Incremental Recomputations in Distributed Materialized Views

Sandy Ganza

January 31, 2014
1 Introduction
   - Terminology

2 Incremental Recomputations in Materialized Views
   - The View Maintenance Problem - Dimensions
   - A Mechanism for Efficient Materialized View Updates
   - Production Rules for Incremental View Maintenance
   - Self-Maintainability of Views

3 View Maintenance Policies
   - Policies

4 Incremental Recomputations in Distributed Materialized Views
   - Consistency in Incremental View Maintenance
   - Eager Compensating Algorithms (ECA)
   - The Strobe Algorithms

5 Conclusion
What is a view?

A relation that is derived from a set of base relations
A function that maps a set of base tables to a derived table
A view can be used as a table
Function recomputed every time the view is referenced
View results are virtual tables and are not stored on the disk
What is a view?

- A relation that is derived from a set of base relations
What is a view?

- A relation that is derived from a set of base relations
- A function that maps a set of base tables to a derived table
What is a view?

- A relation that is derived from a set of base relations
- A function that maps a set of base tables to a derived table
- A view can be used as a table
What is a view?

- A relation that is derived from a set of base relations
- A function that maps a set of base tables to a derived table
- A view can be used as a table
- Function recomputed every time the view is referenced
What is a view?

- A relation that is derived from a set of base relations
- A function that maps a set of base tables to a derived table
- A view can be used as a table
- Function recomputed every time the view is referenced
- View results are virtual tables and are not stored on the disk
Drawbacks of views

- Query executed every time the view is invoked
- Poor performance for repeated and complex queries

Sandy Ganza
Drawbacks of views

- Query executed every time the view is invoked
Drawbacks of views

- Query executed every time the view is invoked
- Poor performance for repeated and complex queries
What is a materialized view (MV)?

A view that has been precomputed and persisted.
A copy of the data defined by the view - data cache.
Query definition not executed on each reference to the view.
What is a materialized view (MV)?

- A view that has been precomputed and persisted
What is a materialized view (MV)?

- A view that has been precomputed and persisted
- A copy of the data defined by the view - data cache
What is a materialized view (MV)?

- A view that has been precomputed and persisted
- A copy of the data defined by the view - **data cache**
- Query definition not executed on each reference to the view
Why are MVs needed?

- Provide fast access to data, like caches
- Performance benefits in computation-intensive environments like data warehouses, where fast response time is required
- Index structures can be built on MVs
- Used for query optimization and integrity constraint checking

Sandy Ganza

Incremental Recomputations in Distributed Materialized Views
Why are MVs needed?

- Provide fast access to data, like caches
Why are MVs needed?

- Provide fast access to data, like caches
- Performance benefits in computation-intensive environments like data warehouses, where fast response time is required
Why are MVs needed?

- Provide fast access to data, like caches
- Performance benefits in computation-intensive environments like data warehouses, where fast response time is required
- Index structures can be built on MVs
Why are MVs needed?

- Provide fast access to data, like caches
- Performance benefits in computation-intensive environments like data warehouses, where fast response time is required
- Index structures can be built on MVs
- Used for query optimization and integrity constraint checking
The consistency problem in MVs
The consistency problem in MVs

- MV data may become obsolete when base data changes
The consistency problem in MVs

- MV data may become obsolete when base data changes
- Important to update the MV → **view maintenance**
View maintenance approaches

Approach 1:
Fully recompute the MV from scratch
Often costly and inefficient

Approach 2:
Only recompute changes (deltas) in the MV → incremental view maintenance
Often cheaper and more efficient
View maintenance approaches

- **Approach 1:**
  - Fully recompute the MV from scratch
  - Often costly and inefficient
View maintenance approaches

- **Approach 1:**
  - Fully recompute the MV from scratch
  - Often costly and inefficient

- **Approach 2:**
  - Only recompute changes (deltas) in the MV → incremental view maintenance
  - Often cheaper and more efficient
Traditional database systems

**Properties**
- MV and base relations controlled by the same database system
- Base relations understand view management
- Base relations have information regarding the view
Distributed database systems

Properties

- MV and the base relations are decoupled e.g. in data warehouses
- *Immediate view maintenance*, therefore, not possible

Materialized View

Base Tables

Sources
Challenges of maintaining distributed MVs

- Data sources are autonomous
- MVs span multiple sources
- Transactions contain updates from one or multiple sources
- Difficult to achieve consistency
Incremental recomputations - in a nutshell

