Practical Incremental View Maintenance

CPS 296.1

Topics in Database Systems

Roadmap

- Zhuge et al. “View Maintenance in a Warehousing Environment.” *SIGMOD*, 1995
  - Identified the problem of changing base table states
  - Proposed the idea of compensation
- Salem et al. “How to Roll a Join: Asynchronous Incremental View Maintenance.” *SIGMOD*, 2000
  - Proposed the idea of asynchronous change propagation based on compensation
  - Prototyped in a commercial DBMS

Data warehousing

- The ETL process
  - Extract data from operational data sources
  - Transform (cleanse and integrate) data
  - Load data into a central warehouse
- Data warehouse data = materialized views over source data
  - Supports fast OLAP (On-Line Analytical Processing)
  - Needs to be kept up-to-date w.r.t. source data
    - ➢The view maintenance problem!

Correct view maintenance

Source:

R:

W	X
1	2

S:

X	Y
2	4
2	3

Warehouse:

$\pi_W (R \bowtie S)$:

W
1
1

- Update U_1 = insert(S, [2, 3]) occurs at the source and is reported to the warehouse
- Warehouse sends $Q_1 = \pi_W (R \bowtie [2, 3])$ to the source
  - Recall change propagation equations
    $R \bowtie (S \oplus \Delta S) = (R \bowtie S) \oplus (R \bowtie \Delta S)$
    $\pi_W (T \oplus \Delta T) = \pi_W (T) \oplus \pi_W (\Delta T)$
- Source evaluates Q_1 and returns answer A_1 = [1]
- Warehouse receives A_1 and adds it to the view

Observations

- To maintain a warehouse view, we may need to send queries back to the sources
  - For a join view, we need to join a delta with the other base tables
  - Source queries are not needed for selection and projection views (assuming minimal deltas, i.e., no over-delete)
- … Unless we store enough information at the warehouse to make it self-maintainable
  - Thursday

A maintenance anomaly

Source:

R:

W	X
1	2
4	2

S:

X	Y
2	3

Warehouse:

$\pi_W (R \bowtie S)$:

W
1
4
4

Wrong!

- Source executes and sends U_1 = insert(S, [2, 3])
- Warehouse receives U_1 and sends $Q_1 = \pi_W (R \bowtie [2, 3])$
- Source executes and sends U_2 = insert(R, [4, 2])
- Warehouse receives U_2 and sends $Q_2 = \pi_W ([4, 2] \bowtie S)$
- Source receives and evaluates Q_1, returns A_1 = [1], [4]
- Warehouse receives A_1 and adds [1], [4] to the view
- Source receives and evaluates Q_2, returns A_2 = [4]
- Warehouse receives A_2 and adds [4] to the view

Another maintenance anomaly

Source:

R:

W	X
~~1~~	~~2~~

S:

X	Y
~~2~~	~~3~~

Warehouse:

$\pi_W (R \bowtie S)$:

W
1 (Wrong!)

- Source executes and sends U_1 = delete(R, [1, 2])
- Warehouse receives U_1 and sends $Q_1 = \pi_W ([1, 2] \bowtie S)$
- Source executes and sends U_2 = delete(S, [2, 3])
- Warehouse receives U_2 and sends $Q_2 = \pi_W (R \bowtie [2, 3])$
- Source receives and evaluates Q_1, returns $A_1 = \varnothing$
- Warehouse receives A_1 and does nothing
- Source receives and evaluates Q_2, returns $A_2 = \varnothing$
- Warehouse receives A_2 and does nothing

What went wrong?

- Change propagation equations should be evaluated over the original state of the base tables
  - Example: $R \bowtie (S \oplus \Delta S) = (R \bowtie S) \oplus (R \bowtie \Delta S)$, where $(R \bowtie \Delta S)$ should read the state of R at the time when ΔS occurs
- But when source receives the maintenance query, base tables might have changed already
  - Example: R changes after the warehouse receives ΔS and before the source receives $(R \bowtie \Delta S)$

Solution: compensation

- Augment maintenance queries with compensating queries to offset the effect of concurrent updates
- Example
  - Warehouse receives ΔS
  - Warehouse sends $Q_1 = (R \bowtie \Delta S)$ to source
  - Warehouse receives ΔR before receiving answer to Q_1
  - Instead of just sending $Q_2 = (\Delta R \bowtie S)$, warehouse sends $Q_2 = (\Delta R \bowtie S) \ominus (\Delta R \bowtie \Delta S)$, where the term $(\Delta R \bowtie \Delta S)$ compensates for Q_1

ECA (Eager Compensating Algorithm)

