defer all work involving other servers S_2, S_3... until we have already returned success for the base update to the client. This ensures that clients experience low latency on their updates. Deferred view maintenance introduces several challenges:

- We must develop a scalable architecture for storing, maintaining and querying views.
- We do not have the luxury of being able to abort the (already committed) client transaction that updated the base table if failures prevent us from updating the view. Thus, we must ensure that views get updated, even in the presence of failures.
- We must define the consistency guarantees provided by views, which may be out of date with respect to the base table in complex ways.
- Since data in our system is replicated to geographically distant datacenters, we need to efficiently replicate the views as well.

In PNUTS, data records are stored in tables, much like relational tables, and tables are partitioned across many servers. In this paper, we explore two mechanisms for maintaining views. The first type, **Remote View Tables (RVTs)**, stores each view in its own PNUTS table, separate from the base tables the views are defined over. Because this view table will be partitioned according to the view's key, not the base table's primary key, view records will likely be stored on different servers than the corresponding base records; this is why these views are called "remote." Remote views are maintained asynchronously, delaying the generation and application of view updates so as to minimize impact on client update latency. We leverage PNUTS' existing asynchronous replication mechanism to asynchronously propagate updates to RVTs.

The second type of view mechanism, **Local View Tables (LVTs)**, keeps view records corresponding to a base record on the same server as the base record. For example, in a view for counting the products for sale by category, each server maintains a local count, grouped by category. To find the total count for, say, the "Toys" category, we must retrieve the partial counts from each server and sum them. Although LVTs are maintained synchronously, the maintenance work is performed on the same server as the original update; and the final aggregation of values across partitions is delayed until query time.

Failure recovery, system testing and performance optimization are already quite difficult in a VLSD system, so we have tried to reuse or extend existing, stable mechanisms in PNUTS whenever possible. In fact, in some cases where two alternative mechanisms are possible, we have preferred the mechanism that is simpler or best uses existing capabilities over a higher performing alternative.

In this paper, we study asynchronous view maintenance over a very large scale, horizontally partitioned distributed database, and make the following contributions:

- Two mechanisms (local and remote view tables) for deferred view maintenance.
- A characterization of the types of indexes and views that can be maintained by these mechanisms.
- A consistency model for views that are maintained asynchronously.
- An experimental evaluation that quantifies the costs of local and remote view maintenance, and their relative benefits for query performance.

The rest of this paper is organized as follows. In Section 2, we present an overview of PNUTS. Then, in Section 3, we define RVTs and LVTs, and show how they are maintained. Section 4 describes which views we do and do not support. Then, in Section 5, we present the rationale for the design choices we have made. In Section 6 we describe our consistency model. Next, in Section 7 we present an experimental evaluation of our techniques. Section 8 discusses related work. Finally, in Section 9 we present our conclusions.

2. SYSTEM OVERVIEW

In this section we provide an overview of PNUTS as well as details pertinent to our view maintenance design (e.g. updates, replication, fault tolerance, consistency). A comprehensive description of PNUTS can be found in [10].

2.1 System Architecture

One of the main goals of PNUTS is elastic scalability, which means that we can add capacity easily by adding more servers. This means that every component on the data path of a request must be horizontally scalable, including our view maintainers.

Figure 1 shows the PNUTS architecture. A data table contains records identified by primary key. The table is horizontally partitioned and stored on multiple *storage servers* using either range or hash partitioning. A storage server typically holds many partitions (e.g., a few hundred 1 GB partitions) and partitions can be moved for load balancing or recovery. Each storage server stores partitions locally in a database; in our implementation we use MySQL. The *query router* accepts requests from clients, determines which storage server currently holds the appropriate partition, and forwards the request to the storage server. If the request is a read request, the storage server's query processor executes the request and returns results by retrieving data from the storage layer (which includes both disk and in-memory cache).

Write transactions are insert, update or delete requests for a single record. If the request is one of these write requests, then:

1. The storage server's update processor forces a log record for the write to the write-ahead log on the *log manager*. At this point, the write is committed.
2. The update processor writes the change to the local storage layer, and returns success to the client.
3. Asynchronously, the log manager delivers the write to record replicas in remote datacenters.

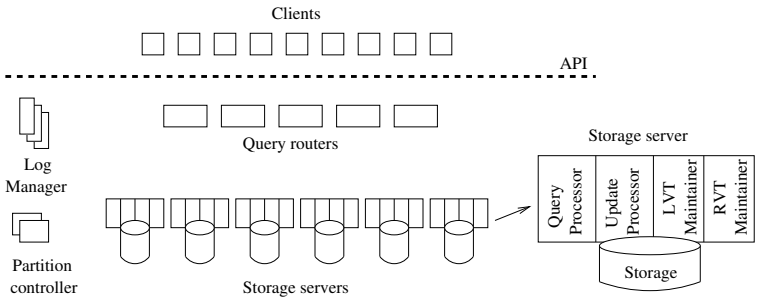

Figure 1: System architecture

Log Manager The log manager serves the dual purposes of reliably *logging* the write and *delivering* it to replicas. The log manager is separate from the storage server to ensure that updates survive even if storage servers, which are inexpensive commodity servers, fail. The log manager writes the log records to two independent disks, ensuring that log records are reliably stored and delivered to remote replicas despite log server failures. There is a cluster of log manager servers at each PNUTS replica, and log records for writes are forced to the local log manager in the datacenter where the write request is executed. Because database writes involve forcing a record to the log manager's disk, we want a log manager to be in the same datacenter as the storage server so that client writes can be low latency.

We have globally distributed users, and thus we must have globally distributed copies of the database so that those users can have low latency access. Also, remote replicas provide disaster recovery. However, cross-region communication is expensive, and we have made the decision in PNUTS to do replication asynchronously to mask cross-region latency for user update operations. The log manager is the component that is responsible for this asynchronous replication. Since we already have a scalable mechanism for asynchronously propagating updates to replicas, it makes sense to reuse this mechanism to asynchronously propagate updates to materialized views, and that is the approach we have taken. Our techniques can be applied to other VLSD databases that have asynchronous replication, including replication mechanisms different than our own.

View Maintainers The RVT and LVT maintainers in Figure 1 are responsible for maintaining the materialized views, and are the main focus of this paper. We describe them in Section 3. The RVT and LVT maintainers are horizontally scalable, so adding more maintainers adds view maintenance capacity. We have decided to run them on storage servers to avoid provisioning and operating separate servers. Although the updates for a given base record will arrive at the view maintainers in the order they are applied to the base record, the asynchronous nature of view maintenance means that views may out of date or temporarily inconsistent with regard to the base table. We discuss these issues in Section 6.

2.2 Data and query model

As a running example in this and future sections, we use the shopping example described in the introduction. For this example, we define two tables; a list of items for sale:

- `Items(ItemID, Name, Description, Category, Date, Price, SellerID)`.

and a list of reviews of items:

- `Reviews(ReviewID, ItemID, Date, Subject, Rating, Text, Reviewer)`.

The underlined attributes are primary keys of these tables; furthermore, `Reviews(ItemID)` is a foreign key for `Items(ItemID)`.

PNUTS tables can be organized as hash tables or ordered tables. In order to support evolving web applications, schemas are flexible: new attributes can be added at any time, and records can be sparse, with a subset of the attributes. To enhance scalability, PNUTS does not support constraints other than primary key uniqueness, such as referential integrity or uniqueness constraints on non-key fields. The application can enforce these constraints if desired. These kinds of feature limitations in PNUTS are shared by other VLSD databases.

