Consistency Results For View Maintenance On Web Data Platforms

Hans-Arno Jacobsen*, Patrick Lee
University of Toronto

Ramana Yerneni
Yahoo!, Inc.

Appendix to "*View Maintenance in Web Data Platforms.*"
Working Technical Communication April 15, 2009: Available from
`http://www.eecg.utoronto.ca/~jacobsen/crvmWDPs.pdf`.

1 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 [2, 1]. 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. We illustrate fundamental problems in maintaining views consistently in WDPs with examples and show the violation of the various levels of consistency. The examples were introduced in [3] in Section 2.

1.1 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 definitions on the base state B_i.

Each update, insert, and delete operation effects 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

*Part of this research was conducted while the author was on sabbatical at Yahoo!, Inc. in 2008.

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 another base state may contain $\{(u_3, u_1)\}$. That is because, the record for the last insert operation does not have the same key as that of any other base record under consideration. So, 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 effect, 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.

1.2 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. Formally, for each view record r,

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

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 view record r, for each intermediate view state V_j, 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 view record r, for each pair of intermediate view states V_{j_1} and V_{j_2} such that $V_{j_1} \leq V_{j_2}$, 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

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

Complete consistency: We say that the system achieves *complete consistency* if and only if it achieves *strong consistency* and every view record reflects all the changes in the base state (i.e., every record in the view computed from every base state is presented in some view state). Formally, for each view record r, for each base state B_i with $B_0 \leq B_i \leq B_f$, 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. In particular, views and other derived data is maintained in most systems in a manner that the information in the views agrees with the base state. However, there is not much benefit in maintaining views in a manner that reflects every possible base state. 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.

1.3 Violation of consistency

In this section we illustrate how the three fundamental issues presented in [3] Section 2.2 may lead to inconsistencies in the maintenance of views in WDPs. In [3], we 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, where 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.□

Note that 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.□

Note that 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: $u_1(R(k_1, x_1, y \rightarrow 10))$, $u_2(R(k_1, x_1, y \rightarrow 20))$, and $u_3(R(k_1, x_1, y \rightarrow 15))$. 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 vio-

lation 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 to be propagated to a view, they can be applied out of order and cause consistency violations.

2 Classes of Views

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

2.1 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.

2.1.1 COUNT view

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
```

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 Algorithm 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 test&set insert operation is

successfully completed, then the program is done. On the other hand, if the test&set 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 test&set 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.[1] If the program determines the log entry was processed on a prior occasion, then it simply ignores that entry. Otherwise, the program issues a test&set update operation (Line 7) to increment the view record's count. Once again, if the test&set update operation succeeds, then the program is done. However, if the test&set 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 test&set 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 Algorithm 2 and Algorithm 3, respectively. The

[1] If the test&set is signature-based, then, as described in [3], $hasProcess$ must compare the signature, sn, against the log entry identified by its stream ID and entry timestamp, sid and eid, respectively. If the test&set is counter-based, then $hasProcess$ always return $False$ since duplicate base log entries are undetectable.

propagation of a delete operation on the base table is quite similar to 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.[2]

The update programs for SUM and AVG views are quite similar to those of COUNT views.[3] For instance, the SUM view's propagation are shown in Algorithm 4-6. 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.

Algorithm 4 Insert propagation for SUM view

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

Algorithm 5 Delete propagation for SUM view

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

2.1.2 Convergence and consistency

Below, we show that the sum, count and avg views computation converge and are strongly consistent.

[2] Not all view update propagation follow this structure.
[3] AVG stores the sum and the count in each view record.

Algorithm 6 Update propagation for SUM view

