View Self-Maintenance

CPS 296.1
Topics in Database Systems

Self-maintainable views

- A view is self-maintainable if it can be maintained without accessing the base tables
  - That is, using just the base table deltas and the old content of the view itself
- Advantages of self-maintainable views
  - Efficiency: no need to access base tables
  - Simplicity: no problem with changing base table states

Examples

- Self-maintainable
  - $V = \sigma_p R$
    - $\nabla V = \sigma_p (\nabla R)$, $\Delta V = \sigma_p (\Delta R)$
  - $V = \max(R)$ w.r.t. ΔR
    - $\nabla V = V$, $\Delta V = \max(V, \Delta R)$
- Not self-maintainable
  - $V = R \bowtie S$ w.r.t. insertions
    - $\Delta V = (\Delta R \bowtie S) \oplus (R \bowtie \Delta S) \oplus (\Delta R \bowtie \Delta S)$
    - What about deletions?
  - $V = \max(R)$ w.r.t. ∇R
    - If $V \subseteq \nabla R$, then V must be recomputed as $\max(R)$

Making a view self-maintainable

- If V is not self-maintainable, add a set of auxiliary views $\mathscr{A}$ such that V and $\mathscr{A}$ taken together can be maintained without accessing any base tables
  - That is, using just the base table deltas and the old content of V and $\mathscr{A}$ itself
- Example
  - $V = \max(R)$ is not self-maintainable
  - Add auxiliary view $A = R$
  - V and A together are self-maintainable
  - Why not just $A = \text{second\_max}(R)$?

A more interesting example

- Store(store_id, city, state, manager)
- Sale(sale_id, store_id, day, month, year)
- Line(line_id, sale_id, item_id, price)
- Item(item_id, item_name, category, supplier)
- $V = \pi_{\text{manager, month, sale\_id, line\_id, item\_id, item\_name, price}}$ $\sigma_{\text{state = “CA” AND year = 1996 AND category = “toy”}}$ $(\text{Store} \bowtie_{\text{store\_id}} \text{Sale} \bowtie_{\text{sale\_id}} \text{Line} \bowtie_{\text{item\_id}} \text{Item})$
  - Not self-maintainable because of joins

Naïve approach

- Add auxiliary views that simply copy base tables
  - A_{Store} = Store
  - A_{Sale} = Sale
  - A_{Line} = Line
  - A_{Item} = Item
- Implemented by most commercial data warehouses
- Certainly correct, but very inefficient
  - All copies are self-maintainable by themselves
  - V is maintainable (even computable) from these copies

A smarter approach

- $V = \pi_{\text{manager, month, sale\_id, line\_id, item\_id, item\_name , price}}$
  $\sigma_{\text{state = “CA” AND year = 1996 AND category = “toy”}}$
  $(\text{Store} \bowtie_{\text{store\_id}} \text{Sale} \bowtie_{\text{sale\_id}} \text{Line} \bowtie_{\text{item\_id}} \text{Item})$
- Push selection/projection into auxiliary views
  - $A_{\text{Store}} = \pi_{\text{store\_id, manager}}\, \sigma_{\text{state = “CA”}}\, \text{Store}$
  - $A_{\text{Sale}} = \pi_{\text{sale\_id, store\_id, month}}\, \sigma_{\text{year = 1996}}\, \text{Sale}$
  - $A_{\text{Line}} = \text{Line}$
  - $A_{\text{Item}} = \pi_{\text{item\_id, item\_name}}\, \sigma_{\text{category = “toy”}}\, \text{Item}$
- Correct, and less inefficient
  - All select-project views are self-maintainable themselves
  - V is maintainable (even computable) from these views

More information

- Key and foreign-key constraints
- Insert/delete/update patterns
  - Append-only tables, updateable columns, etc.

- Store(store_id, city, state, manager)
- Sale(sale_id, store_id, day, month, year)
- Line(line_id, sale_id, item_id, price)
- Item(item_id, item_name, category, supplier)
  - Also, columns referenced in selection/join conditions are not updated

Better auxiliary views

Given the additional constraints

- $A_{\text{Store}} = \pi_{\text{store\_id, manager}}\, \sigma_{\text{state = “CA”}}\, \text{Store}$
  - Same as before
- $A_{\text{Sale}} = \pi_{\text{sale\_id, store\_id, month}}\, \sigma_{\text{year = 1996}}\, \text{Sale}$
  $\ltimes_{\text{store\_id}} A_{\text{Store}}$
  - Note the extra semijoin
- $A_{\text{Item}} = \pi_{\text{item\_id, item\_name}}\, \sigma_{\text{category = “toy”}}\, \text{Item}$
  - Same as before
- No A_{Line} needed

Why the extra semijoin?