PNUTS supports queries against one table at a time. The system provides a call-level API with calls to create, read, update, or delete records. For example, clients can retrieve item #5543 with `get('Items',5543)` and update its price with `set('Items',5543,'Price=100')`. Clients can scan a table, and range scan ordered tables. Although it is possible to layer a declarative/SQL-style query language on top of this API, this restricted set of operations does not require a more complex language. PNUTS does not support join queries against base tables. Single-table queries are significantly easier to plan and execute in a massive scale database. Indeed, one motivation for materialized views is to reduce the problem of join queries to one of looking up records from a single table. PNUTS also does not support group-by-aggregation on base tables (which is another reason to pursue materialized views).

2.2.1 Consistency

PNUTS provides only per-record consistency guarantees, such that updates to a given record are transactional but no ACID-style guarantees are offered across records. For example, we can update item #5543's `Price` and `Category` in a single atomic transaction, but cannot transactionally update the price of all items in the "Electronics" category. This decision improves scalability; we want to avoid managing locks or transaction state across servers. It is also significantly easier to recover from failure when all transactions are single record. One impact of this decision is that range and table scans do not guarantee a consistent snapshot of the data. For example, we might insert item #5111 and not see it in a subsequent scan. This weak consistency model is sufficient for many web applications.

Record Mastery Our consistency model is implemented using *per-record mastership*. Of the replicas for a record, one replica is designated the *master*. All updates to a record are applied first at the master, and then propagated in order to all replicas. The master serializes updates to a record, and prevents conflicting writes to the same record from different clients by holding a lock on the record during the log force and local update operations. The log manager will deliver the updates made to a given record to all replicas in the order they were made at the master. This delivery is guaranteed even if the master fails after forcing the log record, because the log record is replicated within the log manager itself. Failure handling for PNUTS is described in more detail in [10], including handling failures of the log manager, temporarily forcing mastership changes to allow writes after a master fails, and other details. We note that because our materialized views are stored as PNUTS tables, the same recovery techniques that are used for base tables can be used unmodified for views.

Record Timelines Per-record consistency means there is no notion of a table-wide version or snapshot. Instead, we can reason only about a single record's timeline, produced by the sequence of writes to the master copy of the record. Each point on a record's timeline can be identified by the record's *version*. Each replica of a record follows the same timeline, though due to asynchrony, at any time, each replica may be at a different version. The master replica is always at the latest version.

In Section 6, we formalize the per-record model (including types of reads against a record timeline), and then extend per-record consistency to views.

3. MAINTENANCE ARCHITECTURE

We now describe a concrete design for our view maintenance mechanism. Our basic approach is to construct a **remote view table (RVT)**, which is a materialized view table that is maintained asynchronously. In some cases, it is useful to construct a **local view table (LVT)**, where the view records are co-located with the base records. In particular, it is often useful to construct an LVT to serve as a group-by aggregate view over a RVT. As will become clear, the fundamental distinction between RVTs and LVTs is that an RVT's record keys are partitioned differently than the base table on which it is defined, while an LVT's keys are partitioned the same way. In this section, we describe RVTs and LVTs, and how they might be combined. We also discuss the fault tolerance of our mechanisms.

3.1 Remote view tables

A remote view table (RVT) is a table separate from the base tables that stores a materialized view over the base tables. For example, we can create an index over price:[1]:

View 1:

```
CREATE VIEW ByPrice
SELECT Price, ItemID, Name, Category
FROM Items;
```

The RVT is a normal PNUTS table that stores the index entries and is maintained asynchronously by the RVT maintainer. The RVT is partitioned on the view key (`Price,ItemID` in the above example.) Note the base primary key, `ItemID`, is in the key of `ByPrice` to ensure uniqueness. Useful RVTs will usually be partitioned differently from the base table, and the view records will likely be on different storage servers than their corresponding base records. This is the reason we call them "remote view tables."

RVTs are maintained asynchronously to minimize the impact on client update latency. Asynchrony also means RVTs can be stale with respect to the base table. In many web applications, however, some staleness is acceptable. For example, a user may post a review of an item, and must see his own review upon refreshing the page, but it is acceptable if other users do not see that review immediately. This tolerance of staleness already makes it possible for us to replicate base tables asynchronously, and similarly, to maintain views asynchronously to improve client latency.

3.1.1 Maintaining RVTs

When a client initiates an update of the base table, the update processor on the storage server writes the update to the log manager (as described in Section 2.1), but now adds to this update any information necessary (i.e. old field values) to maintain views. Thus, the storage server must know the view definitions[2].

Because the log manager implements PNUTS data replication, the committed update will be delivered to remote replicas. In addition, the RVT maintainer on storage server S_1 subscribes to the log manager, and receives log records for base table updates made on S_1. This approach makes RVT maintainers horizontally scalable; adding more storage servers adds more view maintenance capacity. For each log record, the RVT maintainer generates corresponding updates to the RVT. The RVT is a normal PNUTS table, except the only "client" allowed to update the RVT is the view maintainer. For example, we might change the price of item #5543 from $90 to $100. To properly update View 1, when the storage server writes to the log manager, it must include both the old and new price in the log record. Then, when the RVT maintainer receives that log record, it will delete the `(90,5543)` record from the index and insert the `(100,5543)` record. Note the delete and insert operations are separate log messages. 90 and 100 may map to different partitions on different storage servers, and to improve scalability, PNUTS maintains a separate log per partition. It may be possible to keep a single log and deliver the log message to multiple interested storage servers. However, examining the scalability implications of a single log is beyond the scope of this paper and deferred for future work.

3.2 Local view tables

In the local view tables (LVT) approach, we construct a materialized view over each base table partition. In the shopping example, we can define a view that counts the number of items in each category as follows:

View 2:

```
CREATE VIEW ByCategory AS
SELECT Category, COUNT(*)
FROM Items
GROUP BY Category;
```

[1]Although our system does not use SQL as the language to define views, we will use SQL definitions in this paper to aid clarity.

[2]While we could store definitions elsewhere, we would have to make an extra call to acquire these definitions, before overwriting the old record.

An LVT computes this query for each base partition, and stores the results in the partition itself. Because the `Items` base table is partitioned on `ItemID`, and not `Category`, members of a given category (e.g. "Toys") will be scattered across many base table partitions. To find the count of items in the category "Toys," our query processor would contact each storage server, retrieve the LVT records representing the "Toys" count for each partition, and then take the sum of these partial results. LVTs are kept up to date with the base data, and thus avoid the staleness issue of RVTs. However, the query cost of LVTs is high, since we must access every storage server holding a partition of the base table to compute the entire result.

3.2.1 Maintaining LVTs

For an LVT, the view maintenance is performed as part of the request that updates a base record, ensuring that the view is up-to-date. After the update processor commits the PNUTS-level update request by writing to the log manager, it opens a local transaction in the server's storage layer (e.g. MySQL) to update both the base record and any LVTs. When that local transaction commits, success is returned to the client for the write call. Thus, the update processor encapsulates the LVT maintainer. LVTs add some latency to the client's update transaction; and the amount of latency added increases with the number of LVTs. We can potentially log updates at the storage layer and batch update the LVTs, but this complicates the consistency model (see Section 6.3).

3.3 Combining local and remote view tables

Consider our shopping example again. We might want to compute the number of items for sale at each price. The View 1 `ByPrice` RVT has the items sorted by price, but has not aggregated the count per price value. We can layer an LVT on top of the RVT to materialize the aggregate:

View 3:

```
CREATE VIEW CountByPrice
SELECT Price, COUNT(*)
FROM ByPrice
GROUP BY Price;
```

Because `ByPrice` is an RVT stored in a normal table, we can create an LVT over it as easily as creating a LVT over a base table. Querying this LVT is cheaper than an LVT over a base table: since the partitioning key of the RVT is also the group by attribute(s) of the LVT, then the data will be "pre-grouped" and the aggregate for each group can be retrieved from a single storage server. For this reason, one of the main applications of LVTs in our system is to materialize aggregates over RVTs.