**Incremental Join**

\[ V_{\text{old}} = R \ ▭ \Join S \downarrow \text{insert tuples} \]

\[ V_{\text{new}} = (R \cup \Delta R) \ ▭ \Join S \downarrow \text{join distributive w.r.t union} \]

\[ V_{\text{new}} = (R \ ▭ \Join S) \cup (\Delta R \ ▭ \Join S) \downarrow \text{if } \Delta R = \Delta R \ ▭ \Join S \]

\[ V_{\text{new}} = V_{\text{old}} \cup \Delta R \]

Sandy Ganza

Incremental Recomputations in Distributed Materialized Views
Incremental recomputations - in a nutshell

**Incremental Join**

\[ V_{old} = R \bowtie S \]
Incremental recomputations - in a nutshell

**Incremental Join**

\[ V_{old} = R \bowtie S \]

\[ \downarrow \text{insert tuples } \triangle_R \]
Incremental recomputations - in a nutshell

**Incremental Join**

\[ V_{old} = R \bowtie S \]

\[ \downarrow \text{insert tuples } \triangle R \]

\[ V_{new} = (R \cup \triangle R) \bowtie S \]
Incremental recomputations - in a nutshell

**Incremental Join**

\[ V_{old} = R \bowtie S \]

↓ insert tuples \( \triangle R \)

\[ V_{new} = (R \cup \triangle R) \bowtie S \]

↓ join distributive w.r.t union
Incremental recomputations - in a nutshell

**Incremental Join**

\[ V_{old} = R \bowtie S \]

\[ \downarrow \text{insert tuples } \triangle R \]

\[ V_{new} = (R \cup \triangle R) \bowtie S \]

\[ \downarrow \text{join distributive w.r.t union} \]

\[ V_{new} = (R \bowtie S) \cup (\triangle R \bowtie S) \]
Incremental recomputations - in a nutshell

Incremental Join

\[ V_{old} = R \bowtie S \]

↓ insert tuples \( \triangle_R \)

\[ V_{new} = (R \cup \triangle_R) \bowtie S \]

↓ join distributive w.r.t union

\[ V_{new} = (R \bowtie S) \cup (\triangle_R \bowtie S) \]

↓ if \( \triangle_R = \triangle_R \bowtie S \)
Incremental recomputations - in a nutshell

Incremental Join

\[ V_{old} = R \bowtie S \]

\[ \downarrow \text{insert tuples } \triangle R \]

\[ V_{new} = (R \cup \triangle R) \bowtie S \]

\[ \downarrow \text{join distributive w.r.t union} \]

\[ V_{new} = (R \bowtie S) \cup (\triangle R \bowtie S) \]

\[ \downarrow \text{if } \triangle R = \triangle R \bowtie S \]

\[ V_{new} = V_{old} \cup \triangle R \]
The View Maintenance Problem - Dimensions

A Mechanism for Efficient Materialized View Updates
Production Rules for Incremental View Maintenance
Self-Maintainability of Views

The View Maintenance Problem - Dimensions

- Information
- Modification
- Language
- Instance

Dimensions

- Amount of Information
- Expressiveness of View
- Definition Language
- Other Views
- Insertions
- Deletions
- Updates
- Sets of each
- Group Updates
- Change view definition
- Type of Modification
- Integrity Constraints
- Base Relations
- Materialized View
- Subqueries
- Aggregation
- Union
- Difference
- Outer-Joins
- Chronicle Algebra
- Recursion

Figure 1: The problem space
A Mechanism for Efficient Materialized View Updates

Two components of the mechanism

- Detect updates that do not affect the MV - irrelevant updates
- For relevant updates, use a differential algorithm to re-evaluate the MV
Example (Irrelevant update detection)

Consider relations $r$ and $s$ with $R = \{A,B\}$ and $S = \{C,D\}$. Let the view be defined as

$$v = \pi_{A,D}(\sigma(A>5) \land (C<10) \land (B=C)(r \times s))$$

Selection condition $= C(Y)$, where $Y$ is a set of attributes from the relations. $C(A,B,C) = (A > 5) \land (C < 10) \land (B = C)$. Given,

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th></th>
<th>C</th>
<th>D</th>
<th></th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>6</td>
<td>8</td>
<td>$s$</td>
<td>11</td>
<td>30</td>
<td>$v$</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>20</td>
<td></td>
<td>8</td>
<td>20</td>
<td></td>
<td>2</td>
<td>30</td>
</tr>
</tbody>
</table>

- inserting tuple $(7,8)$ into $r$ is **relevant**
- inserting tuple $(1,5)$ into $r$ is **irrelevant**
**Differential re-evaluation algorithm**

- Identifies tuples to be inserted/deleted from current view instance

**Assumption**

The net effect of updates from all committed transactions are captured
Example (Select views)

A select view is defined by $V = \sigma_C(Y)(R)$, where: $C = \text{selection condition}$, $Y \subseteq R$.