$A_{\text{Sale}} = (\pi_{\text{sale\_id, store\_id, month}}\, \sigma_{\text{year = 1996}}\, \text{Sale}) \ltimes_{\text{store\_id}} A_{\text{Store}}$

- Sale deltas do not need to be joined with Sale
- Line and Item deltas are always joined with Sale and Store together
  - Computable from $A_{\text{Sale}} \bowtie_{\text{store\_id}} A_{\text{Store}}$ (semijoin does not hurt)
- ΔStore cannot join with existing Sale tuples
  - Because every existing Sale references an existing store_id
- ∇Store cannot join with existing Sale tuples
  - Because if it does, it would violate the foreign-key constraint
  - If it cascades, join with A_{Sale} to find sale_id’s to delete from V

Why no A_{Line}?

- Line deltas do not need to be joined with Line
- ΔItem and ΔSale cannot join with existing Line tuples
  - Because every existing Line references an existing item_id and an existing sale_id
- ∇Item and ∇Sale cannot join with existing Line tuples
  - Because if they do, they would violate the foreign-key constraints
  - If they cascade, delete from V deleted item_id’s and sale_id’s
- Store deltas cannot join with existing Line tuples
  - Because they cannot even join with existing Sale tuples

What about updates?

- In most view maintenance literature, an update is treated as a deletion followed by an insertion
- Approach becomes problematic if we want to exploit foreign-key constraints

- Example: updating Store.manager
  - ∇Store = [123, “Fremont”, “CA”, “Amy”]
  - ΔStore = [123, “Fremont”, “CA”, “Ben”]
  - Applying ∇Store and ΔStore separately would temporarily violate the foreign-key constraint from Sale.store_id to Store.store_id
  - ➢ Must treat update as one operation

Characterizing updates

- Exposed update
  - Changes the value of a column referenced in select/join conditions of the view
  - May cause insertion into or deletion from the view
- Protected update
  - Not exposed, but changes the value of a column that is included in the final projection of the view
  - Causes the view column to be updated
- Ignorable update
  - Neither exposed nor protected
  - No effect on the view

Auxiliary views re-examined

- Assume no exposed updates
- For protected updates on Sale, Item, or Line, simply update all *V* tuples with the affected sale_id's, item_id's, or line_id's
- For protected updates on Store, join with A_{Sale} to find all sale_id's associated with the updated stores, and then update *V* tuples with these sale_id's

What if exposed updates are allowed?

- Say Sale.year may be updated
- Must add auxiliary view

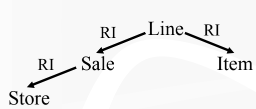
$$A_{\text{Line}} = \pi_{\text{line\_id, sale\_id, item\_id, price}}\ \text{Line} \triangleright\!\!<_{\text{item\_id}} A_{\text{Item}}$$

  - Any Line can be a 1996 sale after a Sale.year update

Self-maintenance algorithm

- How to generate definitions for auxiliary views
- How to maintain the original view
- How to maintain the auxiliary views
- Quass et al. "Making Views Self-Maintainable for Data Warehousing." *PDIS*, 1996

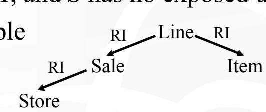
Join graph of a view



- Node *R*: base table *R*
- Directed edge $R \rightarrow S$: join condition of the form $R.A = S.K$, where *K* is a key of *S*
  - The edge is further annotated with *RI* if there is a foreign-key constraint from *R.A* to *S.K*

Dep(*R*)

- Dep(*R*) = { *S* | there is an edge $R \rightarrow S$ annotated with *RI*, and *S* has no exposed updates }
- Example



  - Dep(Store) = ∅
  - Dep(Sale) = { Store }
  - Dep(Item) = ∅
  - Dep(Line) = { Sale, Item }

Intuition behind Dep(R)

A_R can be semijoined with A_S for every S in Dep(R)

- If r in R does not semijoin with A_S, then
  - r must join with some existing s in S not in A_S (foreign-key constraint)
  - r cannot join with ΔS (key constraint on S)
  - s will never contribute to V (no exposed updates on S)
  - ➢ r will never contribute to V

Dep$^+$(R)

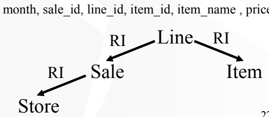
- Dep$^+$(R) is the transitive closure of Dep(R)
  - That is, Dep$^+$(R) $\leftarrow$ Dep(R), and
  - If S is in Dep$^+$(R), then so are tables in Dep(S)
- Example
  - Dep$^+$(Store) = $\varnothing$
  - Dep$^+$(Sale) = { Store }
  - Dep$^+$(Item) = $\varnothing$
  - Dep$^+$(Line) = { Sale, Item, Store }

Intuition behind Dep$^+$(R)