3.4 Fault tolerance

Since view tables are stored as normal tables, we can leverage our existing fault tolerance mechanisms. In particular, PNUTS stores several copies of table data. After a failure, partitions can be recovered from these copies in a way that preserves our consistency guarantees. Since view records are stored in PNUTS tablets, they are replicated for fault tolerance (including LVT records, which are replicated along with their containing base table partition.) Metadata about views (such as view definitions or partitioning) is stored and replicated by the partition controller for fault tolerance. A full discussion of these recovery mechanisms are outside the scope of this paper and are presented in [10].

In addition to making view storage itself fault tolerant, we must ensure that after a base table update the view is definitely updated despite failures. For LVTs, the view maintenance is done as part of the same local (MySQL) transaction that updates the base record, and commits if that local transaction commits. If the local transaction aborts, or the server fails, the locally modified base and view records are marked as "incomplete," and the server waits for the log message from the log manager to continue. Upon receiving this log message, the update processor will again open a local MySQL transaction and attempt to write both the base and LVT records. This process of recovering individual writes from the log manager's log record continues until the update is successfully applied; then the records' "incomplete" marks are removed and processing continues normally.

For RVTs, we utilize the reliability of the log manager. Base table updates, once stored in the log manager, are guaranteed to be delivered (and if necessary, redelivered) to the RVT maintainer until all view updates are committed.

4. VIEW TYPES

The RVT/LVT view mechanisms described in Section 3 can be used to implement a limited but useful set of view definitions: **indexes**, **equijoins**, **selections** and **group-by-aggregates**. We chose to support these view definitions because the first three can be maintained using a single mechanism (index maintenance, as described below); and the fourth is sufficiently important for applications that it justified adding a second mechanism. In this section, we examine how to implement these view definitions, and discuss view definitions that we do not support.

4.1 Indexes

An index is a type of view that is a projection and reordering of a base table. Recall View 1, an index on price. We can implement `ByPrice` as an RVT `ByPrice(Price, ItemID, Name, Category)`, with one index record for every item, and items with the same price distinguished by the `ItemID`. The RVT would be ordered by price. Although only `Price` and `ItemID` are strictly necessary to make the index usable for queries (i.e. use `ItemID` to query `Items` for additional attributes), including `Name` and `Category` allows us to satisfy some queries using the index only. If a query that is answered by an index requires attributes that are not stored in the index, we look up the base record, using the primary key which is stored in the RVT record.

Whenever a base table record is written, the view is updated. An example is shown in Figure 2. If item #5543 is inserted into `Items`, a new record is inserted into the RVT. As shown in Figure 2(a), the sequence of events is that the RVT maintainer receives the log record for the base table update, and (acting as a PNUTS client) writes a record with key (90,5542) into the RVT table (since `(Price,ItemID)` is the key of the RVT table). The storage server holding the appropriate RVT index partition generates a log record reflecting the index write, just like any other write. Now consider a case where item #5543's `Price` changes from $90 to $100. The RVT maintainer will delete the $90 entry from the RVT table, and insert a $100 entry into the RVT. This process is shown in Figure 2(b). Because the RVT partition for the $90 record is likely different than the partition for

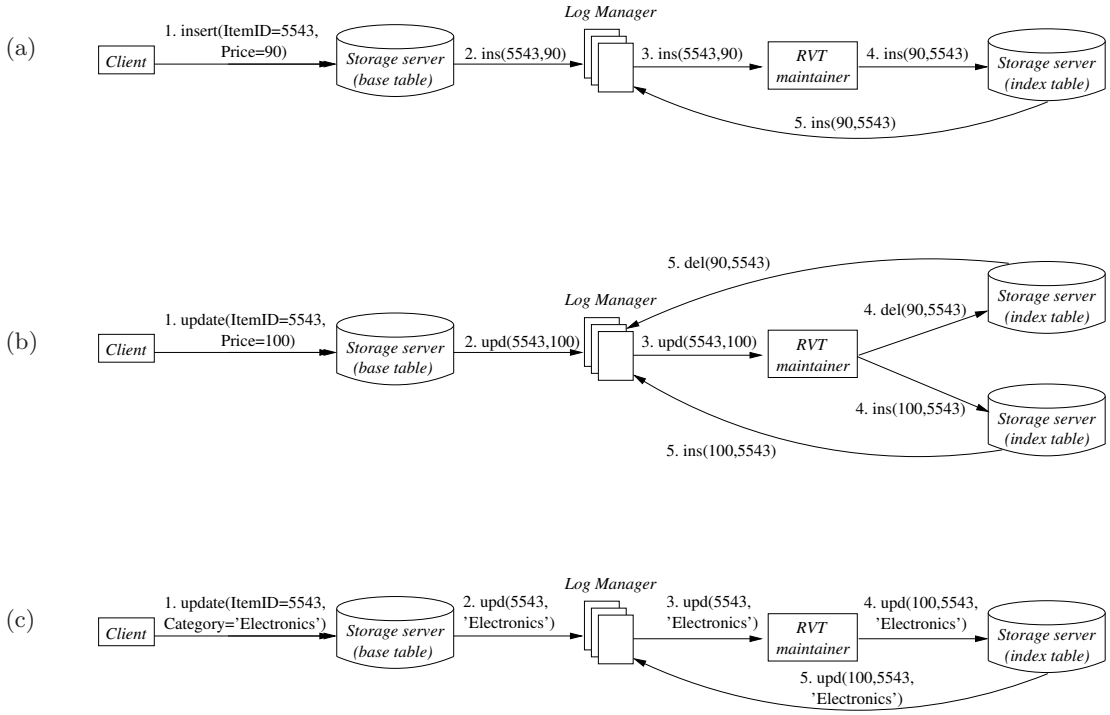

Figure 2: Maintaining an index: (a) a new base record, (b) updating the index key, and (c) updating a non-key attribute. (For clarity, query routers are omitted.)

the $100 record, the RVT maintainer must do two writes to the RVT table, and this results in two separate log records sent to the log manager from two different storage servers. Of course, if the update to the base record is to a field other than the index key field, then we only need to do an update to the index record. For example, as shown in Figure 2(c), if item #5543's category changes, item #5543's view record is updated to the new category.

We query the index with a range scan. For example, to find all items with price $100, we would scan the index table, starting at the first $100 item and continuing until we found the last. Because the RVT is ordered, the $100 items would be clustered (as would every other price.)

It is also possible to implement an index as an LVT, by constructing a local index over each base table partition. However, the cost of querying an LVT by accessing all partitions on all storage servers will be much higher than querying the RVT and accessing only the necessary tablets.

4.2 Equijoins

We might want to join the `Items` and `Reviews` tables. We do so by co-locating (or clustering) joining records in the same partitions, but not actually joining them until query time.

View 4:

```
CREATE VIEW ItemsReviews AS
(SELECT ItemID, Name, Description, Category, Date,
Price, Seller
FROM Items)
UNION
(SELECT ItemID, ReviewID, Date, Subject, Rating,
Text, Reviewer
FROM Reviews);
```

This equijoin view can be implemented as an RVT defined as two `ItemID` indexes (one over `Items` and one over `Reviews`) stored in the same RVT table. Our flexible schemas and support for sparse records allow us to materialize this view in one table, even though the two joined relations have different schemas; each view record will only have values for the columns of the base table it came from.

The RVT is sorted on `ItemID`, and we specify that the system not split items with the same `ItemID` into different partitions. Thus, the RVT clusters all `Items` and `Reviews` records with the same `ItemID` in the same partition. The view is maintained in the same way as the indexes of Section 4.1: whenever there is a change to either `Items` or `Reviews`, we update the `ItemsReviews` RVT.

To query the equijoin view, the query processor scans the RVT partition for a given `ItemID`, and joins the `Items` and `Reviews` records it finds. We do not materialize the joined records; although this would save query time, it increases the complexity of the view maintenance. We prefer to have a single mechanism (effectively, "index maintenance") rather than two mechanisms ("index maintenance" and "join view maintenance"). Recall that we often choose a simpler, easier to test approach versus a more complex option.