If $\triangle_r$ and $\nabla_r$ are inserted and deleted tuples respectively, the new view state $v'$ is given by: $v' = v \cup \sigma_C(Y)(\triangle_r) - \sigma_C(Y)(\nabla_r)$. This corresponds to the sequence of operations:

\[
\text{insert}(V, \sigma_C(Y)(\triangle_r)) \\
\text{delete}(V, \sigma_C(Y)(\nabla_r))
\]

- Cheaper to update the MV by this sequence of operations, when $|v| \gg |d_r|$
**Example (Project views)**

A project view is defined by $V = \pi_X(R)$, where $X \subseteq R$. Given a relation $R = \{A, B\}$ and a view definition $\pi_A(R)$, with

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>r:</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

$V: |

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

- $delete(R,\{(4, 5)\})$ on $r$ results into $delete(V,\{4\})$
- $delete(R,\{(1, 2)\})$ on relation $r$ though, leads to an inconsistent view
- **Solutions:** multiplicity counter, projection of keys in the view
A join view is defined by $V = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_P$.

**Example (Join views - insert operations)**

R and S are two relation schemes with $R = \{A, B\}$ and $S = \{B, C\}$. If a view $V = R \bowtie S$ is defined and a view $v$ is materialized. Assume relation $r$ is modified by inserting tuples $\triangle r$. Modified relation $r' = r \cup \triangle r$ and new state of MV $v'$ is:

$$v' = r' \bowtie s = (r \cup \triangle r) \bowtie s = (r \bowtie s) \cup (\triangle r \bowtie s)$$

If $\triangle r = \triangle r \bowtie s$, then $v' = v \cup \triangle r$.

- MV is modified by inserting deltas into relation $v$
- Cheaper than recomputing the whole join from scratch
Example (Join views - delete operations)

Let the view definition be $V = R \bowtie S$ and $r^i = r - \nabla r$. The new state $v^i$ is given by:

$$v^i = r^i \bowtie s = (r - \nabla r) \bowtie s = (r \bowtie s) - (\nabla r \bowtie s)$$

If $\nabla r = \nabla r \bowtie s$, then $v^i = v - \nabla r$.

- MV is updated by deleting deltas $\nabla r$ from $v$
- When $|v| \gg |\nabla r|$, cheaper than recomputing MV from scratch
Example (Select-Project-Join (SPJ) views)

If \( R = \{A, B\} \) and \( S = \{B, C\} \), and view \( V = \pi_A(\sigma_C(Y)(R \bowtie S)) \).

Let \( r^i = r \cup \triangle_r \). New MV is:

\[
\begin{align*}
\nu^i &= \pi_A(\sigma_C(Y)(r^i \bowtie s)) = \pi_A(\sigma_C(Y)((r \cup \triangle_r) \bowtie s)) = \\
&= \pi_A(\sigma_C(Y)(r \bowtie s)) \cup \pi_A(\sigma_C(Y)(\triangle_r \bowtie s)) = \nu \cup \pi_A(\sigma_C(Y)(\triangle_r \bowtie s))
\end{align*}
\]

If \( \triangle_r = \pi_A(\sigma_C(Y)(\triangle_r \bowtie s)) \), then \( \nu^i = \nu \cup \triangle_r \).

- MV is updated by inserting deltas \( \triangle_r \) into relation \( \nu \)
Production rules for incremental view maintenance

- Used to automatically maintain derived data e.g. views
- User: Initially enters view definition as SQL `select` expression
- Information about keys for the view’s base tables also needed
- System: Automatically derives production rules to maintain the MV
- Rules produced for `insert`, `delete`, and `update` operations
Deriving Production Rules for Incremental View Maintenance

Production rules in database systems allow specifying rules for maintaining a given virtual view. Howver, for some operations substantial recomputation may be required. Writing a correct set of rules for efficiently maintaining a given view can be a difficult problem. Efficiency is achieved by expressing the sets of changes made to the base tables, which are accessible through logical tables provided by the rule language.

