View Maintenance in Web Data Platforms

Hans-Arno Jacobsen #, Patrick Lee #, Ramana Yerneni *

University of Toronto, Canada   * Yahoo!, Inc., U.S.A

***Abstract*— Modern Web Data Platforms (WDPs) handle large amount of data and activity through massively distributed infrastructures. To achieve performance and availability at Internet scale, WDPs restrict querying capability, and provide weaker consistency guarantees than traditional ACID transactions.**

The sheer volume of parallel processing without ACID transaction guarantees, and the large number of independent components in WDPs pose special challenges for view maintenance with respect to concurrent update propagation and correct execution of non-idempotent view updates in the presence of failures.

In this paper, we introduce a novel consistency framework for deferred view maintenance that embodies the weaker consistency primitives prevalent in modern WDPs. Based on this model, we identify techniques to achieve consistent view maintenance for different classes of views. Our analysis covers aggregate, key-foreign-key join, and select-project views.

I. Introduction

Efficient view maintenance is vital for the integration and selective distribution of large volumes of data across wide-area networks and administrative boundaries such data warehouses [9], [12], parallel and distributed databases [10], [14], and WDPs [16]–[20]. In these environments, view maintenance can minimize the amount of data transferred across the network, reduce query latency for queries over data exposed as views, implement secondary indexes, and provide notification and alerting services. Moreover, especially in wide-area networked environments, views can serve to conform to privacy regulations that disallow certain data from leaving specific geographic regions, while requiring other data to be exposed for availability, performance, and operational reasons.

Many existing view maintenance solutions aim to establish strict consistency guarantees for the data exposed in views by exploiting the mechanisms of traditional database technology such as transactions, two-phase commit, locking, versioning, snapshot consistency, and ordered update logs [8]–[10]. However, in many emerging data management contexts, these mechanisms are either not available, or only supported under more relaxed consistency considerations [16]–[21]. The implications this has on maintaining views are at best unclear.

The objectives of this paper are to clarify these points by studying view maintenance in the context of *Web Data Platforms* (WDPs), an emerging breed of massively distributed data management services for supporting large-scale Internet applications. WDPs trade off consistency for availability, scalability and performance [16]–[20]. This trend is common for a growing number of commercial and non-commercial storage platforms, such as Amazon's Dynamo and SimpleDB [17], Google's BigTable [16], Yahoo!'s PNUTS [18], Facebook's Cassandra [20], Project Voldemort [19] (used by LinkedIn), and Xerox's Bayou intended as a versatile replicated storage system [21]. These platforms provide data management services for applications that manage petabytes of data, and support high-availability and low data access latency for billions of operations per day in face of network partitions and load surges. WDPs are massively distributed systems that replicate data asynchronously across several data centers involving thousands of machines worldwide. This massive scale, the inherent parallelism, the asynchronous update processing, and the lack of many traditional database mechanisms available in WDP impose several challenges not addressed by existing view maintenance approaches.

Much of the existing works on view maintenance disregards implications of failures during update propagation on the correctness and consistency of view states [1]–[8]. However, the large number of components comprising WDPs, orchestrated over unstable and hostile wide-area network boundaries, makes failure the norm, not the exception [17]. Ignoring failures can therefore have dire consequence for propagating updates resulting in view data inconsistency. Furthermore, due to asynchronous processing, view managers may propagate base table updates in an order different from the order they were processed on base tables. Under the strict consistency considerations of traditional view maintenance approaches, this surfaces at the view as out-of-order view states, or, yet worse, as invalid view states. Finally, the lack of transactions to isolate the parallel processing of updates coupled with the inherent parallelism of WDPs can result in races of view table updates that compute arbitrary results, again resulting in lack of convergence and introduction of inconsistencies.

To address these problems, we make the following contributions after characterizing WDPs in Sec. II. In Sec. III, we introduce a novel consistency model to characterize view maintenance computations in the context of WDPs. The new model is aligned with the weaker consistency considerations characteristic for this context. In this section, we also present a detailed analysis of challenges for view maintenance resulting from the constraints of modern WDPs. These challenges are *concurrent update propagation*, *non-Idempotent view updates*, and *out-of-order update propagation*. We identify solution techniques to address these challenges in Sec. IV, and illustrate by way of example how they effect view maintenance. In Sec. V, we establish the convergence and consistency results for several classes of views with respect to our consistency model. In Sec. VI, we experimentally quantify the impact of our solution techniques on system performance and derive insights relevant for the provisioning of WDPs supporting view

CSRG Technical Report #599. University of Toronto. September 30, 2009

maintenance. Finally, in Sec. VII, we place our work within the context of the plethora of prior work on view maintenance.

II. Web Data Platforms

In this section we summarize the key properties of WDPs, an emerging breed of data management systems, and describe the implications of these properties for the maintenance of derived data such as database views. We first characterize WDPs by reviewing their main design choices, then highlight implications and challenges for view maintenance. Finally, we describe the architectural framework of a typical WDP.

A. Characteristics of WDPs

WDPs are large-scale data management systems comprising thousands of machines. Typically, a WDP is comprised of services for data persistence, request routing, logging, replication, and load management [16]–[20]. WDPs target the data management needs of applications like Web-based email, online shopping, online catalogs, and shopping-cart management, which are used by hundreds of millions of users over the Web daily. Examples of WDPs described in the literature are Amazon's Dynamo [17] and SimpleDB product, Google's BigTable [16], Yahoo's PNUTS [18], Facebook's Cassandra [20], and Project Voldemort [19] used by LinkedIn.

For many web applications, a weaker consistency guarantee on a per-record basis is sufficient. WDPs exploit this weaker requirement to offer high performance and massive scalability. Firstly, complex operations such as joins are disallowed in favor of simple operations such as record get. Secondly, WDPs avoid global coordination. In particular, full-fledged ACID-transaction support is not offered, including isolation and atomic commitment of multi-update transactions. Instead, primitive single record update operations are offered.

WDPs achieve high performance and scalability through horizontal partitioning of data across many *storage units*. Each record is independently accessible in a parallel manner. Parallelism also extends to propagating base updates to views; view tables as well as base tables are partitioned and processed in this manner. A view record may be dependent on a set of base records that may reside in multiple partitions on multiple storage units. At the same time, each base record may be relevant to multiple view records that reside on multiple storage units. Thus, the mapping of base records/partitions to the view records/partitions is many-to-many.

Yet another aspect of the high-performance and scalability requirements of WDPs is that synchronous operations are generally avoided. In particular, the maintenance of derived data, such as views and replicas, is performed asynchronously to reduce the latency for the operations on the base data.

B. Challenges for Consistent View Maintenance

The above properties of WDPs (parallel processing of base and view updates, lack of ACID transaction support, and deferred view maintenance) pose several challenges to consistent view maintenance in these systems. We discuss three fundamental challenges in this regard:

1) Concurrent update propagation: In WDPs, base updates can be propagated to views through multiple view managers to achieve scalable view maintenance. That is, for a given view that is residing on multiple storage managers, there can be multiple view managers that are propagating updates from the relevant base tables that are residing on multiple storage units. The base update processing, and the update propagation work are done in parallel. Given that WDPs may not provide adequate isolation and concurrency control (like ACID transactions in conventional systems), concurrent update propagation/processing can lead to inconsistencies in the state of the view tables.
2) Non-idempotent view updates: In a typical WDP, as a large number of components (e.g., storage units) are involved in handling ultra-scale data and operations, there is higher likelihood of component failures in WDPs than in conventional database systems. As such, we need to consider the effects of system-component failures on view maintenance. In the presence of failures, update programs that involve non-idempotent view updates (e.g., in the case of COUNT and SUM views) can lead to inconsistent view maintenance.
3) Out-of-order update propagation: The flow of base updates to views is through multiple view managers in a parallel and distributed setting employed by WDPs. Without appropriate flow control, in terms of the flow of base updates to corresponding view records through various view managers, view maintenance can become inconsistent due to out-of-order update propagation.