Note that we materialize the outerjoin. For example, if item #5543 has no reviews, it is still stored in the equijoin view. While we could save space by storing only the inner join, we would pay with extra maintenance cost. For example, when the first review for item #5543 is inserted into the `ItemsReviews` RVT, we would also have to look up item #5543 itself in `Items` to add it to the view. We want to avoid the need for such expensive lookups in our view maintenance.

We can create a join view that joins three or more tables, as long as all of the tables join on the same attribute. For example, we cannot join `Items` and `Reviews` with a table

- `ReviewerProfiles(ReviewerID,Name,Description)`

since `ReviewerProfiles` cannot be partitioned by `ItemID` and `Items` cannot be partitioned by `ReviewerID`. We can also create self-equijoins, that is, join a table with itself using an equality condition (e.g. `SELECT * FROM R,R WHERE R.A=R.B`). In this case, if we join a table with itself C times on C different attributes, each base record will appear in the view C times.

LVTs are not appropriate for join views. Because we are combining two base tables with likely different primary keys, the base tables are partitioned differently. An LVT must be partitioned in the same way as the base table(s) on which it is based. Since we cannot partition an LVT according to the two different partitioning schemes of the two base tables, we cannot construct an LVT that provides any benefit for joins.

4.3 Selection views

Selection views contain a subset of the records of a base table. For example, if "Electronics" devices are frequently accessed, then we might create:

View 5:

```
CREATE VIEW ElectronicsItems
SELECT *
FROM Items
WHERE Category='Electronics';
```

Such a view would be implemented as an RVT and maintained like an index (Section 4.1), except only updates to `Items` records with `Category='Electronics'` would generate view updates. Although a selection view could be implemented as an LVT, the cost of querying all storage servers to collect view records is high and reduces or eliminates the performance benefit of having a small subset of the base table's records for easy access. We include selection views for completeness, although such views are not widely used in our applications.

4.4 Group-by-aggregation views

Group-by-aggregation views are the one view type we support which cannot be easily maintained by the RVT index maintenance mechanism of Section 4.1. That mechanism treats a view record as a transformed replica of a single base record. In aggregation, multiple base records contribute to a view record, requiring us to maintain state in the view record about each of the base records in order to properly update the view record. However, aggregation is so important to web applications that we were willing to accept the system complexity of a second mechanism. In particular, we use LVTs to support aggregation, either by maintaining an LVT over a base table or by maintaining an LVT over an RVT (such as an index or equijoin view).

Consider a view which computes the average rating for an item:

View 6:

```
CREATE VIEW AverageRatings
SELECT ItemID, SUM(Ratings), COUNT(Ratings)
FROM Reviews
GROUP BY ItemID;
```

We could build an LVT directly on `Reviews`. For example, in one partition of `Reviews` there might be five reviews of item #5543. In that partition, we would keep an LVT record `(5543,17,5)`. In another `Reviews` partition, there might be 3 reviews of #5543, and an LVT record `(5543,7,3)`. Whenever we have an insert, delete or update of a review for item #5543 in a `Reviews` partition, we synchronously update the LVT record for that partition.

We support the SQL92 aggregates COUNT, SUM, AVG, MIN and MAX. At query time, each aggregate can be computed from an LVT by combining the partial values from each partition; the exception is AVG which must be computed from COUNT and SUM. Maintaining the aggregates is straightforward: whenever a base record changes we update the aggregate as appropriate. One subtlety is that for MIN and MAX, after an update we may have to scan the partition to find the new MIN or MAX; we do this scan as part of the update transaction to ensure consistency.

We can avoid querying all partitions for their LVT record for item #5543 by instead building the LVT on top of another RVT. For example, View 4 (equijoin of `Items` and `Reviews`) already clusters reviews by `ItemID`, and we could define View 6 over the `ItemsReviews` RVT instead of over the `Items` base table. Then, only the single aggregate record corresponding to the item #5543 group needs be retrieved.

Note that instead of maintaining an LVT, at query time we can compute the average rating over the `ItemsReviews` RVT by scanning all the records in the group and computing the aggregate. This is a physical design decision: do we choose to save maintenance cost by aggregating at query time, or do we choose to save query cost by pre-materializing the aggregate when we update the `ItemsReviews` table? The decision is left to the application designer.

It is possible to use an RVT to materialize the aggregate without an LVT. For example, we could create an RVT where each record represents one `ItemID`, and stores the average `Rating` for all of the `Reviews` records for that `ItemID`. We chose not to take this approach, which would require a new RVT maintenance mechanism (different than the existing mechanism described in Section 4.1). In order to consistently maintain the aggregate, we need to know the version numbers of the base records contributing to a view record (see Section 6). Once we must maintain state for all the base records, it is simpler to create an LVT-over-RVT, and avoid adding a new mechanism specifically for RVT aggregates.

4.5 Unsupported view definitions

There are several view types that we have chosen not to implement, because they cannot be both maintained and queried cheaply.

- *Joins of three or more tables not all joined on the same attribute.* As described above, unless the records to be joined are all co-located in the same view partition (defined by a single join attribute serving as the partitioning key) queries become expensive distributed joins.
- *Joins that are not equijoins.* We partition joining records by the join attributes, so we can only easily find joining records that have the same value for the joining attribute. Maintaining general joins is known to be expensive, especially in distributed databases [16] and we cannot afford the cost at our scale.
- *Full SQL99 aggregate functions.* The complexity of properly maintaining functions such as percentile or standard deviation led us to defer these functions to future work.

5. DESIGN RATIONALE

Two main considerations drive our architecture for maintaining views in a VLSD system. First, we observe that there is a tradeoff between *client latency*, *system throughput* and *view staleness*. For example, we could synchronously update the base table and the views, eliminating view staleness but adding client latency on base table writes. Similarly, we could group-commit updates to view tables to improve system throughput, trading off an increase in view staleness as view updates wait longer to be applied. Given that our system supports end-user web pages, we need to minimize client latency; thus, our design seeks the best tradeoff between system throughput and view staleness. The second consideration driving our design is system complexity. A VLSD database is already quite complex, and we aim for system simplicity unless the performance gains of a more complex alternative are truly significant.

5.1 Choice 1: Should clients or the system maintain views?

Clients can maintain views by creating their own view tables and updating them whenever they update the base table. We could even encapsulate this functionality in a client library so it is not reimplemented for every application. We must ensure that the view is updated, even if the client fails. The client could maintain a redo log of view updates to be performed, possibly storing this log in our system as a regular table. The redo log would be replayed after a failure to ensure all view updates occurred. While this is in general a feasible approach, we observe that PNUTS already keeps a log, and has mechanisms to redo logged updates, so it is redundant for the client to keep a second redo log. As we do not wish to expose the log directly to clients (hiding the log is a common design choice in DBMSs), it makes more sense for the view maintenance to be implemented as a system mechanism rather than a client mechanism.

5.2 Choice 2: How eagerly should we update views?

There is a spectrum of choices for when to update a view:

- **Synchronous view updates:** Update the view as part of the client's base table transaction. LVTs use this approach, but we avoid any synchronous cross-server view maintenance to reduce client latency.
- **Lazy updates:** For example, for View 4, if we insert a new item into `Items`, we return success to the client, and later insert the item into the `ItemsReviews` view. Our RVT mechanism follows this approach.
- **Batched lazy updates:** Like lazy updates, except we batch multiple view updates. For example, we might collect several new items before inserting them into the `ItemsReviews` view to improve throughput via group commit. We decided against this approach to minimize view staleness. We can always improve throughput by adding more servers.
- **Periodic view refresh:** For example, once a day we could drop and recreate `ItemsReviews`, and then scan `Items` and `Reviews` to repopulate the view. Using efficient table scan and bulk load techniques, we could achieve very high throughput, but the view staleness is highest in this approach. Moreover, analysis of one of our anticipated workloads shows that only a small fraction (5-10%) of the base table is likely updated in a given day, meaning that the effort to rebuild the view for the remaining 90-95% of the base table is repeated across refreshes.

