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

Table of Contents

What is a view?

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

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)?

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?

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 $\rightarrow$ **view maintenance**

View maintenance approaches

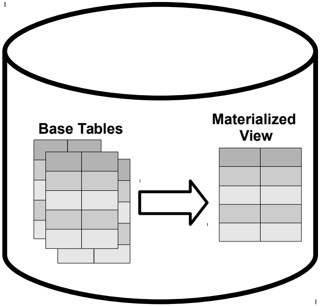
View maintenance approaches

- **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 $\rightarrow$ **incremental view maintenance**
  - Often cheaper and more efficient

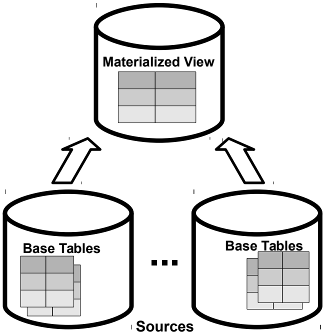
Traditional database systems



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

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

Table of Contents

Incremental recomputations - in a nutshell

Incremental Join

Incremental recomputations - in a nutshell

Incremental Join

$V_{old} = R \bowtie S$

Incremental recomputations - in a nutshell

Incremental Join

$V_{old} = R \bowtie S$

$\downarrow$ insert tuples $\triangle_R$

Incremental recomputations - in a nutshell

Incremental Join

$V_{old} = R \bowtie S$

$\downarrow$ insert tuples $\triangle_R$

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

Incremental recomputations - in a nutshell

Incremental Join

$V_{old} = R \bowtie S$

$\downarrow$ insert tuples $\triangle_R$

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

$\downarrow$ join distributive w.r.t union

Incremental recomputations - in a nutshell

Incremental Join

$V_{old} = R \bowtie S$

$\downarrow$ insert tuples $\triangle_R$

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

$\downarrow$ join distributive w.r.t union

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

Incremental recomputations - in a nutshell

Incremental Join

$$V_{old} = R \bowtie S$$

$\downarrow$ insert tuples $\triangle_R$

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

$\downarrow$ join distributive w.r.t union

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

$\downarrow$ if $\triangle_R = \triangle_R \bowtie S$

Incremental recomputations - in a nutshell

Incremental Join

$V_{old} = R \bowtie S$

$\downarrow$ insert tuples $\triangle_R$

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

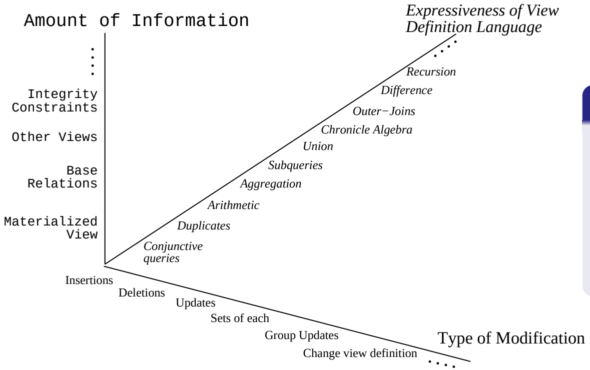
$\downarrow$ join distributive w.r.t union

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

$\downarrow$ if $\triangle_R = \triangle_R \bowtie S$

$V_{new} = V_{old} \cup \triangle_R$

The View Maintenance Problem - Dimensions



Dimensions

- Information
- Modification
- Language
- Instance

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

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

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

	A	B		C	D		A	D
r:	6	8	*s:*	11	30	*v:*	6	20
	2	20		8	20		2	30

- 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 = selection condition, Y $\subseteq$ R.
If $\triangle_r$ and $\bigtriangledown_r$ are inserted and deleted tuples respectively, the new view state v^i is given by: $v^i = v \cup \sigma_{C(y)}(\triangle_r)$ - $\sigma_{C(Y)}(\bigtriangledown_r)$. This corresponds to the sequence of operations:

$$insert(V, \sigma_{C(Y)}(\triangle_r))$$

$$delete(V, \sigma_{C(Y)}(\bigtriangledown_r))$$

- 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

r:

A	B
1	2
1	3
4	5

v:

A
1
4
2

- *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 $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_P$.

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 $r^i = r \cup \triangle_r$ and new state of MV v^i is:

$$v^i = r^i \bowtie s = (r \cup \triangle_r) \bowtie s = (r \bowtie s) \cup (\triangle_r \bowtie s)$$

If $\triangle_r = \triangle_r \bowtie s$, then $v^i = v \cup \triangle_r$.