We will illustrate the above three challenges through examples and point out the various kinds of consistency problems they cause for view maintenance in Sec. III.

C. Typical WDP Architecture

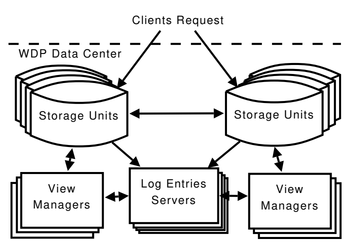

Fig. 1. Typical WDP architecture

Fig. 1 shows the components of a typical WDP. The system structure resembles the Dynamo [17], PNUTS [18], BigTable [16], Cassandra [20], and Project Voldemort [19] architectures. The records in the system are stored at *storage units* and application requests are routed to the appropriate *storage units*. Asynchronous update propagation, for deferred view maintenance, is achieved through the collection and propagation of log records that represent the stream of base-record updates. *View managers* request base-table updates from the *log entries servers* and propagate these updates to the view tables using the algorithms described in Sec. V.

The system structure of Fig. 1 is what we used for the evaluation of the solution techniques described in this paper. We could have performed our experiments using existing DHT storage implementations. However, as evidenced by Dynamo's overlay construction [17] and the PNUTS design [18], the high latency cost associated with $O(log(n))$ average hop counts is unsuitable for large-scale WDPs.

III. Consistency Model

In this section, we define a consistency model for view maintenance in a manner that is compatible with record time-line semantics adopted by WDPs [17], [18]. In particular, we formally define the notion of agreement between the state of a view record and the state of the base records that it is derived from. In our consistency model, we define various levels of consistency to accommodate scalable view maintenance in WDPs. First, we present our notation for records and their states in a WDP. Then, we relate the state of view records to base records. Finally, we specify what it means for view maintenance to achieve convergence and various levels of consistency. Then, we return to the fundamental problems in maintaining views consistently in WDPs, introduced in Sec. II-B, as they aim to achieve scalable data processing by adopting weak-consistency mechanisms. We illustrate these problems with examples and the associated violation of the various levels of consistency in our model.

A. Notation

In our model, B represents the base data, i.e., the set of records in the base tables. V represents the view data, i.e., the set of records in the view tables. Each record in our model is designated by a table the record belongs to and a key for the record. B_i represents a base state and V_j represents a view state. $V_j(r)$ represents the state of a record r in a view state, V_j. $View(B_i)$ represents the set of view records obtained by applying the view definition, $View$, on the base state B_i. $View(B_i)(r)$ represents the state of a record r when applying the view definition, $View$, on the base state B_i. This notation will become important when defining the levels of consistency, below.

Each update, insert, and delete operation affects a single record and produces a new version of the record. Each record in our model has a time-line corresponding to the sequence of versions of the record. Each base state B_i effectively contains a particular version of each base record, and each view state V_j contains a particular version of each view record. We say $B_i \leq B_j$ if for all base records, the version of the record in B_j is not earlier than the version of the same record in B_i. Note that there could be two base states B_i and B_j such that neither $B_i \leq B_j$ nor $B_j \leq B_i$ is true. Similarly, we say $V_i \leq V_j$ if for all view records, the version of the record in V_j is not earlier than the version of the same record in V_i.

To understand the notion of agreement between view states and base states, and the associated levels of consistency, let us consider an initial base state B_0 and a final base state B_f that is achieved after a sequence of updates, inserts and deletes are performed on the base records. The intermediate base states B_i are obtained by considering intermediate versions of the base records. That is, for a base state B_i between B_0 and B_f, at least one base record has a version earlier than its version in B_f and at least one base record has a version later than its version in B_0. Effectively, the intermediate base state B_i can be obtained by applying to B_0 a prefix of a sequence of updates, inserts and deletes that agrees with the sequence of updates, inserts and deletes that produces B_f. We say that two sequences of updates agree with each other if they do not change the order of updates, inserts and deletes on any particular record. That is, if two sequence of updates, inserts and deletes produce the same time-line for each record they operate on, they agree with each other.

We illustrate the notion of base states and how they change as updates are executed by the following example. Consider a base table `Friends(UID, FID)`, with both attributes as the key. Let the initial state of the base table contain a single record $\{(u_3, u_2)\}$. Consider a stream of updates: $i_1(u_3, u_5)$, $d_2(u_3, u_2)$, $i_3(u_3, u_1)$. That is, the first change is an insert, the second one is a delete, and the third one is an insert.

The initial base state has just one record $\{(u_3, u_2)\}$. The final base state contains $\{(u_3, u_5), (u_3, u_1)\}$. An intermediate base state is $\{(u_3, u_5)\}$. Another intermediate base state is $\{(u_3, u_5), (u_3, u_2)\}$. What may be less obvious is that yet another intermediate base state may be $\{(u_3, u_2), (u_3, u_1)\}$, as the record for the last insert operation does not have the same key as that of any other base record under consideration. Thus, the record it inserted can appear in an intermediate base state without the effects of the other two operations (on the other base records). In essence, we can shuffle the three updates because they do not share keys, and then consider the prefixes of the sequences of updates to identify the intermediate base states.

As with the base states, we formulate the notion of an initial view state V_0, a final view state V_f, and intermediate view states V_j for the view records. When the base updates, inserts and deletes are propagated to the view manager, it performs a sequence of view updates, inserts and deletes on view records. With respect to these view updates, we consider the initial, final and intermediate view states.

B. Levels of Consistency

We define the following levels of consistency with respect to view states and base states.

Convergence: We say that the system achieves *convergence* if and only if the final state of all view records correspond to the final base state. That is, V_f and $View(B_f)$ are equal. Formally, for each view record r, [1]

$$V_f(r) = (View(B_f))(r).$$

Weak consistency: We say that the system achieves *weak consistency* if and only if it achieves *convergence* and every state of every view record is valid (i.e., corresponds to some base state). Formally, for each intermediate view state V_j, for

[1]To re-iterate, the interpretation of this notation is that *convergence* is achieved if every record in V_f agrees (left side), in a one-to-one correspondence (i.e., has the same value), with the record that results from applying the view definition, $View$, on the final base state, B_f (right side). See Sec. III-A for the definition of the notation.

each view record r, there exists a base state B_i with $B_0 \leq B_i \leq B_f$, such that

$$V_j(r) = (View(B_i))(r).$$

Strong consistency: We say that the system achieves *strong consistency* if and only if it achieves *weak consistency* and every pair of states of every view record are correctly ordered (i.e., correspond to two base states in the same order.) Formally, for each pair of intermediate view states V_{j_1} and V_{j_2} such that $V_{j_1} \leq V_{j_2}$, for each view record r, there exist a pair of base states B_{i_1} and B_{i_2}, with $B_0 \leq B_{i_1} \leq B_{i_2} \leq B_f$, such that

$$V_{j_1}(r) = (View(B_{i_1}))(r) \text{ and } V_{j_2}(r) = (View(B_{i_2}))(r).$$

Complete consistency: We say that the system achieves *complete consistency* if and only if it achieves *strong consistency* and every base state is reflected in some view state. Formally, for each base state B_i with $B_0 \leq B_i \leq B_f$, for each view record r, there exists a view state V_j, such that

$$V_j(r) = (View(B_i))(r).$$

While we have defined four levels of consistency, in practical terms the complete consistency level is not useful to achieve. There is not much benefit in maintaining views in a manner that reflects every possible base state in WDPs. What is more useful is to achieve eventual agreement of the view with the base state and ensuring that the view information corresponds to valid base states and in the right order. Thus, the discussions in this paper centers around the problems and solutions in achieving convergence, weak consistency and strong consistency for various classes of views.

