**A Short Recap on Data Cubes**

Data Cube: A Lattice of Cuboids

- **Sales Data Cube:**
  - 0-D (apex) cuboid: all
  - 1-D cuboids: e.g., `time, item, location` or `time, item, supplier`
  - 2-D cuboids: e.g., `time, item, location, supplier`
  - 3-D cuboids: e.g., `time, item, location, supplier`
  - 4-D (base) cuboid: `time, item, location, supplier`

---

**Multidimensional Data**

- Sales volume as a function of product, month, and region
  - Dimensions: Product, Location, Time
  - Hierarchical summarization paths:
    - Industry Region Year
    - Category Country Quarter
    - Product City Month Week
    - Office Day

---

**Example of Star Schema**

- **Sales Fact Table**
  - `time_key`, `item_key`, `branch_key`, `location_key`
  - Measures:
    - `units_sold`, `dollars_sold`, `avg_sales`
  - **Item**
    - `item_key`, `item_name`, `brand`, `supplier_type`
  - **Branch**
    - `branch_key`, `branch_name`, `branch_type`
  - **Location**
    - `location_key`, `street`, `city`, `state_or_province`, `country`
A Sample Data Cube

Total sales of TV's in 1st quarter in USA

Date
1Qtr 2Qtr 3Qtr 4Qtr

Product
PC TV VCR

Country
U.S.A Canada Mexico

Total annual sales of TV in U.S.A.

Cuboids Corresponding to the Cube

0-D (apex) cuboid

1-D cuboids

2-D cuboids

3-D (base) cuboid

Typical OLAP Operations

Dice

Roll-up

Slice

Drill-down

Pivot

Data Cubes: Ancestor – Descendent relation

0-D (apex) cuboid

1-D cuboids

2-D cuboids

3-D (base) cuboid
Chapter 4: Data Cube Computation and Data Generalization

- Efficient Computation of Data Cubes
- Exploration and Discovery in Multidimensional Databases
- Attribute-Oriented Induction — An Alternative Data Generalization Method

Efficient Computation of Data Cubes

- Preliminary cube computation tricks (Agarwal et al.'96)
- Computing full/iceberg cubes: 3 methodologies
  - Top-Down: Multi-Way array aggregation (Zhao, Deshpande & Naughton, SIGMOD'97)
  - Bottom-Up:
    - Bottom-up computation: BUC (Beyer & Ramarkrishnan, SIGMOD'99)
    - H-cubing technique (Han, Pei, Dong & Wang: SIGMOD'01)
  - Integrating Top-Down and Bottom-Up:
    - Star-cubing algorithm (Xin, Han, Li & Wah: VLDB'03)
- High-dimensional OLAP: A Minimal Cubing Approach (Li, et al. VLDB'04)
- Computing alternative kinds of cubes:
  - Partial cube, closed cube, approximate cube, etc.

Iceberg Cube

Computing only the cuboid cells whose count or other aggregates satisfying the condition like

```
HAVING COUNT(*) >= minsup
```

Motivation

- Only a small portion of cube cells may be "above the water" in a sparse cube
- Only calculate "interesting" cells—data above certain threshold
- Avoid explosive growth of the cube
  - Suppose 100 dimensions, only 1 base cell. How many aggregate cells if count >= 1? What about count >= 2?

```
compute cube sales_iceberg as
  select month, city, customer_group, count(*)
from salesinfo
cube by month, city, customer_group
having count(*) >= minsup
```
Closed Cubes

Database of 100 dimensions has 2 base cells:
{(a_1, a_2, a_3, ..., a_{100}): 10, (a_1, a_2, b_3, ..., b_{100}): 10}

⇒ 2^{101} - 6 not so interesting aggregate cells:
{(a_1, a_2, a_3, ..., *) : 10, (a_1, a_2, *, a_4, ..., a_{100}): 10, ..., (a_1, a_2, a_3, *, ..., *) : 10}

The only 3 interesting aggregate cells would be:
{(a_1, a_2, a_3, ..., a_{100}): 10, (a_1, a_2, b_3, ..., b_{100}): 10, (a_1, a_2, *, ..., *): 20}

Closed Cubes

A cell c, is a closed cell, if there exists no cell d such that d is a specialization (descendant) of cell c (i.e., replacing a * in c with a non-* value), and d has the same measure value as c (i.e., d will have strictly smaller measure value than c).

A closed cube is a data cube consisting of only closed cells.

For example the previous three form a lattice of closed cells for a closed cube.