- 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 $r^i = r - \bigtriangledown_r$. The new state v^i is given by:

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

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

- MV is updated by deleting deltas $\bigtriangledown_r$ from v
- When $|v| \gg |\bigtriangledown_r|$, 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:

$$v^i = \pi_A(\sigma_{C(Y)}(r^i \bowtie s)) = \pi_A(\sigma_{C(Y)}((r \cup \triangle_r) \bowtie s)) = \pi_A(\sigma_{C(Y)}(r \bowtie s)) \cup \pi_A(\sigma_{C(Y)}(\triangle_r \bowtie s)) = v \cup \pi_A(\sigma_{C(Y)}(\triangle_r \bowtie s))$$

If $\triangle_r = \pi_A(\sigma_{C(Y)}(\triangle_r \bowtie s))$, then $v^i = v \cup \triangle_r$.

- MV is updated by inserting deltas $\triangle_r$ into relation v

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

System structure

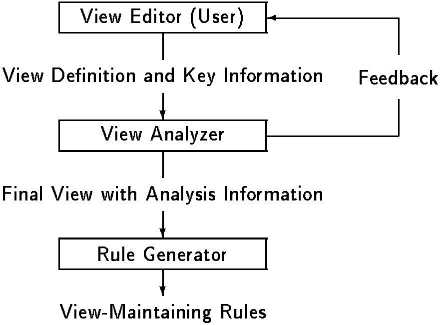


Figure : Rule derivation system

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

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*

Definition

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

Syntax

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

- 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

Transition tables

Definition

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 $C_1, \ldots, C_n$ *from* $T_1, \ldots, T_m$ *where* P
where $T_1, \ldots, T_m$ are top-level table references,
$C_1, \ldots, C_n$ are columns of $T_1, \ldots, T_m$, 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

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

Rule

create rule ins-$T_i - V$
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

Rule

create rule del-$T_i - V$
when deleted from T_i
then delete from V
where $< C_1, ..., C_n >$ in
(select $C_1, ..., C_n$
from old $T_i, .., deletedT_i$,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**

Self-maintainability of views

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

- 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 $R_1, R_2, ..., R_n$ 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

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

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

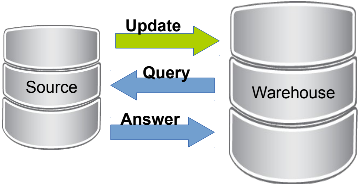
Consistency levels

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:

r_1:

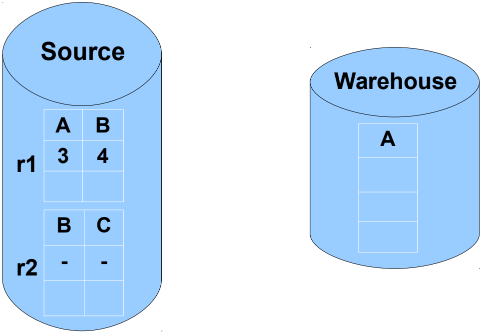
A	B
3	4

and r_2:

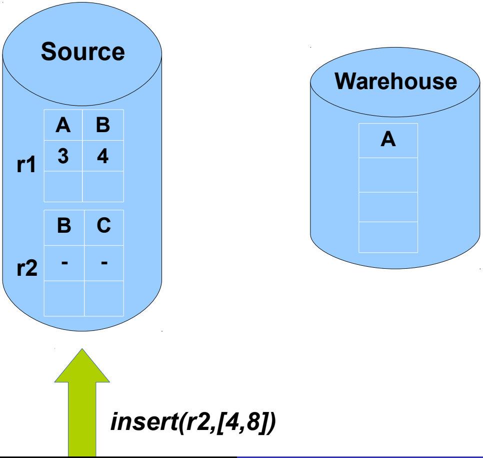
B	C
-	-

Let view definition be $V = \Pi_A(r_1 \bowtie r_2)$. Two consecutive updates happen at the source:
$U_1 = insert(r_2,[4,8])$ and $U_2 = insert(r_1,[5,4])$. The materialized view (MV) is initially empty $MV = \emptyset$.

