CompSci 590.6
Understanding Data: Theory and Applications

Lecture 5
Index for ROLAP Cube
and
An Algorithm for MOLAP Cube

Instructor: Sudeepa Roy
Email: sudeepa@cs.duke.edu
Today’s Paper(s)

Paper 1
**Index Selection for OLAP**
Gupta-Harinarayan-Rajaraman-Ullman
ICDE 1997

Paper 2
**An Array-Based Algorithm for Simultaneous Multidimensional Aggregates**
Zhao-Deshpande-Naughton
SIGMOD 1997
Paper#1

• Recall Lecture 3 (selective materialization)
• Materialized views for cubes
  – Greedy algorithm
  – By a subset of the authors
• This paper
  – Data cubes with indexes on the materialized views
Running Example

• From TPC-D (again)
• part (p), supplier (s), customer (c), sales
  – The business buys a part from a supplier and sells it to a customer
• p, s, c: Dimensions or attributes
• sub-cube on 1 or 2 out of 3 dimensions
Queries Considered

• Each dimension \((p, s, c)\)
  – as a selection attribute (in \texttt{WHERE}, \(\sigma\)),
  – or as an output attribute (in \texttt{GROUP-BY}, \(\Upsilon\))

• Example
  – Find the “sales” to each customer of a given “part” = ‘widget’ bought from a given “supplier” = ‘widgets-r-us’
  – Denoted by \(Q = \Upsilon_c \sigma_{ps}\)
  – The order of dimensions in \(\Upsilon, \sigma\) is assumed to be non-important
  – Any subcube that has all the output and selection attributes can answer such queries
Indexes

- **B-tree indexes or variants**
- **For subcube** $ps$, we can construct
  - $I_{ps}$: search key is a concatenation of $p$ and $s$
  - $I_{sp}$: search key is a concatenation of $s$ and $p$
- **Order matters**
  - Given a value of $p$, $I_{ps}$ can efficiently retrieve those rows in subcube $ps$ that have this value
  - Cannot do so “efficiently” given a value of $s$
- $I_{X_1, X_2, ..., X_k}$ can efficiently answer a query that has some prefix of $X_1, X_2, ..., X_k$ in its $\sigma$
Cost Model

- Cost of answering a query = #rows processed
- Consider $Q_1 = \gamma_p \sigma_s$
- How can we answer $Q_1$?
- using ps
  - = 0.8M
- using psc
  - = 6M
- using ps and index $I_{sp}$
  - The avg. no. of rows per s value = $|ps|/|s| = 0.8/0.01 = 80$
What to materialize?

• Which subcube and indexes?
• Assume all queries are equi-probable
  – Queries associated with ps are
    – $\gamma_p \sigma_s$
    – $\gamma_{ps} \sigma_{\{}$
    – $\gamma_{\{} \sigma_{ps}$
    – $\gamma_s \sigma_p$
• Cannot materialize everything
• Suppose
  – all subcubes and indexes require 80M rows
  – You can store only 25M rows
Simple Two-step Approach

- Divide available space for cubes and indexes
  - say equally
- Use greedy algo to select views (Lecture 3)
  - say psc, ps, sc, c, s, p, none
- Then select indexes
  - say $I_{csp}, I_{pcs}$
- 1.18M rows per query on average
1-Greedy Approach

- One step
- Greedily choose
  - subcube
  - or the index on a subcube (if the subcube is already chosen)
- \( \text{psc} \rightarrow \text{l}_{\text{csp}} \rightarrow \text{ps} \rightarrow \text{l}_{\text{pcs}} \rightarrow \text{l}_{\text{spc}} \rightarrow \text{c} \rightarrow \text{s} \rightarrow \text{p} \rightarrow \text{none} \)
- average query cost = 0.74M rows
  - 40% savings
  - \( \frac{3}{4} \) to index \( \frac{1}{4} \) to cube
  - hard to decide a priori
- But still can be improved
Slice Queries

- $\gamma_c \sigma_{p='widget'} R$
  - slice through the subcube $pc$
- $\gamma_{G1,...Gk} \sigma_{S1,...,Sl}$ associated with the subcube $G1..GkS1...Sl$
  - smallest cube that can answer this query
- An $r$-dimensional subcube has $2^r$ slice queries
  - each dimension can go to either $\gamma$ or $\sigma$
- every query is a slice query
- An $n$-dimensional cube has $\binom{n}{r}$ $r$-dimensional subcubes
- Total slice queries for a data cube = $3^n$
  - summing over all $r = 0$ to $n$
How many indexes per cube?