C. Violation of Consistency

In this section we illustrate how the three fundamental issues presented in Sec. II-B may lead to inconsistencies in the maintenance of views in WDPs. In Sec. IV, we will show how a set of solutions we developed can re-establish consistency.

Example 1 – **Concurrent update propagation**: Consider a base table $R(\underline{K}, X, Y)$ and a view $D(\underline{X}, S)$ defined as `SELECT` X, `SUM(`Y`) FROM` R `GROUP BY` X; note that K is the key attribute in R and X is the key attribute in D. Consider two base updates: $u_1(R(k_2, x_1 \rightarrow x_2, 200))$ and $u_2(R(k_4, x_2 \rightarrow x_1, 400))$. The initial and final base states along with the initial view state are shown below.

$$B_0 = \{(k_1, x_1, 100), (k_2, x_1, 200), (k_3, x_2, 300), (k_4, x_2, 400)\}$$
$$B_f = \{(k_1, x_1, 100), (k_2, x_2, 200), (k_3, x_2, 300), (k_4, x_1, 400)\}$$
$$V_0 = \{(x_1, 300), (x_2, 700)\}$$

Let two view managers (or two concurrent threads of a view manager) propagate the two updates concurrently. As the view managers execute the update programs to propagate the updates, they access the view records and issue updates. Let the view manager processing the first update u_1 first read the x_1 and x_2 view records. Then, let it add 200 to the SUM for x_2 and subtract 200 from the SUM for x_1. Concurrently, let the second view manager processing the update u_2 read the x_1 and x_2 view records. Then, let it add 400 to the SUM for x_1 and subtract 400 from the SUM for x_2. Let the concurrent interleaving of these operations be such that all the read operations happen first and then the update operations of the first view manager and then the update operations of the second view manager. In such a case, the final state of the view records, V_f will be $\{(x_1, 700), (x_2, 300)\}$. This final view state does not agree with the final base state B_f. In particular, B_f is $\{(k_1, x_1, 100), (k_2, x_2, 200), (k_3, x_2, 300), (k_4, x_1, 400)\}$. The view definition applied to B_f yields $View(B_f) = \{(x_1, 500), (x_2, 500)\}$. Both the x_1 view record and the x_2 view record do not converge to their final states, with respect to the final base state. Thus, concurrent update propagation, in the absence of appropriate isolation support, can lead to loss of convergence in view maintenance.□

The crux of the problem above is that some of the view updates performed when propagating the base updates may be "overwritten" by other concurrent activity. In effect, we have a case of "lost updates" in the presence of concurrent update propagation. Accordingly, the view states achieved may not reflect all the base updates properly and thus we can lose the convergence property for view maintenance.

Example 2 – **Non-idempotent view updates**: Consider a base table $R(\underline{K}, X, Y)$ and a view $D(\underline{X}, S)$ defined as `SELECT` X, `SUM(`Y`) FROM` R `GROUP BY` X. Note that K is the key attribute in R and X is the key attribute in D. Consider the base insert $i_1(R(k_3, x_1, 100))$. The initial and final base states along with the initial view state are shown below.

$$B_0 = \{(k_1, x_1, 100), (k_2, x_1, 200)\}$$
$$B_f = \{(k_1, x_1, 100), (k_2, x_1, 200), (k_3, x_1, 100)\}$$
$$V_0 = \{(x_1, 300)\}$$

Let the view manager propagate the update $i_1(R(k_3, x_1, 100))$ by increasing the SUM for the x_1 record by 100. However, after it has issued the view update, and the system completed the update operation, let the view manager die due to a system/component failure. Later, when the view manager recovers and resumes update propagation, it will have to redrive the processing of the i_1 update, because it might not have registered the fact that its earlier processing of that update had completed. Thus, in its second attempt, it again increases the SUM for the x_1 record by 100, arriving at the view record $(x_1, 500)$. Note that this view record does not correspond to either B_0 or B_f. Thus, we have an invalid view state that violates the weak-consistency level in our model.□

The crux of the problem above is that the view manager does not know that the view update issued before it failed had completed. Effectively, the repeated view update when the view manager resumes after failure is an "illegal update". Therefore, the view state becomes invalid. In order to avoid invalid states, and thus preserve the weak-consistency level in our model, view-maintenance schemes need a solution to the problem of processing non-idempotent view updates in the presence of failures. The above example illustrates how non-idempotent view updates can lead to invalid view states, thus violating weak consistency. Other variations of the example may also demonstrate the loss of convergence in terms of the final view state not agreeing with the final base state.

Example 3 – **Out-of-order update propagation**: Consider a base table $R(\underline{K}, X, Y)$ and a view $D(\underline{K}, X, Y)$ defined as SELECT K, X, Y FROM R WHERE $Y < 25$. Note that K is the key attribute in R and D. Let the initial state be $B_0 = \{(k_1, x_1, 5)\}$, $V_0 = \{(k_1, x_1, 5)\}$. Consider three base updates: $u_1(R(k_1, x_1, y \to 10))$, $u_2(R(k_1, x_1, y \to 20))$, and $u_3(R(k_1, x_1, y \to 15))$. The final base state is $B_f = \{(k_1, x_1, 15)\}$. Let two view managers (or two concurrent threads of a view manager) propagate the updates. In particular, let the first view manager propagate u_1 and then u_3, while the second view manager propagates u_2. To illustrate the specific problems with the out-of-order update propagation, let us assume that the two view managers do not encounter isolation problems with respect to the concurrent execution of their update programs. Thus, the view update operations can be as follows: the second view manager can completely process u_2 before the first view manager starts processing u_1. The view record has the following sequence of states: $(k_1, x_1, 5)$, $(k_1, x_1, 20)$, $(k_1, x_1, 10)$, $(k_1, x_1, 15)$. The final state of the view record does agree with the final base state. So, the view converges to the desired final state. Moreover, all the states of the view record are valid, because they correspond to some base state. The $(k_1, x_1, 5)$ view record corresponds to B_0, the $(k_1, x_1, 15)$ view record corresponds to B_f, the $(k_1, x_1, 20)$ view record corresponds to the intermediate base state that can be obtained by applying u_1 and u_2 to B_0, and the $(k_1, x_1, 10)$ view record corresponds to the intermediate base state that can be obtained by applying u_1 to B_0. Thus, all the view states are valid and so weak consistency is not violated. However, the sequence of view states does not have strong consistency. In particular, note that the view-record state $(k_1, x_1, 10)$ is later than the view-record state $(k_1, x_1, 20)$, but they correspond to intermediate base states in the wrong order.□

The above example illustrates how out-of-order propagation can lead to violation of strong consistency. Other variations of the above example can illustrate how out-of-order propagation can also lead to violation of other consistency levels. For instance, if in the above example, u_1 and u_3 are completely propagated by the first view manager before the second manager propagates u_2, then the final view state obtained may be $(k_1, x_1, 20)$, violating convergence of the view.