Closed Cubes

Closed cube lattice:

- (a_1, a_2, *, ..., *): 20
- (a_1, a_2, a_3, ..., a_{100}): 10
- (a_1, a_2, b_3, ..., b_{100}): 10

Preliminary Tricks (Agarwal et al. VLDB’96)

- Sorting, hashing, and grouping operations are applied to the dimension attributes in order to reorder and cluster related tuples
- Aggregates may be computed from previously computed aggregates, rather than from the base fact table
  - Smallest-child: computing a cuboid from the smallest, previously computed cuboid
  - Cache-results: caching results of a cuboid from which other cuboids are computed to reduce disk I/Os
  - Amortize-scans: computing as many as possible cuboids at the same time to amortize disk reads
  - Share-sorts: sharing sorting costs across multiple cuboids when a sort-based method is used
  - Share-partitions: sharing the partitioning cost across multiple cuboids when hash-based algorithms are used
Multi-Way Array Aggregation

- Array-based "bottom-up" algorithm
- Using multi-dimensional chunks
- No direct tuple comparisons
- Simultaneous aggregation on multiple dimensions
- Intermediate aggregate values are re-used for computing ancestor cuboids
- Cannot do Apriori pruning: No iceberg optimization

Multi-way Array Aggregation for Cube Computation (MOLAP)

- Partition arrays into chunks (a small subcube which fits in memory).
- Compressed sparse array addressing: (chunk_id, offset)
- Compute aggregates in “multiway” by visiting cube cells in the order which minimizes the # of times to visit each cell, and reduces memory access and storage cost.

What is the best traversing order to do multi-way aggregation?
Multi-Way Array Aggregation for Cube Computation (Cont.)

- Method: the planes should be sorted and computed according to their size in ascending order
  - Idea: keep the smallest plane in the main memory, fetch and compute only one chunk at a time for the largest plane
  - Limitation of the method: computing well only for a small number of dimensions
  - If there are a large number of dimensions, “top-down” computation and iceberg cube computation methods can be explored

Bottom-Up Computation (BUC)

- BUC (Beyer & Ramakrishnan, SIGMOD’99)
- Bottom-up cube computation (Note: top-down in our view!)
- Divides dimensions into partitions and facilitates iceberg pruning
  - If a partition does not satisfy min_sup, its descendants can be pruned
  - If minsup = 1 ⇒ compute full CUBE!
- No simultaneous aggregation

BUC: Partitioning

- Usually, entire data set can’t fit in main memory
- Sort distinct values, partition into blocks that fit
- Continue processing
- Optimizations
  - Partitioning
    - External Sorting, Hashing, Counting Sort
  - Ordering dimensions to encourage pruning
    - Cardinality, Skew, Correlation
  - Collapsing duplicates
    - Can’t do holistic aggregates anymore!

H-Cubing: Using H-Tree Structure

- Bottom-up computation
- Exploring an H-tree structure
- If the current computation of an H-tree cannot pass min_sup, do not proceed further (pruning)
- No simultaneous aggregation
H-tree: A Prefix Hyper-tree

H-Cubing: Computing Cells Involving Dimension City

Computing Cells Involving Month But No City

Computing Cells Involving Only Cust_grp

Check header table directly
Star-Cubing: An Integrating Method

- Integrate the top-down and bottom-up methods
- **Explore shared dimensions**
  - E.g., dimension A is the shared dimension of ACD and AD
  - ABD/AB means cuboid ABD has shared dimensions AB
- Allows for shared computations
  - e.g., cuboid AB is computed simultaneously as ABD
- Aggregate in a top-down manner but with the bottom-up sub-layer underneath which will allow Apriori pruning
- Shared dimensions grow in bottom-up fashion

Iceberg Pruning in Shared Dimensions

- Anti-monotonic property of shared dimensions
  - If the measure is anti-monotonic, and if the aggregate value on a shared dimension does not satisfy the iceberg condition, then all the cells extended from this shared dimension cannot satisfy the condition either
  - Intuition: if we can compute the shared dimensions before the actual cuboid, we can use them to do Apriori pruning
  - Problem: how to prune while still aggregate simultaneously on multiple dimensions?

Cell Trees

- Use a tree structure similar to H-tree to represent cuboids
- Collapses common prefixes to save memory
- Keep count at node
- Traverse the tree to retrieve a particular tuple

Star Attributes and Star Nodes

- Intuition: If a single-dimensional aggregate on an attribute value \( p \) does not satisfy the iceberg condition, it is useless to distinguish them during the iceberg computation
  - E.g., \( b_j, b_3, b_4, c_j, c_2, c_4, d_j, d_2, d_3 \)
  - Solution: Replace such attributes by a * . Such attributes are star attributes, and the corresponding nodes in the cell tree are star nodes
Example: Star Reduction

- Suppose \( \text{minsup} = 2 \)
- Perform one-dimensional aggregation. Replace attribute values whose count < 2 with * and collapse all *'s together
- Resulting table has all such attributes replaced with the star-attribute
- With regards to the iceberg computation, this new table is a loseless compression of the original table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>*</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>a1</td>
<td>b1</td>
<td>*</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>a1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>a2</td>
<td>*</td>
<td>c3</td>
<td>d4</td>
<td>1</td>
</tr>
</tbody>
</table>

Resulting table has all such attributes replaced with the star-attribute

With regards to the iceberg computation, this new table is a loseless compression of the original table

Star Tree

- Given the new compressed table, it is possible to construct the corresponding cell tree—called star tree
- Keep a star table at the side for easy lookup of star attributes
- The star tree is a loseless compression of the original cell tree

Star-Cubing Algorithm—DFS on Lattice Tree

- Multi-Way Aggregation

- Base-Tree

- BCD-Tree

- ACD/A-Tree

- ABD/AB-Tree

- ABC/ABC-Tree

Multi-Way Aggregation
Star-Cubing Algorithm—DFS on Star-Tree

- Start depth-first search at the root of the base star tree.
- At each new node in the DFS, create corresponding star trees that are descendents of the current tree according to the integrated traversal ordering.
  - E.g., in the base tree, when DFS reaches a1, the ACD/A tree is created.
  - When DFS reaches b*, the ABD/AD tree is created.
- The counts in the base tree are carried over to the new trees.