```
propagateUpdate(R(k, x→x', y→y'), sid, eid):
    if x = x' then
      propagateInsert(R(k, x, y' - y), sid, eid)
    else
      propagateDelete(R(k, x, y), sid, eid)
5:    propagateInsert(R(k, x', y'), sid, eid)
```

Theorem 1: The sum view is guaranteed to converge.

Proof: We will first define some notations then proceed to inductively prove the convergence property.

Let

$$U = \bigcup_i \{k | R(k, x, y) \in B_i\}$$

be the set of all possible base entry keys (i.e. the base entry key's domain). We can then define $f(\beta, k, x)$ as the contribution by the base entry identified by k towards the view entry identified by x for $View(B_\beta)$. More formally,

$$f(\beta, k, x) = \begin{cases} y & \text{if } \exists y,\ R(k, x, y) \in B_\beta \\ 0 & \text{otherwise} \end{cases}$$

The property stated below immediately follows from this definition

$$View(B_\beta)(x) = \sum_{k' \in U} f(\beta, k', x)$$

Now, let us assume that the view is initially correct (i.e. $View(B_o) = V_o$). Suppose for any arbitrary number of base table operations, α, that the view converges. We now show that the view converges after $\alpha + 1$ operations.

case 1: insert operation i(R(k, x, y))

Since the insert operation is the $(\alpha+1)$-th operation, this implies that $f(\alpha, k, x) = 0$. Hence the base table entry identified by k did not contribute to any view entry in $View(B_\alpha)$. All other base table entries are not effected by the insert operation and $f(\alpha, k, x) = 0$, therefore

$$View(B_\alpha)(x) = \sum_{k' \in U} f(\alpha, k', x) = \sum_{k' \in U \setminus \{k\}} f(\alpha + 1, k', x)$$

By the induction hypothesis, $V_\alpha = View(B_\alpha)$. Thus adding y to $V_\alpha(x)$

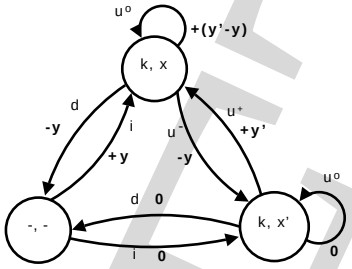

Figure 1: Valid sum view operation sequences with respect to view key x

results in

$$\begin{aligned} V_{\alpha+1}(x) = V_\alpha(x) + y &= View(B_\alpha)(x) + y \\ &= \sum_{k' \in U} f(\alpha, k', x) + y \\ &= \sum_{k' \in U \setminus \{k\}} f(\alpha+1, k', x) + y \\ &= \sum_{k' \in U \setminus \{k\}} f(\alpha+1, k', x) + f(\alpha+1, k, x) \\ &= \sum_{k' \in U} f(\alpha+1, k', x) \\ &= View(B_{\alpha+1})(x) \end{aligned}$$

Similar arguments can be applied to the other cases. As such, we will only note the differences between them. Figure 1 shows the state transition diagram that characterizes all valid sequences of operations on the base record identified by key k, from the perspective of view record with key x.

case 2: delete operation d(R(k, x, y))

$$f(\alpha, k, x) = y \quad \text{and} \quad f(\alpha+1, k, x) = 0$$

case 3a: update operation u(R(k, x, y → y'))

$$f(\alpha, k, x) = y \quad \text{and} \quad f(\alpha+1, k, x) = y'$$

case 3b: update operation u(R(k, x → x', y → y'))

$$\begin{aligned} f(\alpha, k, x) = y \quad &\text{and} \quad f(\alpha+1, k, x) = 0 \\ f(\alpha, k, x') = 0 \quad &\text{and} \quad f(\alpha+1, k, x') = y' \end{aligned}$$

□

Weak and subsequently strong consistency is established by constructing for every valid view state a base state that agrees with the view. We also show that

this construction is order-preserving in terms of the definition of strong consistency.

Theorem 2: The sum view computation is strongly consistent.

Proof: First we establish weak consistency by showing that every view state is valid. We show, given any view record, (x, n_1), there exists a base state, B_1, such that $B_0 \leq B_1 \leq B_f$ and $View(B_1)(x) = (x, n_1)$.

Let S_1 be the sequence of base operations propagated by the view manager to take x from V_0 to (x, n_1) and let S_3 be the operations that do not effect x.

Now, construct a new sequence of operations, S_2, that extends S_1 by interleaving it with S_3 operations in order to satisfy the record-time-line requirement.[4]

The base state B_1 is constructed as follows: $B_1 = S_2(B_0)$, where B_1 is resulted from applying the base table operations in S_2 to B_0. For B_1, $View(B_1)(x) = (x, n_1)$, establishing weak consistency for the view computation.

To show strong consistency we proceed in the same manner. We show, given a subsequent state of the same view record, (x, n_2), there exists a base state, B_2, such that $B_0 \leq B_1 \leq B_2 \leq B_f$ with $View(B_1)(x) = (x, n_1)$ and $View(B_2)(x) = (x, n_2)$. B_1 is as above.

To construct B_2, let S_1' be the sequence of base operations propagated by the view manager to take x from the view state with (x, n_1) to the subsequent view state with (x, n_2) and S_3 is as before.

Now, construct a new sequence of operations, S_2' by interleaving S_1' with the operations in S_3 to satisfy the record-time-line requirement, as illustrated above. B_2 is then constructed similar to B_1. That is $B_2 = S_2'(B_1)$.

Thus, $B_0 \leq B_1 \leq B_2 \leq B_f$, $View(B_1)(x) = (x, n_1)$, and $View(B_2)(x) = (x, n_2)$ establish strong consistency for the view computation. □

Theorem 3: The count view converges and is strongly consistent.

Proof: The count view a special case of the sum view, where

$$f(\alpha, k, x) = 1 \Leftrightarrow R(k, x, y) \in B_\alpha$$

Therefore, our consistency results for the sum view is also applicable to the count view. □

Theorem 4: The avg view converges and is strongly consistent.

Proof: As explained in Section 2.1.1, the avg view is constructed by storing the sum and the count in each view record. Since both the sum and count views converge, the avg view must also converge. Furthermore, because both the sum

[4] The interleaving of S_1 with S_3 is required, as illustrated by this example. Assume an initial state with $R = \{\}$ and $V = \{\}$ and an update stream: $i_1(k_1, x_2, y_2)$, $d_2(k_1, x_2, y_2)$, $i_3(k_1, x_1, y_1)$, $d_4(k_1, x_1, y_1)$, $i_5(k_1, x_1, y_1)$. Given the intermediate view state $(x_1, 1)$ (after propagating i_3), $S_1 = \{i_3\}$ and $S_3 = \{i_1, d_2\}$. Would we construct the base state B_1 as simply $B_1 = S_1$, B_1 would not reflect the updates i_1 and d_2, thus violate the record-time-line requirement for the base record with key, $k1$.

and count views are strongly consistent, the avg view is also strongly consistent. □

2.1.3 MIN and MAX views

Algorithm 7 Insert propagation for MIN view