5.3 Choice 3: How do we replicate view tables?

In our approach, RVTs are normal tables except that they are updated by the RVT maintainer and not clients. Similarly, LVT records are normal records stored inside base table partitions (with a special key to distinguish them from other base records). We decided to store RVTs and LVTs using normal PNUTS mechanisms so that we could reuse the existing replication and failure recovery mechanisms. If a storage server fails, we can recover its lost partitions (both base and view) by copying them from another database replica, and replaying any updates in the log manager that have not yet been applied to this replica.

Note that this approach introduces some inefficiency in the log shipping process, as we must send log records for both the base updates and the view updates to remote replicas. An alternative approach would be to implement special, non-replicated views. For example, when an update to `Items` arrives at a replica, that replica updates its copy of `ItemsReviews`, but the `ItemsReviews` updates are not themselves shipped between replicas. However, this requires two mechanisms for storing and recovering tables (one for replicated base tables and one for special view tables), whereas our approach uses the same mechanisms for both kinds of data. Since replication and recovery are already quite difficult to implement and test in a VLSD system, we opted for system simplicity over improved performance.

6. CONSISTENCY MODEL

Because our system has both asynchrony and replication, it can be hard for application developers to reason about the state of the data in views they are accessing. We have defined a consistency model for PNUTS that lets us hide many of the complex details for replication of base tables [10]. In this section, we review our timeline consistency model for base tables, and then extend it to views.

6.1 Record-Level Consistency Model

As discussed Section 2.2.1, PNUTS provides per-record consistency and, to enhance scalability, does not support multi-record transactions or ACID guarantees. We now discuss the PNUTS consistency model.

Each record is named by its primary key r. Each replica of r (designated r_i) has an associated version number σ_i. Writing a record updates the latest version. When reading a record, an application can request:

- `ReadAny(`r`)` - read and return any version of r.
- `ReadCritical(`r`,`σ'`)` - read and return any version of r such that the returned record r_i with version σ_i is at least as new as σ' (e.g. $\sigma_i \geq \sigma'$).
- `ReadLatest(`r`)` - read and return the latest version of r.

`ReadAny` is the fastest option as the local replica can always be used, and in many cases a stale copy is acceptable. `ReadCritical` can be used to ensure that a particular user does not see data "move backwards" in time, so that we return a version at least as new as the last version they read or wrote. `ReadCritical` may require accessing the master replica if the

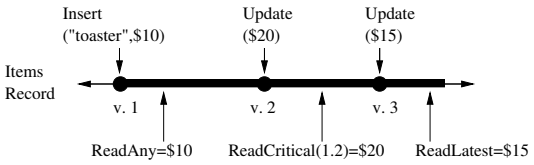

Figure 3: `Items` Record Timeline

Figure 4: Records in `Items`, `Reviews` and `ItemsReviews`. Arrows represent dependence between view and base records.

local replica is too old. `ReadLatest` always accesses the master replica. Thus, these calls allow the application to trade staleness for performance.

Figure 3 shows a sample sequence of read/write operations on a (simplified) `Items` record. Suppose a user Stella inserts a toaster for sale for $10, and later updates the price to $20 and then $15. Each write increments the record version number, from $v.1$ to $v.2$ and then $v.3$. Both Stella and a potential buyer, Bill, do reads on the record. For example, Bill may use `ReadAny` in his initial search for toasters and get version $v.1$. Stella may use `ReadCritical`($v.2$) after updating the price the first time to check if her write succeeded. Bill, who continues to check on the item, may switch to using `ReadLatest` to ensure he sees the latest price.

6.2 Maintaining view consistency

Our goal is to extend the base consistency model to include views. Then, we can hide the details of which, if any, view replica is used to serve the data, and minimize any penalty to consistency of using the view instead of the base table. How does the notion of per-record consistency carry over to the views case? Unlike our base consistency model, where the timelines of all records are independent, with views multiple records are now connected: view records are connected to the base records over which they are defined. However, we can extend our consistency model in a very natural way, by defining the timeline of a view record in terms of the timeline of the base records on which it depends. We now formalize this connection, and then argue that maintaining consistency for views is not too expensive.

6.2.1 Base and view record relationships

When we inquire about the version of a view record, we are really inquiring about the version of the underlying base data. Suppose Bill buys the toaster in our shopping example and inserts a positive review of it in the `Reviews` table. The toaster then breaks and he updates his review to be negative. He then wants to read all reviews of the toaster, a query best answered by the `ItemsReviews` view (View 4). When he reads the view, we had better respond with his updated review, or he will become confused. Thus, we want Bill's `ReadCritical` operation against the view to reflect his update to the base table.

If the information in a view record vr comes from a base record r, we say that vr is *dependent* on r while r is *incident* on vr. View definitions are either *one-to-one* or *many-to-one*, depending on how many base records may be incident on a given view record. Indexes are one-to-one, as every index entry has a single incident base record. Selection views are similarly one-to-one. Group-by aggregates are many-to-one; each view record is dependent on multiple base records. While joins may seem to be many-to-one, in our framework they are one-to-one, since our join views are really co-located indexes. An example for the `ItemsReviews` join view is shown in Figure 4.

It is possible to construct views on top of views. We have already described constructing an LVT on top of an RVT. More generally, any view could be constructed on top of an incident table that is a base table or another view.

6.2.2 Cost of view maintenance

We now argue that the set of view definitions we support are efficiently maintainable. Supporting general views can be quite expensive, as maintaining the view after a base table update can require executing complex queries against the base tables or against auxiliary structures. However, we have limited the set of views that we support, and those views can be maintained after a base table update using only the old and new value of the updated base record.

Consider a base record br that is updated.

- **Indexes** - The record br corresponds to a single index view record vr. The view record vr has as a key the value of some secondary attribute a of br. If a is updated, then we must delete vr and insert a new view record keyed by the new value of a. If some attribute besides a is updated, then we only need to update vr. Thus, an update to br causes at most two updates to the view, and those updates only require knowledge of the old and new values of br.
- **Equijoins** - Equijoins are maintained in the same way as indexes. Thus, as with indexes, a base table update requires at most two view updates, and making those updates only requires the old and new value of the updated base record.
- **Selection views** - If br did not previously pass the view's selection condition but is updated to pass the condition, then a record must be inserted into the view. If br did pass the condition but is updated to no longer pass the condition, a view record must be deleted. In either case, a single view update is required, and only br is required to properly make the update. Of course, if br did not pass the condition before and after the update, or did pass the condition before and after the update, no view update is required.
- **Group-by-aggregation views** - For SUM and COUNT aggregates, the view can be updated by knowing only the change in value of the aggregated field of br, and the value of the grouped attribute. Since AVG can be maintained using SUM and COUNT, the same argument holds for AVG.

The log record for br's update includes the old and new value of the updated attribute, any grouping information

for aggregate views, and other base record attributes stored in the view but not modified by the update. Thus, when *br* is updated, we can examine just the log record in order to maintain each view. View maintenance cost therefore scales with the number of views and base updates, but does not depend on the size of the base or view tables.

The exceptions to the above argument are the MIN and MAX aggregates. If we update a record that was the MIN (or MAX) to have a larger (respectively, smaller) value, we must query the base table to recompute MIN (or MAX) which may be the same or a different value as before. However, note that because in our system aggregates are implemented as LVTs, and LVTs are maintained per partition, only a single partition must be scanned to find the new MIN (or MAX), which is cheaper than scanning the entire relation.

6.3 Read consistency for views

We can either read a single view record, or conduct a range scan of the view.

