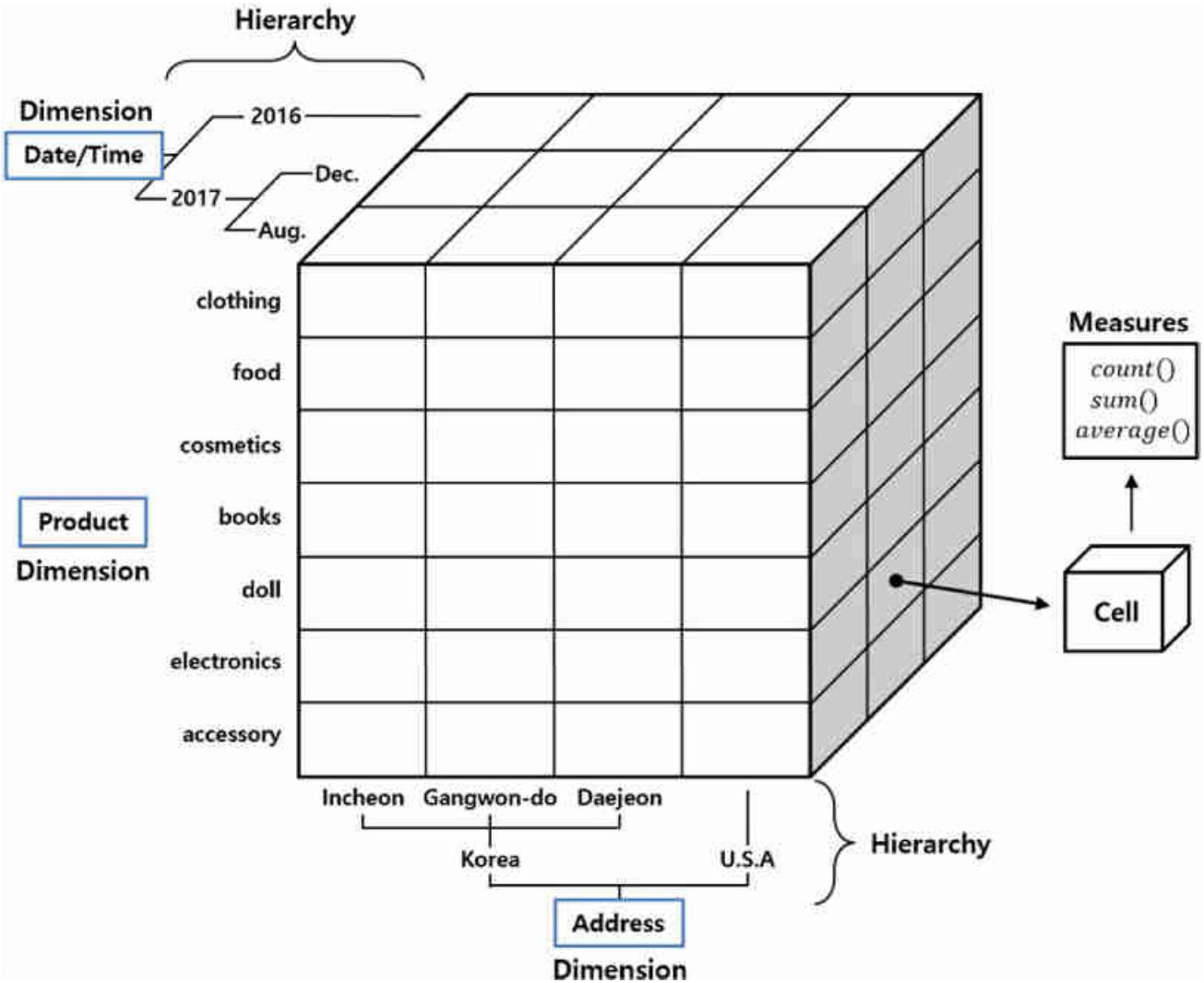
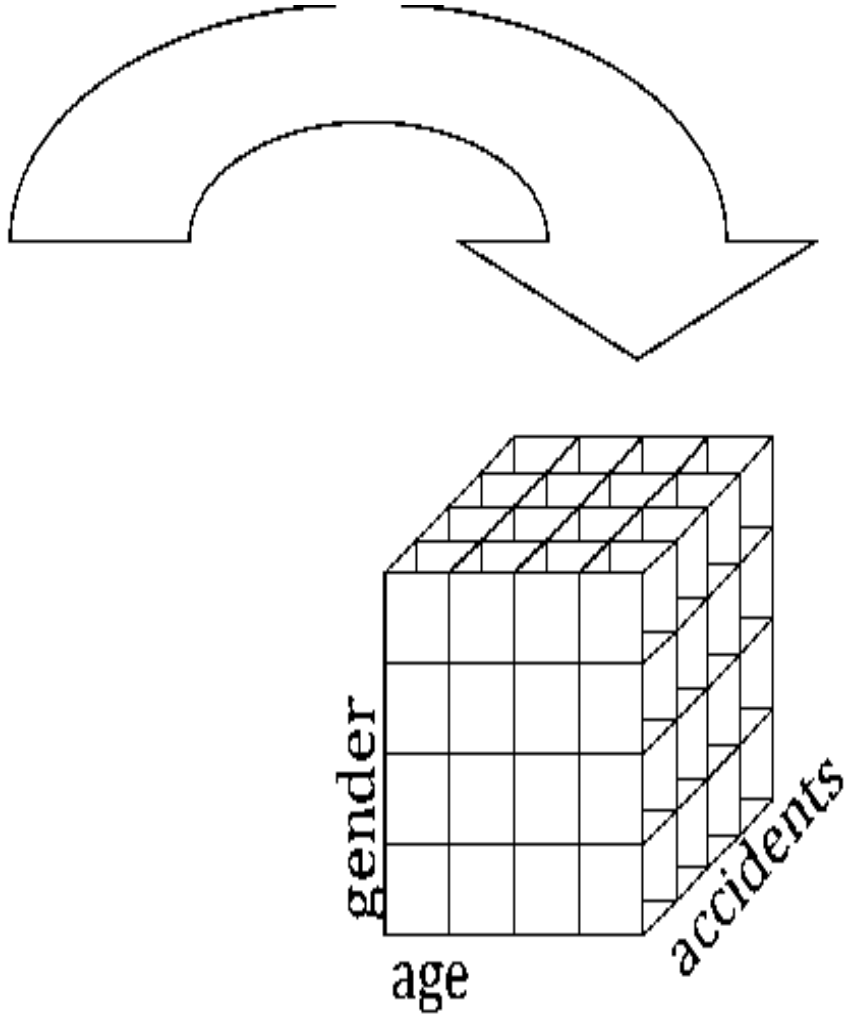