Multi-Way Star-Tree Aggregation

- Start depth-first search at the root of the base star tree.
- At each new node in the DFS, create corresponding star trees that are descendents of the current tree according to the integrated traversal ordering.
  - E.g., in the base tree, when DFS reaches a1, the ACD/A tree is created.
  - When DFS reaches b*, the ABD/AD tree is created.
- The counts in the base tree are carried over to the new trees.

Multi-Way Aggregation (2)

- When DFS reaches a leaf node (e.g., d*), start backtracking.
- On every backtracking branch, the count in the corresponding trees are output, the tree is destroyed, and the node in the base tree is destroyed.
- Example:
  - When traversing from d* back to c*, the alb*c*/alb*c* tree is output and destroyed.
  - When traversing from c* back to b*, the alb*D/alb* tree is output and destroyed.
  - When at b*, jump to b1 and repeat similar process.

The Curse of Dimensionality

- None of the previous cubing method can handle high dimensionality!
- A database of 600k tuples. Each dimension has cardinality of 100 and zipf of 2.
Motivation of High-D OLAP

- Challenge to current cubing methods:
  - The "curse of dimensionality" problem
  - Iceberg cube and compressed cubes: only delay the inevitable explosion
  - Full materialization: still significant overhead in accessing results on disk
- High-D OLAP is needed in applications
  - Science and engineering analysis
  - Bio-data analysis: thousands of genes
  - Statistical surveys: hundreds of variables

Fast High-D OLAP with Minimal Cubing

- **Observation:** OLAP occurs only on a small subset of dimensions at a time
- **Semi-Online Computational Model**
  1. Partition the set of dimensions into **shell fragments**
  2. Compute data cubes for each shell fragment while retaining **inverted indices** or **value-list indices**
  3. Given the pre-computed **fragment cubes**, dynamically compute cube cells of the high-dimensional data cube **online**

Properties of Proposed Method

- Partitions the data vertically
- Reduces high-dimensional cube into a set of lower dimensional cubes
- Online re-construction of original high-dimensional space
- Lossless reduction
- Offers tradeoffs between the amount of pre-processing and the speed of online computation

Example Computation

- Let the cube aggregation function be **count**

<table>
<thead>
<tr>
<th>tid</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
</tr>
<tr>
<td>2</td>
<td>a1</td>
<td>b2</td>
<td>c1</td>
<td>d2</td>
<td>e1</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>b2</td>
<td>c1</td>
<td>d1</td>
<td>e2</td>
</tr>
<tr>
<td>4</td>
<td>a2</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e2</td>
</tr>
<tr>
<td>5</td>
<td>a2</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e3</td>
</tr>
</tbody>
</table>

- Divide the 5 dimensions into 2 shell fragments: (A, B, C) and (D, E)
1-D Inverted Indices

- Build traditional inverted index or RID list

<table>
<thead>
<tr>
<th>Attribute Value</th>
<th>TID List</th>
<th>List Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>1 2 3</td>
<td>3</td>
</tr>
<tr>
<td>a2</td>
<td>4 5</td>
<td>2</td>
</tr>
<tr>
<td>b1</td>
<td>1 4 5</td>
<td>3</td>
</tr>
<tr>
<td>b2</td>
<td>2 3</td>
<td>2</td>
</tr>
<tr>
<td>c1</td>
<td>1 2 3 4 5</td>
<td>5</td>
</tr>
<tr>
<td>d1</td>
<td>1 3 4 5</td>
<td>4</td>
</tr>
<tr>
<td>d2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>e1</td>
<td>1 2</td>
<td>2</td>
</tr>
<tr>
<td>e2</td>
<td>3 4</td>
<td>2</td>
</tr>
<tr>
<td>e3</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Shell Fragment Cubes

- Generalize the 1-D inverted indices to multi-dimensional ones in the data cube sense

<table>
<thead>
<tr>
<th>Cell</th>
<th>Intersection</th>
<th>TID List</th>
<th>List Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1 b1</td>
<td>1 2 3 ∩ 1 4 5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a1 b2</td>
<td>1 2 3 ∩ 2 3</td>
<td>2 3</td>
<td>2</td>
</tr>
<tr>
<td>a2 b1</td>
<td>4 5 ∩ 1 4 5</td>
<td>4 5</td>
<td>2</td>
</tr>
<tr>
<td>a2 b2</td>
<td>4 5 ∩ 2 3</td>
<td>⊤</td>
<td>0</td>
</tr>
</tbody>
</table>

Shell Fragment Cubes (2)

