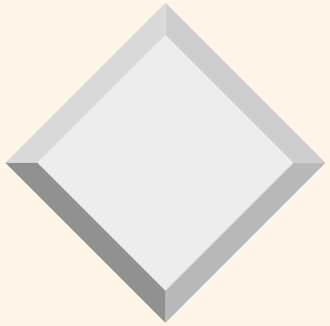


Decision Support

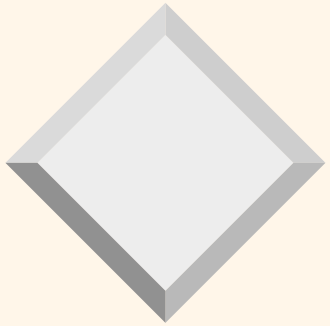
Chapter 25

Introduction

- Increasingly, organizations are analyzing current and historical data to identify useful patterns and support business strategies.
- Emphasis is on complex, interactive, exploratory analysis of very large datasets created by integrating data from across all parts of an enterprise; data is fairly static.
  - Contrast such **On-Line Analytic Processing (OLAP)** with traditional **On-line Transaction Processing (OLTP):** mostly long queries, instead of short update Xacts.

Three Complementary Trends

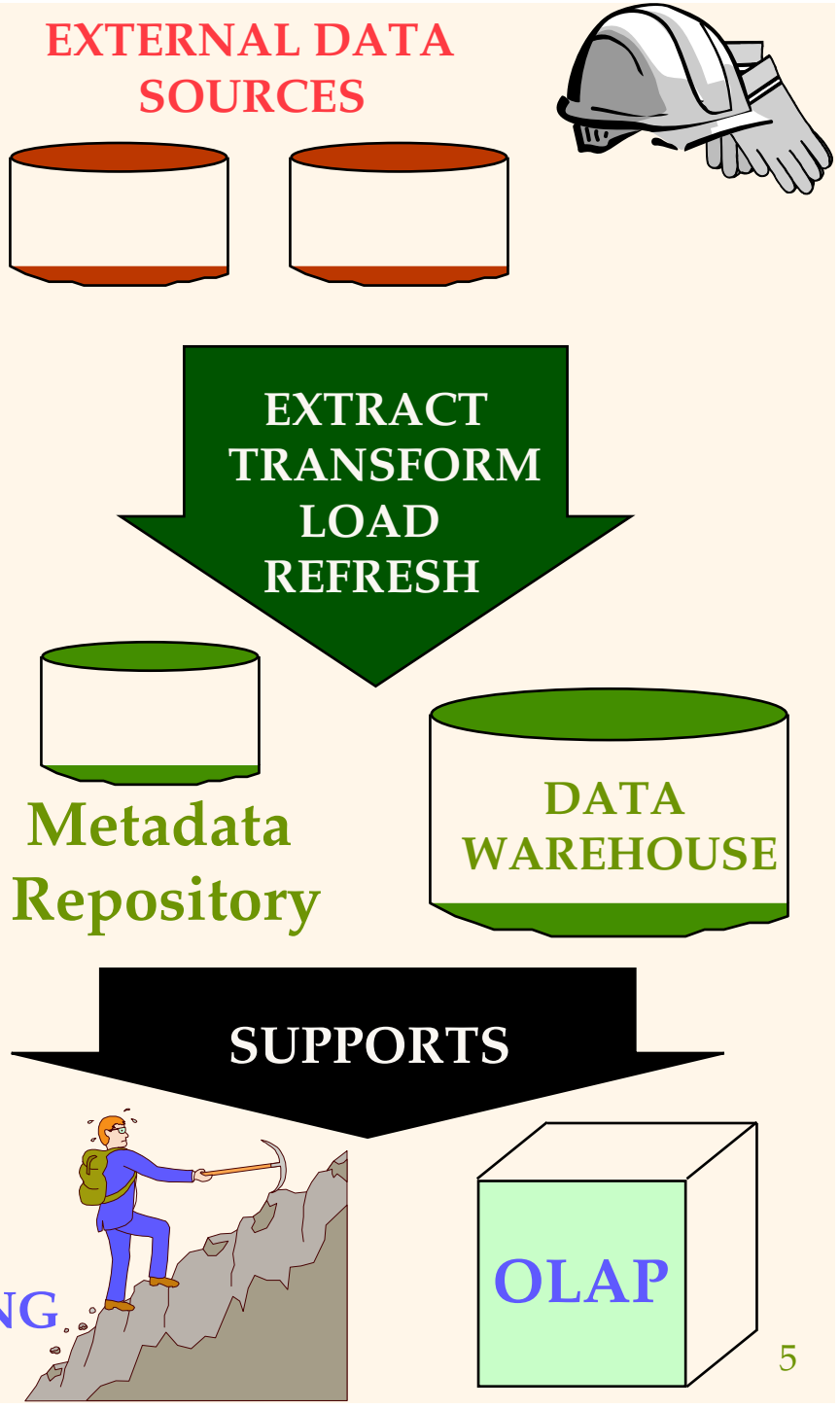
- Data Warehousing: Consolidate data from many sources in one large repository.
  - Loading, periodic synchronization of replicas.
  - Semantic integration.
- OLAP:
  - Complex SQL queries and views.
  - Queries based on spreadsheet-style operations and "multidimensional" view of data.
  - Interactive and "online" queries.
- Data Mining: Exploratory search for interesting trends and anomalies. (Another lecture!)



