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. \( \nabla R \)
  - \( \Delta V = \max(V, \Delta R) \)
- Not self-maintainable
  - \( V = R >\sigma S \) w.r.t. insertions
  - \( \Delta V = (\Delta R >\sigma S) \oplus (R >\sigma S) \oplus (\Delta R >\sigma S) \)
  - What about deletions?
  - \( V = \max(R) \) w.r.t. \( \nabla R \)
  - 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 \( A \) such that \( V \) and \( A \) 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 \( A \) itself
- Example
  - \( V = \max(R) \) is not self-maintainable
  - Add auxiliary view \( A = R \)
  - \( V \) and \( A \) together are self-maintainable
  - Why not just \( A = \text{second\_max}(R) \)?

Naïve approach

- Add auxiliary views that simply copy base tables
  - \( A_{\text{store}} = \text{Store} \)
  - \( A_{\text{sale}} = \text{Sale} \)
  - \( A_{\text{line}} = \text{Line} \)
  - \( A_{\text{item}} = \text{Item} \)
- 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 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)
- \( V = \pi_{\text{manager}, \text{month}, \text{sale\_id}, \text{line\_id}, \text{item\_id}, \text{item\_name}, \text{price}} \)
  - \( \sigma_{\text{state} = \text{“CA”} \text{ AND year = 1996 AND category = “toy”}} \)
  - (Store >\sigma_{\text{store\_id}} \text{Sale} >\sigma_{\text{sale\_id}} \text{Line} >\sigma_{\text{item\_id}} \text{Item})
- Not self-maintainable because of joins
A smarter approach

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

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

Why the extra semijoin?

\( A_{\text{Sale}} = (\pi_{\text{sale_id, store_id, month \sigma_{\text{year} = 1996}}} \text{Sale}) \bowtie_{\text{store_id}} A_{\text{Store}} \)
- 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)
- \( \Delta_{\text{Store}} \) cannot join with existing Sale tuples
  - Because every existing Sale references an existing store_id
- \( V \setminus \text{Store} \) cannot join with existing Sale tuples
  - Because if it does, it would violate the foreign-key constraint
  - If it cascades, join with \( \Delta_{\text{Sale}} \) to find sale_id’s to delete from \( V \)

Why no \( A_{\text{Line}} \)?

- Line deltas do not need to be joined with Line
- \( \Delta_{\text{Item}} \) and \( \Delta_{\text{Sale}} \) cannot join with existing Line tuples
  - Because every existing Line references an existing item_id and an existing sale_id
- \( V \setminus \text{Item} \) and \( V \setminus \text{Sale} \) 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
  - \( V \setminus \text{Store} = [123, \text{“Fremont”}, \text{“CA”}, \text{“Amy”}] \)
  - \( \Delta_{\text{Store}} = [123, \text{“Fremont”}, \text{“CA”}, \text{“Ben”}] \)
  - Applying \( V \setminus \text{Store} \) and \( \Delta_{\text{Store}} \) 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 \( A_{\text{Sale}} \) 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
  \[ A_{\text{Line}} = \pi_{\text{line_id, sale_id, item_id, price}} \text{Line} \]
  – 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


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\( (R) \)

• \( \text{Dep}(R) = \{ S \mid \text{there is an edge } R \rightarrow S \text{ annotated with } RI, \text{ and } S \text{ has no exposed updates} \} \)
• Example
  \[ \begin{align*}
  \text{Dep}(\text{Store}) &= \emptyset \\
  \text{Dep}(\text{Sale}) &= \{ \text{Store} \} \\
  \text{Dep}(\text{Item}) &= \emptyset \\
  \text{Dep}(\text{Line}) &= \{ \text{Sale, Item} \}
  \end{align*} \]
Intuition behind Dep\( (R) \)

\( A_R \) can be semijointed 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 \( \Delta S \) (key constraint on \( S \))
  - \( s \) will never contribute to \( V \) (no exposed updates on \( S \))

\( \Rightarrow r \) will never contribute to \( V \)

\( \text{Dep}(R) \)

- \( \text{Dep}(R) \) is the transitive closure of \( \text{Dep}(R) \)
  - That is, \( \text{Dep}(S) \leftrightarrow \text{Dep}(R) \), and
  - If \( S \) is in \( \text{Dep}(R) \), then so are tables in \( \text{Dep}(S) \)

\[ \text{Example} \]
- \( \text{Dep}(\text{Store}) = \emptyset \)
- \( \text{Dep}(\text{Sale}) = \{ \text{Store} \} \)
- \( \text{Dep}(\text{Item}) = \emptyset \)
- \( \text{Dep}(\text{Line}) = \{ \text{Sale, Item, Store} \} \)

Intuition behind \( \text{Dep}^+(R) \)

- If \( \text{Dep}^+(R) \) 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 \( R \rightarrow S_1 \rightarrow \ldots \rightarrow S_k \rightarrow S \)
  - \( \Delta S \) cannot join with existing \( S_i \) tuples, and therefore cannot join with existing \( S_{i-1} , \ldots , S_i , \) and \( R \) tuples

\( \text{Need}(R) \)

- If the key of \( R \) is preserved in \( V \)
  \( \text{Need}(R) = \emptyset \)

- Otherwise, if there exists \( S \) s.t. \( S \rightarrow R \)
  \( \text{Need}(R) = \{ S \} \cup \text{Need}(S) \)

- Otherwise, \( \text{Need}(R) = \) all tables except \( R \) itself

\[ \text{Example} \]
- \( V = \pi \text{manager, month, sale_id, line_id, item_id, item_name, price} (\ldots) \)
  - \( \text{Need}(\text{Store}) = \{ \text{Sale} \} \)
  - \( \text{Need}(\text{Sale}) = \emptyset \)
  - \( \text{Need}(\text{Item}) = \emptyset \)
  - \( \text{Need}(\text{Line}) = \emptyset \)

Generating auxiliary views

For each \( R \)

- If \( \text{Dep}^+(R) \) includes all other tables and \( R \) is not contained in any \( \text{Need}(S) \), 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 \( R \)
  - \( \text{Semijoin} A_R \) with \( A_S \) for every \( S \) in \( \text{Dep}(R) \)

\( \Rightarrow \) 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, $A_k$’s, and/or $V$
  – Inserts
  – Deletes
  – Updates (protected and exposed)

Strategy for inserts

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

Strategy for deletes

• Rewrite terms to reference $V$ whenever possible
  – If key($R$) is preserved in $V$, then $\ldots \triangleright \Delta R \triangleright \ldots = V \triangleright \text{key($R$)} \Delta R$
  – If key($R$) is not preserved in $V$, but there is a chain join $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_k \rightarrow R$, and key($S_j$) is preserved in $V$, then $\ldots \triangleright \Delta R \triangleright \ldots = V \triangleright \text{key($S_j$)} \Delta R$

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
  – $\Delta A_k = (\pi \Delta R) \triangleright \ldots \triangleright \Delta A_k \triangleright \ldots$
  – Since $S$ is in $\text{Dep}(R)$, $\Delta S$ has no effect on $A_k$
• Deletion
  – $\forall A_k = A_k \triangleright \forall R$
  – $\forall A_k = A_k \triangleright \forall A_k$
  – $A_k$ preserves the key of $R$ and the foreign key reference to $S$
• Protected updates
  – For a protected update on $R$, just update $A_k$ because $A_k$ preserves the key
  – Protected updates on $S$ do not affect $A_k$

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