The results may indicate that, in order to improve the efficiency of view maintenance, further interaction with the system is necessary prior to rule generation. In particular, the system is necessary when a view is created. Initially, the system is used when a view is created. The user is provided with the results of this analysis.

Production rules generated from a view definition can be used to maintain materialized views. Howevor, for some operations substantial recomputation may be required. Writing a correct set of rules for efficiently maintaining a given view can be a difficult problem. Efficiency is achieved by expressing the sets of changes made to the base tables, which are accessible through logical tables provided by the rule language.

The results may indicate that, in order to improve the efficiency of view maintenance, further interaction with the system is necessary prior to rule generation. In particular, the system is necessary when a view is created. Initially, the system is used when a view is created. The user is provided with the results of this analysis.

Final View with Analysis Information

View-Maintaining Rules

Figure: Rule derivation system
Production rule language

- **Set-oriented**, SQL-base production rule language
- “Usual” database functionality available
- Rules based on notion of transitions
Production rule language

- **Set-oriented, SQL-base production rule language**
- “Usual” database functionality available
- Rules based on notion of *transitions*

**Definition**

A **Transition** is a database state change resulting from a sequence of data manipulation operations.
Production rule language

- Set-oriented, SQL-base production rule language
- "Usual" database functionality available
- Rules based on notion of transitions

Definition

A **Transition** is a database state change resulting from a sequence of data manipulation operations

Syntax

```plaintext
create rule name
when transition predicate
then action
[precedes rule-list]
```
Production rule language

- Set-oriented, SQL-base production rule language
- “Usual” database functionality available
- Rules based on notion of transitions

Syntax

create rule name
when transition predicate
then action
[precedes rule-list]

Definition

A Transition is a database state change resulting from a sequence of data manipulation operations

- Transition predicate specifies operations on tables: inserted into T, deleted from T, or updated T
- Rule triggered when at least one of the operations occurs in transaction
A transition table is a logical table that reflects changes that have occurred during a transition.
Transition tables

Definition

A **transition table** is a logical table that reflects changes that have occurred during a transition.

- **Transition table “inserted T”:** current tuples of table T inserted by the transition
- **“deleted T”:** pre-transition tuples of T deleted by the transition
- **“old updated T”:** pre-transition tuples of T updated by the transition
- **“new updated T”:** current tuples of T updated by the transition
View analysis process

For each list of table references in the view definition, the system:

- Computes “bound columns” of the table references
- Determines “safety” of each table reference

View definition

```
define view V(Col-List):
select C1,...,Cn from T1,...,Tm where P
where T1,...,Tm are top-level table references,
C1,...,Cn are columns of T1,...,Tm, and P is a predicate
```
View analysis process

For each list of table references in the view definition, the system:

- Computes “bound columns” of the table references
- Determines “safety” of each table reference

