Warehousing

- The most common form of information integration: copy sources into a single DB and try to keep it up-to-date.
- Usual method: periodic reconstruction of the warehouse, perhaps overnight.

OLTP Versus OLAP

- Most database operations are of a type called *on-line transaction processing* (*OLTP*).
  - Short, simple queries and frequent updates involving one or a small number of tuples.
  - Examples: answering queries from a Web interface, recording sales at cash-registers, selling airline tickets.

- Of increasing importance are operations of the *on-line analytic processing* (*OLAP*) type.
  - Few, but very complex and time-consuming queries (can run for hours).
  - Updates are infrequent, and/or the answer to the query is not dependent on having an absolutely up-to-date database.
  - Example: Amazon analyzes purchases by all its customers to come up with an individual screen with products of likely interest to the customer.
  - Example: Analysts at Wal-Mart look for items with increasing sales at stores in some region.
- Common architecture: Local databases, say one per branch store, handle OLTP, while a warehouse integrating information from all branches handles OLAP.
- The most complex OLAP queries are often referred to as *data mining*.

Star Schemas

Commonly, the data at a warehouse is of two types:

1. *Fact Data*: Very large, accumulation of facts such as sales.

    - Often "insert-only"; once there, a tuple remains.

2. *Dimension Data*: Smaller, generally static, information about the entities involved in the facts.

Example

Suppose we wanted to record every sale of beer at all bars: the bar, the beer, the drinker who bought the beer, the day and time, the price charged.

- Fact data is in a relation with schema:

```
Sales(bar, beer, drinker, day, time, price)
```

- Dimension data could include a relation for bars, e.g. `Bars(name, addr)`, one for beers, and one for drinkers.

  ```
  Bars(bar, addr, lic)
  Beers(beer, manf)
  Drinkers(drinker, addr, phone)
  ```

Two Approaches to Modeling Warehouses

1. *ROLAP* (Relational OLAP): relational database system tuned for star schemas, e.g. using special index structures such as:

   - "Bitmap indexes" (for each key of a dimension table, e.g., bar name, a bit-vector telling which tuples of the fact table have that value).

   - *Materialized views* = answers to general queries from which more specific queries can be answered with less work than if we had to work from the raw data.

2. *MOLAP* (Multidimensional OLAP): A specialized model based on a "cube" view of data.

ROLAP

Typical queries begin with a complete "star join," for example:

```
SELECT *
FROM Sales, Bars, Beers, Drinkers
WHERE Sales.bar = Bars.bar AND
    Sales.beer = Beers.beer AND
    Sales.drinker = Drinkers.drinker;
```

- Typical OLAP query will:
  1. Do all or part of the star join.
  2. Filter interesting tuples based on fact and/or dimension data.
  3. Group by one or more dimensions.
  4. Aggregate the result.
- Example: "For each bar in Palo Alto, find the total sale of each beer manufactured by Anheuser-Busch."

Performance Issues

- If the fact table is large, queries will take much too long.
- Materialized views can help.

Example

For the question about bars in Palo Alto and beers by Anheuser-Busch, we would be aided by the materialized view:

```
  CREATE VIEW BABMS(bar, addr, beer, manf,
sales) AS
      SELECT bar, addr, beer, manf,
          SUM(price) AS sales
      FROM Sales NATURAL JOIN
          Bars NATURAL JOIN Beers
      GROUP BY bar, addr, beer, manf;
```

MOLAP

Based on "data cube": keys of dimension tables form axes of the cube.

- Example: for our running example, we might have four dimensions: bar, beer, drinker, and time.
- Dependent attributes (price of the sale in our example) appear at the points of the cube.
- But the cube also includes aggregations (sums, typically) along the margins.
  - Example: in our 4-dimensional cube, we would have the sum over each bar, each beer, each drinker, and each time instant (perhaps group by day).
  - We would also have aggregations by all subsets of the dimensions, e.g., by each bar and beer, or by each beer, drinker, and day.

Slicing and Dicing

- *Slice* = select a value along one dimension, e.g., a particular bar.
- *Dice* = the same thing along another dimension, e.g., a particular beer.

Drill-Down and Roll-Up

- *Drill-down* = "de-aggregate" = break an aggregate into its constituents.
  - Example: having determined that Joe's Bar in Palo Alto is selling very few Anheuser-Busch beers, break down his sales by the particular beer.
- *Roll-up* = aggregate along one dimension.
  - Example: given a table of how much Budweiser each drinker consumes at each bar, roll it up into a table of amount consumed by each drinker.

Performance

As with ROLAP, materialized views can help.

- Data-cubes invite materialized views that are aggregations in one or more dimensions.
- Dimensions need not be aggregated completely. Rather, grouping by attributes of the dimension table is possible.
  - Example: a materialized view might aggregate by drinker completely, by beer not at all, by time according to the day, and by bar only according to the city of the bar.
  - Example: time is a really interesting dimension, since there are natural groupings, such as weeks and months, that are not commensurate.

Data Mining

Large-scale queries designed to extract patterns from data.

- Big example: "association-rules" or "frequent itemsets."

Market-Basket Data

An important source of data for association rules is *market baskets*.

- As a customer passes through the checkout, we learn what items they buy together, e.g., hamburger and ketchup.
- Gives us data with schema `Baskets(bid, item)`.
- Marketers would like to know what items people buy together.
  - ✦ Example: if people tend to buy hamburger and ketchup together, put them near each other, with potato chips between.
  - ✦ Example: run a sale on hamburger and raise the price of ketchup.

Simplest Problem: Find the Frequent Pairs of Items

Given a *support threshold* s, we could ask:

- Find the pairs of items that appear together in at least s baskets.

```
SELECT b1.item, b2.item
FROM Baskets b1, Baskets b2
WHERE b1.bid = b2.bid AND
    b1.item < b2.item
GROUP BY b1.item, b2.item
HAVING COUNT(*) >= s;
```

A-Priori Trick

- Above query is prohibitively expensive for large data.
- *A-priori algorithm* uses the fact that a pair (i, j) cannot have support s unless i and j both have support s by themselves.
- More efficient implementation uses an intermediate relation *Baskets1*.

```
INSERT INTO Baskets1(bid, item)
    SELECT * FROM Baskets
    WHERE item IN (
        SELECT item
        FROM Baskets
        GROUP BY item
        HAVING COUNT(*) >= s
    );
```

- Then run the query for pairs on *Baskets1* instead of `Baskets`.