- Warehouse maintains *UQS* (Unanswered Query Set)
  - That is, the set of queries that were sent by the warehouse, but whose answers have not been received
- When warehouse receives source update U_i
  - Formulate a maintenance query Q_i based on U_i
  - For each query in *UQS*, formulate a compensating query Q' based on U_i, and augment Q_i with $\ominus Q'$
  - Send the augmented Q_i to the source
- Assumption: If A_j is received after U_i, then A_j has seen the effect of U_i
  - Send the message in the same transaction
  - Assume in-order message delivery

A note on negative deltas

- If tuples are allowed to have negative counts, then we can capture all changes in a single ΔR rather than the pair ∇R and ΔR
- Everything continues to work
  - $\oplus$ adds counts of matching tuples
  - $\ominus$ subtracts counts of matching tuples
  - $\times$ multiplies tuple counts
    - Yes, negative times negative is positive

ECA example

Source:

R:

W	X
1	2
4	2

S:

X	Y
2	5

T:

Y	Z
5	3

Warehouse:

$\pi_W (R \bowtie S \bowtie T)$:

W
4
1

- U_1 = insert(R, [4, 2])
- $Q_1 = \pi_W ([4, 2] \bowtie S \bowtie T)$
- U_2 = insert(T, [5, 3])
- $Q_2 = \pi_W (R \bowtie S \bowtie [5, 3])$ $\boxed{\ominus\, \pi_W ([4, 2] \bowtie S \bowtie [5, 3])}$ for Q_1
- U_3 = insert(S, [2, 5])
- $Q_3 = \pi_W (R \bowtie [2, 5] \bowtie T)$ $\boxed{\ominus\, \pi_W ([4, 2] \bowtie [2, 5] \bowtie T)}$ for Q_1
  $\boxed{\ominus\, (\pi_W (R \bowtie [2, 5] \bowtie [5, 3]) \ominus \pi_W ([4, 2] \bowtie [2, 5] \bowtie [5, 3]))}$ for Q_2
- $A_1 = [4]$; $A_2 = [1]$; $A_3 = \varnothing$

Summary

- Problem: changing base table states
- Solution: compensation
- Trick: negative counts

- Lots, lots of follow-on work
  - More efficient algorithms
  - Multi-source version
  - Parallel version

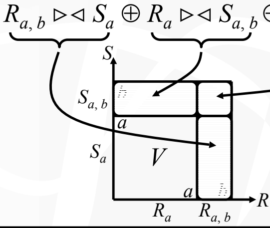
Traditional database view maintenance

- Incremental maintenance is executed as an atomic transaction
  - Blocks updates to base tables
  - Blocks reads of views
  - ➢Could be broken into separate propagation and apply phases
- The maintenance transaction is synchronous and needs to see particular states of the base tables around the time of the refresh

Synchronous propagation using pre-states

Griffin and Libkin

- $V = R \rhd\lhd S$
- a: last refresh time; b: current refresh time
- $V_{a,b} = R_{a,b} \rhd\lhd S_a \oplus R_a \rhd\lhd S_{a,b} \oplus R_{a,b} \rhd\lhd S_{a,b}$

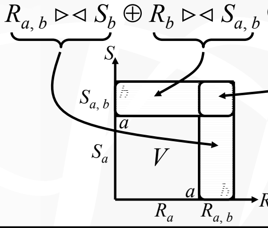


Pre-states are hard to obtain sometimes

Synchronous propagation using after-states

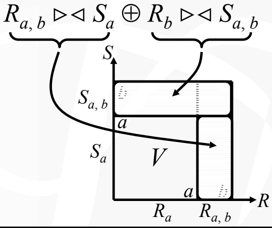
Oracle (?)

- $V = R \rhd\lhd S$
- a: last refresh time; b: current refresh time
- $V_{a,b} = R_{a,b} \rhd\lhd S_b \oplus R_b \rhd\lhd S_{a,b} \ominus R_{a,b} \rhd\lhd S_{a,b}$



Synchronous propagation using mixed states

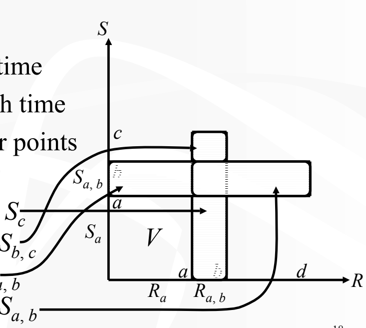
- $V = R \rhd\lhd S$
- a: last refresh time; b: current refresh time
- $V_{a,b} = R_{a,b} \rhd\lhd S_a \oplus R_b \rhd\lhd S_{a,b}$