This method doesn’t support maintenance of views with duplicates

View definition

```
define view V(Col-List):
    select C_1,...,C_n from T_1,...,T_m where P

where T_1,...,T_m are top-level table references, 
C_1,...,C_n are columns of T_1,...,T_m, and P is a predicate```

Sandy Ganza
Incremental Recomputations in Distributed Materialized Views
Bound columns & Duplicate analysis

- Bound columns used to determine whether the view may contain duplicates

**Property (Bound columns lemma for top-level tables)**

If two tuples in the cross-product of top-level tables $T_1, \ldots, T_m$ satisfy predicate $P$ and differ in their **bound columns**, then the tuples also must differ in view columns $C_1, \ldots, C_n$.
Bound columns & Duplicate analysis

- Bound columns used to determine whether the view may contain duplicates

Property (Bound columns lemma for top-level tables)

If two tuples in the cross-product of top-level tables $T_1, \ldots, T_m$ satisfy predicate $P$ and differ in their bound columns, then the tuples also must differ in view columns $C_1, \ldots, C_n$.

- Duplicate analysis is only done when the view’s definition doesn’t contain distinct

Theorem

If the set of bound columns includes a key for every top-level table, then $V$ will not contain duplicates.
Safety analysis

- To generate incremental view maintenance rules for operations on a table, the reference to that table has to be safe.
- Safety of top-level table references is similar to duplicate analysis.

**Theorem**

*If table reference \( T_i \) is safe, then insert, delete, and update operations on \( T_i \) can be reflected by incremental changes to \( V \).*
Rule generation

- Last phase of the rule derivation process
- First consider safe table references, then unsafe references
- For each table reference generate 4 rules: one triggered by **inserted**, one by **deleted**, and two by **updated**
Rule for inserted

create rule ins-$T_i$ $\rightarrow$ $V$
when inserted into $T_i$
then insert into $V$
(select $C_1,...,C_n$
from old $T_1$,$...,inserted T_i$,$...,T_m$
where P and $<C_1,...,C_n>$ not in inserted $V$)

- Use **inserted $T_i$** instead of $T_i$ to propagate insertions
- Insertion theorem says insertions cannot create duplicates in the view, however
- Check whether a tuple may not have been already inserted by a different rule, to avoid duplicates. Use transition table **inserted $V$** for this
Rule for deleted

create rule del- $T_i \rightarrow V$
when deleted from $T_i$
then delete from $V$
where $< C_1, \ldots, C_n >$ in
(select $C_1, \ldots, C_n$
from old $T_i, \ldots, deleted T_i, old \ T_m$
where $P-old$)

- Deletion theorem says deleted tuples should no longer be in the view, however
- Check if other tables haven’t been modified. Consider pre-transition values of all other tables by use of $P-old$.
Rule for updated

- Update operations on base tables cause delete and/or insert operations on views
- Two rules are triggered by updated:
  - One to perform deletions
  - The second to perform insertions
- The two rules are similar to rules for deleted and inserted
Definition

A **self-maintainable** view is a view that can be maintained using only the content of the view and the database modifications (deltas), without using underlying tables.

- **Self-maintainability** is defined with respect to one of the three modification types (**insertions, deletions, or updates**).
Self-maintainability with respect to insertions difficult to achieve

Observations
- Impossible to self-maintain a Select-Project-Join (SPJ) view w.r.t insertions, because inserted tuples could originate from other base tables.
- All SP views (don’t involve joins) are self-maintainable w.r.t insertions.
Self-maintainability with respect to deletions

An SPJ view that joins base relations $R_1, R_2, \ldots, R_n$ is said to be self-maintainable w.r.t deletions in $R_1$, if the following sufficient condition holds:

**Condition**

For some key candidate of $R_1$, each key attribute is either retained in the view, or each key attribute is equated to a constant in the view definition.
Self-maintainability with respect to updates

- To achieve self-maintainability, updates are modeled directly rather than as deletions followed by insertions.
- Valuable information from deleted tuples helps to insert tuples.
- Self-maintainability depends on the attributes being updated.
- An SPJ joining two or more distinct relations $R_1, R_2, \ldots, R_n$ is said to be self-maintainable w.r.t updates to $R_1$ if and only if:

**Condition**

The updated attributes are **unexposed** and not **distinguished**, or the updated attributes are unexposed and the view is self-maintainable w.r.t updates.
Table of Contents

1 Introduction
   • Terminology

2 Incremental Recomputations in Materialized Views
   • The View Maintenance Problem - Dimensions
   • A Mechanism for Efficient Materialized View Updates
   • Production Rules for Incremental View Maintenance
   • Self-Maintainability of Views

3 View Maintenance Policies
   • Policies

4 Incremental Recomputations in Distributed Materialized Views
   • Consistency in Incremental View Maintenance
   • Eager Compensating Algorithms (ECA)
   • The Strobe Algorithms

5 Conclusion
View maintenance policies

*When* and *how* are views maintained?
View maintenance policies

When and how are views maintained?

- **Immediate views**: refresh view after every update transaction
  - Allows fast querying
  - Update transaction overhead & not applicable in distributed environments
View maintenance policies

When and how are views maintained?

- **Immediate views**: refresh view after every update transaction
  - ✅ Allows fast querying
  - 😞 Update transaction overhead & not applicable in distributed environments

- **Deferred views**: refresh view when queried (on-demand)
  - ✅ Fast update transactions & batched updates possible
  - 😞 Slow querying & view may become inconsistent with it’s definition
View maintenance policies

When and how are views maintained?

- **Immediate views**: refresh view after every update transaction
  - 🎁 Allows fast querying
  - 😞 Update transaction overhead & not applicable in distributed environments

- **Deferred views**: refresh view when queried (on-demand)
  - 🎁 Fast update transactions & batched updates possible
  - 😞 Slow querying & view may become inconsistent with it’s definition

- **Snapshot views**: refresh view periodically (e.g. weekly)
  - 🎁 Fast querying & fast updates
  - 😞 Queries may read obsolete data
Table of Contents

1 Introduction
   • Terminology

2 Incremental Recomputations in Materialized Views
   • The View Maintenance Problem - Dimensions
   • A Mechanism for Efficient Materialized View Updates
   • Production Rules for Incremental View Maintenance
   • Self-Maintainability of Views

3 View Maintenance Policies
   • Policies

4 Incremental Recomputations in Distributed Materialized Views
   • Consistency in Incremental View Maintenance
   • Eager Compensating Algorithms (ECA)
   • The Strobe Algorithms

5 Conclusion
Overview

- In distributed environments the MV is decoupled from sources
- Incremental maintenance in response to updates can’t be triggered by update transactions
- Maintenance anomalies possible
- Which levels of consistency exist?
- Two classes of incremental view maintenance algorithms in distributed environments
Consistency in Incremental View Maintenance

Eager Compensating Algorithms (ECA)
The Strobe Algorithms

Consistency levels

The consistency is defined between the warehouse (source data) and the materialized view

- **Convergence**: For finite executions, the view is consistent with the source data after the last update and all activity is complete

- **Weak consistency**: Convergence holds & for every state of the view, there is a valid source state in a corresponding order

- **Strong consistency**: Weak consistency holds. Furthermore, for every state of a view, there exists a valid source start

- **Completeness**: Strong consistency holds & between the states of the view and those of the sources, there is complete order-preserving mapping
Update processing in a single source model

- When source is updated, it sends an update message to the warehouse (view)
- Warehouse (WH) queries source for additional data necessary to make changes
- Source evaluates queries and sends answers to WH
- During the evaluation, dirty reads may occur
View maintenance anomaly over a single source

Example (View maintenance anomaly)

Assume two relations $r_1$ and $r_2$ at the source with $r_2$ initially empty:

\[
\begin{align*}
  r_1 &: \frac{A}{3} \frac{B}{4} \\
  r_2 &: \frac{B}{-} \frac{C}{-}
\end{align*}
\]

Let view definition be $V = \Pi_A(r_1 \bowtie r_2)$. Two consecutive updates happen at the source:

$U_1 = \text{insert}(r_2, [4, 8])$ and $U_2 = \text{insert}(r_1, [5, 4])$. The materialized view (MV) is initially empty $MV = \emptyset$. 
View maintenance anomaly over a single source

- **Source**
  - r1: A 3 B 4
  - r2: B C - -

- **Warehouse**
  - A

The diagram illustrates the concept of view maintenance anomaly in a single source scenario.
View maintenance anomaly over a single source