- Compute all cuboids for data cubes ABC and DE while retaining the inverted indices
- For example, shell fragment cube ABC contains 7 cuboids:
  - A, B, C
  - AB, AC, BC
  - ABC
- This completes the offline computation stage

Shell Fragment Cubes (3)

- Given a database of T tuples, D dimensions, and F shell fragment size, the fragment cubes’ space requirement is:

\[
O(T \left\lceil \frac{D}{F} \right\rceil 2^F - 1)
\]

- For F < 5, the growth is sub-linear.
Shell Fragment Cubes (4)

- Shell fragments do not have to be disjoint
- Fragment groupings can be arbitrary to allow for maximum online performance
  - Known common combinations (e.g., <city, state>) should be grouped together.
- Shell fragment sizes can be adjusted for optimal balance between offline and online computation

ID_Measure Table

- If measures other than count are present, store in ID_measure table separate from the shell fragments

<table>
<thead>
<tr>
<th>tid</th>
<th>count</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>30</td>
</tr>
</tbody>
</table>

The Frag-Shells Algorithm

1. Partition set of dimension \((A_1, \ldots, A_n)\) into a set of \(k\) fragments \((P_1, \ldots, P_k)\).
2. Scan base table once and do the following
   3. insert <tid, measure> into ID_measure table.
   4. for each attribute value \(a_i\) of each dimension \(A_i\)
   5. build inverted index entry \(<a_i, tidlist>\)
   6. For each fragment partition \(P_i\)
   7. build local fragment cube \(S_i\) by intersecting tid-lists in bottom-up fashion.

Frag-Shells (2)

Dimensions

<table>
<thead>
<tr>
<th>Cubes and Cuboids</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC Cube</td>
</tr>
<tr>
<td>DEF Cube</td>
</tr>
<tr>
<td>D Cuboid</td>
</tr>
<tr>
<td>DE Cuboid</td>
</tr>
<tr>
<td>EF Cuboid</td>
</tr>
<tr>
<td>D Cuboid</td>
</tr>
<tr>
<td>Tuple-ID List</td>
</tr>
<tr>
<td>d1 e1</td>
</tr>
<tr>
<td>d1 e2</td>
</tr>
<tr>
<td>d2 e1</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Online Query Computation

- A query has the general form \( \langle a_1, a_2, \ldots, a_n ; M \rangle \)
- Each \( a_i \) has 3 possible values
  1. Instantiated value
  2. Aggregate * function
  3. Inquire ? function
- For example, \( \langle 3 \ ? \ ? \ * \ 1 : \text{count} \rangle \) returns a 2-D data cube.

Online Query Computation (2)

- Given the fragment cubes, process a query as follows
  1. Divide the query into fragment, same as the shell
  2. Fetch the corresponding TID list for each fragment from the fragment cube
  3. Intersect the TID lists from each fragment to construct \textit{instantiated base table}
  4. Compute the data cube using the base table with any cubing algorithm

Online Query Computation (3)

Experiment: Size vs. Dimensionality (50 and 100 cardinality)

- (50-C): 10^6 tuples, 0 skew, 50 cardinality, fragment size 3.
- (100-C): 10^7 tuples, 2 skew, 100 cardinality, fragment size 2.
Experiment: Size vs. Shell-Fragment Size

- (50-D): 10^5 tuples, 50 dimensions, 0 skew, 50 cardinality.
- (100-D): 10^5 tuples, 100 dimensions, 2 skew, 25 cardinality.

Experiment: Run-time vs. Shell-Fragment Size

- 10^5 tuples, 20 dimensions, 10 cardinality, skew 1, fragment size 3, 3 instantiated dimensions.

Experiment: I/O vs. Shell-Fragment Size

- (10-D): 10^5 tuples, 10 dimensions, 10 cardinality, 0 skew, 4 inst., 4 query.
- (20-D): 10^5 tuples, 20 dimensions, 10 cardinality, 1 skew, 3 inst., 4 query.

Experiment: I/O vs. # of Instantiated Dimensions

- 10^5 tuples, 10 dimensions, 10 cardinality, 0 skew, fragment size 1, 7 total relevant dimensions.
Experiments on Real World Data

- UCI Forest CoverType data set
  - 54 dimensions, 581K tuples
  - Shell fragments of size 2 took 33 seconds and 325MB to compute
  - 3-D subquery with 1 instantiate D: 85ms~1.4 sec.
- Longitudinal Study of Vocational Rehab. Data
  - 24 dimensions, 8818 tuples
  - Shell fragments of size 3 took 0.9 seconds and 60MB to compute
  - 5-D query with 0 instantiated D: 227ms~2.6 sec.

Comparisons to Related Work

- [Harinarayan96] computes low-dimensional cuboids by further aggregation of high-dimensional cuboids. Opposite of our method’s direction.
- Inverted indexing structures [Witten99] focus on single dimensional data or multi-dimensional data with no aggregation.
- Tree-stripping [Berchtold00] uses similar vertical partitioning of database but no aggregation.