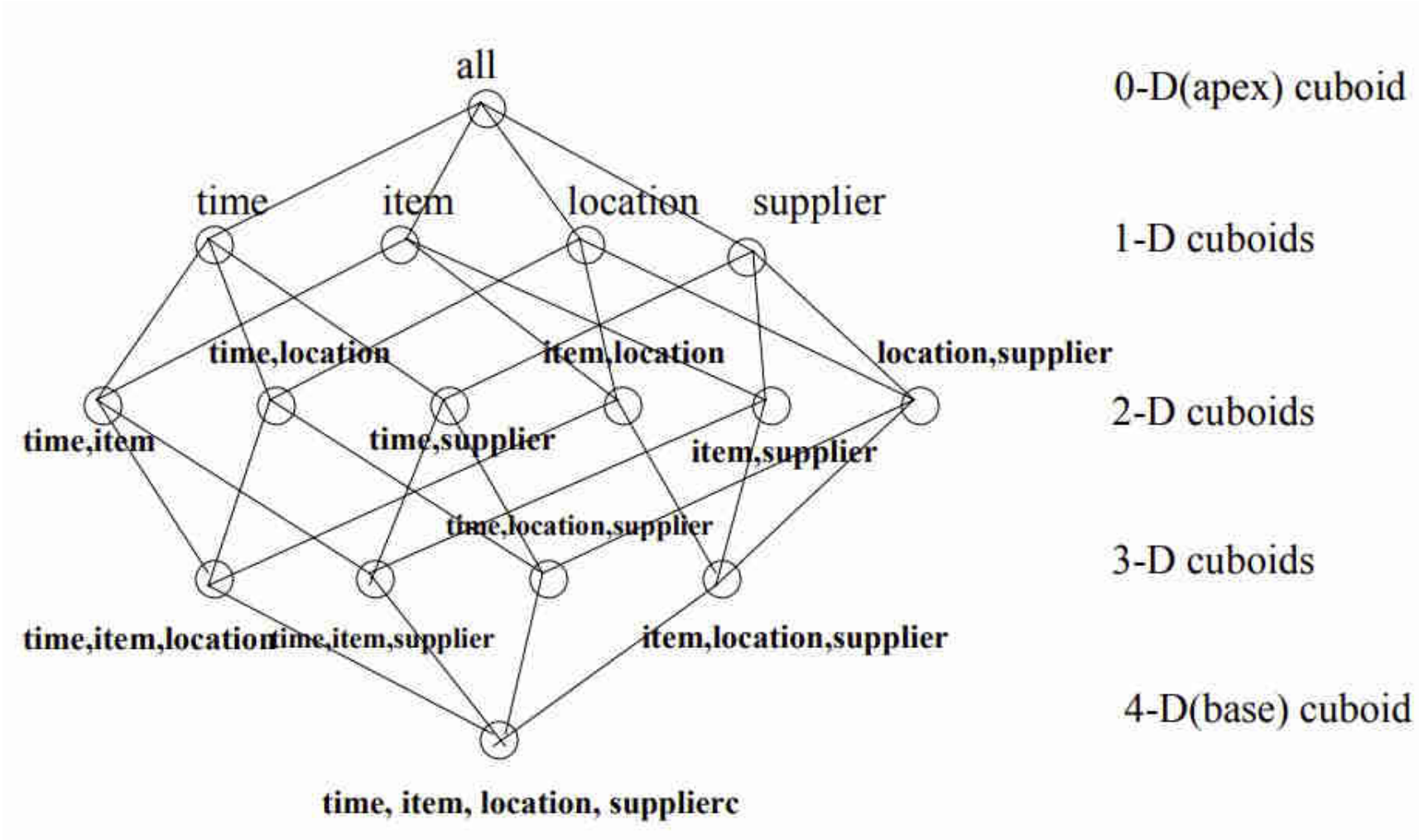
Unit 4

