Lecture 9: Decision support, OLAP

Rasmus Pagh

Reading: RG25, [WOS04, sec. 1+2], [DSTW03, sec. 1+2+3.0]

Today’s lecture

- Multi-dimensional data and OLAP.
- Bitmap indexing.
- Materialized views: Use and maintenance.

- Exercises.
- ”On producing join results early”.
- Some other uses of sampling.

Background

- Data of an organization is often a gold mine!
  - Data can identify useful patterns that can be used to form a proper business strategy.
- Two main approaches:
  - Data mining: Automated ”mining” of patterns.
  - OLAP: Interactive investigation of data. (This lecture.)
- OLAP = On-Line Analytic Processing
  OLTP = On-Line Transaction Processing

Multi-dimensional data model

- Tuples with d attributes can be viewed as points in a d-dimensional ”cube”.
- **Example**: (DBT,’John Doe’,13) can be viewed as the point with coordinates DBT, ’John Doe’, and 13 along the dimensions Course, Name, and Grade.
- Natural view of data, specifically for *aggregation* queries. Dimensions often have several *granularities*.
- **Example**: We may want to compute the average grade of all SDT courses.

Sample OLAP query (in SQL)

```
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
```

Get fine-grained aggregate data

```
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.timeid=L.timeid
GROUP BY L.state
```

Get dimension ”coordinates” +aggregates

SQL:1999 CUBE operator

- Similar to GROUP BY, but includes groupings according to *all subsets* of the given attributes.
- **Example**:

```
SELECT year,state,SUM(sales)
FROM Salesview
GROUP BY CUBE (year,state)
```

returns tuples of the form:

- `(1999,Iowa,29438)`
- `(1999,NULL,314974)`
- `(NULL,Iowa,213891)`
- `(NULL,NULL,3217919)`

Typical OLAP query

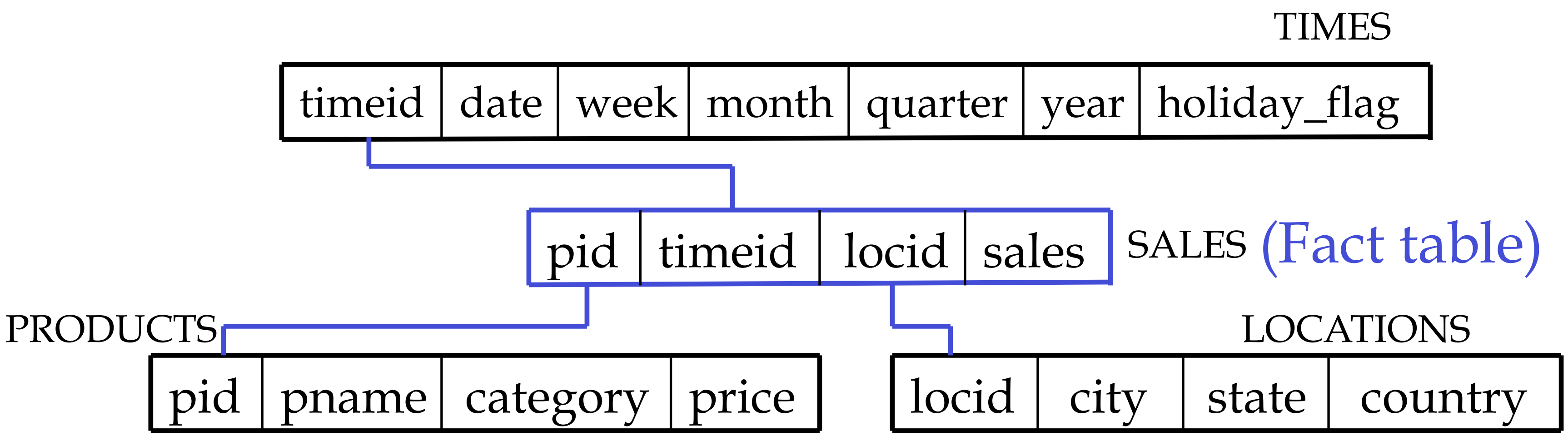
- Selection on one or more dimensions (e.g. select only sales to a certain customer group).
- Grouping by one or more dimensions (e.g., group sales by quarter).
- Aggregation over each group (e.g. total sales revenue).