```
Source
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Warehouse

| A |
```

\[\text{insert}(r2, [4,8])\]
View maintenance anomaly over a single source
View maintenance anomaly over a single source
View maintenance anomaly over a single source

Update_1

Q1=\Pi_A(r1 \bowtie [4,8])
View maintenance anomaly over a single source

$\text{Source}$

$r1$

$A$  $B$
$3$  $4$

$r2$

$B$  $C$
$4$  $8$

$\text{Warehouse}$

$Q1=\Pi_A(r1 \bowtie [4,8])$

$\text{Update}_1$

$insert(r1, [5,4])$

Sandy Ganza
View maintenance anomaly over a single source

Source

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>r2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Warehouse

Q1 = \Pi_A(r1 \Join [4,8])

Update_1

Update_2
View maintenance anomaly over a single source
View maintenance anomaly over a single source

### Source
- r1
  - A: 3, 4, 5
  - B: 4, 8
- r2
  - B: 4, 8
  - C: 4, 8

### Warehouse
- A: 3
- 5

\[ Q_1 = \Pi_A (r_1 \bowtie [4,8]) \]
\[ Q_2 = \Pi_A ([5,4] \bowtie r_2) \]
\[ A_1 = ([3],[5]) \]
View maintenance anomaly over a single source

Source
- r1
  - A: 3, 4
  - B: 5, 4
- r2
  - B: 4, 8

Warehouse
- A: 3, 5
- B: 4, 8
- C: 5, 5

Update_1
- Q1 = \(\Pi_A (r1 \bowtie [4,8])\)
- Update_2
- Q2 = \(\Pi_A ([5,4] \bowtie r2)\)
- A1 = ([3], [5])
- A2 = ([5])

Sandy Ganza
Incremental Recomputations in Distributed Materialized Views
Compensating queries as a solution

Definition

A **compensating query** is added to queries sent to source to offset the effect of concurrent queries.

Solution to the view maintenance anomaly

WH receives $U_2$ before $A_1$ and infers that $Q_1$ will be evaluated on incorrect data, since messages are supposed to be delivered in order. WH therefore sends **compensation query** $Q_2$ to undo the effect of $U_2$ on $A_1$.

$$Q_2 = \Pi_A([5, 4] \bowtie r_2) - \Pi_A([5, 4] \bowtie [4, 8])$$
Query compensation

**Source**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>r2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

**Warehouse**

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
</table>

Update 1

\[ Q_1 = \Pi_A (r_1 \bowtie [4,8]) \]

Update 2

\[ Q_2 = \Pi_A ([5,4] \bowtie r_2 \bowtie ([5,4] \bowtie [4,8])) \]
Query compensation

Source

Update_1
Q1=Π_A(r1⋈[4,8])

Update_2
Q2=Π_A([5,4]⋈r2)⋈-[5,4]⋈[4,8])

A1 = ([3],[5])

Warehouse

r1
A B
3 4
5 4

r2
B C
4 8

Update_1

Update_2

r1
A
3
4
5

r2
B
4
8

Source

Warehouse
Query compensation

Source

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Warehouse

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Update_1

Q1=\Pi_A(r1 \bowtie [4,8])

Update_2

Q2=\Pi_A([5,4] \bowtie r2 \bowtie [5,4] \bowtie [4,8])

A1 = ([3],[5])

A2 = ∅
The Strobe Algorithms

- Maintenance of consistency in multi-source environments
- Updates arriving at the warehouse may need to be integrated with data from other sources before being stored
- Important to know if and how sources run transactions
- Three transaction scenarios possible: *single update*, *source-local*, or *global transactions*
- Corresponding Strobe algorithms for the transaction scenarios are:
  - Strobe algorithm
  - Transaction-Strobe algorithm
  - Global-Strobe algorithm
The Strobe algorithm

- Updates are not performed directly on the view. They are processed but kept in an actions list \( AL \).
- Actions in \( AL \) only applied to MV when consistent state can be guaranteed.
- \( AL \) consists of insert and delete actions.
- A set called \( \text{pending}(Q) \) stores updates that occur during query processing.
- Delete actions are added to \( AL \) straight away.
- Insert action is added after compensation of query Q has terminated.
Example (Strobe)

Let UQS be the unanswered query set. Operation $key_{-}delete(R, U_i)$ deletes tuples from relation $R$ whose key attributes are the same as $U_i$. $V(U)$ is the view expression $V$ with tuple $U$ substituted for $U$’s relation. If we have relations $r_1$, $r_2$ and $r_3$ residing on sources $x$, $y$ and $z$ respectively, let view $V$ be defined as $V = r_1 \bowtie r_2 \bowtie r_3$. Given that:

$$
\begin{align*}
& r_1: \quad 1 & A & 2 & B \\
& r_2: \quad - & B & C \\
& r_3: \quad 3 & C & 4 & D
\end{align*}
$$

Initially the materialized view is $MV = \emptyset$. Given two updates: $U_1 = insert(r_2,[2,3])$ and $U_2 = delete(r_1,[1,2])$. 
**Strobe algorithm**

![Diagram showing the Strobe algorithm process]

**Warehouse**

Source x

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Source y

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source z

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**AL = []**

Insert r2 with [2,3]

**Consistency in Incremental View Maintenance**

**Eager Compensating Algorithms (ECA)**

**The Strobe Algorithms**
Strobe algorithm

 Warehouse
Source x
A B
1 2
Source y
B C
2 3
Source z
C D
3 4
Update_1
AL = []
pending(Q1) = Ø
Q1 = r1 ⋈ [2,3] ⋈ r3
Strobe algorithm

Q1,1 = r1 ⋈ [2,3]

AL = []
Strobe algorithm

Warehouse
Source x
Source y
Source z
A B
1 2
B C
2 3
C D
3 4
A B C D
AL = 
Update_1
Q1 = r1  [2,3]⋈   r3⋈
pending(Q1) = Ø
Q1,1=r1  [2,3]⋈
A1,1=[1,2,3]

AL = []
pending(Q1) = Ø
Q1 = r1  [2,3] ⋈ r3
Strobe algorithm

Warehouse

Source x

Source y

Source z

Update_1

Q1,1 = r1 \bowtie [2,3]

A,1 = [1,2,3]

pending(Q1) = Ø

Q1,2 = [1,2,3] \bowtie r3

A = [ ]

Warehouse

Source x

Source y

Source z

AL = []

pending(Q1) = Ø

Q1 = r1 \bowtie [2,3] \bowtie r3
Strobe algorithm

Warehouse
Source x
Source y
Source z

A B
1 2
B C
2 3
C D
3 4
A B C D

AL = [
Update_1
Q1,1=r1⋈[2,3]
A1,1=[1,2,3]

Q1= r1 ⋈ [2,3] ⋈ r3
pending(Q1) = Ø
Q1,2=[1,2,3] ⋈ r3
update(r1, [1,2])
Strobe algorithm

Warehouse

Source x

Source y

Source z

A B
- -
B C
2 3
C D
3 4
A B C D

AL = []

pending(Q1) = {Update_2}

Q1 = r1 \bowtie [2,3] \bowtie r3

Q1,1 = r1 \bowtie [2,3]

A1,1 = [1,2,3]

Update_2

Update_1

Q1,2 = [1,2,3] \bowtie r3

AL = []
Consistency in Incremental View Maintenance
Eager Compensating Algorithms (ECA)
The Strobe Algorithms

**Strobe algorithm**