Tuning For Data WareHouse


Hierarchy
Dimension
Date/Time
2016
2017
Dec.
Aug.
clothing
food
cosmetics
books
doll
electronics
accessory
Product
Dimension
Measures
count()
sum()
average()
Cell
Incheon
Gangwon-do
Daejeon
Korea
U.S.A
Hierarchy
Address
Dimension

gender	age	accident
Male	27	3
Male	37	1
Male	37	0
Male	37	1
Male	49	2
Male	39	4
Male	43	0
Male	41	2
Male	49	1
Male	44	2
Male	43	3
Male	50	4
Male	60	0
Female	26	0
Female	39	0
Female	45	2
Female	41	2
Female	39	1
Female	37	0
Female	43	1




all
time
item
location
supplier
time,location
item,location
location,supplier
time,item
time,supplier
item,supplier
time,location,supplier
time,item,location
time,item,supplier
item,location,supplier
time, item, location, supplierc
0-D(apex) cuboid
1-D cuboids
2-D cuboids
3-D cuboids
4-D(base) cuboid

Data Generalization

- Data generalization is the process of creating a more broad categorization of data in a database, essentially 'zooming out' from the data to create a more general picture of trends or insights it provides.
- It replaces a specific data value with one that is less precise
- It is widely applicable and used technique in data mining, analysis, and secure storage.

Original Data:

Ages: 26, 28, 31, 33, 37, 42, 42, 46, 48, 49, 54, 57, 57, 58, 59

Generalized Data:

Ages:

20-29 (2)

30-39 (3)

40-49 (5)

50-59 (5)

Data Generalization → Why it is needed?

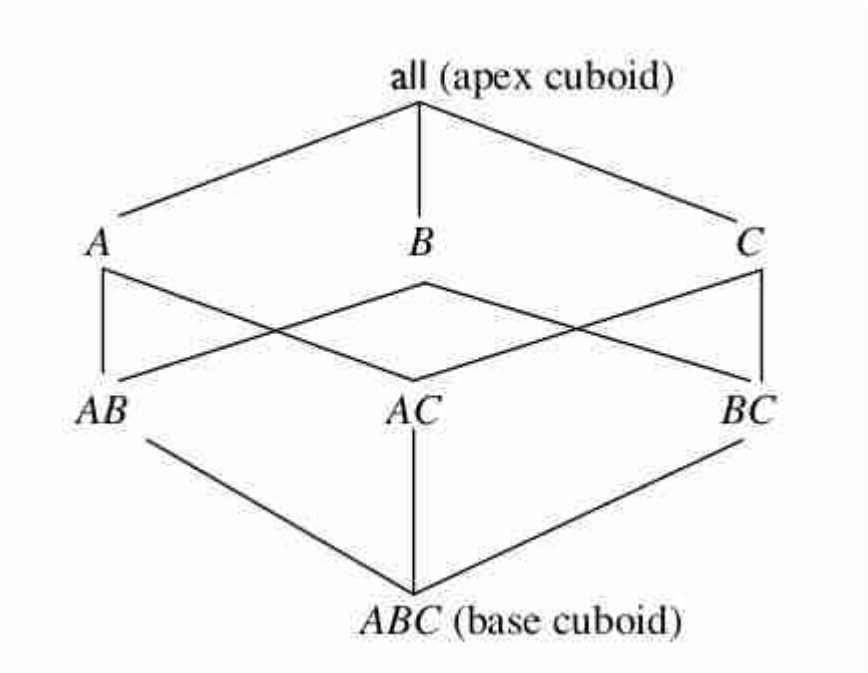
- When we need to analyze data that you've collected, but also need to ensure the privacy of the individuals who are included in that data. It's a powerful way of abstracting personal information while retaining the usefulness of the data points. In the age example above, generalizing age data based on each decade gives a general picture of where the individuals in the data set fall, and still allows you to use that data for relatively accurate targeting or analysis.

Computation of Selected Cuboids

There are three choices for data cube materialization given a base cuboid:

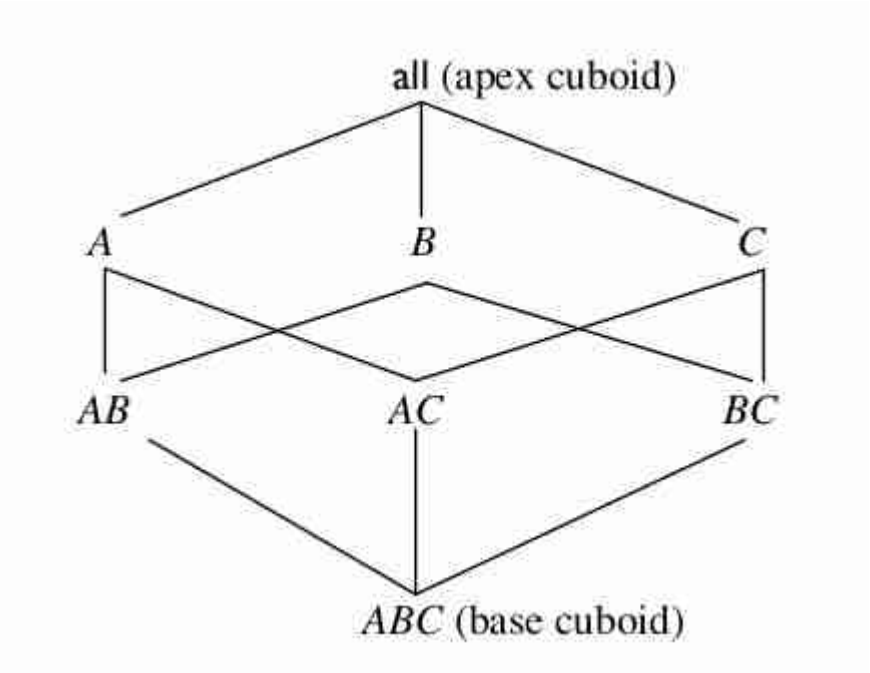
- **No materialization:** Do not pre-compute any of the cuboids. This leads to computing expensive multidimensional aggregates, which can be extremely slow.
- **Full materialization:** Pre-compute all of the cuboids. The resulting lattice of computed cuboids is referred to as the full cube.
- **Partial materialization:** Selectively compute a proper subset of the whole set of possible cuboids.