Further Implementation Considerations

- Incremental Update:
  - Append more TIDs to inverted list
  - Add <tid: measure> to ID_measure table
- Incremental adding new dimensions
  - Form new inverted list and add new fragments
- Bitmap indexing
  - May further improve space usage and speed
- Inverted index compression
  - Store as d-gaps
  - Explore more IR compression methods
Computing Cubes with Non-Antimonotonic Iceberg Conditions

- Most cubing algorithms cannot compute cubes with non-antimonotonic iceberg conditions efficiently
- Example
  
  ```sql
  CREATE CUBE Sales_Iceberg AS
  SELECT month, city, cust_grp,
  AVG(price), COUNT(*)
  FROM Sales_Infor
  CUBEBY month, city, cust_grp
  HAVING AVG(price) >= 800 AND
  COUNT(*) >= 50
  ```
- Needs to study how to push constraint into the cubing process

Non-Anti-Monotonic Iceberg Condition

- Anti-monotonic: if a process fails a condition, continue processing will still fail
- The cubing query with avg is non-anti-monotonic!
  - (Mar, *, *, 600, 1800) fails the HAVING clause
  - (Mar, *, Bus, 1300, 360) passes the clause

From Average to Top-k Average

- Let (*, Van, *) cover 1,000 records
  - Avg(price) is the average price of those 1000 sales
  - Avg50(price) is the average price of the top-50 sales (top-50 according to the sales price)
- Top-k average is anti-monotonic
  - The top 50 sales in Van. is with avg(price) <= 800 → the top 50 deals in Van. during Feb. must be with avg(price) <= 800

Binning for Top-k Average

- Computing top-k avg is costly with large k
- Binning idea
  - Avg50(c) >= 800
  - Large value collapsing: use a sum and a count to summarize records with measure >= 800
    - If count>=800, no need to check “small” records
  - Small value binning: a group of bins
    - One bin covers a range, e.g., 600~800, 400~600, etc.
    - Register a sum and a count for each bin
Computing Approximate top-k average

Suppose for (*, Van, *), we have

<table>
<thead>
<tr>
<th>Range</th>
<th>Sum</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over 800</td>
<td>28000</td>
<td>20</td>
</tr>
<tr>
<td>600–800</td>
<td>10600</td>
<td>15</td>
</tr>
<tr>
<td>400–600</td>
<td>15200</td>
<td>30</td>
</tr>
</tbody>
</table>

Top 50

The cell may pass the HAVING clause

\[
\text{Approximate avg}^{50}(\cdot) = \frac{(28000 + 10600 + 600 \times 15)}{50} = 952
\]

**Weakened Conditions Facilitate Pushing**

- Accumulate quant-info for cells to compute average iceberg cubes efficiently
- Three pieces: sum, count, top-k bins
- Use top-k bins to estimate/prune descendants
- Use sum and count to consolidate current cell

\[
\begin{array}{c|c|c|c|c|c|c}
\text{Month} & \text{City} & \text{Cost}_{\text{grp}} & \text{Prod} & \text{Cost} & \text{Price} \\
\hline
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{array}
\]

\[\text{Strongest} \quad \text{Weakest}\]

\[\text{Approximate avg}^{50}(\cdot) \quad \text{real avg}^{50}(\cdot) \quad \text{avg}(\cdot)\]

- Anti-monotonic, can be computed efficiently
- Anti-monotonic, but computationally costly
- Not anti-monotonic

**Computing Iceberg Cubes with Other Complex Measures**

- Computing other complex measures
  - Key point: find a function which is weaker but ensures certain anti-monotonicity
  - Examples
    - \(\text{Avg}(\cdot) \leq v: \quad \text{avg}_k(c) \leq v\) (bottom-k avg)
    - \(\text{Avg}(\cdot) \geq v\) only (no count): \(\text{max(price)} \geq v\)
    - \(\text{Sum(profit)}\) (profit can be negative):
      - \(p_{\text{sum}}(c) \geq v\) if \(p_{\text{count}}(c) \geq k\), or otherwise, \(\text{sum}^k(c) \geq v\)
    - Others: conjunctions of multiple conditions

**Compressed Cubes: Condensed or Closed Cubes**

- Iceberg cube cannot solve all the problems
  - Suppose 100 dimensions, only 1 base cell with count = 10. How many aggregate (non-base) cells if count >= 10?
- Condensed cube
  - Only need to store one cell \((a_{j1}, a_{j2}, \ldots, a_{j100}, 10)\), which represents all the corresponding aggregate cells
  - Adv.
    - Fully precomputed cube without compression
    - Efficient computation of the minimal condensed cube
- Closed cube
  - Dong Xin, Jiawei Han, Zheng Shao, and Hongyan Liu, "C-Cubing: Efficient Computation of Closed Cubes by Aggregation-Based Checking", ICDE’06.
Chapter 4: Data Cube Computation and Data Generalization

- Efficient Computation of Data Cubes
- Exploration and Discovery in Multidimensional Databases
- Attribute-Oriented Induction — An Alternative Data Generalization Method