Indexing aggregations

- Simple OLAP queries involve a range selection (a>10) and an aggregation such as SUM(x) or COUNT.
- B-trees can be extended to compute such aggregates *without* inspecting all tuples in the range.
  - Version 1 (dynamic): Along with each pointer to a substructure, store the corresponding aggregate.
  - Version 2 (append-only): Augment each tuple with an aggregate over all tuples with smaller a-value.

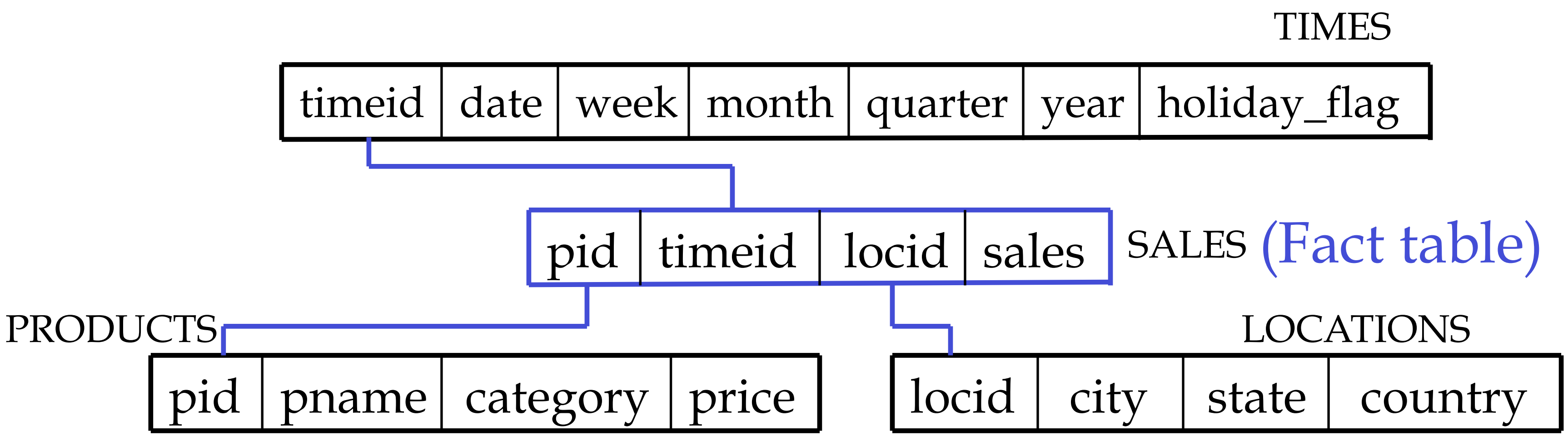
Relational set-up of OLAP

- **Star schema:** A ”fact table”, plus a number of ”dimension tables” whose keys are foreign keys of the fact table.
- Example from RG’s slides:



Problem session

- How would you efficiently answer the query: *"Find the total sales of raincoats for each location last year."*
- You may assume suitable indexes.



Queries on star schemas

Special considerations, 1:

- Dimension tables are often small, especially when considering only the tuples matching a select operation.
  - Common assumption that all dimension tables fit in RAM.
  - Complete join can then be made in a single scan of the fact table, using pipelining.

Queries on star schemas

Special considerations, 2:

- Number of relations can be large - join size estimation is difficult.
- Two interesting cases:
  - If the join with some dimension table results in much fewer tuples than in the fact table, we may perform this join first using an index (if available).
  - Otherwise we may use the pipelined join that scans the whole fact table.