```
propagateInsert(R(k, x, y), sid, eid):
    loop
      foundEntry ← read(D(x, ?m, ?count), ?sn)
      if foundEntry then
        if y < m then
 5:       succeed ← updateTAS(D(x, y, 1), sn, sn + 1)
        else if y = m then
          succeed ← updateTAS(D(x, m, count + 1), sn, sn + 1)
        else
          break
10:   else
        succeed ← insertTAS(D(x, y), 0)
      if succeed then
        break
```

Algorithm 8 Delete propagation for MIN view

```
propagateDelete(R(k, x, y), sid, eid):
    loop
      foundEntry ← read(D(x, ?m, ?count), ?sn)
      if not foundEntry then
        break
 5:   if y ≠ m then
        break
      if count > 1 then
        succeed ← updateTAS(D(x, m, count - 1), sn, sn + 1)
      else
10:     foundBaseMin ← read(R(-, x, min(y) → baseMin))
        if foundBaseMin then
          succeed ← updateTAS(D(x, baseMin, 1), sn, sn + 1)
        else
          succeed ← deleteTAS(D(x, m, count), sn)
15:   if succeed then
        break
```

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.

Algorithm 9 Update propagation for MIN view

propagateUpdate(R(k, x→x', y→y'), sid, eid):
    propagateDelete(R(k, x, y), sid, eid)
    propagateInsert(R(k, x', y'), sid, eid)

Below we prove that MIN-view computation is weakly consistent. Convergence can be established by induction in a similar manner.

Theorem 5: The MIN computation is weakly consistent.

Proof: We show that all view states are valid. There are two cases.

1. Propagation requires base table access: A new minimum is computed by scanning the base table. This scan is guaranteed to see a cross-section of records going forward in time. Moreover, the scan was triggered after B_0 and can not see beyond the final state, B_f. Therefore the minimum value computed is valid.

2. Propagation gets by with local computation: A new minimum is installed due to the propagation of a base table update that holds a value smaller than the current minimum in the view. Therefore, the newly installed value corresponds to a valid base record state.

□

While the MIN view computation 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.

2.2 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 Algorithm 10. 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 10 K-FK join view update propagation.

```
propagateInsert(R(x, y)):
    read(S(y, ?z))
    insert(V(x, y, z))

propagateInsert(S(y, z)):
    {no operation}

propagateDelete(R(x, y)):
    delete(V(x, ?y, ?z))

propagateDelete(S(y, z)):
    {no operation}

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

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

We conclude the discussion of K-FK join view with the convergence and consistency result.

Theorem 6: The key-foreign key join converges.