The crux of the problem above is that if we allow the updates on a base record to "race with each other" through multiple view managers maintaining the same view table, they can be applied out of order and cause consistency violations.

IV. Solution Techniques

In the previous section we have described the various challenges to consistent view maintenance in typical WDPs. In this section, we discuss the solutions to these problems.

A. Concurrent Update Propagation

As observed in the previous section, a major challenge to consistent view maintenance in WDPs is concurrent update propagation. There are typically multiple view managers that propagate base updates to the views. In such a setting, it is possible that there are multiple view managers that receive updates from different partitions of a base table and propagate them to the same view record. That is, a view record may be concurrently accessed and updated by multiple view managers handling different base updates being propagated.

As we have seen in the previous section, uncontrolled concurrent update propagation can lead to inconsistent view maintenance. Conventional systems overcome this problem by relying on ACID-transaction capabilities that ensure appropriate serializable execution of the concurrent operations. However, in typical WDPs it is too expensive to provide ACID-transaction capabilities. Therefore, to achieve scalability of data and operations, no ACID transactions with serializable execution order are supported.

We have developed simple techniques for consistent view maintenance through concurrent update propagation, using basic "test-and-set" (TAS) primitives that can be supported by WDPs in a scalable manner. The techniques we employ derive from optimistic locking [22]. We illustrate these techniques by revisiting the example of the previous section.

Example 1 (cont.'d): Let two view managers (or two concurrent threads of a view manager) propagate the two updates, u_1, u_2, concurrently. As the view managers execute the update programs to propagate the updates, they access the view records and issue updates. Let the view managers test the versions of the records they access before they perform the update operations. For instance, the view manager processing the first update u_1, reads the x_1 and x_2 view records, because the k_1 base record is changing x_1 to x_2. Later on when it adds 200 to the SUM for x_2, it tests that the x_2 record has not been modified between the time it read the record earlier and is now updating it. If there had been an intervening update, perhaps due to another concurrent update propagation (by another view manager or another thread of activity), the view manager simply reads the x_2 record again and issues an update that increases it by 200. In a similar way, it can protect the integrity of its update of the x_1 record from interference by other concurrent update-propagation activity.□

The important thing to note here is that the simple TAS capability described above can be easily supported by typical WDPs by keeping track of a signature indicating the state of the data record. In particular, a read operation returns the record and its signature; a TAS operation checks if the signature of the record has changed since the earlier read that the update depends on. If the signature has changed, the TAS operation fails; otherwise, the operation succeeds and it updates the signature. Note that the implementation of the TAS operation does not require any global synchronization/coordination like in the case of ACID-transaction support. Each TAS operation can be managed by the storage unit that stores the record being updated, in an efficient way using short-duration latches that just last for the duration of the update operation (not an entire transaction).

As part of the TAS operation's arguments, it is the view manager's responsibility to provide a meaningful signature that captures the state of the view record. This also serves to reduce the computational burden on the storage units. The signature can be initialized to any value when performing a TAS insert operation. When performing a TAS update operation, the only requirement is that the new signature must be different from the retrieved signature. In its simplest form, a TAS signature can be a single integer representing the current version number of the view record. Henceforth, we will call this *counter-based TAS* (cnt-TAS). The signature is initialized to zero in a TAS insert operation. As for TAS update operation, a new signature is generated by incrementing the retrieved version number by one.

Signatures that satisfy the properties specified in Sec. IV-B can also be used by TAS operations. Henceforth, we will call this *signature-based TAS* (sig-TAS). The basic construction of such signatures is described in Sec. IV-B. Optimization to the signature construction is discussed in Sec. IV-D.

B. Non-idempotent View Updates

As seen in the previous section, when view-update programs involve non-idempotent operations (like in the case of propagating base updates to a SUM view), system failures can lead to redriving the update programs and consequently end up with view states that are invalid.

The main problem at hand is that when a failure occurs and later the view manager resumes processing updates again, it may have to redrive the last update that it was processing at the time of the failure. If the update actions issued just before the failure occurrence actually had been completed by the system, but the view manager had not registered the completion of the issued updates, the redriving of the update program may lead to an invalid view state. If only the view manager can determine that the non-idempotent update operations performed as part of the update program were already completed earlier, it could avoid repeating the non-idempotent operations and thus not produce invalid view states. We present a technique based on *update signatures* that ensures the detection of repeated non-idempotent operations and guarantees "exactly once" semantics. We discuss properties and the construction of update signatures below.

The core intuition of our technique is to tag each non-idempotent update operation in an update program with an update signature. When this operation is issued by the view manager, the system keeps track of the signature as it processes the operation in an atomic way. That is, the signature of the operation is registered in the same logical operation as the corresponding update operation (typically, under a latch held by the storage unit when processing the update operation). If the same update operation is later issued again by the view manager, the update signature is checked to see if the update operation had been processed earlier. If so, the second attempt is rejected with the error code indicating that the update had been processed earlier.

The key property of an update signature is that it is sensitive to the base table update that is being propagated. That is, if the same base update is being propagated again by the view manager (due to a redrive caused by a failure during the earlier attempt), the update signature generated will be the same. Thus, if the system has actually processed the update earlier and the update is being redriven, the duplicate update signature can help the system avoid the repeated processing of the non-idempotent operation.

The second important property of update signatures is that two different base updates being propagated should have two different update signatures. Otherwise, when the second of these updates is being processed by the view manager, its update operations on the view tables will be rejected because of duplicated signatures.

Thus, by using update signatures that are unique and that are tied to the base table updates being propagated, we can avoid the problem of constructing invalid view states in the presence of failures. The following example illustrates how the inconsistency in view maintenance, described in the previous section, can be avoided through update signatures.

Example 2 (cont.'d): Let the view manager propagate the update $i_1(R(k_3, x_1,\ 100))$ with update signature us_1 and encounter a failure at the end of propagating that update. In particular, the view record changes from $(x_1, 300)$ to $(x_1, 400)$, but the view manager may not have registered the fact that the update propagation has completed. On resumption after the failure, the view manager propagates the same update again, and tags it with the same signature us_1. This time, the view update that attempts to add 100 to the sum value associated with the x_1 view record will be rejected with the error code indicating that the update had already been executed previously. Thus, the view manager avoids constructing the invalid view state that includes $(x_1, 500)$.□

In some situations, a single base update can lead to multiple view updates. There are two main cases to consider: (1) the effect of a single base update on a single view leads to multiple updates on that view; (2) multiple views are defined on a base table and so a single base update leads to multiple view updates on these different views. A simple example for case (1) is to consider the base update $u(R(k_3, x_1 \rightarrow x_2, 100))$ and consider its propagation to the above SUM view. This single base update leads to two updates on two different records (the x_1 record and the x_2 record) in the SUM view. We need to have two different update signatures associated with these two update operations; if we have the update signature be sensitive only to the base update, the second operation will be erroneously rejected as a duplicate. That is, update signatures should be sensitive to the particular view update in the update program that is being executed in propagating a base update.