Kinds of Exceptions and their Computation

- Parameters
  - SelfExp: surprise of cell relative to other cells at same level of aggregation
  - InExp: surprise beneath the cell
  - PathExp: surprise beneath cell for each drill-down path
- Computation of exception indicator (modeling fitting and computing SelfExp, InExp, and PathExp values) can be overlapped with cube construction
- Exception themselves can be stored, indexed and retrieved like precomputed aggregates

Discovery-Driven Exploration of Data Cubes

- Hypothesis-driven
  - exploration by user, huge search space
- Discovery-driven (Sarawagi, et al.’98)
  - Effective navigation of large OLAP data cubes
  - pre-compute measures indicating exceptions, guide user in the data analysis, at all levels of aggregation
  - Exception: significantly different from the value anticipated, based on a statistical model
  - Visual cues such as background color are used to reflect the degree of exception of each cell

Examples: Discovery-Driven Data Cubes
Complex Aggregation at Multiple Granularities: Multi-Feature Cubes

- Multi-feature cubes (Ross, et al. 1998): Compute complex queries involving multiple dependent aggregates at multiple granularities
- Ex. Grouping by all subsets of (item, region, month), find the maximum price in 1997 for each group, and the total sales among all maximum price tuples
  ```
  select item, region, month, max(price), sum(R.sales)
  from purchases
  where year = 1997
  cube by item, region, month: R
  such that R.price = max(price)
  ```
- Continuing the last example, among the max price tuples, find the min and max shelf live, and find the fraction of the total sales due to tuple that have min shelf life within the set of all max price tuples

Cube-Gradient (Cubegrade)

- Analysis of changes of sophisticated measures in multi-dimensional spaces
- Query: changes of average house price in Vancouver in '00 comparing against '99
- Answer: Apts in West went down 20%, houses in Metrotown went up 10%
- Cubegrade problem by Imielinski et al.
  - Changes in dimensions → changes in measures
  - Drill-down, roll-up, and mutation

From Cubegrade to Multi-dimensional Constrained Gradients in Data Cubes

- Significantly more expressive than association rules
- Capture trends in user-specified measures
- Serious challenges:
  - Many trivial cells in a cube → "significance constraint" to prune trivial cells
  - Numerate pairs of cells → "probe constraint" to select a subset of cells to examine
  - Only interesting changes wanted → "gradient constraint" to capture significant changes

MD Constrained Gradient Mining

- Significance constraint $C_{sig}$: (cnt $\geq 100$)
- Probe constraint $C_{prb}$: (city="Van", cust_grp="busi", prod_grp="*")
- Gradient constraint $C_{grad}(C_g, C_p)$: $(\text{avg}_{price}(C_g)/\text{avg}_{price}(C_p) \geq 1.3)$
Efficient Computing Cube-gradients

- Compute probe cells using $C_{\text{sig}}$ and $C_{\text{prb}}$
  - The set of probe cells $P$ is often very small
- Use probe $P$ and constraints to find gradients
  - Pushing selection deeply
  - Set-oriented processing for probe cells
  - Iceberg growing from low to high dimensionalities
  - Dynamic pruning probe cells during growth
  - Incorporating efficient iceberg cubing method

Chapter 4: Data Cube Computation and Data Generalization

- Efficient Computation of Data Cubes
- Exploration and Discovery in Multidimensional Databases
- Attribute-Oriented Induction — An Alternative Data Generalization Method

What is Concept Description?

- Descriptive vs. predictive data mining
  - Descriptive mining: describes concepts or task-relevant data sets in concise, summarative, informative, discriminative forms
  - Predictive mining: Based on data and analysis, constructs models for the database, and predicts the trend and properties of unknown data
- Concept description:
  - Characterization: provides a concise and succinct summarization of the given collection of data
  - Comparison: provides descriptions comparing two or more collections of data

Data Generalization and Summarization-based Characterization

- Data generalization
  - A process which abstracts a large set of task-relevant data in a database from a low conceptual levels to higher ones.
- Approaches:
  - Data cube approach(OLAP approach)
  - Attribute-oriented induction approach
Concept Description vs. OLAP

- **Similarity:**
  - Data generalization
  - Presentation of data summarization at multiple levels of abstraction.
  - Interactive drilling, pivoting, slicing and dicing.

- **Differences:**
  - Can handle complex data types of the attributes and their aggregations
  - Automated desired level allocation.
  - Dimension relevance analysis and ranking when there are many relevant dimensions.
  - Sophisticated typing on dimensions and measures.
  - Analytical characterization: data dispersion analysis

Attribute-Oriented Induction

- Proposed in 1989 (KDD '89 workshop)
- Not confined to categorical data nor particular measures
- How it is done?
  - Collect the task-relevant data (**initial relation**) using a relational database query
  - Perform generalization by attribute removal or attribute generalization
  - Apply aggregation by merging identical, generalized tuples and accumulating their respective counts
  - Interactive presentation with users