Data Warehousing

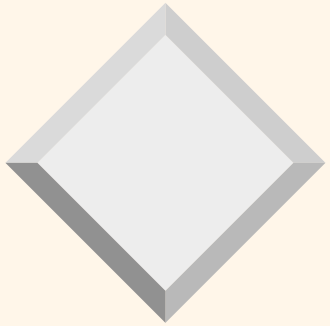
Data Warehousing

- Integrated data spanning long time periods, often augmented with summary information.
- Several gigabytes to terabytes common.
- Interactive response times expected for complex queries; ad-hoc updates uncommon.



Warehousing Issues

- Semantic Integration: When getting data from multiple sources, must eliminate mismatches, e.g., different currencies, schemas.
- Heterogeneous Sources: Must access data from a variety of source formats and repositories.
  - Replication capabilities can be exploited here.
- Load, Refresh, Purge: Must load data, periodically refresh it, and purge too-old data.
- Metadata Management: Must keep track of source, loading time, and other information for all data in the warehouse.

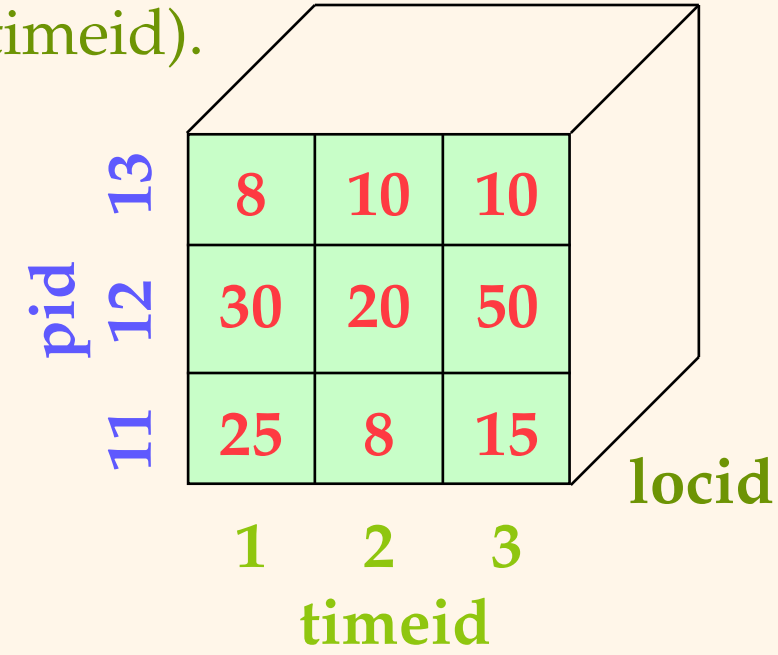


OLAP

Multidimensional Data Model

- Collection of numeric measures, which depend on a set of dimensions.
  - E.g., measure **Sales**, dimensions **Product** (key: pid), **Location** (locid), and **Time** (timeid).

Slice locid=1 is shown:



pid	timeid	locid	sales
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
13	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35