Proof: We show (i) $(x, y, z) \in View(B_f) \Rightarrow (x, y, z) \in V_f$ and (ii) $(x, y, z) \notin View(B_f) \Rightarrow (x, y, z) \notin V_f$. We consider inserts and deletes, the proof including updates follows the same scheme, introducing several additional cases.

(i) $(x, y, z) \in View(B_f)$ implies that

a. either the last operation on $R(x, y)$ is $i(R(x, y))$ and $(y, z) \in S$ when the view manager processes the insert due to key-foreign key constraint,

b. or initially $(x, y) \in R$ implying that $(y, z) \in S$ initially as well, due to key-foreign key constraint and no other operations effect either $R(x, y)$ or $S(x, y)$.

For a., the view manager inserts the tuple (x, y, z) into the view and never changes it. For b., $(x, y, z) \in V_0$ and it never changes. Thus, $(x, y, z) \in V_f$.

(ii) $(x, y, z) \notin View(B_f)$ implies that

a. either the last operation on $R(x, y)$ is $d(R(x, y))$,

b. or initially $(x, y) \notin R_0$ and no other operations yield $(x, y) \in R$.

For a., the view manager deletes the tuple (x, y, z) from the view. For b., $(x, y, z) \notin V_0$ and it is never inserted. Thus, $(x, y, z) \notin V_f$. $\square$

We now show the K-FK join view is weakly consistent, but violates strong consistency. That is an observer of the view will only observe valid view states, but may notice view states that are out-of-sequence with respect to base states.

Theorem 7: The key-foreign key join is weakly consistent.

Proof: We proof weak consistency by contradiction. Assume (x, y, z) is an invalid view record. We consider inserts and deletes, the proof including updates follows the same scheme, introducing several additional cases.

There are two cases to consider:

a. $(x, y) \notin R_0$ and $(y, z) \notin S_0$

b. $(x, y) \notin R_0$ and $(y, z) \in S_0$

The additional two cases, where $(x, y) \in R_0$ are trivial. $(y, z) \in S_0$ contradicts the assumption outright. $(y, z) \notin S_0$ is not possible due to key-foreign key constraint requirement.

For a., $i(R(x, y))$ comes to the view manager and a base table check on S yields (y, z), resulting in $(x, y, z) \in V$ (due to key-foreign key constraint the S tuple must exist at that point, otherwise the constraint would be violated.) Now, construct a base state, B, starting from R_0 and include all operations up to and including $i(R(x, y))$ on R, and starting from S_0 and include all operations up to and including $i(S(x, y))$ on S. For B, we have $B_0 \leq B$ and $View(B)(x) = (x, y, z)$, a valid view record.

For b., $i(R(x, y))$ comes to the view manager and a base table check on S yields (y, z), resulting in $(x, y, z) \in V$. B is constructed in the exact same manner as above, only considering operations on R. $\square$

We show violation of strong consistency with an example. The initial base state is: $R = \{\} S = \{(y, z)\}$. The base table update stream is: $i_1(R(x, y))$, $u_2(S(y, z \rightarrow z''))$, $u_3(S(y, z'' \rightarrow z'''))$.

The view manager processes the update stream after all operations took effect on the base state, which is, $R = \{(x, y)\}, S = \{(y, z''')\}$, at that point.

The update stream takes the initial view state from $V_0 = \{\}$ to $V_1 = \{(x, y, z''')\}$ via i_1 and from V_1 to $V_2 = \{(x, y, z'')\}$ via u_2 and so on. While the view computation eventually converges, from the given view state sequence it follows that there exists no B_i, B_j with $B_i \leq B_j$ such that $V_1 = View(B_i)(x), V_2 = View(B_j)(x)$, and $V_1 \leq V_2$. V_1 only agrees with the final base state and V_2 agrees with an earlier base state, violating strong consistency.

For example, given the initial base state, $R(\underline{K}, X, Y) = \{(k_1, x_1, y_1)\}$, the initial view state, $V = \{(k_1, x_1, y_1)\}$ (the view is a selection view), and the update stream: $u_1(k_1, x_1, y_1 \rightarrow y_2)$ and $u_2(k_1, x_1 \rightarrow x_2, y_2)$. If the flow control principle is implemented the final view state is $V_f = \{k_1, x_2, y_2\}$. The below state sequences illustrate violation of weak consistency and lack of convergence.