Queries on star schemas

Special considerations, 3:

- Often data can be assumed to be *static* (a snapshot of the operational data). This means that we have the option to *precompute*. Will be used in two ways:
  - Indexing
  - Pre-aggregation

Indexing low cardinality attributes

- Suppose there are only 4 different locations in our previous example.
- Then we may represent the locations of N fact table tuples using only 2N bits.
- However, an (unclustered) index on location seems to require at least N log N bits.
- Can we get by reading less data?

Basic bitmap index

- For each possible value (of ”location”) and each tuple, store a bit that is 1 iff the tuple contains the given value.
- Store these bits **ordered by column**.
- In the context of this example, the bitmap index is a **join index** that can be used when computing joins.
- How?

L1?	L2?	L3?	L4?	RID
0	0	1	0	1
1	0	0	0	2
1	0	0	0	3
0	1	0	0	4
0	0	0	1	5

Gain of bitmap indexes

- How much can **at most** be gained by using bitmap indexes to do a star join (with a selection on each dimension table), compared to using B-tree indexes?
  - Theoretically 1 bit/tuple vs log N bits/tuple.
  - Typically 1 bit/tuple vs 32 bits/tuple.
- Main case where there is no gain:
  - A single dimension is very selective.
  - (Usually only the case for high cardinality attributes.)

Compressed bitmap indexes

- If there are many possible values for an attribute (it has ”high cardinality”), basic bitmap indexing is not space efficient (nor time efficient).
- **Observation:** A column will have few 1s, on average. It should be possible to ”compress” long sequences of 0s.
- **How to compress?** Usual compression algorithms consume too much computation time. Need simpler approach.

Word-aligned hybrid (WAH) coding

[WOS04]

- <u>In a nutshell</u>:
  - Split the bitmap B into pieces of 31 bits.
  - A 32-bit word in the encoding contains one of the following, depending on the value of its first bit:
    - A number specifying the length of an interval of bits where all bits of B are zeros.
    - A piece of B (31 bits).
  - The conjunction (”AND”) or disjunction (”OR”) of two compressed bitmaps can be computed in linear time (O(1) ops/word).

WAH analysis

- Let N be the number of rows of the indexed relation, and c the cardinality of the indexed attribute.
- At most N WAH words will encode a piece of the bitmap.
- Reasonable assumption:
  - All (or most) gaps between consecutive 1s can be encoded using 31 bits.
  - Thus, at most N+c gaps.
- Total space usage: 2N+c words.
- Compares favorably to B-trees.

WAH experiments

- Implemented in the ”FastBit” system.
- Compared against compressed bitmap index of Oracle: Considerable speedup due to simpler program.
- Compared against not using a bitmap index (MySQL): Speedup several orders of magnitude.

De-confuser

- The book talks about join indexes in two distinct senses:
  - Precomputing the join result (”basic join index”).
  - Precomputing *projections* of the join result onto tuples of pairs of relations.
- In both cases, compact row IDs (rids) are used to represent the tuples forming each result tuple.
- It is for the latter that we can use bitmap indexing with advantage – use to perform rid intersection.

Bitmap join index in Oracle

- Similar to defining a join!
  - A join index is an index on a join result.
- Example:
  A bitmap join index that allow us to find the sales in a given state:

```
CREATE BITMAP INDEX ON
   sales(locations.state)
FROM sales, locations
WHERE sales.locid=locations.locid
```

- Can index even index multiple attributes in a multi-way join.

Next: Materialized views

- An SQL *view* is similar to a macro.
- Example:

```
CREATE VIEW MyView AS
SELECT *
FROM    Sales S, Times T, Locations L
WHERE   S.timeid=T.timeid AND
        S.locid=L.locid
```

- A query on MyView is transformed into a query that performs the join of Sales, Times, and Locations.
- In contrast, a **materialized view** *physically* stores the query result. Additionally: can be indexed!