For case (2), once again it is clear that the update signatures associated with the different view tables updated in response to a single base update need to be different. Otherwise, only the first of these view updates gets executed and the subsequent updates (on the other view tables) will be rejected. In summary, update signatures need to have the following properties in order to ensure consistent view maintenance:

1) Update signatures for two view-update operations associated with two different base updates must be different.
2) Update signatures for two update operations on two different views that are updated based on the same base update must be different.
3) Update signatures for two update operations on the same view when a single base update is being propagated must be different.
4) Update signatures for two update operations on the same view that are updated based on the same base entry must impose partial temporal ordering.
5) Given the same base update, the same view, and the same update operation in the view update program, the update signature must be same.

A simple signature construction procedure can be derived directly from the specified properties. Consider the signature to be a triple, consisting of the view table ID, the view record ID, and the complete list of base table log entries that have been propagated to the view record. Each base table log entry can be represented by a pair, the log-stream ID and log-entry timestamp. The log-stream ID indicates where the base table log entry originated (e.g., which storage unit performed the base table update) and the log-entry timestamp indicates when the base table update was performed. The storage and transmission costs of such signatures are high. Therefore, in Sec. IV-D, we develop a more efficient signature scheme, if certain requirements are met.

C. *Out-of-order Update Propagation*

As discussed in the previous section, even if there is appropriate concurrency control with respect to multiple view managers accessing and modifying view records as base updates are propagated to them, there can be problems with consistent view maintenance due to out-of-order update propagation. Without proper flow control, a sequence of updates on a base record can be propagated to a view record out of order. To avoid the associated consistency problems due to out-of-order update propagation, we have developed a simple solution that involves controlling the view-update flow without significantly reducing the degree of parallel processing. In particular, we formulate a rule that can be used in the design of the update flow (in terms of the paths through the view managers taken by base updates to their corresponding view updates). The rule is as follows: *All base updates on a single base record must go through the same view manager to update the same view record.*

Basically, we allow for multiple view managers to maintain a view, and we allow for multiple view managers to subscribe to the updates on a base table to propagate its updates in a parallel manner. However, we restrict the update propagation flow so that out-of-order propagation is avoided. To implement the above rule for flow control, we leverage the essential distributed nature of WDPs. We consider the case of large base tables and view tables being partitioned across multiple storage units. In particular, we use the flow control mechanism to regulate how the updates on each partition are processed by different view managers to propagate updates to different partitions of the related view tables. The flow-control rule is implemented by requiring that in the many-to-many mapping between the set of base-table partitions and the view-table partitions, through the view managers, there is a single view manager on the path from any base-table partition to any view-table partition.

Based on the simple implementation of the flow-control rule, we achieve a high degree of parallel processing in view maintenance, through multiple view managers propagating updates to views. In particular, we allow for the updates of a single base record to go through multiple view managers to effect different views. We also allow for the updates of different base records to go through multiple view managers to effect the same view.

The following example illustrates how our flow-control rule avoids consistency problems in view maintenance.

Example 3 (cont.'d): Let two view managers (or two concurrent threads of a view manager) propagate the updates. With the flow control in place, all the k_1 base record updates (namely, u_1, u_3 and u_5 go to one view manager while the k_2 base record updates (namely, u_2 and u_4) can go to the other view manager. Accordingly, the k_1 view record has the following sequence of states: $(k_1, x_1, 5)$, $(k_1, x_1, 10)$, $(k_1, x_1, 20)$, $(k_1, x_1, 15)$. The k_2 view record has the following sequence of states: $(k_2, x_2, 5)$, $(k_2, x_2, 22)$, $(k_2, x_2, 17)$. Thus, due to the flow control of the update propagation, we achieve strong consistency. In particular, the view states for the two view records achieve convergence, all states are valid, and all state transitions agree with base-state record time-lines.□

D. *Non-idempotent View Updates Revisited*

As we will show in Sec. V, the above solutions are rarely applied independently, but are combined in the view update programs. This observation lets us significantly reduce the size of update signatures. As explained in Sec. IV-A, TAS operations can use the signatures defined in Sec. IV-B to ensure the correctness of concurrent updates. Since a TAS operation must provide the view table name and the record key to indicate which entry is to be modified, the signature's view table ID and view record ID are implicitly encoded as part of the TAS operation. Thus, the signature only needs to store the complete list of base-table log entries. Furthermore, when flow control is applied, view mangers must process log entries from each log stream in sequential order. Hence, the timestamp within each log stream must be monotonically increasing. Exploiting this fact, it is sufficient to maintain for each log stream the entry with the largest timestamp. A view manager can check if a log entry has been processed by comparing the entry's timestamp to the corresponding log stream's timestamp stored within the signature. Assuming the log stream IDs form an integer range, we can also eliminate the need to explicitly store the log-stream IDs by keeping the log entries as a sorted list.

The final signature is reduced to a simple form of a vector clock as first described by Lamport in [23]. The storage requirement is one integer (4 bytes) per log stream. Standard compression techniques can then be applied to further reduce the signature size. Thus, we can generate space-efficient signatures when used in conjunction with TAS operations and flow control.

V. CLASSES OF VIEWS

In this section we present a systematic analysis of consistent view maintenance for different classes of views.

A. Aggregate Views

We consider the standard set of aggregate views, namely COUNT, MIN, MAX, SUM, and AVG. For COUNT views, we present a detailed analysis, and summarize the results for the other aggregate views.

Algorithm 1 Insert propagation for COUNT view

```
propagateInsert(R(k, x, y), sid, eid):
      loop
        foundEntry ← read(D(x, ?c), ?sn)
        if foundEntry then
          if hasProcessed(sn, sid, eid) then
 5:         break
          sn' ← generateSignature(sn, sid, eid)
          succeed ← updateTAS(D(x, c + 1), sn, sn')
        else
          sn' ← generateSignature(NULL, sid, eid)
10:       succeed ← insertTAS(D(x, 1), sn')
        if succeed then
          break
```

1) COUNT View: Consider a COUNT view D defined on a base table $R(\underline{K}, X, Y)$ as $D(\underline{X}, C)$ = `Select` X, `COUNT(`Y`) As` C `From` R `Group-by` X. The update propagation algorithms are given in Alg. 1-3, respectively.

We shall walk through the update program that handles insert on the base table R. It first reads (Line 2) the view record whose count needs to be incremented due to the insert operation on the base table. If the view record is not present, $foundEntry$ will be set to $False$; otherwise, it is set to $True$ and the view record's value and signature are store in c and sn, respectively.

In the case where $foundEntry$ is $False$, the program (Line 10) attempts to insert a new view record with C value of 1. If the TAS insert operation is successfully completed, then the program is done. On the other hand, if the TAS insert operation failed, then the failure implies a view record with the same key already exists within the view table; this can only occur if another view manager inserted that view record after this program executes the read operation in Line 2 but before it executes the TAS insert operation in Line 10. The failure will cause the program to go back to the top of the loop and re-drive the processing of the same log entry.

When $foundEntry$ is $True$, the program (Line 4) must first check if the log entry has already been processed.[2] If the program determines the log entry was processed on a prior occasion, then it simply ignores that entry. Otherwise, the program issues a TAS update operation (Line 7) to increment the view record's count. Once again, if the TAS update operation succeeds, then the program is done. However, if the TAS update operation failed, then the program re-enters the loop and the log entry is re-processed from scratch. We conclude the discussion of COUNT view insert propagation by noting the TAS update failure can only occur if: (1) the view record is missing because another view manager deleted it, or (2) the view record's signature has been modified by another view manager.