Simpler, but harder to implement

Asynchronous propagation

- $V = R \rhd\lhd S$
- a: last refresh time
- b: target refresh time
- c, d: some later points in time
- $V_{a,b} = R_{a,b} \rhd\lhd S_c$
  $\ominus R_{a,b} \rhd\lhd S_{b,c}$
  $\oplus R_d \rhd\lhd S_{a,b}$
  $\ominus R_{a,d} \rhd\lhd S_{a,b}$



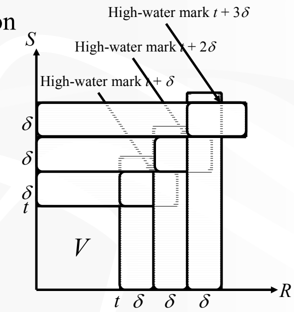
Advantages of asynchronous propagation

$\underbrace{R_{a,b} \rhd\lhd S_c}_{\text{forward query}} \ominus \underbrace{R_{a,b} \rhd\lhd S_{b,c}}_{\text{compensation query}} \oplus \underbrace{R_d \rhd\lhd S_{a,b}}_{\text{forward query}} \ominus \underbrace{R_{a,d} \rhd\lhd S_{a,b}}_{\text{compensation query}}$

- Flexibility: Reading of base tables can happen any later time (independent of *a* and *b*)
  - Although every read of a base table must be properly compensated
- More concurrency: Each term can be evaluated in a different transaction

Continuous propagation process

- Choose a propagation interval length δ
- Asynchronously compute $V_{t,\, t+\delta}$
  - May be executed after $t + \delta$
- $t \leftarrow t + \delta$
- Repeat
- δ is tunable



Apply process

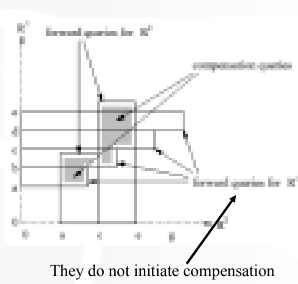
- Delta timestamps
  - Assume that base table delta tuples are timestamped by transactions that update them
  - Compute timestamps for view delta tuples
    - Join returns the smallest timestamp (may sound counterintuitive, but works with compensation)
- An independent apply process can refresh the view to any time *t* before the current high-water mark
- Without these timestamps, the apply process can refresh the view only to a high-water mark

Example of timestamp computation

- $V = R \rhd\lhd S$
  - Last refreshed at time *t*; need to refresh to time *t'*
- At time *a*, insert(*R*, *x*); at time *b*, insert(*S*, *y*)
  - $t < a < b < t'$
- At time $c > t'$, calculate forward query $R_{t,\, t'} \rhd\lhd S_c$
  - Adds *xy* to view delta with timestamp *a*
- Compensate by $R_{t,\, t'} \rhd\lhd S_{t',\, c}$
  - Empty
- At time $d > t'$, calculate forward query $R_d \rhd\lhd S_{t,\, t'}$
  - Adds *xy* to view delta with timestamp *b* ← Correct effect
- Compensate by $R_{t,\, d} \rhd\lhd S_{t,\, t'}$
  - Subtracts *xy* from view delta with timestamp *a*

Rolling propagation

- Flexibility: Different base tables may be updated at different rates, so allow each base table's propagation interval to be tuned independently
- Efficiency: Do not set target refresh time for a view in advance; less compensation work



Complications

- Regions that require compensation may not be rectangular
  - A query always returns a rectangular region
  - So multiple compensation queries may be needed
- High-water marks are no longer determined in advance
  - The current high-water mark must be calculated as the beginning of the oldest query that has not been completely compensated
    - Prior to this point, all forward queries have been completely compensated

Implementation issues

- Detecting and timestamping base table deltas
  - Log-based approach
  - Trigger-based approach
- Determining the evaluation time of a query (or the base table state that it reads)

Log-based approach

- Used by the paper on DB2
- A tool continuously examines the database transaction log and populates base table deltas
  - Transaction ID
  - Commit sequence number (unique "timestamp")
  - Commit timestamp (not necessarily unique)
- Advantage: does not disrupt normal database operations
- Disadvantage: needs to scan through many unnecessary log entries if we are only interested in a few base tables

Trigger-based approach

- Define a trigger on the base table that fires whenever the table is updated and populates the delta table
  - What is the timestamp then?
    - A regular trigger has no access to the commit sequence number because it is not known until commit time
    - A commit trigger (fired at commit time) is required but is not a standard DBMS feature
- Disadvantage: interferes with normal database operations

Determining query evaluation time

- We need the commit sequence number of the transaction in which a propagation query is evaluated
- But it is difficult to tell which log entries belong to this particular transaction

➢ Hack: make this transaction write a unique value into a special table

➢ These solutions are very system-dependent!

Next time

- All the continuous changing base table states give me headaches!

➢ Self-maintainable views—do not rely on base tables for view maintenance!