• e.g. 4 with subcube ps
  – \( I_p(ps), I_s(ps), I_{ps}(ps), I_{sp}(ps) \)
• order matters in an index
• #Index for a view with m attr
  – \[ = \sum_{r=0}^{m} \binom{m}{r} r! \rightarrow (e-1)m! \]
• Total #indexes for a n-dimensional cube
  – about 3n!
• Total #fat indexes (same attr in view and index)
  – about 2n!
  – where index attributes are permutations of cube attributes
Materializing Views with Indexes

• **Input**
  – a set of views
  – each view has a set of indexes
  – a set of queries to be supported
  – cost $c(Q, V, J)$ of answering query $Q$ using view $V$ and index $J$
    • estimated
  – amount of space available $S$

• **Goal:**
  – select a set of views and indexes that will minimize the total cost to answer the queries not exceeding the space $S$

• **NP-hard**
  – Lecture 3
  – even if no index and unit cost
r-Greedy

• Use greedy algorithms
• r-Greedy
  – Generalization of 1-greedy
  – Instead of choosing at most one index/view with max benefit per unit space...
  – ...choose “at most r views” or index (for chosen views) every step with max benefit per unit space as a set
  – Runtime: $O(km^r)$
    • m: Number of structures (views/index) in query graph
    • k: Number of structures selected
    • Max size (assuming unit size): $S + r - 1$ units
  – Only practical for $r \leq 4$
An Array-Based Algorithm for Simultaneous Multidimensional Aggregates
Zhao-Deshpande-Naughton
SIGMOD’97

Acknowledgement:
The following slides have been prepared using the slides by Manuel Calimlim, in CS632-Advanced Database Systems, Spring 2000, Cornell University
ROLAP vs MOLAP cube

• **ROLAP = Relational OLAP**
  - All algorithms so far were for ROLAP
  - A cell in the space is a tuple
  - e.g. (shoes, WestTown, 3-July-96, $34)

• **MOLAP = Multi-dimensional OLAP**
  - Data in sparse arrays
  - just stores the data value $34
  - The position in the array encodes (shoes, WestTown, 3-July-96)

• This paper: MOLAP algorithm for cube

• Similar example
  - Dimensions = product, store, time
  - Measure = sales
ROLAP Cube

In ROLAP systems, 3 main ideas for efficiently computing the CUBE

1. Group related tuples together (using sorting or hashing)
2. Use grouping performed on sub-aggregates to speed computation
3. Compute an aggregate from another aggregate rather than the base table
MOLAP cube

• No “bring together related values”
  – Data values are stored in their own fixed location
  – Rather, visit those values in the right order so that the computation is efficient
• Simultaneously compute spatially-delimited partial aggregates
  – so that a cell is not visited for each sub-aggregate
• Store arrays efficiently on disk
  – “chunk” them into pieces
  – do compression to avoid wasting space on cells with no data
Multidimensional Array Storage

Data is stored in large, sparse arrays, which leads to certain problems:

1. The array may be too big for memory
2. Many of the cells may be empty and the array will be too sparse
Chunking Arrays

Sarawagi-Stonebraker, ICDE’94: Efficient Organization of Large Multidimensional Arrays

Why chunk?
• A simple row major or column major layout (partitioning by dimension) will favor certain dimensions over others
• e.g. assume (store, day) – row major
  – to access a day may need multiple block read from disk

What is chunking?
• Divide an n-dimensional array into smaller n-dimensional chunks and store each chunk as one object on disk
Chunks

\[
\begin{array}{ccc}
  & C_B & \\
C_A & & C_A \\
  & C_B & \\
\end{array}
\]

Dimension A

Dimension B
Array Compression

• No compression for dense arrays
  – more than 40% filled with data
  – fixed length chunks
  – assign a null value to invalid cells
  – Still compression since none of the dimension values are stored

• Compression for sparse array
  – less than 40% filled, most cells invalid
  – use “Chunk-offset compression”
  – for each valid entry, store (offsetInChunk, data) where offsetInChunk is the offset from the start of the chunk
  – e.g. for 3-D array, convert address (i, j, k) into an offset
  – chunks will be of variable length – needs metadata for each chunk and data file
Naïve Array Cubing Algorithm

- Multiple passes
- Compute each group-by in a separate pass with minimum memory
- No overlap of computation and minimizing I/O cost
- Similar to ROLAP, each aggregation is computed from its parent in the lattice.
- Each chunk is aggregated completely and then written to disk before moving on to the next chunk.
Naïve Array Cubing Algorithm

- Compute AB
  - sweep through the C-plane if no chunks
- Suppose ABC is stored in a no. of chunks
  - they are numbered in dimension order (ABC)
  - need to sweep chunk by chunk
  - To compute group by for $a_0 b_0$, need to sum over 4 chunks for $c_0, c_1, c_2, c_3$
Naïve Array Cubing Algorithm

- Multiple aggregates in cube
- Compute A from AB or AC, not from ABC
- Embed a “minimum spanning tree” to the lattice – min size parent
Problems with Naïve approach

- Each sub aggregate is calculated independently
- E.g. this algorithm will compute AB from ABC, then rescan ABC to calculate AC, then rescan ABC to calculate BC
- We need a method to simultaneously compute all children of a parent in a single pass over the parent
Single-Pass Multi-Way Array Cubing Algorithm

• The order of scanning is vitally important in determining how much memory is needed to compute the aggregates.

