Data Warehousing and Decision Support

Chapter ~~23~~, Part B

25

Views and Decision Support

- OLAP queries are typically aggregate queries.
  - Precomputation is essential for interactive response times.
  - The CUBE is in fact a collection of aggregate queries, and precomputation is especially important: lots of work on what is best to precompute given a limited amount of space to store precomputed results.
- Warehouses can be thought of as a collection of asynchronously replicated tables and periodically maintained views.
  - Has renewed interest in view maintenance!

View Modification (Evaluate On Demand)

View

```
CREATE VIEW RegionalSales(category,sales,state)
    AS SELECT P.category, S.sales, L.state
       FROM Products P, Sales S, Locations L
       WHERE P.pid=S.pid AND S.locid=L.locid
```

Query

```
SELECT R.category, R.state, SUM(R.sales)
FROM RegionalSales AS R GROUP BY R.category, R.state
```

Modified Query

```
SELECT R.category, R.state, SUM(R.sales)
FROM (SELECT P.category, S.sales, L.state
      FROM Products P, Sales S, Locations L
      WHERE P.pid=S.pid AND S.locid=L.locid) AS R
GROUP BY R.category, R.state
```

View Materialization (Precomputation)

- Suppose we precompute RegionalSales and store it with a clustered B+ tree index on [category,state,sales].
  - Then, previous query can be answered by an index-only scan.

```
SELECT R.state, SUM(R.sales)
FROM RegionalSales R
WHERE R.category="Laptop"
GROUP BY R.state
```

Index on precomputed view is great!

```
SELECT R.state, SUM(R.sales)
FROM RegionalSales R
WHERE R. state="Wisconsin"
GROUP BY R.category
```

Index is less useful (must scan entire leaf level).

Materialized Views

- A view whose tuples are stored in the database is said to be materialized.
  - Provides fast access, like a (very high-level) cache.
  - Need to maintain the view as the underlying tables change.
  - Ideally, we want incremental view maintenance algorithms.
- Close relationship to data warehousing, OLAP, (asynchronously) maintaining distributed databases, checking integrity constraints, and evaluating rules and triggers.

Issues in View Materialization

- What views should we materialize, and what indexes should we build on the precomputed results?
- Given a query and a set of materialized views, can we use the materialized views to answer the query?
- How frequently should we refresh materialized views to make them consistent with the underlying tables? (And how can we do this incrementally?)

View Maintenance

- Two steps:
  - Propagate: Compute changes to view when data changes.
  - Refresh: Apply changes to the materialized view table.
- Maintenance policy: Controls when we do refresh.
  - Immediate: As part of the transaction that modifies the underlying data tables. (**+** Materialized view is always consistent; **-** updates are slowed)
  - Deferred: Some time later, in a separate transaction. (**-** View becomes inconsistent; **+** can scale to maintain many views without slowing updates)

Deferred Maintenance

- Three flavors:
  - Lazy: Delay refresh until next query on view; then refresh before answering the query.
  - Periodic (Snapshot): Refresh periodically. Queries possibly answered using outdated version of view tuples. Widely used, especially for asynchronous replication in distributed databases, and for warehouse applications.
  - Event-based: E.g., Refresh after a fixed number of updates to underlying data tables.

Snapshots in Oracle 7

- A snapshot is a local materialization of a view on data stored at a master site.
  - Periodically refreshed by re-computing view entirely.
  - Incremental "fast refresh" for "simple snapshots" (each row in view based on single row in a single underlying data table; no DISTINCT, GROUP BY, or aggregate ops; no sub-queries, joins, or set ops)
    - Changes to master recorded in a log by a trigger to support this.

Issues in View Maintenance (1)

expensive_parts(pno) :- parts(pno, cost), cost > 1000

- What information is available? (Base relations, materialized view, ICs). Suppose parts(p5,5000) is inserted:
  - Only materialized view available: Add p5 if it isn't there.
  - Parts table is available: If there isn't already a parts tuple p5 with cost >1000, add p5 to view.
    - May not be available if the view is in a data warehouse!
  - If we know pno is key for parts: Can infer that p5 is not already in view, must insert it.