- If Dep$^+$(R) includes all tables in V other than R itself, then A_R is not needed for processing inserts
- Every S is reachable from R from a chain of foreign-key joins, say $R \rightarrow S_1 \rightarrow \ldots \rightarrow S_k \rightarrow S$
  - ΔS cannot join with existing S_k tuples, and therefore cannot join with exiting $S_{k-1}, \ldots, S_1$, and R tuples

Need(R)

- If the key of R is preserved in V
  Need(R) = $\varnothing$
- Otherwise, if there exists S s.t. $S \rightarrow R$
  Need(R) = { S } $\cup$ Need(S)
- Otherwise, Need(R) = all tables except R itself
- Example $V = \pi_{\text{manager, month, sale\_id, line\_id, item\_id, item\_name , price}}(\ldots)$
  - Need(Store) = { Sale }
  - Need(Sale) = $\varnothing$
  - Need(Item) = $\varnothing$
  - Need(Line) = $\varnothing$



Intuition behind Need(R)

- If S appears in Need(R) then A_S may be needed for processing deletes and protected updates on R
- To process a delete or a protected update on R, we need to identify V tuples that are affected by this modification
  - If R's key is preserved in V, we know which tuples are affected
  - Otherwise, we can join the modification with A_S to find the S keys of the affected V tuples

Generating auxiliary views

For each R

- If Dep$^+$(R) includes all other tables and R is not contained in any Need(S), then A_R is not needed
  - Only happens for the root of the join graph
- Otherwise, push selection and projection down into A_R as much as possible, but preserve the key of R
- Semijoin A_R with A_S for every S in Dep(R)

➢ No recursive definition if join graph is a tree

Maintaining the original view

- Basic strategy: start with regular change propagation equations, rewrite the change terms to reference only deltas, A_R's, and/or V
  - Inserts
  - Deletes
  - Updates (protected and exposed)

Strategy for inserts

- Eliminate terms that are guaranteed to be $\varnothing$
  - If there is a foreign-key join from $R.A$ to $S.K$, then $\ldots \bowtie R \bowtie \ldots \bowtie \Delta S \bowtie \ldots = \varnothing$
- In the remaining terms, replace R's with A_R's
  - Rewrite $\ldots \bowtie R \bowtie \ldots \bowtie S \bowtie \ldots$ as $\ldots \bowtie A_R \bowtie \ldots \bowtie A_S \bowtie \ldots$
  - Note that in the remaining terms, R always appears together with S, so the semijoin with A_S is harmless

Strategy for deletes

- Rewrite terms to reference V whenever possible
  - If key(R) is preserved in V, then $\ldots \bowtie \nabla R \bowtie \ldots = V \rtimes_{\text{key}(R)} \nabla R$
  - If key(R) is not preserved in V, but there is a chain join $S_1 \to S_2 \to \ldots \to S_k \to R$, and key($S_1$) is preserved in V, then

$$\ldots \bowtie \nabla R \bowtie \ldots = V \rtimes_{\text{key}(S_1)} (A_{S_1} \rtimes_{\text{key}(S_2)} (A_{S_2} \rtimes_{\text{key}(S_3)} (\ldots (A_{S_k} \rtimes_{\text{key}(R)} \nabla R) \ldots)))$$

Strategy for updates

- Protected updates
  - Similar to deletes
  - Rewrite using V, using additional joins as necessary to recover preserved keys
- Exposed updates
  - Treated as deletes followed by inserts
  - Cannot exploit foreign-key constraints

Maintaining auxiliary views

- Insertion
  - $\Delta A_R = (\pi \sigma \Delta R) \rtimes \ldots \rtimes A_S \rtimes \ldots$
  - Since S is in Dep(R), ΔS has no effect on A_R
- Deletion
  - $\nabla A_R = A_R \rtimes \nabla R$
  - $\nabla A_R = A_R \rtimes \nabla A_S$
  - A_R preserves the key of R and the foreign key reference to S
- Protected updates
  - For a protected update on R, just update A_R because A_R preserves the key
  - Protected updates on S do not affect A_R

Recap

- Bottom line: use constraints to simplify view maintenance
  - Start with change propagation equations
  - Using constraints, simplify equations or rewrite them to reference the view itself
  - Examine remaining terms and see if tables can be joined to form auxiliary views
    - Joins (or semijoins) serve as additional filters
  - Don't forget to check that auxiliary views themselves are self-maintainable!

Compile- vs. run-time self-maintenance

- Compile-time self-maintenance (this paper)
  - Views are always self-maintainable, no matter what the current database state is and what changes may occur in the future
  - Strong guarantee, but large auxiliary views
- Run-time self-maintenance
  - Look at each change and the current view content, decide whether it is possible to self-maintain the view
    - Example: $V = \max(R)$
    - Example: most updates are protected, but some are exposed
  - Base tables are accessed only when necessary