Algorithm 2 Delete propagation for COUNT view

```
propagateDelete(R(k, x, y), sid, eid):
      loop
        foundEntry ← read(D(x, ?c), ?sn)
        if foundEntry then
          if hasProcessed(sn, sid, eid) then
 5:         break
          sn' ← generateSignature(sn, sid, eid)
          if c = 1 then
            succeed ← deleteTAS(D(x, c), sn)
          else
10:         succeed ← updateTAS(D(x, c - 1), sn, sn')
        else
          break
        if succeed then
          break
```

Algorithm 3 Update propagation for COUNT view

```
propagateUpdate(R(k, x→x', y→y'), sid, eid):
      if x ≠ x' then
        propagateDelete(R(k, x, y), sid, eid)
        propagateInsert(R(k, x', y'), sid, eid)
```

The programs to handle the propagation of the delete and update operations on the base table are shown in Alg. 2 and Alg. 3, respectively. The propagation of a delete operation on the base table resembles the insert propagation. Finally, the update program to propagate base-table updates to a COUNT view has the same structure as a delete operation followed by an insert operation.[3]

We now establish the convergence and consistency results for the COUNT view. Due to space constraints, we will present the following result without proof. We refer the reader to [24] for the complete proof.

Theorem 1: The count view is strongly consistent.

SUM and AVG views are also strongly consistent. The update programs for SUM and AVG views are akin to those of COUNT views.[4] For instance, in the case of SUM views,

[2] If the TAS is signature-based, then, as described in Sec. IV-D, $hasProcess$ must compare the signature, sn, against the log entry identified by its stream ID and entry timestamp, sid and eid, respectively. If the TAS is counter-based, then $hasProcess$ always return $False$ since duplicate base log entries are undetectable.

[3] Not all view update propagation follow this structure.

[4] AVG stores the sum and the count in each view record.

rather than incrementing and decrementing the C value, the sum value is added to and subtracted from C. Error handling and concurrent update propagation are exactly the same.

2) MIN and MAX Views: Update propagation for MIN and MAX views are similar. We briefly discuss MIN views. The discussion also applies to MAX views.

To maintain a MIN view the update programs keep track of the MIN value in each view record. Whenever a base-table insert operation is propagated, the MIN is changed if the new value is smaller than the MIN; otherwise it is ignored. In the case of a base-table delete operation, the value is ignored if it is larger than the MIN. If the deleted value is equal to the MIN, a new MIN value is computed from the base-table records. Update operations on the base table are propagated to the MIN view by combining the logic of the delete operations and insert operations.

We will present convergence and consistency result for MIN views without proof. The complete proof is provided in [24].

Theorem 2: The MIN view is weakly consistent.

While the MIN view converges and all view states computed are valid, out-of-order results are possible, thus violating strong consistency. We illustrate this through an example.

Let the initial base table state be $B_0 = \{(k_1, x_1, 4)\}$. Let the following sequence of updates be processed on B_0: $d_1(k_1, x_1, 4)$, $i_2(k_2,\ x_1, 2)$, and $u_3(k_2, x_1, y \rightarrow 5)$. The resulting base state is $B_f = \{(k_1, x_1, 5)\}$. The view manager sees the same sequence of base-table updates, however, after the base state has transitioned to B_f. The view undergoes the following state transitions: $V_0 = \{(x_1, 4)\}$ (VM processes d_1 by scanning base table in state B_f), $V_1 = \{(x_1, 5)\}$ (VM processes i_2 locally), $V_2 = \{(x_1, 2)\}$ (VM processes u_3 by scanning the base state), $V_3 = \{(x_1, 5)\}$. While the computation converges, V_2 agrees with an earlier base state than V_1, thus violating strong consistency.

However, had the update program, instead of doing a local computation in V_1, when seeing i_2, also scanned the base table to compute the potentially new minimum, the state transitioned would have been strongly consistent. In that case $V_2 = \{(x_1, 5)\}$, since the base table scan would have found 5 as the MIN value. This underlines the classical trade-off between efficiency (optimized update propagation, by avoiding base-table scans through local MIN value update) and consistency (weak versus strong). In particular, an update program for MIN views that always scans the base table whenever a new MIN value is to be computed (whenever the existing MIN value is deleted or a lower value is being inserted) achieves strong consistency, but is not as efficient as the update program that only scans the base table whenever the MIN value is deleted.

B. Key-Foreign Key Join Views

In this section we summarize our analysis of the key-foreign key join view (K-FK). The K-FK join view is defined over two base tables $R(\underline{X}, Y)$ and $S(\underline{Y}, Z)$, where Y is the foreign key in R and $D(\underline{X}, Y, Z) = R(X, Y) \bowtie S(Y, Z)$, The foreign key constraint implies that given $(x, y) \in R \Rightarrow (y, z) \in S$.

The K-FK join view update propagation algorithms for base table inserts, deletes, and updates are given in Alg. 4. Just like the MIN-view, the K-FK join view requires base table access to propagate certain updates, which prevents strong consistency. The update programs are simplified for presentation suppressing the error handling and concurrent update propagation logic. The update programs exploit the fact that K-FK constraints are satisfied. Constraints are tracked by a separate system service and are instantiated in the base tables when the view manager receives an update. That is $(x, y) \in R \Rightarrow (y, z) \in S$. For example, propagating inserts or deletes on S has no effect on the view table, as the K-FK constraint guarantees that there is no matching record in R.

Also note that propagating an update, $u(S(y, z_1 \rightarrow z_2))$, on S to D is, unlike in the other view update propagation cases in this section, not a delete–insert pair. A delete-insert pair would have no effect on the view (both operations are no-ops), but there could be multiple records in the view from earlier inserts that need to reflect the changes for z. In the program we denote the fact that multiple records are read and updated in D with a "$*$" next to the read/update operation. In the full implementation, the read requires a scan of the entire base table, as the key is not provided as input, unless an index is maintained on D.

Algorithm 4 K-FK join view update propagation.

```
propagateInsert(R(x, y)):          propagateInsert(S(y, z)):
    read(S(y, ?z))                     {no operation}
    insert(V(x, y, z))

propagateDelete(R(x, y)):          propagateDelete(S(y, z)):
    delete(V(x, ?y, ?z))               {no operation}

propagateUpdate(R(x, y→y')):       propagateUpdate(S(y, z→z')):
    read(S(y', ?z))                    read*(V(?x, y, z))
    update(V(x, y→y', z))              update*(V(x, y, z→z'))
```

We conclude the discussion of K-FK join view with the convergence and consistency result. Refer to [24] for the complete proof.

Theorem 3: The K-FK join view is weakly consistent.

C. Selection, Projection and Set Operation Views

Selection, projection and set operation views are strongly consistent. Update propagation appears seemingly simple. We summarize a few subtleties below. Refer to [24] for a detailed analysis.

For key-preserving projection and selection views, the update program applies the selection condition to the base table update, projects out any values not required by the view, and inserts the record into the view. Base table deletes and updates are propagated in the same manner. Update signatures are not required since the view update programs are idempotent. Concurrent update propagation is not an issue, since two base table updates on the same key cannot race due to flow control, thus avoiding conflicts. However, it is crucial that the flow control principle is respected, as, otherwise, the view computation violates weak consistency and may not converge.