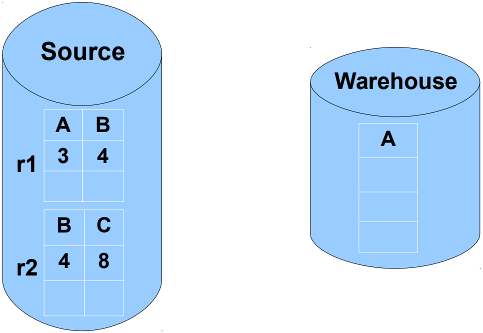
View maintenance anomaly over a single source



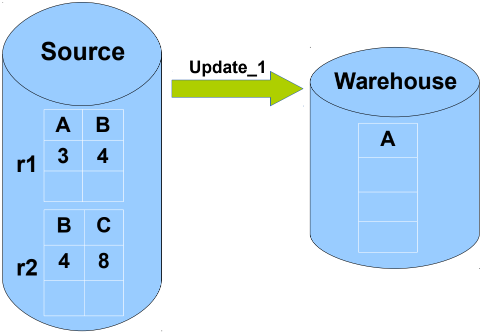
View maintenance anomaly over a single source



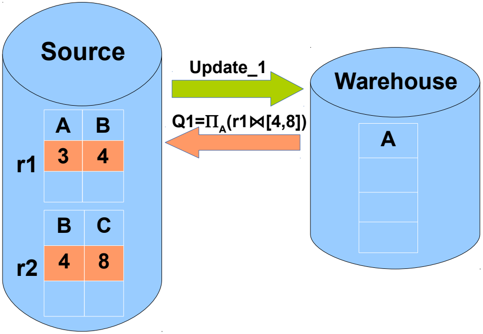
View maintenance anomaly over a single source



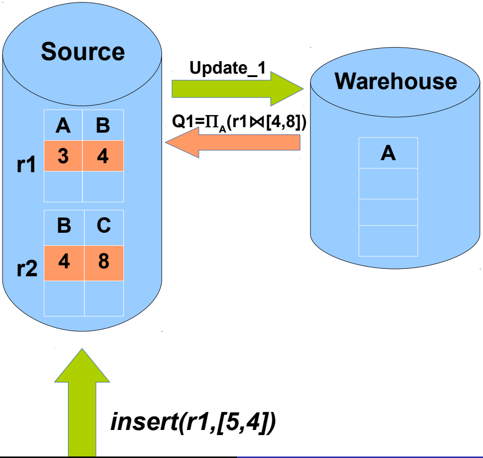
View maintenance anomaly over a single source



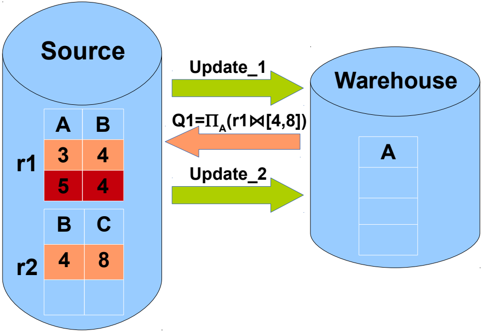
View maintenance anomaly over a single source



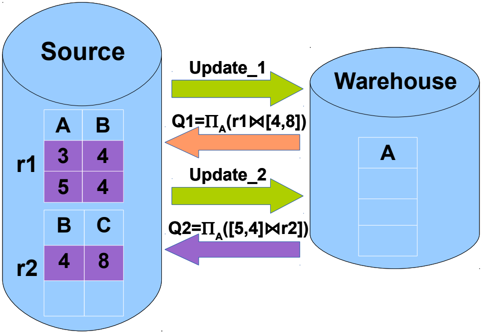
View maintenance anomaly over a single source



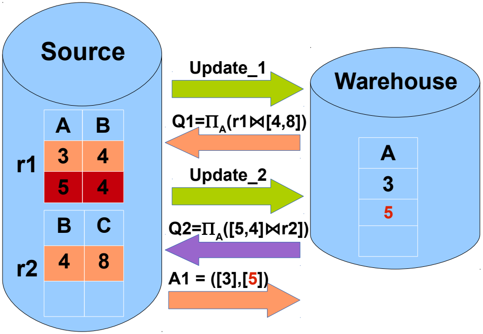
View maintenance anomaly over a single source