Cube Materialization



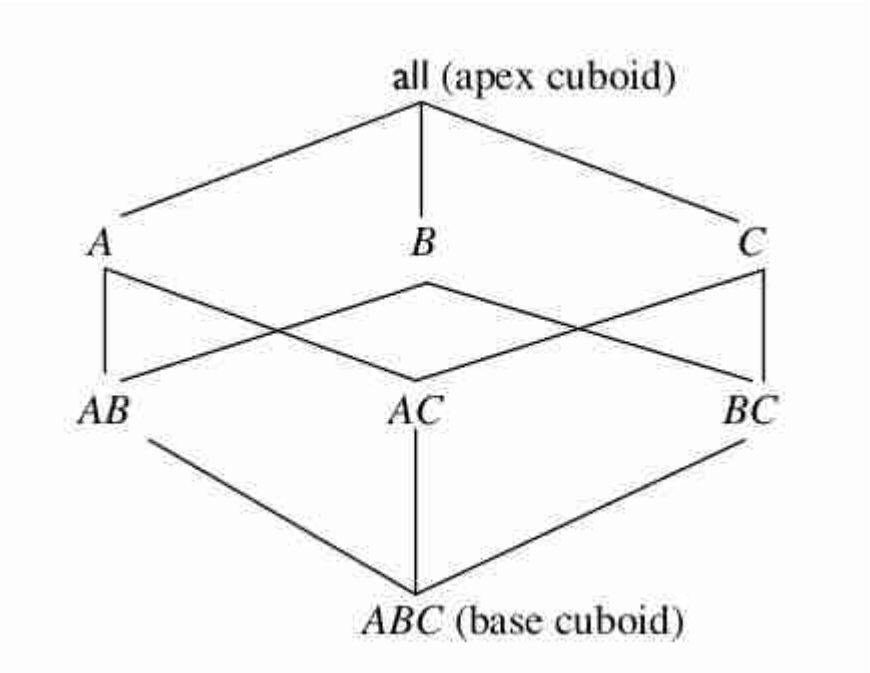
- Figure shows a 3-D data cube for the dimensions A, B, and C, and an aggregate measure, M.
- Commonly used measures include count() , sum() , min() , max() , and total sales().
- A data cube is a lattice of cuboids. Each cuboid represents a group-by.
- ABC is the base cuboid, containing all three of the dimensions. Here, the aggregate measure, M, is computed for each possible combination of the three dimensions.

Cube Materialization



- To drill down in the data cube, we move from the apex cuboid downward in the lattice.
- To roll up, we move from the base cuboid upward.
- A cell in the base cuboid is a base cell.
- A cell from a nonbase cuboid is an aggregate cell

Cube Materialization → Base and aggregate cells



- An aggregate cell aggregates over one or more dimensions, where each aggregated dimension is indicated by a * in the cell notation.
- Suppose we have an n-dimensional data cube. Let a = (a_1 , a_2 , . . . , a_n, measure) be a cell from one of the cuboids making up the data cube. We say that a is an m-dimensional cell (i.e., from an m-dimensional cuboid) if exactly m (m ≤ n) values among (a_1 , a_2 , . . . , a_n) are not *.
- If m = n, then a is a base cell; otherwise, it is an aggregate cell

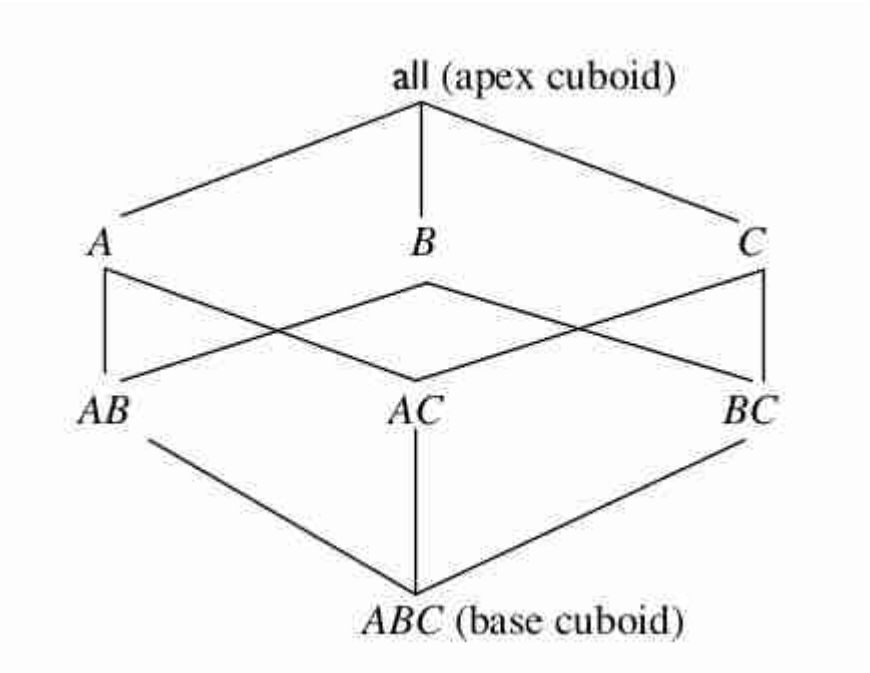
Cube Materialization → Base and aggregate cells