Issues in View Maintenance (2)

expensive_parts(pno) :- parts(pno, cost), cost > 1000

- What changes are propagated? (Inserts, deletes, updates). Suppose parts(p1,3000) is deleted:
  - Only materialized view available: If p1 is in view, no way to tell whether to delete it. (Why?)
    - If count(#derivations) is maintained for each view tuple, can tell whether to delete p1 (decrement count and delete if = 0).
  - Parts table is available: If there is no other tuple p1 with cost >1000 in parts, delete p1 from view.
  - If we know pno is key for parts: Can infer that p1 is currently in view, and must be deleted.

Issues in View Maintenance (3)

- View definition language? (Conjunctive queries, SQL subset, duplicates, aggregates, recursion)

```
Supp_parts(pno) :- suppliers(sno, pno), parts(pno, cost)
```

- Suppose parts(p5,5000) is inserted:
  - Can't tell whether to insert p5 into view if we're only given the materialized view.

Incremental Maintenance Alg: One Rule, Inserts

View(X,Y) :- Rel1(X,Z), Rel2(Z,Y)

- Step 0: For each tuple in the materialized view, store a "derivation count".
- Step 1: Rewrite this rule using Seminaive rewriting, set "delta_old" relations for Rel1 and Rel2 to be the inserted tuples.
- Step 2: Compute the "delta_new" relations for the view relation.
  - Important: Don't remove duplicates! For each new tuple, maintain a "derivation count".
- Step 3: Refresh the stored view by doing "multiset union" of the new and old view tuples. (I.e., update the derivation counts of existing tuples, and add the new tuples that weren't in the view earlier.)

Incremental Maintenance Alg: One Rule, Deletes

View(X,Y) :- Rel1(X,Z), Rel2(Z,Y)

- Steps 0 - 2: As for inserts.
- Step 3: Refresh the stored view by doing "multiset difference" of the new and old view tuples.
  - To update the derivation counts of existing tuples, we must now subtract the derivation counts of the new tuples from the counts of existing tuples.

Incremental Maintenance Alg: General

- The "counting" algorithm can be generalized to views defined by multiple rules. In fact, it can be generalized to SQL queries with duplicate semantics, negation, and aggregation.
  - Try and do this! The extension is straightforward.

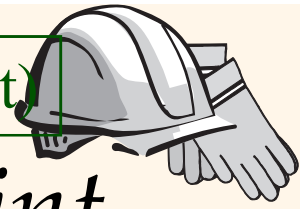
Maintaining Warehouse Views

view(sno) :- r1(sno, pno), r2(pno, cost)

- Main twist: The views are in the data warehouse, and the source tables are somewhere else (operational DBMS, legacy sources, …).
  - Warehouse is notified whenever source tables are updated. (e.g., when a tuple is added to r2)
  - Warehouse may need additional information about source tables to process the update (e.g., what is in r1 currently?)
  - The source responds with the additional info, and the warehouse incrementally refreshes the view.

Problem: New source updates between Steps 1 and 3!

view(sno) :- r1(sno, pno), r2(pno, cost)



Example of Warehouse View Maint.

- Initially, we have r1(1,2), r2 empty
- insert r2(2,3) at source; notify warehouse
- Warehouse asks ?r1(sno,2)
  - Checking to find sno's to insert into view
- insert r1(4,2) at source; notify warehouse
- Warehouse asks ?r2(2,cost)
  - Checking to see if we need to increment count for view(4)
- Source gets first warehouse query, and returns sno=1, sno=4; these values go into view (with derivation counts of 1 each)
- Source gets second query, and says Yes, so count for 4 is incremented in the view
  - **But this is wrong! Correct count for view(4) is 1.**

Warehouse View Maintenance

- Alternative 1: Evaluate view from scratch
  - On every source update, or periodically
- Alternative 2: Maintain a copy of each source table at warehouse
- Alternative 3: More fancy algorithms
  - Generate queries to the source that take into account the anomalies due to earlier conflicting updates.