B	(k_1, x_1, y_1)	u_1	(k_1, x_1, y_2)	u_2	(k_1, x_2, y_2)
V^{FC}	(k_1, x_1, y_1)	u_1	(k_1, x_1, y_2)	u_2	(k_1, x_2, y_2)
V^c	(k_1, x_1, y_1)	u_2	$\underline{(k_1, x_2, y_1)}$	u_1	(k_1, x_2, y_2)
V^m	(k_1, x_1, y_1)	u_2'	$\overline{(k_1, x_2, y_2)}$	u_1'	$\underline{(k_1, x_1, y_2)}$

The sequence labeled B is the base state sequence of processing u_1 followed by u_2. The sequence labeled V^{FC} is the view state sequence resulting due to flow control; updates to the same base record are processed by the view manager in the order they occurred in the base state. The sequence labeled V^c is the view state sequence resulting from processing the base updates at the view manager out-of-order. While the computation converges, the second state is invalid, thus violating weak consistency. The final sequence, labeled V^m, is the view state sequence resulting from the same out-of-order processing as V^c. The base updates labeled u_2' and u_1' propagate the entire base record, not just the fields that changed. The out-of-order propagation of u_1' overwrites the x_2 with the old value of x_1 in the message, thus preventing convergence.

A non-key-including projection view is similar to the COUNT-view. An auxiliary "count" field in the view is maintained to keep track of the tuples mapping onto one and the same view tuple from the base table. Essentially, the view counts the values in the base relation resulting from projecting out the key.

2.3 Selection and projection views

Selection and projection views are strongly consistent. Update propagation appears seemingly simple. We summarize a few subtleties below.

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

Algorithm 11 Insert propagation for selection view

```
propagateInsert(R(k, x, y), sid, eid):
    if not condition(R(k, x, y)) then
      return
    loop
      foundEntry ← read(D(k, -, -), ?sn)
 5:   if foundEntry then
        break
      sn' ← generateSignature(NULL, sid, eid)
      succeed ← insertTAS(D(k, x, y), sn')
      if succeed then
10:     break
```

Algorithm 12 Delete propagation for selection view

```
propagateDelete(R(k, x, y), sid, eid):
    if not condition(R(k, x, y)) then
      return
    loop
      foundEntry ← read(D(k, -, -), ?sn)
 5:   if not foundEntry then
        break
      if hasProcessed(sn, sid, eid) then
        break
      succeed ← deleteTAS(D(k, x, y), sn)
10:   if succeed then
        break
```

Algorithm 13 Update propagation for selection view

```
propagateUpdate(R(k, x→x', y→y'), sid, eid):
    loop
      foundEntry ← read(D(k, -, -), ?sn)
      if foundEntry then
        if hasProcessed(sn, sid, eid) then
 5:       break
        if condition(R(k, x', y')) then
          sn' ← generateSignature(sn, sid, eid)
          succeed ← updateTAS(D(k, x', y'), sn, sn')
        else
10:       succeed ← deleteTAS(D(k, x, y), sn)
      else
        if condition(R(k, x', y')) then
          sn' ← generateSignature(NULL, sid, eid)
          succeed ← insertTAS(D(k, x', y'), sn')
15:     else
          break
      if succeed then
        break
```

Algorithm 14 Insert propagation for key-preserving projection view

```
propagateInsert(R(k, x, y), sid, eid):
    loop
      foundEntry ← read(D(k, -), ?sn)
      if foundEntry then
        break
 5:   sn' ← generateSignature(NULL, sid, eid)
      succeed ← insertTAS(D(k, x), sn')
      if succeed then
        break
```

Algorithm 15 Delete propagation for key-preserving projection view

```
propagateDelete(R(k, x, y), sid, eid):
    loop
      foundEntry ← read(D(k, -), ?sn)
      if not foundEntry then
        break
 5:   if hasProcessed(sn, sid, eid) then
        break
      succeed ← deleteTAS(D(k, x), sn)
      if succeed then
        break
```

Algorithm 16 Update propagation for key-preserving projection view

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

Algorithm 17 Insert propagation for non-key-preserving projection view

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

Algorithm 18 Delete propagation for non-key-preserving projection view

```
propagateDelete(R(k, x, y), sid, eid):
    loop
      foundEntry ← read(D(x, k), ?sn)
      if not foundEntry then
        break
 5:   if hasProcessed(sn, sid, eid) then
        break
      succeed ← deleteTAS(D(x, k), sn)
      if succeed then
        break
```

Algorithm 19 Update propagation for non-key-preserving projection view

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

weak consistency and may not converge.

