ICS 421 Spring 2010
Data Warehousing 3

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa

Implementation Issues

- Recall requirements of a data warehouse:
  - Read only (updates via ETL)
  - Ad hoc queries
  - Interactive response times
- How do we support fast response times ?
  - Indexing, new indexes
  - Pre-compute results aka materialization
  - Views

Bitmap Indexes

How many male customers have a rating of 5?

SELECT COUNT(*)
FROM Customer
WHERE
Gender='M' **AND**
Rating=5

CustID	Name	Gender	Rating
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
117	Woo	M	4

M	F
1	0
1	0
0	1
1	0

Bitmap Index for Gender

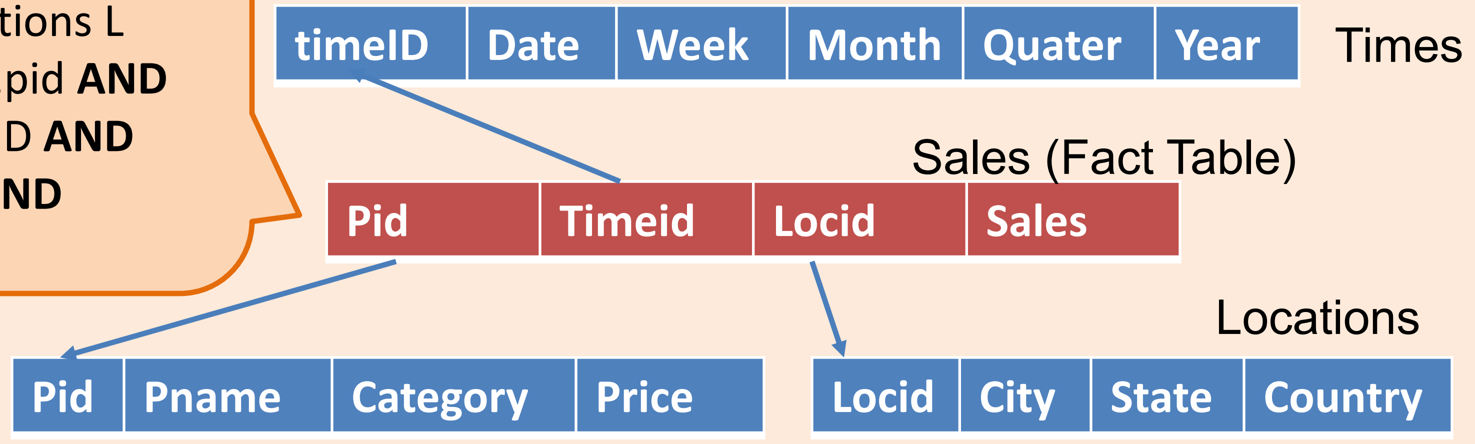
1	2	3	4	5
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Bitmap Index for Rating

- What are the possible QEPs for this query ?
- What if there are indexes on gender and rating ?
- How does bitmap indexes help ?
- Why bitmap indexes and not B+ trees ?

Join Indexes

SELECT S.Sales, T.*, P.*, L.*
FROM Sales S, Times T,
Products P, Locations L
WHERE S.pid=P.pid **AND**
S.Timeid=T.timeID **AND**
S.locid=L.locid **AND**
L.State='HI'



- How do we speed up joins with dimension tables ?
- A join index stores the RIDs of all the join tuples:
  - [RID(Sales), RID(Products), RID(Times), RID(Locations)]
- Variant: if many join queries have predicates on state
  - [Value(Location.state). RID(Sales), RID(Products), RID(Times), RID(Locations)]
  - B+ tree with Location.state as key and the tuple of RIDs as the data entry.

Bitmap Join Index (Oracle)

```
SELECT S.Sales
FROM Sales S,  Locations L
WHERE  S.locid=L.locid
AND L.State='HI'
```

```
CREATE  BITMAP INDEX
myidx
ON Sales(st.state)
FROM Sales S,  Locations L
WHERE  S.locid=L.locid
```

- Create a bitmap index where
  - One bitvector per L.state
  - Each bitvector encodes RIDs of Sales
- Index ANDing of multiple of these bitmap join indexes are efficient!
- What is the difference with a regular bitmap index ?

Views (Evaluate on Demand)

```
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
```

- A *view* is conceptually the same as a relation, but we store a *definition*, rather than a set of tuples.
- Views can be dropped using the DROP VIEW command.
  - How to handle DROP TABLE if there's a view on the table?
    - DROP TABLE command has options to let the user specify this.

Query Rewriting using Views

View Definition

```
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
```

Re-written 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
```

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

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.

Inc. View Maintenance: Inserts

```
CREATE VIEW
expensive_parts(pno)
AS
SELECT pno
FROM parts
WHERE cost > 1000
```

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

Suppose parts(p5,5000) is inserted

- What information is available?
  - 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.

Inc. View Maintenance: Deletes

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

Suppose parts(p1,3000) is delerted

- What information is available?
  - 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.

Inc. Maintenance Algorithm: Inserts

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

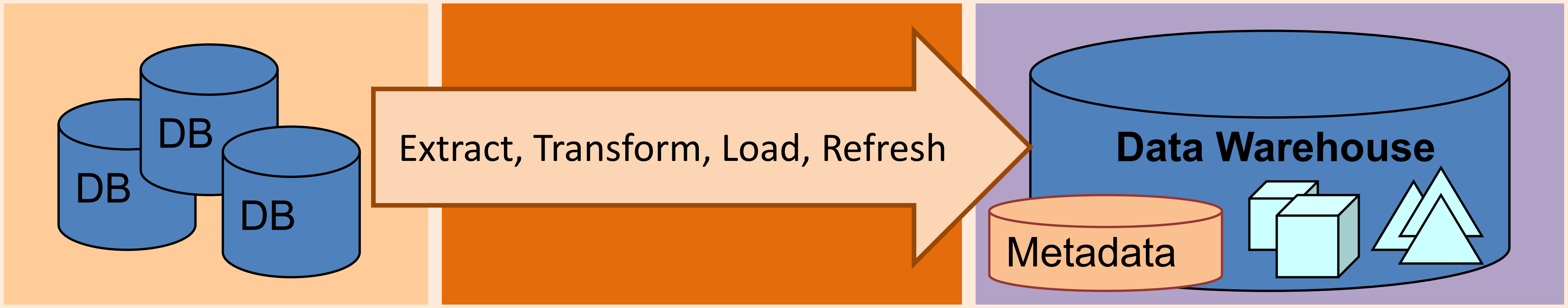
- Step 0: For each tuple in the materialized view, store a "derivation count".
- Step 1: Rewrite this rule using Seminaive rewriting, set "delta_old" relations for Rel1 and Rel2 to be the inserted tuples.
- Step 2: Compute the "delta_new" relations for the view relation.
  - Important: Don't remove duplicates! 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.)

Inc. Maintenance Algorithm: 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.
- 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.

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, …).

1) Warehouse is notified whenever source tables are updated. (e.g., when a tuple is added to r2)
2) Warehouse may need additional information about source tables to process the update (e.g., what is in r1 currently?)
3) The source responds with the additional info, and the warehouse incrementally refreshes the view.

What happens if source is updated between Steps 1 and 3?

Example: Warehouse View Maintenance

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

- 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 Approaches

- 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.