6.3.1 Single record reads

On a per-record basis, our views provide the same consistency guarantees as base tables (Section 6.1). Consider a view record that has a one-to-one relationship with a base record; for example, an index entry in `ByPrice` (View 1). We can define the read operations against an RVT or LVT view record vr in terms of its incident base record r:

- `ReadAny(`vr`)` - return a version of vr that corresponds to any version of r.
- `ReadCritical(`vr`,`σ'`)` - return any version of vr such that the returned version vr_i corresponds to a base record version r_i that is at least as new as σ' (e.g. $\sigma_i \geq \sigma'$).
- `ReadLatest(`vr`)` - read and return the version of vr the corresponds to the latest version of r.

Recall the view maintainer is the only "client" that may write a view record, and only to process an incident base record write. Therefore, there is no need for one-to-one view records to have their own version numbers. Instead, each one-to-one view record inherits the version number of its incident base record. With this inheritance, we support the above read levels by treating view record replicas like base record replicas.

The situation is more complicated for many-to-one views (like the per-`Category` aggregate in View 2). When we read a given view record, there are multiple relevant base record versions (one for each incident base record). If we do not care about the versions of any of these base records, `ReadAny` is sufficient. On the other hand, we may want to specify the version of one or more base records. In this `ReadCritical` case, we specify a vector containing the subset of incident base record versions we care about and require that the aggregate value incorporate base records at least as new as the specified versions. For non-specified base records, the more relaxed constraints of `ReadAny` still apply. In our implementation, where we compute the aggregate over an index at query time, the base record version numbers are readily available in the index records. In other implementations it is possible to achieve the same effect with version vectors, Lamport causal clocks, etc.

Consider the costs of each read level with RVTs. `ReadAny` returns a view record with a version that is guaranteed to be valid (because it is based on the version of an underlying base record, which is also guaranteed by PNUTS to be valid.) For views, just like base table replicas, the record we wish to read may not be present yet, and thus a `ReadAny` may return an empty record (which we regard to be a valid version zero). `ReadCritical` returns the view record unless it is too stale, at which point we must do an additional read of the corresponding base record master to get a satisfying version. This fallback to the master is the same mechanism used to implement `ReadCritical` for base tables. `ReadLatest` has a particularly high cost for RVTs. When examining an RVT record, it is impossible to know whether that record has the latest value without examining the master copy of the base record. Therefore, every `ReadLatest` access of an RVT would also require accessing the base table. Because of this expense, `ReadLatest` should be used with caution for an RVT. Note, however, that this process is still cheaper than scanning the base table looking for records that match the query.

Providing read guarantees is straightforward with LVTs. The LVT is always up-to-date with respect to the local replica of the underlying base table. Thus, any request for `ReadAny`, `ReadCritical` and `ReadLatest` can be satisfied from the LVT if its underlying base table replica has the appropriate version. If the underlying base table replica does not have the correct version, the request must be forwarded to the master copy, just as we would have to do if we were reading the base table partition directly. `ReadAny` and `ReadCritical` calls for an LVT-on-RVT are as cheap as providing those calls for the underlying RVT that the LVT-on-RVT is based on. Providing `ReadLatest` for an LVT-on-RVT is expensive, because providing `ReadLatest` for the underlying RVT is expensive as discussed above.

6.3.2 Range scans of the view

Range scans over base tables provide limited guarantees: every returned record is a valid version, but some records may be missing and some may be stale. Range scans over views provide the same limited guarantees, and any returned view record is dependent on some base record. There are two challenges with range scanning views that are not present for base tables. These stem from view maintenance that requires an insert and delete of view records. An example is when item #5543's `Price` changes from $90 to $100, and we have to delete the $90 entry from the `ByPrice` index and insert the $100 entry. The delete and the insert propagate asynchronously and independently. Thus, at a given time, item #5543 may appear twice in the index (if the insert arrives before the delete) or zero times (if the delete arrives before the index.) We can mask these anomalies from the client. Since each view record carries the base record's primary key, we can easily filter out multiple view records that correspond to a single base record when range scanning a view. To mask the situation where zero view records appear, we can retain tombstones when deleting view records. If we range scan a view and see a tombstone but no corresponding regular view record, we know we have encountered the "delete followed by insert" case. If desired, we can look up the base record (using the primary key stored in the tombstone) to include the record in the scan results. Of course, we need to garbage collect old tombstones. We have not yet implemented the tombstone approach in PNUTS.

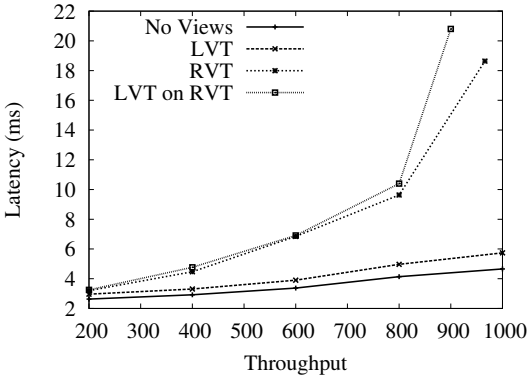

Figure 5: View mechanisms

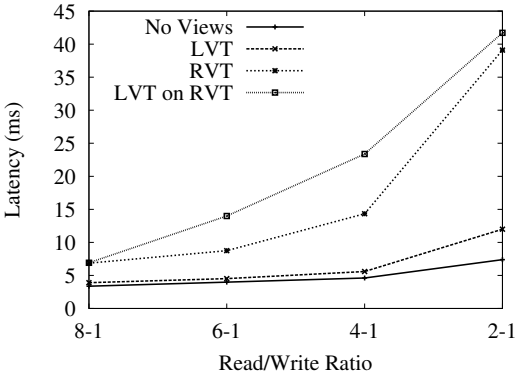

Figure 6: Read/write ratio

7. EVALUATION

We have run a series of experiments to evaluate the costs and benefits of materializing views in PNUTS. We have added a prototype of our view maintenance mechanisms to the production PNUTS code. First, we measure the *view maintenance cost* in terms of three metrics: 1. the impact to the **latency** of the client queries, 2. the impact to the **throughput** of client queries, and 3. the average **staleness** of the view. Second, we measured the **time to query views**, depending on the type of view. In summary, our results show that maintaining views adds load to the system, which decreases somewhat the throughput clients can achieve. However, the impact to client latency is reasonable. Our query results show that querying views is several orders of magnitude cheaper than scanning the table, the only viable alternative to views for join, aggregation and secondary selection queries. We also compare strategies for querying index, group-by-aggregate and join views to find the most efficient approach.

7.1 Experimental setup

PNUTS is implemented in C++ and runs on either FreeBSD or Linux; our experiments used FreeBSD 6.3. Data partitions were stored in MySQL 4.1.23. Our prototype of the RVT maintainer and LVT maintainer were also implemented in C++, and used MySQL to store view definitions. We used synthetic data in our experiments in order to directly control the data distribution and selectivity of our queries. For some experiments, we ran on a minimal setup with one storage server, one router, and one log manager machine; for others, we used three storage servers. We chose this setup to directly isolate the cost, per storage server, of join maintenance. We ran one experiment where we increased the number of storage servers, and observed that maintenance cost per storage server remained constant. In ongoing work we are measuring performance for larger system instances.

7.2 Evaluation of View Maintenance Costs

We loaded each storage server with 10GB of data, consisting of records of size 4KB each. The MySQL buffer pool was 2GB. We chose our read workload such that 90% of reads are served from cache (to simulate a real workload where there would be a working set). All the views we used in this evaluation were "thin" (containing only the indexed attribute and the record primary key) indexes on integer attributes.

In all our experiments, we are I/O bound. Therefore, we can directly see the effects of maintaining views, which add additional I/Os for every base table write.