Basic Principles of Attribute-Oriented Induction

- **Data focusing:** task-relevant data, including dimensions, and the result is the **initial relation**
- **Attribute-removal:** remove attribute \( A \) if there is a large set of distinct values for \( A \) but (1) there is no generalization operator on \( A \), or (2) \( A \)'s higher level concepts are expressed in terms of other attributes
- **Attribute-generalization:** If there is a large set of distinct values for \( A \), and there exists a set of generalization operators on \( A \), then select an operator and generalize \( A \)
- **Attribute-threshold control:** typical 2-8, specified/default
- **Generalized relation threshold control:** control the final relation/rule size

Attribute-Oriented Induction: Basic Algorithm

- **InitialRel:** Query processing of task-relevant data, deriving the **initial relation**.
- **PreGen:** Based on the analysis of the number of distinct values in each attribute, determine generalization plan for each attribute: removal? or how high to generalize?
- **PrimeGen:** Based on the PreGen plan, perform generalization to the right level to derive a "prime generalized relation", accumulating the counts.
- **Presentation:** User interaction: (1) adjust levels by drilling, (2) pivoting, (3) mapping into rules, cross tabs, visualization presentations.
Example

- **DMQL:** Describe general characteristics of graduate students in the Big-University database
  
  **use Big_University_DB**

  **mine characteristics as "Science_Students"**

  in relevance to name, gender, major, birth_place, birth_date, residence, phone#, gpa

  **from student**

  **where** status in "graduate"

- **Corresponding SQL statement:**

  ```sql
  Select name, gender, major, birth_place, birth_date, residence, phone#, gpa
  from student
  where status in {"Msc", "MBA", "PhD"}
  ```

Class Characterization: An Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Gender</th>
<th>Major</th>
<th>Birth-Place</th>
<th>Birth_date</th>
<th>Residence</th>
<th>Phone #</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Woodman</td>
<td>M</td>
<td>CS</td>
<td>Vancouver, BC, Canada</td>
<td>8-12-76</td>
<td>3511 Main St., Richmond</td>
<td>487-4598</td>
<td>3.67</td>
</tr>
<tr>
<td>Scott Lachance</td>
<td>M</td>
<td>CS</td>
<td>Montreal, Que, Canada</td>
<td>28-7-75</td>
<td>345 1st Ave., Richmond</td>
<td>253-9106</td>
<td>3.70</td>
</tr>
<tr>
<td>Laura Lee</td>
<td>F</td>
<td>Physics</td>
<td>Seattle, WA, USA</td>
<td>25-8-70</td>
<td>125 Anita Ave., Burnaby</td>
<td>420-4025</td>
<td>3.48</td>
</tr>
</tbody>
</table>

- **Prime Relation**

<table>
<thead>
<tr>
<th>Gender</th>
<th>Major</th>
<th>Birth_region</th>
<th>Age_range</th>
<th>Residence</th>
<th>GPA</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Science</td>
<td>Canada</td>
<td>20-25</td>
<td>Richmond</td>
<td>Very-good</td>
<td>16</td>
</tr>
<tr>
<td>F</td>
<td>Science</td>
<td>Foreign</td>
<td>25-30</td>
<td>Burnaby</td>
<td>Excellent</td>
<td>22</td>
</tr>
</tbody>
</table>

- **Generalized Relation**

<table>
<thead>
<tr>
<th>Gender</th>
<th>Major</th>
<th>Birth_region</th>
<th>Age_range</th>
<th>Residence</th>
<th>GPA</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Canada</td>
<td>Foreign</td>
<td>20-25</td>
<td>Richmond</td>
<td>Very-good</td>
<td>16</td>
</tr>
<tr>
<td>F</td>
<td>Canada</td>
<td>Foreign</td>
<td>25-30</td>
<td>Burnaby</td>
<td>Excellent</td>
<td>22</td>
</tr>
</tbody>
</table>

Presentation of Generalized Results

- **Generalized relation:**
  Relations where some or all attributes are generalized, with counts or other aggregation values accumulated.

- **Cross tabulation:**
  Mapping results into cross tabulation form (similar to contingency tables).

- **Visualization techniques:**
  Pie charts, bar charts, curves, cubes, and other visual forms.

- **Quantitative characteristic rules:**
  Mapping generalized result into characteristic rules with quantitative information associated with it, e.g.,

  \[
  \text{grad}(x) \land \text{male}(x) \Rightarrow \\
  \text{birth\_region}(x) = \text{Canada}[t: 53\%] \lor \text{birth\_region}(x) = \text{foreign}[t: 47\%].
  \]

Mining Class Comparisons

- **Comparison:** Comparing two or more classes

- **Method:**
  - Partition the set of relevant data into the target class and the contrasting class(es)
  - Generalize both classes to the same high level concepts
  - Compare tuples with the same high level descriptions
  - Present for every tuple its description and two measures
  - Support - distribution within single class
  - Comparison - distribution between classes
  - Highlight the tuples with strong discriminant features

- **Relevance Analysis:**
  - Find attributes (features) which best distinguish different classes
Quantitative Discriminant Rules