```
Warehouse
Source x
Source y
Source z
A B
- -
B C
2 3
C D
3 4
A B C D
AL = [key_delete(MV,Update_2)]
Update_1
Q1 = r1 ⊙ [2,3] ⊙ r3 ⊙ Update_2
pending(Q1) = {Update_2}
Q1,1=r1 [2,3] ⊙ A1,1=[1,2,3]
Q1,2=[1,2,3] r3 ⊙ Update_2
```

Sandy Ganza
Incremental Recomputations in Distributed Materialized Views
Strobe algorithm

Pending(Q1) = {Update_2}
Q1 = r1 ⋈ [2,3] ⋈ r3
AL = [key_delete(MV,Update_2) ]
**Strobe algorithm**

- **Update 1**
  - Source x
    - Q1,1 = r1 \( \bowtie \) [2,3]
  - A1,1 = [1,2,3]

- **Update 2**
  - Source y
    - Q1,2 = [1,2,3] \( \bowtie \) r3
  - A1,2 = ?
    - key_delete(A1,2, Update_2)
  - Warehouse
    - AL = [key_delete(MV,Update_2)]

- pending(Q1) = Ø

- Q1 = r1 \( \bowtie \) [2,3] \( \bowtie \) r3

- AL = [key_delete(MV,Update_2)]

- key_delete(A1,2, Update_2)
**Strobe algorithm**

Warehouse
Source x
Source y
Source z
A B
- -
B C
2 3
C D
3 4
A B C D
- - - -

\[ AL = \text{key_delete(MV,Update}_2) \]

\[ \text{pending}(Q1) = \emptyset \]

\[ Q1 = r1 \bowtie [2,3] \bowtie r3 \]

\[ AL = [\text{key_delete(MV,Update}_2)] \]

\[ \text{key_delete}(A1,2 , \text{Update}_2) \]
Strobe algorithm

Warehouse

Source x

\[
\begin{array}{c|c|c}
  A & B & - \\
\end{array}
\]

Source y

\[
\begin{array}{c|c|c}
  B & C & 2 \\
\end{array}
\]

Source z

\[
\begin{array}{c|c|c}
  C & D & 3 \\
\end{array}
\]

\[Q_1,1 = r_1 \bowtie [2,3]\]
\[A_{1,1} = [1,2,3]\]

Update 2

\[Q_1,2 = [1,2,3] \bowtie r_3\]

Update 1

pending(Q1) = Ø

\[Q_1 = r_1 \bowtie [2,3] \bowtie r_3\]

A_2 = Ø

\[A_2 = [ ]\]
Table of Contents

1 Introduction
   • Terminology

2 Incremental Recomputations in Materialized Views
   • The View Maintenance Problem - Dimensions
   • A Mechanism for Efficient Materialized View Updates
   • Production Rules for Incremental View Maintenance
   • Self-Maintainability of Views

3 View Maintenance Policies
   • Policies

4 Incremental Recomputations in Distributed Materialized Views
   • Consistency in Incremental View Maintenance
   • Eager Compensating Algorithms (ECA)
   • The Strobe Algorithms

5 Conclusion
Materialized views: fast data access & fast querying
Incremental view maintenance often cheap & efficient
Concurrent updates in distributed environments cause maintenance anomalies
Compensation mechanisms used to overcome view maintenance anomalies in distributed systems
Thank You!