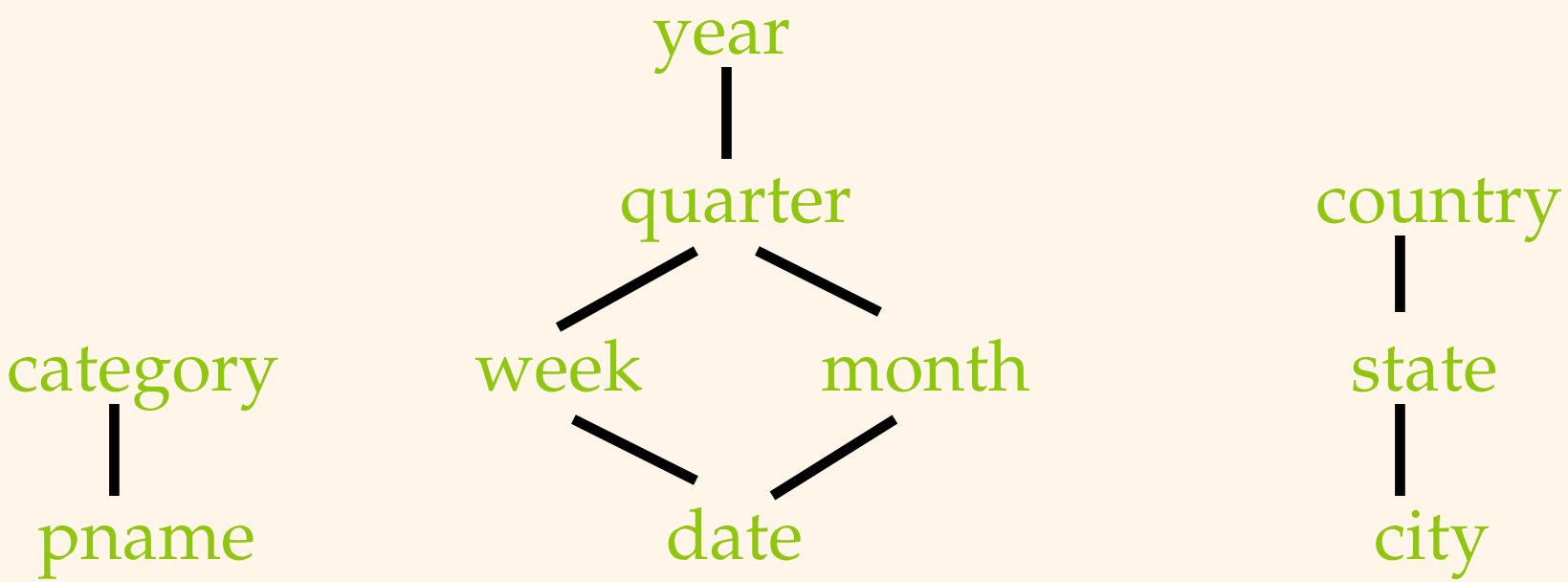
MOLAP vs ROLAP

- Multidimensional data can be stored physically in a (disk-resident, persistent) array; called MOLAP systems. Alternatively, can store as a relation; called ROLAP systems.
- The main relation, which relates dimensions to a measure, is called the fact table. Each dimension can have additional attributes and an associated dimension table.
  - E.g., **Products(pid, pname, category, price)**
  - Fact tables are *much* larger than dimensional tables.

Dimension Hierarchies

- For each dimension, the set of values can be organized in a hierarchy:

PRODUCT **TIME** **LOCATION**



OLAP Queries

- Influenced by SQL and by spreadsheets.
- A common operation is to aggregate a measure over one or more dimensions.
  - Find total sales.
  - Find total sales for each city, or for each state.
  - Find top five products ranked by total sales.
- Roll-up: Aggregating at different levels of a dimension hierarchy.
  - E.g., Given total sales by city, we can roll-up to get sales by state.

OLAP Queries

- Drill-down: The inverse of roll-up.
  - E.g., Given total sales by state, can drill-down to get total sales by city.
  - E.g., Can also drill-down on different dimension to get total sales by product for each state.
- Pivoting: Aggregation on selected dimensions.
  - E.g., Pivoting on Location and Time yields this **cross-tabulation**:
- Slicing and Dicing: Equality and range selections on one or more dimensions.

	WI	CA	Total
1995	63	81	144
1996	38	107	145
1997	75	35	110
Total	176	223	399

Comparison with SQL Queries

- The cross-tabulation obtained by pivoting can also be computed using a collection of SQLqueries:

```
SELECT SUM(S.sales)
FROM   Sales S, Times T, Locations L
WHERE  S.timeid=T.timeid AND S.locid=L.locid
GROUP BY T.year, L.state
```

```
SELECT SUM(S.sales)
FROM   Sales S, Times T
WHERE  S.timeid=T.timeid
GROUP BY T.year
```

```
SELECT SUM(S.sales)
FROM   Sales S, Location L
WHERE  S.locid=L.locid
GROUP BY L.state
```