Consider a data cube with the dimensions month, city, and customer group, and the measure sales. (Jan, * , * , 2800) and (*, Chicago, * , 1200) are 1-D cells; (Jan, * , Business, 150) is a 2-D cell; and (Jan, Chicago, Business, 45) is a 3-D cell. Here, all base cells are 3-D, whereas 1-D and 2-D cells are aggregate cells.

Cube Materialization → Ancestor and descendant cells

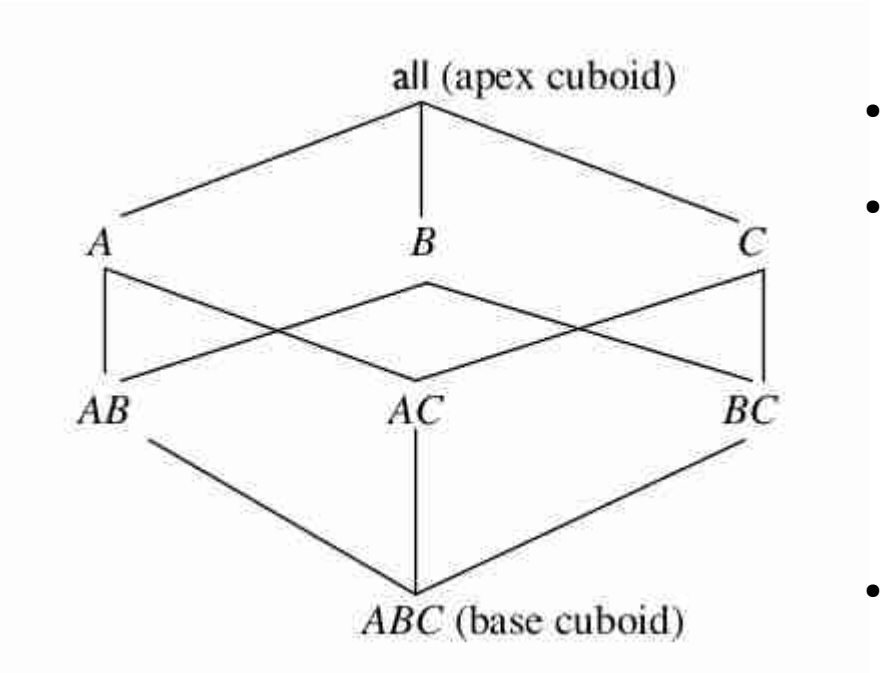
1-D cell a = (Jan, * , * ,2800) and 2-D cell b = (Jan, * , Business, 150) are ancestors of 3-D cell c = (Jan,Chicago, Business, 45); c is a descendant of both a and b; b is a parent of c; and c is a child of b.

Cube Materialization → Full Cube



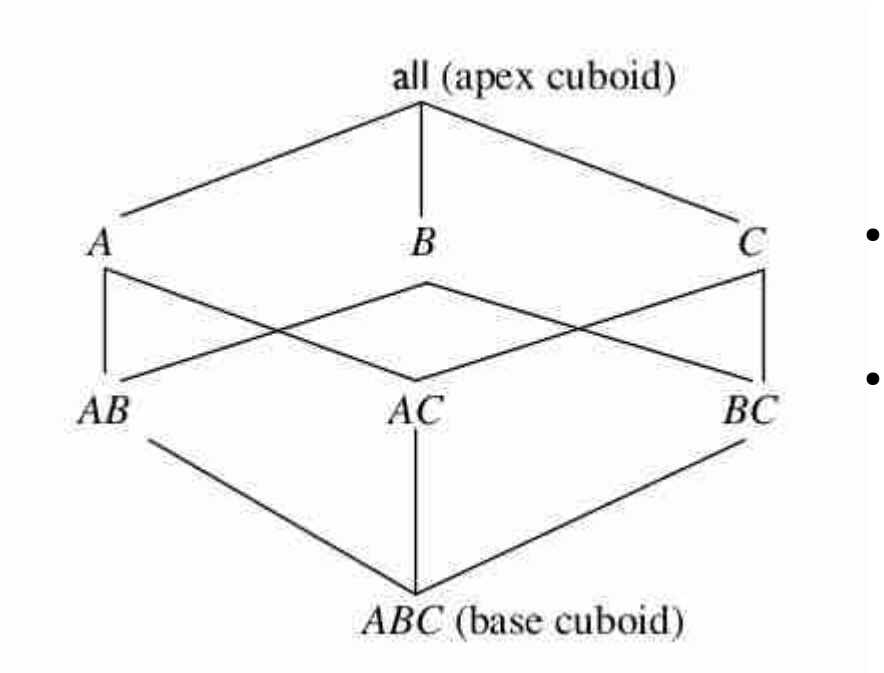
- All the cells of all the cuboids for a given data cube is full cube.
- To ensure fast OLAP, it is sometimes desirable to precompute the full cube