Set operation and non-key-including projection views are similar to the COUNT-view. Auxiliary "count" fields in the view are maintained to keep track of the records mapping onto one and the same view record from the base table: the view counts the values in the base relation resulting from projecting out the key.

VI. Experimental Evaluation

In this section, we present performance experiments that evaluate the effect of various techniques we developed in this paper for aggregate views. In particular, we study the maintenance of SUM views and MIN views to illustrate the important issues. SUM view behavior is similar to COUNT and AVG view behavior. Thus, by conducting experiments for the SUM and MIN views, we understand the performance of all aggregate views.

In our experiments we studied how our solution techniques for aggregate view maintenance scales with the number of view managers. We also studied the performance overhead of using update signatures to achieve correct update propagation in the presence of failures. Our experiments also throw light on the restrictions and the corresponding performance penalties imposed by flow control techniques that are needed to ensure correct view update propagation.

Our experiments are performed on the storage framework described in Sec. II-C. For each storage unit, we used MySQL as the engine to drive the updates. The storage framework was deployed on a cluster of 20 nodes (Linux 2.6, 2 x dual-core Xeon 5160 @ 3.00GHz, 4 GB RAM) connected by a 1 Gbps Ethernet switch. We deployed 20 storage servers, 10 log servers, and 20 machines were used for hosting view managers. We also used one of the machines to host a client application responsible for issuing base-table updates.

In our experiments, we started with a base table that is partitioned across multiple storage units (SUs). We studied the performance of view-update propagation to a SUM view and a MIN view. For each experiment, we initialized the base table with 1 million entries and inserted the corresponding entries directly into the view table. Once the base table and the view table are initialized, we started the view managers and the client application that performed base table updates. We developed the client application to uniformly sample the base table's key space and update the record. Note, the traffic load produced by the client application is set up to provide enough traffic that all the view managers can stay busy throughout the experiment propagating the updates to the views.

Experiments for SUM View: Fig. 2 shows the results of the experiment to study the effect of the view cardinality on the throughput achieved for the SUM view. When there are fewer view entries, we expect a higher probability that two or more view managers are concurrently updating the same view entry. That is, as the cardinality of the view grows, there is less contention and we can achieve more throughput through a larger number of view managers. We see that as we add more view managers to the system, the throughput continues to increase until the contention rate reaches about 750; after which, throughput begins to decrease. This illustrates that we should not blindly increase the number of view managers in hope of increasing throughput.

Fig. 3 studies the effect of the number of view managers on TAS contention for different view cardinalities. As expected, the contention increases as the number of view manager increases. We see that with a modest number of view records (like 1000s), we have very low TAS contention even with a large number of view managers.

Fig. 5 and Fig. 6 show the throughput and contention rate when update signatures are used to protect against failures. We observe that the overhead of update signatures is about 9% on the overall throughput. Thus, we conclude that the slight decrease in throughput is an acceptable trade-off to achieve consistent update propagation through the use of update signatures in the presence of failures.

MIN View Experiments: In the case of MIN view maintenance, the key issue is that the cost of reading the minimum value from the base table can be prohibitive. However, the likelihood of having to incur this cost depends on the sequence of updates on the base table. To study this effect, we varied the distribution of the minimum value in the data.

As shown in Fig. 4, the cost of reading the minimum value from the base table has little impact on throughput if the input updates follow a discrete uniform or a discrete normal distribution since the MIN view records are rarely updated in response to base-table update propagation. In fact, the two data distributions share the same throughput characteristics. However, as expected, in the case of a Zipfian distribution, the cost of reading the minimum value from the base table becomes too expensive and the value of adding more view managers is quickly diminished.

We can avoid the frequency of having to compute a new MIN by reading the base table, when the MIN view is updated, by maintaining a counter field to keep track of the number of occurrences of the MIN value in the base records. For instance, if a base-table record is deleted and it has the MIN value, this update when propagated to the view may suggest that a new MIN value should be computed by reading the base table. However, if the counter on the view records indicates that there are more than one base records with that MIN value, then the update program can simply decrement this counter and avoid reading the base table to compute a new MIN value.

As shown in Fig. 7, the cost of reading the minimum value from the base table is amortized by the reference counters (ref. cnt.) and good throughput scalability is achieved for all three distributions. Reference counters introduce almost no overhead when the data follows a discrete uniform or a discrete normal distribution since the view managers rarely need to modify the min view record. Again, the two data distributions share the same throughput characteristics. For data with Zipfian distribution, the overall throughput is lower than the other two distributions because it incurs a higher

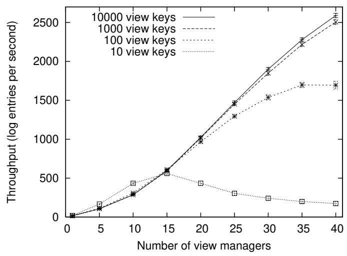

Fig. 2. cnt-TAS SUM view throughput rate

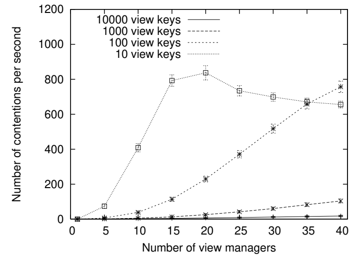

Fig. 3. cnt-TAS SUM view contention rate

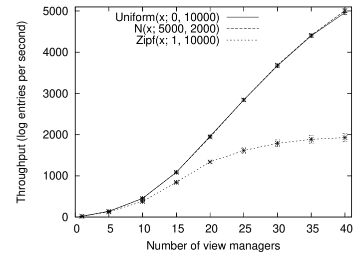

Fig. 4. MIN view throughput rate (without ref. cnt.)

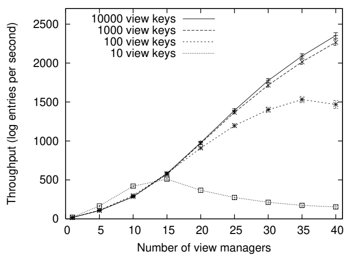

Fig. 5. sig-TAS SUM view throughput rate

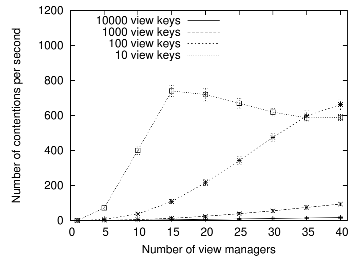

Fig. 6. sig-TAS SUM view contention rate

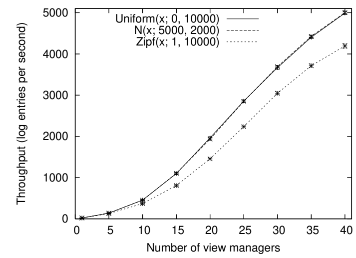

Fig. 7. MIN view throughput rate (with ref. cnt.)

overhead for updating reference counters. This is due to the fact that reference counters are updated much more frequently.

For both test cases (with and without reference counter), the highest observed contention rate was less than 24 TAS contentions per second. Thus TAS failures have little impact on the view managers' overall performance.

Impact of Flow Control: When storage units (SUs) experience uneven workloads, flow control may impact system performance since view managers may have uneven workloads and consequently some view managers may idle. We conducted experiments for SUM view maintenance wherein we explored the effect of unbalanced versus balanced workloads. For the unbalanced workload, we used a highly skewed Zipfian distribution to model the update frequency for different base records. For the balanced workload, we used a uniform distribution of updates on base-table records.