7.2.1 Varying view type

The first experiment measures the cost of our different view mechanisms for a realistic read:write ratio of 8:1. Figure 5 plots client actual throughput vs. latency for the cases of *no views*, *1 RVT*, *1 LVT*, and *1 LVT on RVT*. (Actual throughput is less than the client's attempted throughput when the system is saturated.) At lower throughputs all alternatives have similar latencies, meaning view maintenance has a negligible impact on client performance. As throughput increases *no views* and *LVT* show a gradual increase in latency, while *RVT* and *LVT on RVT* show a steeper increase. Although the LVT and RVT cases both write the same amount of additional data onto the storage unit, LVTs do so with the same number of MySQL transactions as in the base only case, whereas RVTs do so with triple the number (each base write provokes a view delete and view insert). At a throughput of 400 operations/sec, the *RVT* increases latency by 50%. This result illustrates that we need to provision enough capacity to accommodate the extra view maintenance work. Otherwise, when the throughput of the system is high, it will saturate more quickly because of the extra work of view maintenance.

7.2.2 Varying read/write workload

We next measure the impact of the read/write ratio on the impact of view maintenance. We have fixed throughput at 600 operations/sec. Figure 6 plots ratio vs. latency for our 4 cases. As the write percentage increases, the total number of disk accesses increases in all cases. There is, however, a multiplier effect in the view cases. For each additional write on the base table, there are 2 additional writes on the view table. Therefore, the gap between *no views* and all other cases grows with increasing number of writes. For the two RVT cases, we see latency eventually increase drastically, as in the previous experiment. We expect 8:1 to be a typical ratio in our system. If this assumption were to change to, say, 2:1, views would have a much greater impact on client latency, and we would have to provision more resources to handle the extra maintenance load.

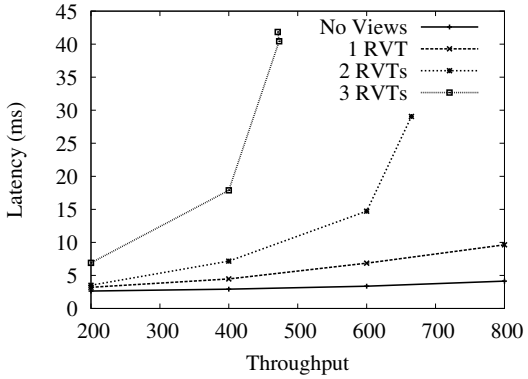

Figure 7: Scaling number of RVTs

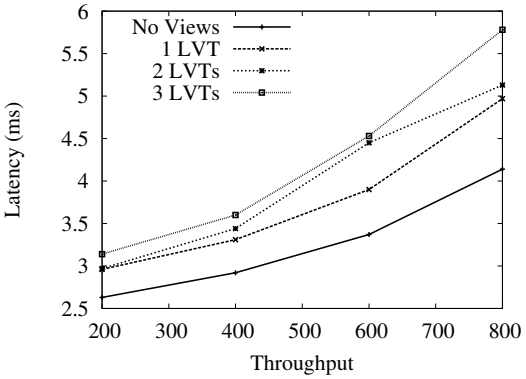

Figure 8: Scaling number of LVTs

7.2.3 Varying number of views

The next two experiments measure the effect on throughput vs. latency as the number of views increases. Figure 7 shows curves for 0,1,2 and 3 *RVTs*, while Figure 8 shows curves for 0,1,2 and 3 *LVTs*. In both cases, for each additional view, each client base write generates an additional 2 view writes on the system (in this experiment, on the same server). For a given throughput, the added disk writes increase latency. As we increase throughput, the greater the number of views, the sooner we reach disk saturation. As before, the effect at higher throughputs is larger for RVTs than LVTs.

7.2.4 View staleness

We measured average staleness for RVTs. We tracked a random sample of base writes and their corresponding view updates. On average, a view record was updated 437 ms after the base record. Thus, while view records are maintained asynchronously, the lag between a base and dependent view write is quite low. Of course, if the system becomes overloaded this staleness would increase.

We provisioned our maintenance experiments so that they could "keep up" with the background work of applying updates to views. It is possible to tolerate client load spikes by delaying the application of view updates, reserving more system capacity for client writes by accumulating a backlog of view updates. The impact is that views become stale until the spike passes and we resume applying updates. This is acceptable for many applications.

We ran experiments allowing backlog to accumulate by prioritizing the storage server to handle client writes over applying log records received from other replicas. We implemented prioritization by varying the ratio of processes accepting client writes versus handling log messages. The more we prioritize the client, the further we extend maximum client throughput. However, the higher the throughput beyond the no-backlog maximum, the faster the backlog accumulates. We observed that for a given throughput, backlog grows linearly with time. The rate of backlog increase depends on the difference D between the storage server's total write capacity (for base and view updates) and the rate of client writes to base tables. If D is high enough to accomodate all view updates, no backlog accumulates. If D is very low, most view updates will be backlogged. In between, backlog grows at a moderate rate.

7.3 Query evaluation

We created a base table B on three storage servers, and loaded records consisting of a primary key, five integer attributes $a_0 \ldots a_4$ and a 4KB additional payload p. Values for a_0 were drawn uniformly randomly from a set of 250 values, a_1 from 500, a_2 from 1000, a_3 from 2000, and a_4 from 4000. We initialized 10 RVTs. We built five index RVTs $R_0 \ldots R_4$ with keys $a_0 \ldots a_4$, respectively. The R indexes included the payload p to support index-only queries. We also built five corresponding "thin" indexes $T_0 \ldots T_4$. These are identical to the R indexes, but omit p. We also initialized 10 LVTs. For B we maintain group-by count aggregate LVTs for each of $a_0 \ldots a_4$. For each T_x thin index we maintain the group-by count aggregate LVT for a_x. The base table B and all RVTs had 64 partitions. We loaded 125,000 records into the base table. This data set, while small, is large enough for us to evaluate the tradeoffs in different query strategies.

7.3.1 Index plans

Our first experiment looked up records by secondary attributes. We varied the selectivity of the query by selecting on attributes $a_0 \ldots a_4$. We consider two alternatives. *Index only*, for attribute a_x, range scans R_x for all records with the desired secondary key and, for each, returns p. *Thin+base* first scans T_x, but must do parallel lookups of the returned keys on B to read p from each. The results (not shown) show two things. First, as expected, the time for both *index only* and *thin+base* increases linearly with the number of returned results. Second, *index only* is more than an order of magnitude faster than *thin+base*, as *thin+base* time is dominated by reading from B. *Index only* plans require only sequential I/O, whereas looking up records in the base table requires both random I/O, and a cross server call to read the base partition. Both strategies are signficantly faster than a full table scan, which required 75 seconds for our table (almost two orders of magnitude slower than *index only*). The cost of a table scan increases with the size of the table, while the cost of an *index only* or *thin+base* plan only increases with the size of the result set. Thus, our index views are a major benefit for secondary attribute queries.