Cube Materialization → Sparse Cube



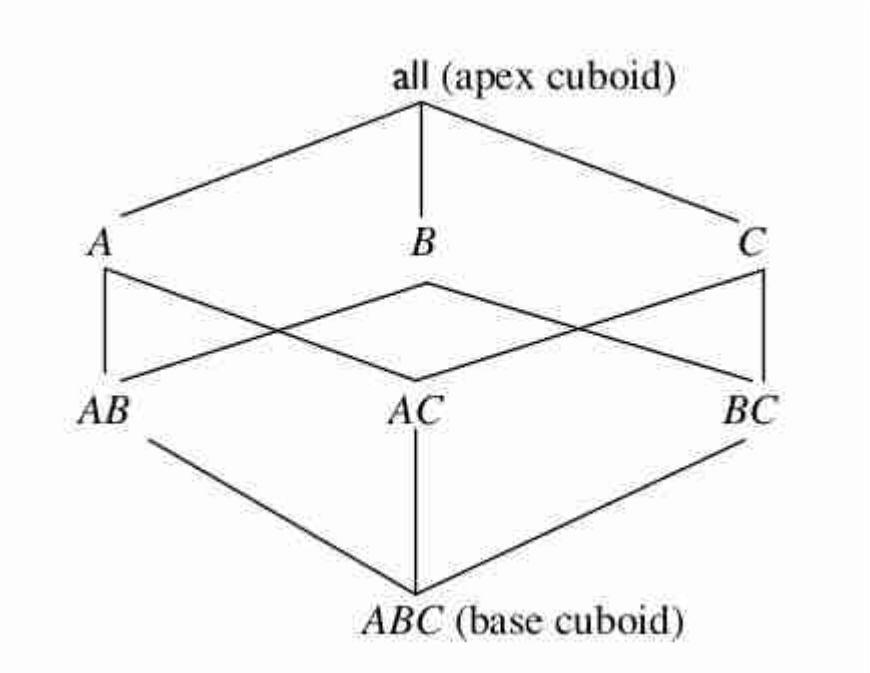
- Many cells in a cuboid may actually be of little or no interest to the data analyst.
- Each cell in a full cube records an aggregate value such as count or sum
- For many cells in a cuboid, the measure value will be zero. When the product of the cardinalities for the dimensions in a cuboid is large relative to the number of nonzero-valued tuples that are stored in the cuboid, then we say that the cuboid is sparse.
- If a cube contains many sparse cuboids, we say that the cube is sparse.

Cube Materialization → Iceberg Cube



- An Iceberg-Cube contains only those cells of the data cube that meet an aggregate condition. It is called an Iceberg-Cube because it contains only some of the cells of the full cube, like the tip of an iceberg.
- The aggregate condition could be, for example, minimum support or a lower bound on average, min or max.
- The purpose of the Iceberg-Cube is to identify and compute only those values that will most likely be required for decision support queries.

Cube Materialization → Iceberg Cube



For a three dimensional data cube, with attributes A, B and C, the Iceberg-Cube problem may be represented as:

```
SELECT A,B,C,Count(*),SUM(X)
FROM TableName
CUBE BY A,B,C
HAVING COUNT(*)>= minsup
```

where minsup is the minimum support

Cube Materialization → Closed Cube

- A closed cube is a data cube consisting of only closed cells.
- A cell, c, is a closed cell if there exists no cell, d, such that ‘d is a descendant of cell c’ and ‘d has the same measure value as c’

Cube Materialization → Cube Shell

- The cuboids involving a small number of dimensions, e.g., 3

Data Cube Computation

- Data cube computation is an essential task in data warehouse implementation.
- The precomputation of all or part of a data cube can greatly reduce the response time and enhance the performance of online analytical processing.
- However, such computation is challenging because it may require substantial computational time and storage space.

Data Cube Computation → Number of cuboids

$$T = \prod_{i=1}^{n} (L_i + 1)$$

Where T= Total Number of Cuboids
L_i= number of levels associated with dimension i

If the cube has 10 dimensions and each dimension has 4 levels, what will be the number of cuboids generated?

Total number of cuboids= $5 \times 5 \times 5 \times 5 \times 5 \times 5 \times 5 \times 5 \times 5 \times 5 = 5^{10} \sim 9.8 \times 10^{6}$

Efficient Data Cube Computation → General Heuristics

- Sorting, hashing, and grouping
- Simultaneous aggregation and caching of intermediate results.
- Aggregation from the smallest child when there exist multiple child cuboids
- The Apriori pruning method can be explored to compute iceberg cubes efficiently.