• A dimension order \( O = (D_{j1}, D_{j2}, ..., D_{jn}) \) defines the order in which dimensions are scanned
  
  – Logical order, independent of physical layout on disk

• \( |D_i| \) = size of dimension \( i \)

• \( |C_i| \) = size of the chunk for dimension \( i \)

• \( |C_i| << |D_i| \) in general
Order determines memory requirement

- \(|C_i| = 4, \ |D_i| = 16\) for all \(i\)
- Dimension order (ABC)
  - For BC, we need 4 chunks
    - 1-4 computes one chunk \(b_0c_0\) of BC
    - give the memory to \(b_1c_0\)
  - For AC, we need 16 chunks
    - allocate space to 4 chunks \(a_0c_0, a_1c_0, a_2c_0, a_3c_0\)
    - after reading 16 chunks (a plane) give the memory to \(a_0c_1, a_1c_1, a_2c_1, a_3c_1\)
  - For AB, we need all 64 chunks
    - allocate memory to all 16 chunks of AB as we read chunks of the cube
    - after aggregation is complete, output those chunks in AB order
Concrete Example

- For BC group-bys, we need 1 chunk (4x4)
- For AC, we need 4 chunks (16x4)
- For AB, we need to keep track of whole slice of the AB plane, so (16x16)
Minimum Memory Spanning Trees (MMST)

MMST for a given dimension order

\[ p = \text{size of the largest common prefix between the current group-by (size n-1) and its parent} \]

\[ \Pi_{i=1 \to p} |D_i| \times \Pi_{i=p+1 \to n-1} |C_i| \]

\[ D_i = 16, C_i = 4 \]

order = (A B C)

Q. What is the optimal dimension order in general?
Effects of Dimension Order

\[ |D_A| = 10, \quad |D_B| = 100, \quad |D_C| = 1000, \quad |D_D| = 10000 \]

\[ |C_A| = |C_B| = |C_C| = |C_D| = 10 \]
Effects of Dimension Order

• The early elements in O (particularly the first one) appear in the most prefixes
  – contribute their dimension sizes to the memory requirements

• The last element in O can never appear in any prefix
  – The total memory requirement for computing the CUBE is independent of the size of the last dimension
Optimal Dimension Order

• Sort them on increasing dimension size
  – order is \((D_{i1}, D_{i2}, ..., D_{in})\)
  – where \(|D_{i1}| \leq |D_{i2}| \leq |D_{i3}| \leq ... \leq |D_{in}|\)

• The total memory requirement will be minimized
  – a formal proof in the paper

• The total memory requirement is Independent of the size of the largest dimension
  – huge benefit if the largest dimension is big

• Extension to multiple passes
  – limited memory, suppose required memory is not
  – Right to left scan – first compute BC so that it is not divided into multiple passes
Results

Figure 5: Naive vs. Multi-way Array Alg.
ROLAP vs. MOLAP

Response Time (Seconds)

Size of the Fourth Dimension

ROLAP Alg

Multi-way Array Alg w/o Loading
ROLAP vs. MOLAP

- Memory constant, ROLAP does multiple passes
- Array dimension size not changing with density
- Largest dimension has no effect

Figure 8: ROLAP vs. Multi-way Array for Data Set 2
Figure 9: ROLAP vs. Multi-way Array for Data Set 3
MOLAP for ROLAP system

We can use the MOLAP algorithm with ROLAP systems:
1. Scan the table and load into an array.
2. Compute the CUBE on the array.
3. Convert results into tables

- Even with the additional cost of conversion between data structures, the MOLAP algorithm
  - runs faster than directly computing the CUBE on the ROLAP tables
  - scales much better

- the multidimensional-array can be used as a query evaluation data structure rather than a persistent storage structure.
Summary

• The multidimensional array of MOLAP should be chunked and compressed

• The Single-Pass Multi-Way Array method simultaneously updates all GROUP-Bys in the CUBE with a single pass over the data
  – assumes required memory is available
  – Multiple passes are needed otherwise

• By minimizing the overlap in prefixes and sorting dimensions in order of increasing size, we can build a MMST that gives a plan for computing the CUBE

• On MOLAP systems, the CUBE is calculated much faster than on ROLAP systems
  – can be used even for cube for ROLAP