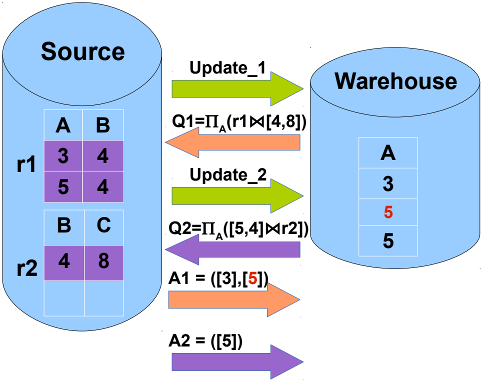
View maintenance anomaly over a single source



View maintenance anomaly over a single source



View maintenance anomaly over a single source



Compensating queries as a solution

Definition

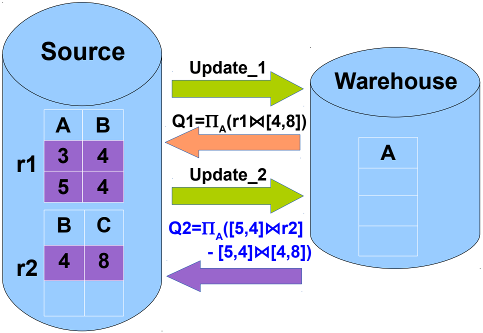
A **compensating query** is added to queries sent to source to offset the effect of concurrent queries

Solution to the view maintenance anomaly

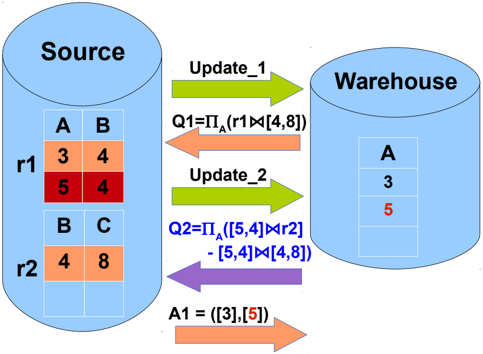
WH receives U_2 before A_1 and infers that Q_1 will be evaluated on incorrect data, since messages are supposed to be delivered in order. WH therefore sends **compensation query** $\mathbf{Q}_2$ to undo the effect of U_2 on A_1

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

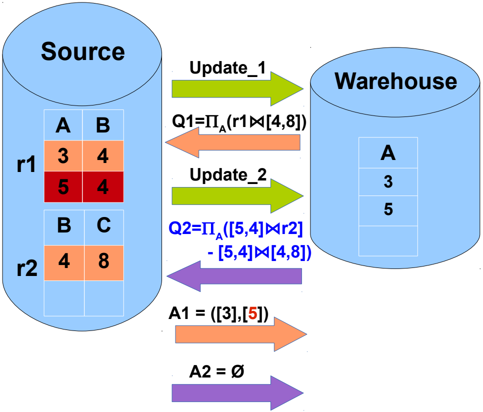
Query compensation



Query compensation



Query compensation



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 **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

Strobe algorithm

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:

r_1:

A	B
1	2

r_2:

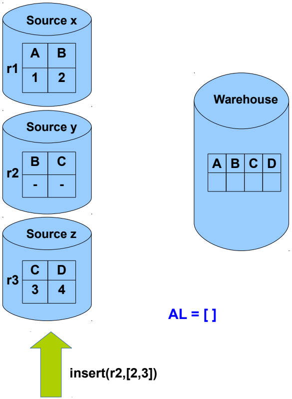
B	C
-	-

and r_3:

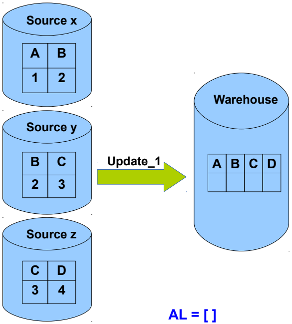
C	D
3	4

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



Strobe algorithm

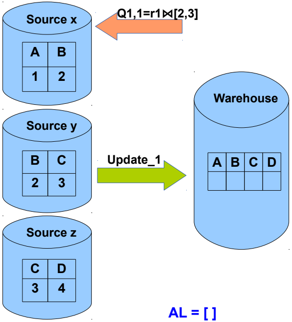


AL = [ ]

pending(Q1) = Ø

Q1 = r1 ⋈ [2,3] ⋈ r3

Strobe algorithm

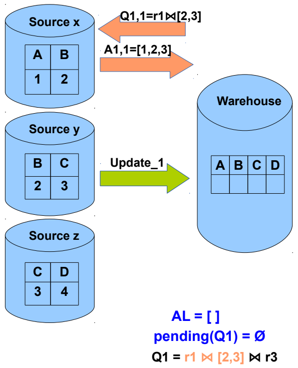


AL = [ ]

