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
    - $\Delta V = (\Delta R \bowtie S) \uplus (R \bowtie \Delta S) \uplus (\Delta R \bowtie \Delta S)$
    - 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 $\mathcal{A}$ such that V and $\mathcal{A}$ 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 $\mathcal{A}$ 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 = 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

- $V = \pi_{\text{manager, month, sale_id, line_id, item_id, item_name, price}}$
  $\sigma_{\text{state = "CA" AND year = 1996 AND category = "toy"}}$
  (Store $\bowtie_{\text{store_id}}$ Sale $\bowtie_{\text{sale_id}}$ Line $\bowtie_{\text{item_id}}$ Item)
- Push selection/projection into auxiliary views
  - $A_{\text{Store}} = \pi_{\text{store_id, manager}}\ \sigma_{\text{state = "CA"}}$ Store
  - $A_{\text{Sale}} = \pi_{\text{sale_id, store_id, month}}\ \sigma_{\text{year = 1996}}$ Sale
  - A_{Line} = Line
  - $A_{\text{Item}} = \pi_{\text{item_id, item_name}}\ \sigma_{\text{category = "toy"}}$ 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

- $A_{\text{Store}} = \pi_{\text{store_id, manager}}\ \sigma_{\text{state = "CA"}}$ Store
  - Same as before
- $A_{\text{Sale}} = \pi_{\text{sale_id, store_id, month}}\ \sigma_{\text{year = 1996}}$ Sale $\rhd\!<_{\text{store_id}} A_{\text{Store}}$
  - Note the extra semijoin
- $A_{\text{Item}} = \pi_{\text{item_id, item_name}}\ \sigma_{\text{category = "toy"}}$ Item
  - Same as before
- No A_{Line} needed

Why the extra semijoin?

$A_{\text{Sale}} = (\pi_{\text{sale_id, store_id, month}}\ \sigma_{\text{year = 1996}}$ Sale) $\rhd\!<_{\text{store_id}} A_{\text{Store}}$

- 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?

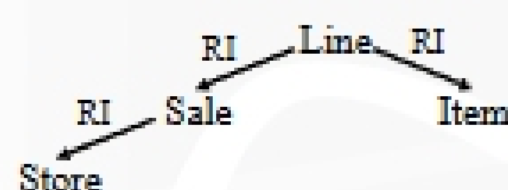
- Say Sale.year may be updated
- Must add auxiliary view

  $A_{Line} = \pi_{line_id,\ sale_id,\ item_id,\ price}\ \text{Line} \rhd\!< _{item_id} A_{Item}$
  - 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

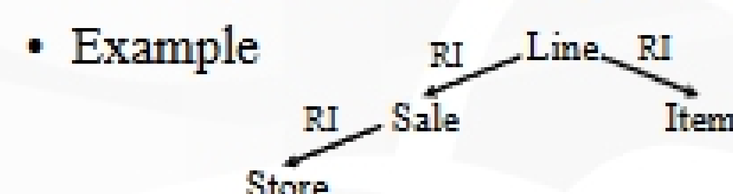
Join graph of a view



- 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) = $\varnothing$
  - Dep(Sale) = { Store }
  - Dep(Item) = $\varnothing$
  - Dep(Line) = { Sale, Item }