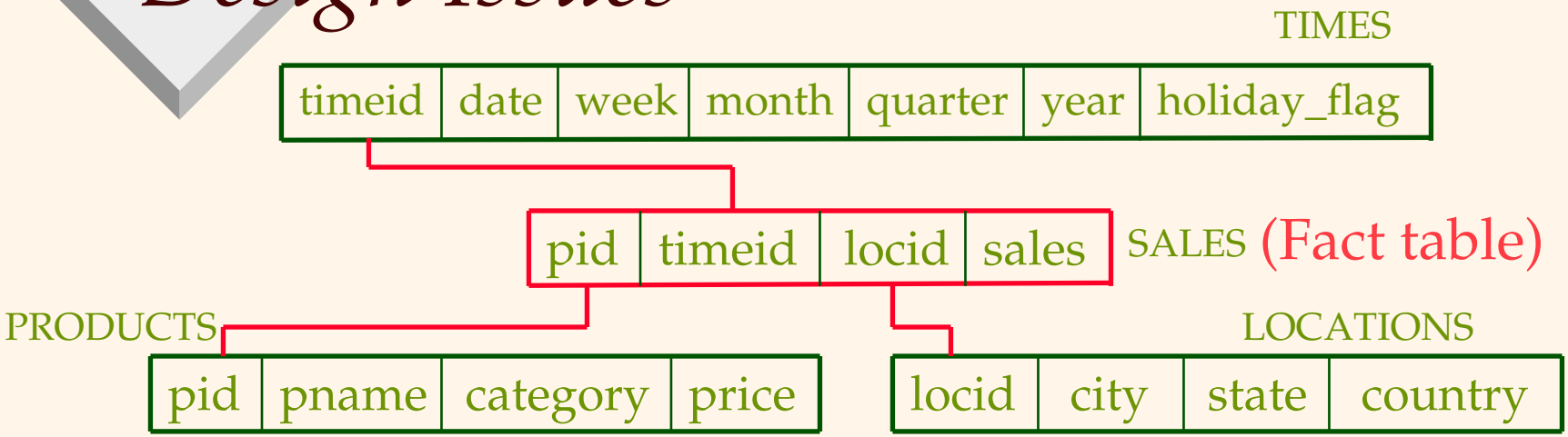
The CUBE Operator

- Generalizing the previous example, if there are k dimensions, we have 2^k possible SQL GROUP BY queries that can be generated through pivoting on a subset of dimensions.
- CUBE pid, locid, timeid BY SUM Sales
  - Equivalent to rolling up Sales on all eight subsets of the set {pid, locid, timeid}; each roll-up corresponds to an SQL query of the form:

```
SELECT SUM(S.sales)
FROM   Sales S
GROUP BY grouping-list
```

Lots of work on optimizing the CUBE operator!

Design Issues

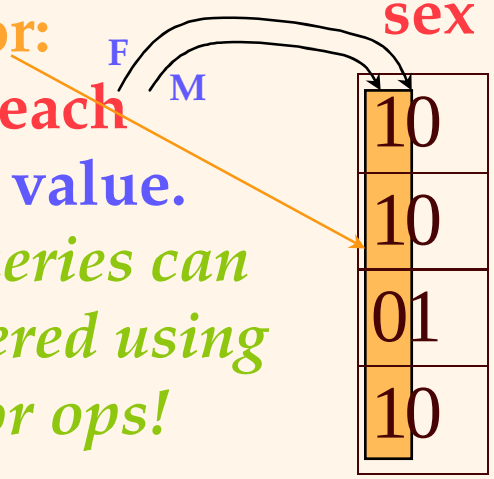


- Fact table in BCNF; dimension tables un-normalized.
  - Dimension tables are small; updates/inserts/deletes are rare. So, anomalies less important than query performance.
- This kind of schema is very common in OLAP applications, and is called a star schema; computing the join of all these relations is called a star join.

Implementation Issues

- New indexing techniques: Bitmap indexes, Join indexes, array representations, compression, precomputation of aggregations, etc.
- E.g., Bitmap index:

Bit-vector:
1 bit for each possible value.
Many queries can be answered using bit-vector ops!