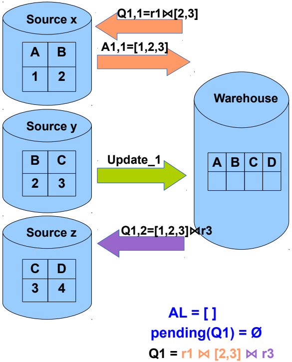
pending(Q1) = Ø

Q1 = r1 ⋈ [2,3] ⋈ r3

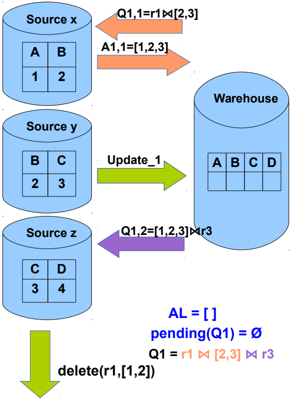
Strobe algorithm



Strobe algorithm



Strobe algorithm

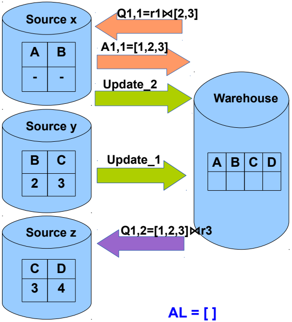


AL = [ ]

pending(Q1) = Ø

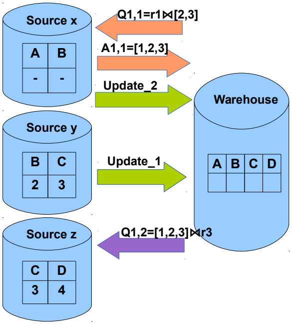
Q1 = r1 ⋈ [2,3] ⋈ r3

Strobe algorithm



AL = [ ]
pending(Q1) = {Update_2}
Q1 = r1 ⋈ [2,3] ⋈ r3

Strobe algorithm

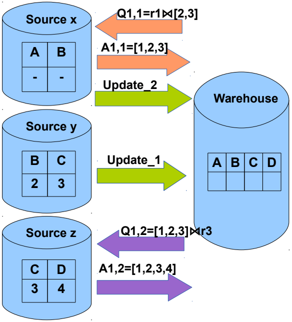


pending(Q1) = {Update_2}

Q1 = r1 ⋈ [2,3] ⋈ r3

AL = [key_delete(MV,Update_2)]

Strobe algorithm

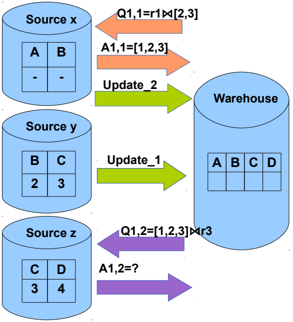


pending(Q1) = {Update_2}

Q1 = r1 ⋈ [2,3] ⋈ r3

AL = [key_delete(MV,Update_2)]

Strobe algorithm



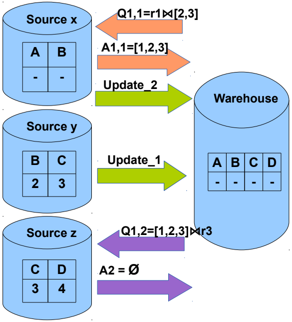
pending(Q1) = Ø

Q1 = r1 ⋈ [2,3] ⋈ r3

AL = [key_delete(MV,Update_2)]

key_delete(A1,2 , Update_2)

Strobe algorithm



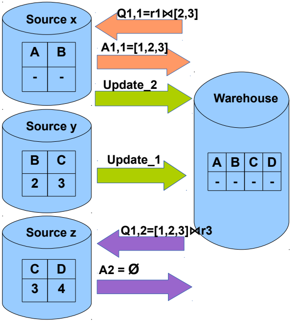
pending(Q1) = Ø

Q1 = r1 ⋈ [2,3] ⋈ r3

AL = [key_delete(MV,Update_2)]

key_delete(A1,2 , Update_2)

Strobe algorithm



pending(Q1) = Ø

Q1 = r1 ⋈ [2,3] ⋈ r3

AL = [ ]

Table of Contents

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!