Using a materialized view

1. DBA grants permission:
   ```
   GRANT CREATE MATERIALIZED VIEW TO hr
   ```
2. Materialized view is created:
   ```
   CREATE MATERIALIZED VIEW SalaryByLocation AS
   SELECT location_id, country_id, SUM(salary) AS s
   FROM Employees NATURAL JOIN Departments
     NATURAL JOIN Locations
   GROUP BY location_id, country_id
   ```
3. Materialized view is used:
   ```
   SELECT country_id, SUM(salary) AS salary
   FROM SalaryByLocation
   GROUP BY country_id
   ```

   Factor 10 faster than direct query on Oracle XEs example DB.

Automatically using mat. views

- Suppose a user does not know about the materialized view and writes directly

```
SELECT location_id, country_id, SUM(salary) AS s
FROM Employees NATURAL JOIN Departments
    NATURAL JOIN Locations
GROUP BY country_id
```

- A smart DBMS will realize that this can be rewritten to a query on the materialized view. (Disabled in Oracle XE...)
- Rewrite capability is a key technique in relational OLAP systems.

”Refreshing” a materialized view

- Any change to the underlying tables may give rise to a change in the materialized view. There are at least three options:
  - Update for every change (”`ON COMMIT`”)
  - Update only on request (”`ON DEMAND`”)
  - Update when the view is accessed (”lazy”)
- RG describes a way of refreshing where recomputing the defining query is often not necessary (”`FAST`”). `((board))`

Exercises

We look at two exercises from RG:

- 25.8.
- 25.10, 2) and 4).

<u>Schema</u>:

`Locations(locid,city,state,country)`

`Products(pid,pname,category,price)`

`Sales(pid,timeid,locid,sales)`

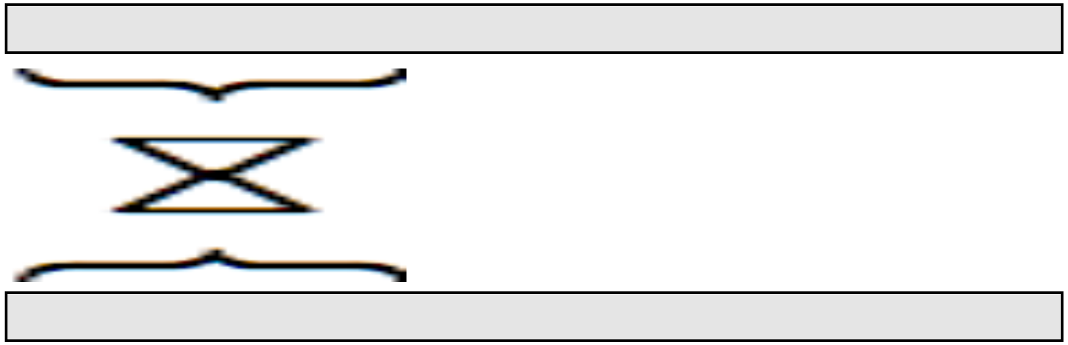
Next: Early join results

[DSTW03]

- Even with suitable materialized views in place, we may need to do some joins.
- **OLAP goal**: Get at least *part* of the result early.
- We consider the simplest case: Joining two relations R and S
- <u>Basic idea</u>: First report $R' \bowtie S'$, where R′ and S′ are samples (subsets) of R and S.
- Assumption: We get a ”random-like” sample by picking the first k tuples.

First approach

- Join samples of size 1,2,4,8,...,N. (TPMMS)
- Report join tuples as soon as they are seen - remember to filter away tuples that were previously reported.



- **Problem session**:
  - How many tuples can we expect after joining the samples of size k?
  - What is the total I/O cost of this?