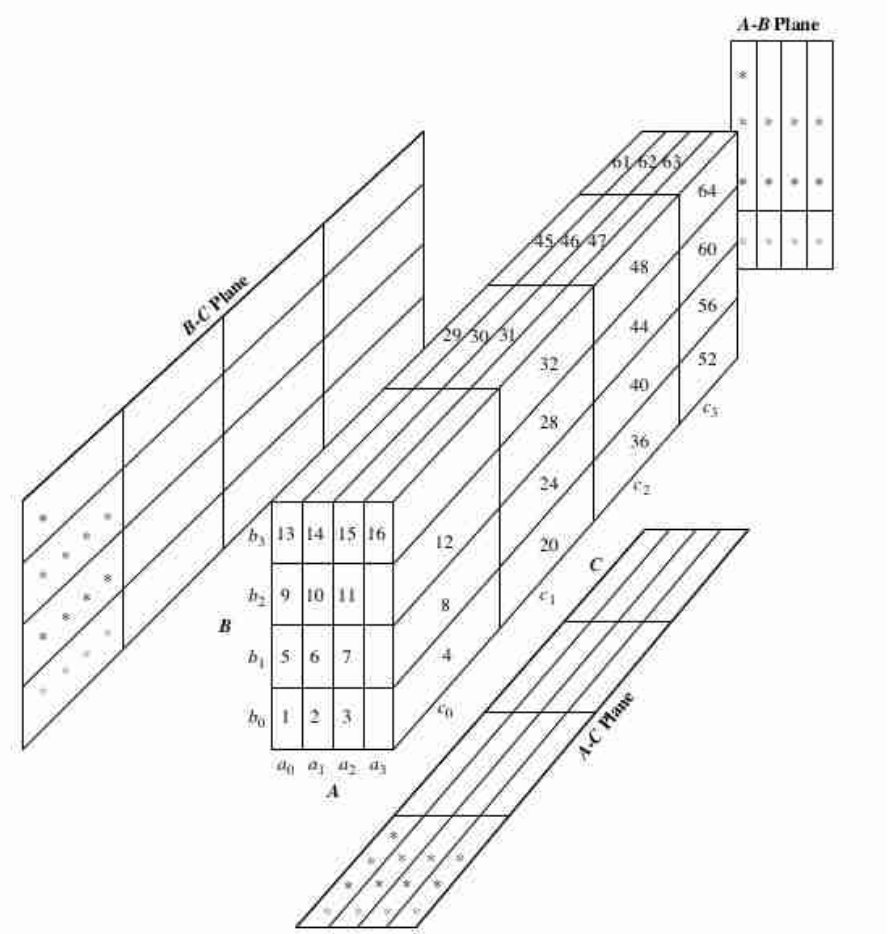
Data Cube Computation → Efficient Methods

1. Multiway Array Aggregation (MultiWay)
2. Bottom-Up Computation (BUC)
3. Star-Cubing
4. High dimension OLAP

Efficient Methods → Multiway Array Aggregation

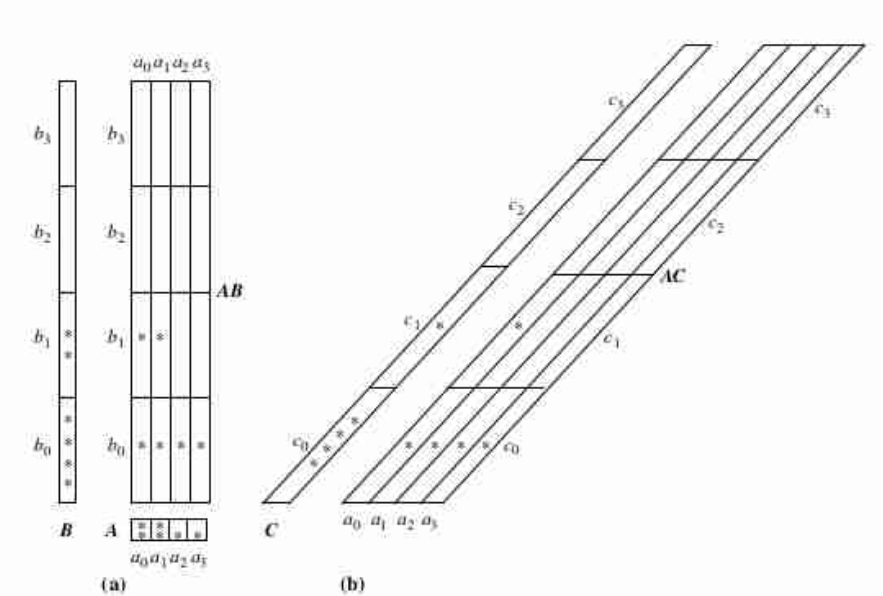
- Computes a full data cube by using a multidimensional array as its basic data structure.
- Uses direct array addressing, where dimension values are accessed with the help of position or index of their corresponding array locations.
- Cannot perform any value-based reordering as an optimization technique.
- *Steps:*
  a) **Partition the array into chunks.**
     Chunking is a method for dividing an n-dimensional array into small n-dimensional chunks, where each chunk(sub cube) is stored as an object on disk.
  b) **Compute aggregates by visiting cube cells.**
     The order in which cells are visited can be optimized so as to minimize the number of times that each cell must be revisited, thereby reducing memory access and storage costs.

Efficient Methods → Multiway Array Aggregation



- Consider a 3-D data array containing the three dimensions A, B, and C. The 3-D array is partitioned into small, memory-based chunks. The array is partitioned into 64 chunks.
- Dimension A is organized into four equalsized partitions: a_0 , a_1 , a_2 , and a_3 . Dimensions B and C are similarly organized into four partitions each.
- Chunks 1, 2, . . . , 64 correspond to the subcubes $a_0b_0c_0$, $a_1b_0c_0$, . . . , $a_3b_3c_3$, respectively.
- Suppose that the cardinality of the dimensions A, B, and C is 40, 400, and 4000, respectively. Thus, the size of the array for each dimension, A, B, and C, is also 40, 400, and 4000, respectively.