7.3.2 Aggregates

In our aggregation experiments, we grouped by one of the attribute values, and computed `COUNT(*)` over the groups. We consider three processing plans: 1) *index scan*: for at-

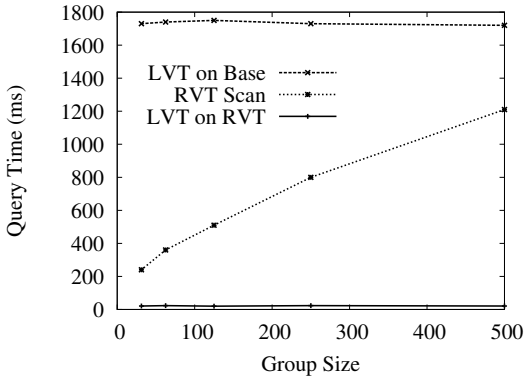

Figure 9: Aggregates, impact of group size

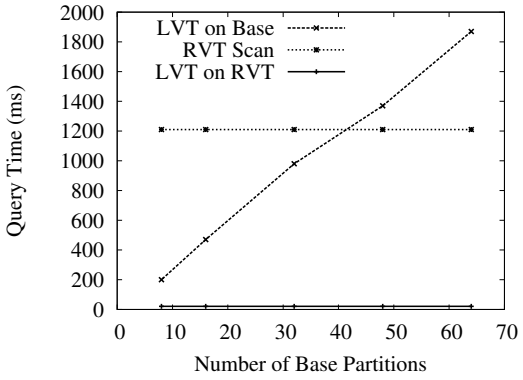

Figure 10: Aggregates, impact of number of partitions

tribute a_x, scanning T_x for the group, 2) *LVT on base*: parallel querying the LVT for a_x on each B partition, and 3) *LVT on RVT*: querying the LVT defined over T_x for a_x. First, we varied the group size (by grouping on $a_0 \ldots a_4$). The resulting average query times are shown in Figure 9. The two LVT approaches are constant across all group sizes. Since the aggregates are pre-computed, the number of records accessed is fixed over all group sizes. *LVT on base* is the most expensive, since all of the base partitions must be accessed. *LVT on RVT* is cheapest, since a single record from a single partition is accessed. *Index scan* is of intermediate cost; although we conduct a range scan, this scan requires only sequential I/O, while the *LVT on base* requires random I/O and is more expensive. Of course, the cost of the index scan increases as the group size increases.

The next experiment fixes group size at 500. Figure 10 shows the number of base table partitions vs. average query time. *Index scan* and *LVT on RVT* are unaffected by the number of partitions, but *LVT on base* query time increases with a greater number of partitions. In fact, for a small enough number of partitions, *LVT on base* beats *index scan*. However, *LVT on RVT* is always the dominant strategy.

7.3.3 Equijoins

To compute an equijoin, we perform an index scan over a join view and then join the records at the query processor. We compared the cost of scanning to the cost of the joining step. We built an equijoin view over two base relations of equal size. The cost of computing the final join depends on the number of joining rows from each base relation. This experiment represents the worse case, where both relations contributed equal numbers of rows and thus the final join cost was highest. The results (not shown) show that the query time is dominated by the index scan. For example, when joining 250 records from one relation with 250 records from another, the final join computation time was only 10% of the scan time. This result validates our decision to defer the actual joining of rows to query time, since joining them is cheap compared to the scan (which would be required even if we materialized the joined rows.) Our implementation was optimized for joining small sets of records. If there were so many records to make the join disk-based, we could pipeline the joining step with the data returning step (e.g. using a symmetric hash join.) We did not evaluate this scenario.

8. RELATED WORK

Although massive scale distributed database systems such as BigTable have just begun to incorporate indexes and materialized views [4], there have been many techniques developed for maintaining materialized views for more traditional relational databases.

Incremental view maintenance - Although materialized views can be refreshed by re-running the view definition query over base tables, it has long been recognized that this approach is often too expensive. Incremental maintenance approaches update the view in response to base table updates. Initial work in the area [3, 5] describes how to determine which base updates impact a view, and what resulting changes need to be applied to a view. We follow their basic approach of filtering base updates based on the view definition and including the key of the base table in view records. However, the set of views we support are restricted, and thus not all their techniques are needed. Similarly, there has also been work on materializing various complex types of views, including views with recursion [13] and views supporting ROLAP operations such as pivot [8].

Deferred maintenance - While incremental maintenance approaches usually update the view synchronously with the base table transaction, deferring the view maintenance can improve efficiency. In this approach, the view is either allowed to remain stale between maintenance operations [19], or the view is brought up to date as necessary when queried [21, 14, 1]. As discussed in Section 5, we examined a batch update approach like that of [19], but decided that our applications demanded more up-to-date views. However, since our applications can tolerate some staleness, we do not need to go through extra work at query time to hide staleness, as in [21, 14]. Systems that try to maintain transactional consistency over materialized views usually lock the view during deferred maintenance, and some work has focused on how to efficiently apply view updates to minimize the time the view is unavailable [9, 20]. Since our consistency is record-level, we do not need to lock the whole view. CouchDB [1] supports views over a collection of schema-free documents, and does incremental maintenance at query time, incorporating document updates performed since the view was last updated. Their views can be any arbitrary computation over the base records, as long as the computation can be incrementally maintained from the latest changes.

Distributed view maintenance - Luo et al. [16] describe a parallel system with base and view tables partitioned over multiple nodes, and describe how maintaining auxiliary structures at view nodes reduces the cost of updating a join view. Our join indexes, which co-locate joining records, replace the need for these structures. Another type of distribution occurs in data warehouses, where the materialized view may be stored in a database system separate from the base tables. After receiving an update from a base table, the warehouse may have to query the base tables at the data sources to determine the correct data to use to update the view. Several algorithms have been proposed [22, 23, 2] that focus on minimizing the communication between the warehouse and the base tables, and on minimizing the time for which the view is unavailable during this maintenance. Some work has focused on how to query sources when their schemas and capabilities may be changing [7]. It is possible to make views *self-maintainable* by keeping additional information at the data warehouse, so that base tables do not need to be queried to maintain the view [18]. In our system, the storage servers holding base table partitions know the view definitions, and publish, along with the base update, enough information to maintain the view.

Index maintenance - There has been work on how to construct [17] and maintain [12] indexes in an online manner. Indexes typically must preserve transactional consistency with the base table. Our asynchronous indexes trade some consistency for more scalability.

9. CONCLUSIONS

Maintaining views is critical to enhance the query power in VLSD databases. We have carefully chosen the set of views that we support to balance several factors, including system complexity, system throughput, view staleness, and usefulness to applications. As we have shown, remote view tables (RVTs) are useful for index, equijoin and selection views, while local view tables (LVTs) are useful for group-by-aggregate views. Both view types defer work until after the client's update of the base table completes. This deferred work complicates both failure recovery and consistency. By reusing existing mechanisms for replicating and recovering base tables, we have made our view tables resilient to failure. By extending the consistency model for base tables to apply to views, we have masked much of the complexity of replication and asynchrony from users. We have also shown that view maintenance is efficient, both theoretically and experimentally. The significant gains to query efficiency (versus scanning the base table) demonstrate the benefits view maintenance can bring to a VLSD database.

10. ACKNOWLEDGMENTS

We would like to thank Shel Finkelstein for his comments, which greatly improved the paper, as did the comments of the anonymous reviewers.

11. REFERENCES

[1] CouchDB. http://couchdb.apache.org/.

[2] D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD*, 1997.

[3] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[4] M. Cafarella et al. Data management projects at Google. *SIGMOD Record*, 34–38(1), March 2008.

[5] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[6] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] S. Chen, B. Liu, and E. A. Rundensteiner. Multiversion-based view maintenance over distributed data sources. *ACM Transactions on Database Systems*, 29:675–709, 2004.

[8] S. Chen and E. A. Rundensteiner. Gpivot: Efficient incremental maintenance of complex rolap views. In *ICDE*, 2005.

[9] L. S. Colby et al. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.

[10] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[11] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[12] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, March 2006.

[13] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[14] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *ICDE*, 2005.

[15] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A structured storage system on a P2P network. In *SIGMOD*, 2008.

[16] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. A comparison of three methods for join view maintenance in parallel RDBMS. In *ICDE*, 2003.

[17] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *SIGMOD*, 1992.

[18] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Conf. on Parallel and Distributed Information Systems*, 1996.

[19] D. Quass and J. Widom. On-line warehouse view maintenance. In *SIGMOD*, 1997.

[20] K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. In *SIGMOD*, 2000.

[21] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, 2007.

[22] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, 1995.

[23] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The strobe algorithms for multi-source warehouse consistency. In *Proc. Conf. on Parallel and Distributed Information Systems*, 1996.