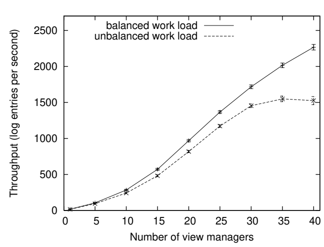

Fig. 8. SU workload

During our experiments, the most heavily loaded storage unit handled 9 times more requests than the storage unit with the lightest workload. Fig. 8 shows the throughput of both balanced and unbalanced workloads. Although the throughput of the balanced workload increases at a near linear rate, the throughput for the unbalanced workload levels off at about 1500 updates per second. We also observed the unbalanced workload has less TAS contentions than the balanced workload. Under both workloads, since there are less than 74 TAS contentions per second, TAS failures have little influence on the view managers' throughput.

Even though our experiments showed that in the case of highly-skewed workloads there can be throughput degradation due to flow control, we can achieve significant throughput by having a large number of view managers that follow flow control to ensure a reasonable level of consistency in view-update propagation.

In summary, from our experiments we conclude:

1) Scalability of SUM view maintenance: The update propagation throughput of SUM view is significantly impacted by the cardinality of the SUM view.
2) TAS contention of SUM view: The contention is quite low if the cardinality of the SUM view is reasonably high.
3) Overhead of update signatures: Update signatures, which ensure correct update propagation in the presence of failures, has very little overhead on the throughput.
4) MIN view throughput: The use of counters to reduce the frequency of having to read the base table in order to compute the MIN value can be very beneficial.
5) Flow control impact: For highly-skewed update distributions flow control can lead to significant reduction of view-update throughput. However, this reduction in throughput can be overcome by having additional view managers in the system.

VII. Related Work

In this section we compare our work to existing *incremental and deferred view maintenance* approaches [1]–[8]. We also contrast our work to existing work on *view maintenance in distributed database systems* [9]–[15].

Incremental view maintenance has been studied extensively over the years [1]–[8]. Incremental view maintenance algorithms compute view differentials as query expressions over base table updates, base states before or after the update, and view states [2]–[5]. Our approach leverages this work, but must get by with very restricted query support since WDPs often only provide record get and set operations.

View maintenance can be broadly classified as immediate versus deferred maintenance. In the immediate case, view maintenance overhead is incurred as part of the update transaction. The base table update triggers the immediate propagation of the change to the view. This tight coupling is not possible since transactions to tie base update and view maintenance operation together do not exist in the context of WDPs. Futhermore, update propagation happens asynchronously, where updates to base tables "trigger" view update propagation that result in updates (inserts, deletes, and changes) to view tables, executed as normal query operations to the system.

Deferred view maintenance delays the update propagation and shifts the burden of updating views from the update transaction to other points in time, such as when a view is in use [1], or when the system is idle [8]. To prevent inconsistencies in the view due changing base tables, deferred view maintenance techniques employ row versioning and version stores, and rely on snapshot isolation. Existing solutions are inapplicable since WPDs do not offer these mechanisms. Recently, Zhou *et al.* [7] proposed dynamic view maintenance techniques to selectively materialize parts of a view most frequently used. This technique is complimentary to our approach.

Seemingly closest to the work presented in this paper are the approaches on view maintenance in distributed databases and data warehouse environments [9]–[15]. Based on the consistency definition by Hull and Zhou [15], Zhuge *et al.* [9] developed a consistency model to characterize convergence properties of deferred view maintenance in multi-source warehouse environments. Their consistency model looks similar to ours at first glance. However, their model is based on characterizing base and view states as database snapshots, whereas our model operates at the record-level without the existence of snapshots to reason about. Their model is therefore much coarser grained and cannot capture the structures we define to characterize consistency at the record-level. For example, a state sequence that may be strongly consistent in our model, could turn out to be invalid in their model. This is not a flaw in the earlier model, which is simply aiming to characterize computation in a different context. Moreover, the earlier approaches are based on transactional guarantees, ordered update propagation, and snapshot isolation [9], [12], which are not available in the context we target.

Luo *et al.* [14] develop techniques to address the problem of efficient join view maintenance in a parallel RDBMS, where relations are partitioned across a cluster. This work is orthogonal to what is presented in this paper.

Chen *et al.* [10] present view maintenance techniques for distributed databases focusing on addressing problems resulting from schema changes of base tables and concurrent update propagation of base table changes. Their solution is based on ACID transactions and multi-versions of relations, which is not an option for WDPs, as motivated above.

VIII. Conclusions

WDPs are massively distributed systems that replicate data asynchronously across several data centers involving thousands of machines worldwide. This massive scale, the inherent parallelism, the asynchronous processing, and the lack of many traditional database mechanisms impose several challenges not addressed by existing view maintenance approaches. We show, for example, that component failures, despite recovery, can lead to the lack of convergence for certain view maintenance computations. We develop a consistency model to characterize when a view record agrees with the base records it is derived from. The model introduces several levels of consistency that help the application developer understand the properties a given view will exhibit as it is maintained. We formally establish the consistency properties for various classes of views based on the consistency model. We also develop several simple solution techniques, well suited to the constraints of WDPs, and offer experimental evidence for the practicality of our solutions.

References

[1] L. Colby et al. Algorithms for deferred view maintenance. In *SIGMOD*, 1996.
[2] J. Blakeley et al. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, 1986.
[3] T. Griffin et al. Incremental maintenance of views with duplicates. *SIGMOD Rec.*, 24(2):328–339, 1995.
[4] A. Gupta et al. Maintaining views incrementally. In *SIGMOD*, 1993.
[5] T. Palpanas et al. Incremental maintenance for non-distributive aggregate functions. In *VLDB*, 2002.
[6] A. Gupta et al. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Eng. Bulletin*, 18(2), 1995.
[7] J. Zhou et al. Lazy maintenance of materialized views. In *VLDB*, 2007.
[8] J. Zhou et al. Dynamic materialized views. In *ICDE*, 2007.
[9] Y. Zhuge et al. The Strobe algorithms for multi-source warehouse consistency. In *PDIS*, 1996.
[10] S. Chen and other. Multiversion-based view maintenance over distributed data sources. *TODS*, 29(4):675–709, 2004.
[11] D. Quass et al. Making views self-maintainable for data warehousing. In *PDIS*, 1996.
[12] D. Agrawal et al. Efficient view maintenance at data warehouses. In *SIGMOD*, 1997.
[13] J. Bailey et al. Incremental view maintenance by base relation tagging in distributed databases. *DPD*, 6(3):287–309, 1998.
[14] G. Luo et al. Comparison of three methods for join view maintenance in parallel RDBMS. In *ICDE*, 2003.
[15] R. Hull et al. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD*, 1996.
[16] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
[17] G. Decandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.
[18] B. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.
[19] Project voldemort. `http://project-voldemort.com`.
[20] A. Lakshman et al. Cassandra: Structured storage system over a p2p network. *`http://www.slideshare.net/jhammerb/data-presentations-cassandra-sigmod`*.
[21] K. Petersen et al. Flexible update propagation for weakly consistent replication. 1997.
[22] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.
[24] H.-A. Jacobsen et al. Consistency results. `http://www.eecg.toronto.edu/~jacobsen/crvmWDPs.pdf`, 03/09.