- $C_j$ = target class
- $q_a$ = a generalized tuple covers some tuples of class
  but can also cover some tuples of contrasting class
- $d$-weight
  - range: $[0, 1]$
  - $d$-weight $= \frac{\text{count}(q_a \in C_j)}{\sum_{i=1}^{\text{count}(q_a \in C_i)}}$
- quantitative discriminant rule form

$$\forall X, target\_class(X) \iff \text{condition}(X) \ [d : d\_weight]$$

Example: Quantitative Discriminant Rule

<table>
<thead>
<tr>
<th>Status</th>
<th>Birth_country</th>
<th>Age_range</th>
<th>Gpa</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graduate</td>
<td>Canada</td>
<td>25-30</td>
<td>Good</td>
<td>90</td>
</tr>
<tr>
<td>Undergraduate</td>
<td>Canada</td>
<td>25-30</td>
<td>Good</td>
<td>210</td>
</tr>
</tbody>
</table>

Count distribution between graduate and undergraduate students for a generalized tuple

- Quantitative discriminant rule

$$\forall X, \text{graduate\_student}(X) \iff\text{birth\_country}(X) = \text{Canada} \land \text{age\_range}(X) = 25-30 \land \text{gpa}(X) = \text{good} \ [d : 30\%]$$

  - where $90/(90 + 210) = 30\%$

Class Description

- Quantitative characteristic rule

$$\forall X, target\_class(X) \Rightarrow \text{condition}(X) \ [t : t\_weight]$$
  - necessary

- Quantitative discriminant rule

$$\forall X, target\_class(X) \Leftarrow \text{condition}(X) \ [d : d\_weight]$$
  - sufficient

- Quantitative description rule

$$\forall X, target\_class(X) \Leftarrow \omega(X) \ [t : w_1, d : w_1'] \lor \ldots \lor \omega(X) \ [t : w_n, d : w_n']$$
  - necessary and sufficient

Example: Quantitative Description Rule

<table>
<thead>
<tr>
<th>Location/Item</th>
<th>Count</th>
<th>$t_wt$</th>
<th>$d_wt$</th>
<th>Count</th>
<th>$t_wt$</th>
<th>$d_wt$</th>
<th>Count</th>
<th>$t_wt$</th>
<th>$d_wt$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Europe</td>
<td>80</td>
<td>25%</td>
<td>40%</td>
<td>240</td>
<td>75%</td>
<td>30%</td>
<td>120</td>
<td>100%</td>
<td>72%</td>
</tr>
<tr>
<td>N_Am</td>
<td>120</td>
<td>17.65%</td>
<td>60%</td>
<td>560</td>
<td>82.35%</td>
<td>70%</td>
<td>680</td>
<td>100%</td>
<td>68%</td>
</tr>
<tr>
<td>Both regions</td>
<td>200</td>
<td>20%</td>
<td>100%</td>
<td>800</td>
<td>80%</td>
<td>100%</td>
<td>1000</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Crosstab showing associated $t$-weight, $d$-weight values and total number (in thousands) of TVs and computers sold at AllElectronics in 1998

- Quantitative description rule for target class $Europe$

$$\forall X, \text{Europe}(X) \iff (item(X) = "TV" \lor [1 : 25\%, d : 40\%]) \lor (item(X) = "computer" \lor [1 : 75\%, d : 30\%])$$
Summary

- Efficient algorithms for computing data cubes
  - Multiway array aggregation
  - BUC
  - H-cubing
  - Star-cubing
  - High-D OLAP by minimal cubing
- Further development of data cube technology
  - Discovery-drive cube
  - Multi-feature cubes
  - Cube-gradient analysis
- Another generalization approach: Attribute-Oriented Induction

References (I)

- S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. VLDB'96
- Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-Dimensional Regression Analysis of Time-Series Data Streams, VLDB'02
- G. Dong, J. Han, J. Lam, J. Pei, K. Wang. Mining Multi-dimensional Constrained Gradients in Data Cubes. VLDB'01
- J. Han, Y. Cai and N. Cercone, Knowledge Discovery in Databases: An Attribute-Oriented Approach, VLDB'92
- J. Han, J. Pei, G. Dong, K. Wang. Efficient Computation of Iceberg Cubes With Complex Measures. SIGMOD'01

References (II)

- L. V. S. Lakshmanan, J. Pei, and J. Han, Quotient Cube: How to Summarize the Semantics of a Data Cube, VLDB'02
- X. Li, J. Han, and H. Gonzalez, High-Dimensional OLAP: A Minimal Cubing Approach, VLDB'04
- K. Ross and D. Srivastava. Fast computation of sparse datacubes. VLDB'97
- K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. EDBT'98
- S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of OLAP data cubes. EDBT'98
- G. Sathe and S. Sarawagi. Intelligent Rollups in Multidimensional OLAP Data. VLDB'01
- D. Xin, J. Han, X. Li, B. W. Wah, Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration, VLDB'03
- D. Xin, J. Han, Z. Shao, H. Liu, C-Cubing: Efficient Computation of Closed Cubes by Aggregation-Based Checking, ICDM'06
- W. Wang, H. Lu, J. Feng, J. X. Yu, Condensed Cube: An Effective Approach to Reducing Data Cube Size. ICDE'02
- Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. SIGMOD'97

SQL Group By statement

From: http://www.w3schools.com/sql/sql_groupby.asp