Progressive mergesort join

- Same idea (increasing sample sizes). [DSTW03]
- <u>New:</u>
  - Re-use as much work as possible
  - Avoid explicit filtering

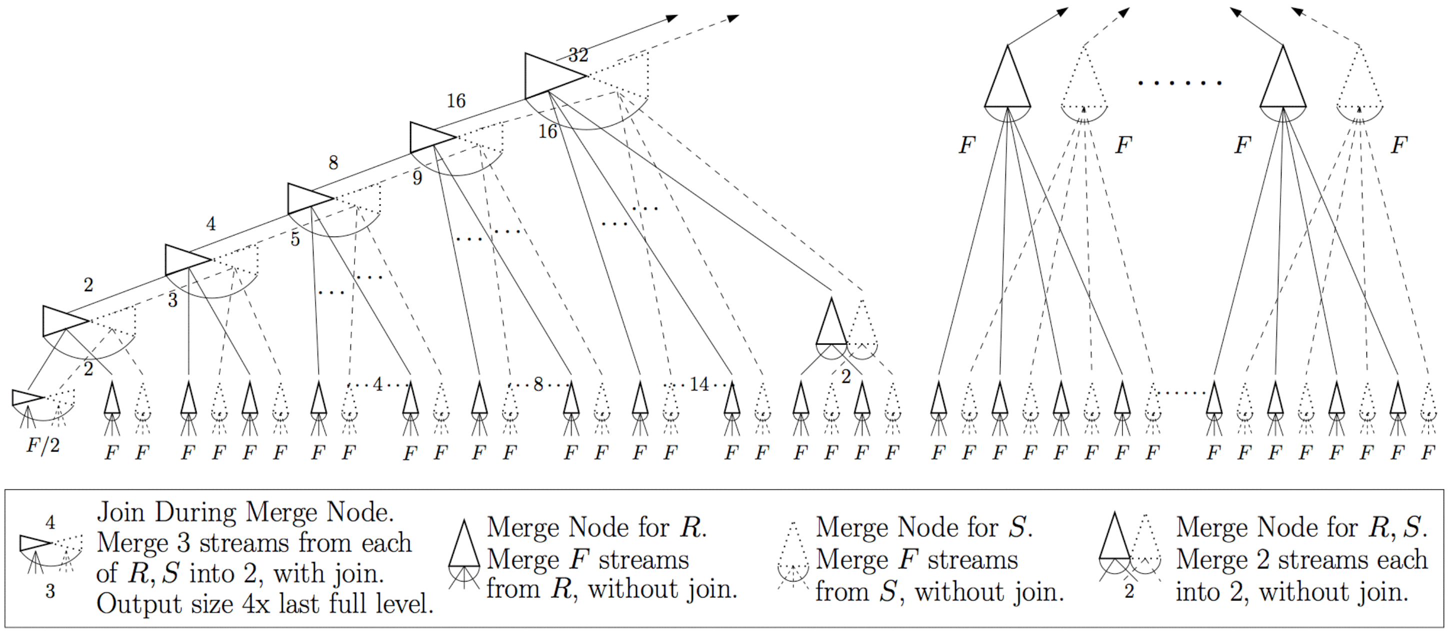


Figure 1: One full level of Progressive Mergesort Join to the next ($F = 64$).

On-line aggregation

- For aggregates like sums and averages, the result on a sample can be used to estimate the result on all data.
- Same principle as used in opinion polls!
- Can give statistical guarantees on an answer, e.g. ”Answer is 3200±180 with 95% probability”.
- The longer the query runs, the smaller the uncertainty gets.
- Possibly ok to terminate before precise answer is known.

”Top K” queries

- Suppose, in a given query result, we are only interested in the K tuples with the largest values of attribute A.
- In Oracle:

```
SELECT *
FROM (SELECT ... ORDER BY A DESC)
WHERE rownum<=K.
```

- If the DBMS knew the ”cutoff” value for A, we could add this as a condition, possibly reducing dramatically the amount of data to be considered.
- Sampling approach: Estimate (conservatively) the right cutoff based on the sample.