sex

10
10
01
10

custid name sex rating

112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
112	Woo	M	4

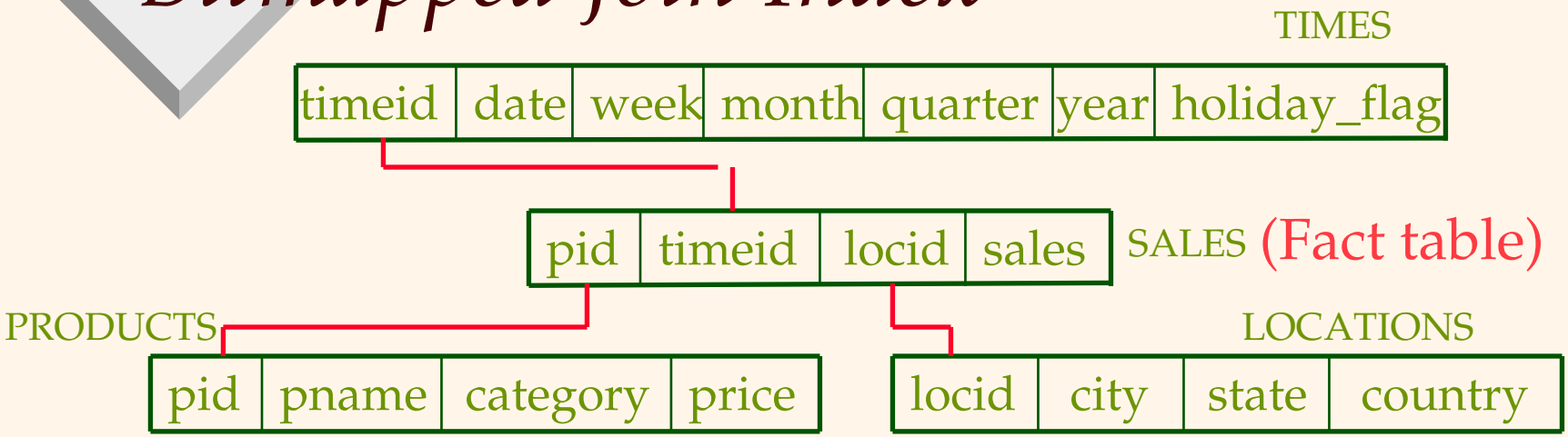
rating

00100
00001
00001
00010

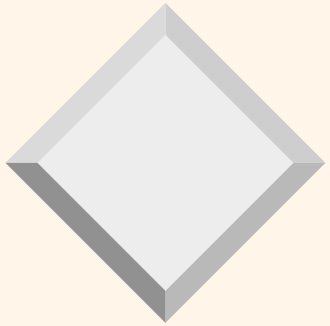
Join Indexes

- Consider the join of Sales, Products, Times, and Locations, possibly with additional selection conditions (e.g., country="USA").
  - A join index can be constructed to speed up such joins. The index contains [s,p,t,l] if there are tuples (with sid) s in Sales, p in Products, t in Times and l in Locations that satisfy the join (and selection) conditions.
- Problem: Number of join indexes can grow rapidly.
  - A variation addresses this problem: For each column with an additional selection (e.g., country), build an index with [c,s] in it if a dimension table tuple with value c in the selection column joins with a Sales tuple with sid s; if indexes are bitmaps, called bitmapped join index.

Bitmapped Join Index



- Consider a query with conditions price=10 and country="USA". Suppose tuple (with sid) s in Sales joins with a tuple p with price=10 and a tuple l with country ="USA". There are two join indexes; one containing [10,s] and the other [USA,s].
- Intersecting these indexes tells us which tuples in Sales are in the join and satisfy the given selection.



Views

Views and Decision Support

- OLAP queries are typically aggregate queries.
  - Pre-computation is essential for interactive response times.
  - The CUBE is in fact a collection of aggregate queries, and pre-computation is especially important: lots of work on what is best to pre-compute given a limited amount of space to store pre-computed 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 (Pre-computation)

- Suppose we pre-compute 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 pre-computed 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

- Algorithm to maintain a materialized view?
- What views should we materialize, and what indexes should we build on the pre-computed results?
- Given a query and a set of materialized views (possibly with some indexes), can we use the materialized views to answer the query?

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

- 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

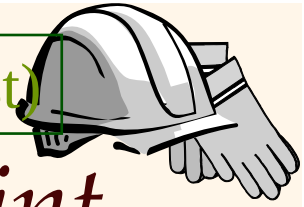
view(sno) :- r1(sno, pno), r2(pno, cost)

- Main twist: The views are in the data warehouse, and the source tables are somewhere else (operational DBMS, legacy sources, …).

Problem: New source updates between Steps 1 and 3!

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.

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.

Summary

- Decision support is an emerging, rapidly growing subarea of databases.
- Involves the creation of large, consolidated data repositories called data warehouses.
- Warehouses exploited using sophisticated analysis techniques: complex SQL queries and OLAP "multidimensional" queries (influenced by both SQL and spreadsheets).
- New techniques for database design, indexing, view maintenance, and interactive querying need to be supported.