Algorithm 20 Insert propagation for key-not-including projection view

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

A non-key-including projection view is similar to the COUNT-view. An auxiliary "count" field in the view is maintained to keep track of the records mapping onto one and the same view record from the base table. Essentially, the view counts the values in the base relation resulting from projecting out the key.

2.4 Set operation views

Views defined as set union, intersection, and difference operation of relations can be maintained by either accessing the base tables or by managing auxiliary data as part of the view table and shifting part of the view computation to the retrieval stage.

For example, consider the base relations, $R(\underline{K}, X)$ and $S(\underline{K}, X)$, the view $D(\underline{K}, \underline{X}, C)$ defined as $D(\underline{K}, \underline{X}, C) = R(K, X) \cap S(K, X)$, where C is the auxiliary data in the view that keeps track of the existing number of tuples for

Algorithm 21 Delete propagation for key-not-including projection view

```
propagateDelete(R(k, x, y), sid, eid):
    loop
      foundEntry ← read(D(x, y, ?count), ?sn)
      if not foundEntry then
        break
 5:   if hasProcessed(sn, sid, eid) then
        break
      if count = 1 then
        succeed ← deleteTAS(D(x, y, count), sn)
      else
10:     sn' ← generateSignature(sn, sid, eid)
        succeed ← updateTAS(D(x, y, count - 1), sn, sn')
      if succeed then
        break
```

Algorithm 22 Update propagation for key-not-including projection view

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

a given key. A query against the view is defined to check the value of C against the number of relations intersected, before returning a tuple. For base tables $R = \{(k_1, x_1), (k_2, x_2)\}$ and $S = \{(k_1, x_2), (k_2, x_2)\}$, the view is $D = \{(k_1, x_1, 1), (k_1, x_2, 1), (k_2, x_2, 2)\}$. Here, the tuple with key, k_2:k_2, is the only true result in the view.

Without auxiliary information, the view would represent the exact result, but an insert on R and S propagated to the view, requires a base table check on S and R, respectively. Propagating updates from R and S require a similar base table check by the view manager, unless auxiliary information is kept. Similar reasoning applies to union and difference view maintenance.

This trade-off is similar to the MIN-view maintenance case, but, unlike for MIN views, set operation-based view maintenance that accesses base state are weakly consistent, while the auxiliary data-based variant without base state access is strongly consistent. This is illustrated by way of the following example.

The stream of base table updates is as follows: $i_1(k_1, x_3)$, $u_2(R(k_1, x_1 \rightarrow x_2))$, $u_3(R(k_1, x_2 \rightarrow x_3))$ and $u_4(k_1, x_3 \rightarrow x_2)$. This results in the following states:

	B_0		intermediate states		B_f
R:	$\{(k_1, x_1)\}$	u_2	$\{(k_1, x_2)\}$	u_3	$\{(k_1, x_3)\}$
S:		i_1	$\{(k_1, x_3)\}$	u_4	$\{(k_1, x_2)\}$

The view manager sees the same update stream after the base table R has transitioned to B_f. The initial and final view states are $V_0 = \{\}$ and $V_f = \{\}$, respectively. The first line below illustrates the view state sequence undergone by the base state access-based view implementation. The second line below illustrates the view state sequence undergone by the auxiliary data-based view implementation. The third line describes the exact view result for this case for clarity.

$\{(k_1, x_3)\}$	$\{(k_1, x_2)\}$	$\{\}$	$\{\}$
$\{(k_1, x_1, 1), (k_1, x_3, 1)\}$	$\{(k_1, x_2, 1), (k_1, x_3, 1)\}$	$\{(k_1, x_3, 2)\}$	$\{(k_1, x_3, 1), (k_1, x_2, 1)\}$
$\{\}$	$\{\}$	$\{(k_1, x_3)\}$	$\{\}$

The view state $\{(k_1, x_3)\}$ that results from propagation of i_1 in V_0 can only agree with a base record value in B_f (the required base record exits in no earlier state), while the later view state $\{(k_1, x_2)\}$ can only agree with an earlier base state. This demonstrates violation of strong consistency. However, the auxiliary data-based implementation does not exhibit this problem and is strongly consistent.

References

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

[2] G. Decandia, D. Hastorun, et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[3] H.-A. Jacobsen, P. Lee, and R. Yerneni. View maintenance in web data platforms. Technical report, (Under submission to VLDB'09), 2009.