Efficient Methods → Multiway Array Aggregation



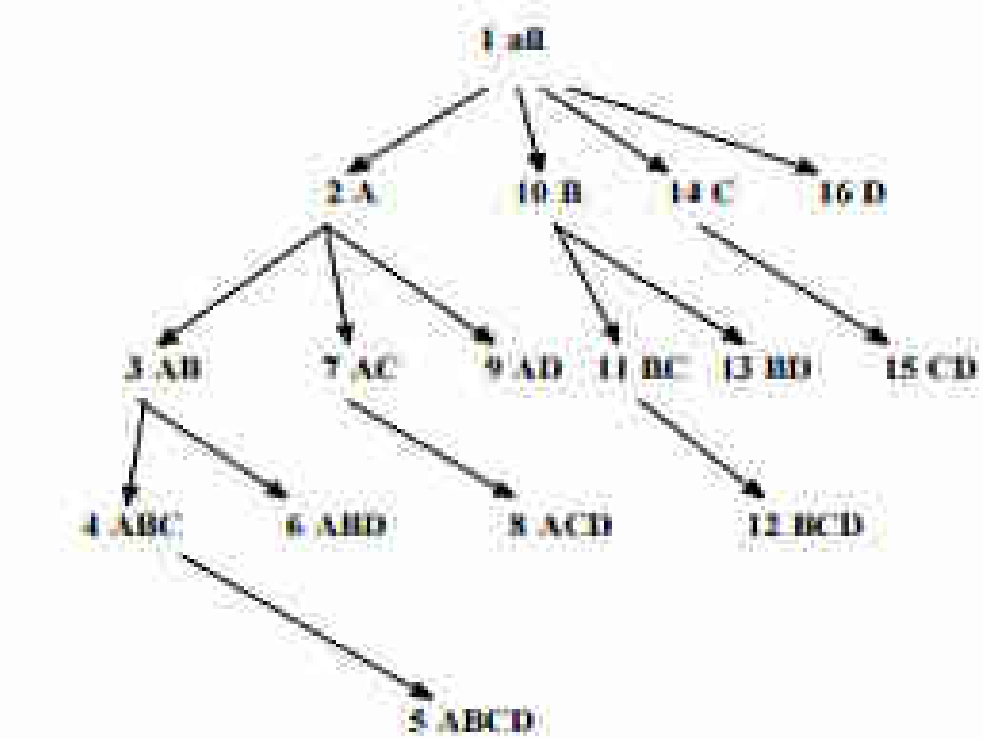
Memory allocation and computation order for computing 1-D cuboids.
(a) The 1-D cuboids, A and B, are aggregated during the computation of the smallest 2-D cuboid, AB.
(b) The 1-D cuboid, C, is aggregated during the computation of the second smallest 2-D cuboid, AC. The *'s represent chunks that, so far, have been aggregated to.

The resulting full cube consists of the following cuboids:

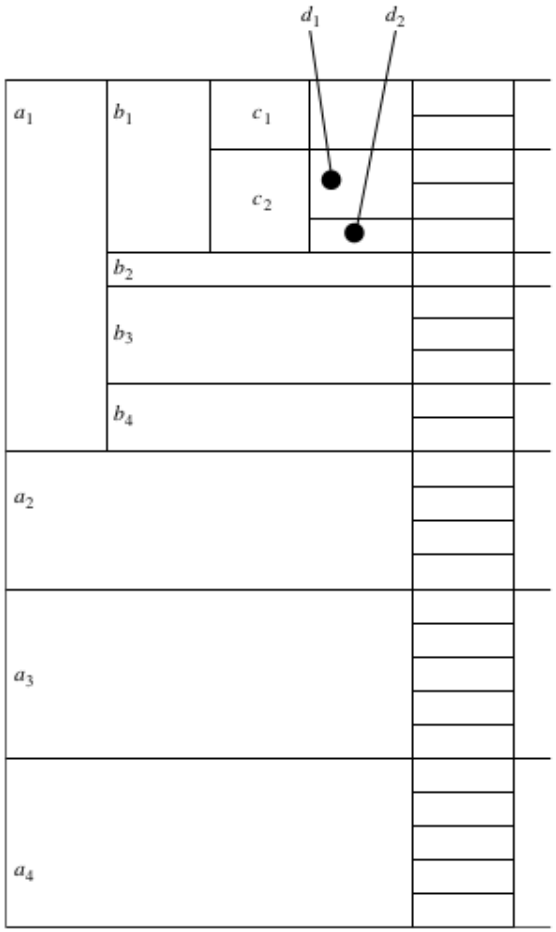
- The base cuboid, denoted by ABC (from which all the other cuboids are directly or indirectly computed). This cube is already computed and corresponds to the given 3-D array.
- The 2-D cuboids, AB, AC, and BC, which respectively correspond to the group-by's AB, AC, and BC. These cuboids must be computed.
- The 1-D cuboids, A, B, and C, which respectively correspond to the group-by's A, B, and C. These cuboids must be computed.
- The 0-D (apex) cuboid, denoted by all, which corresponds to the group-by (); that is, there is no group-by here.

Efficient Methods → Buttom Up Computation (BUC)

- BUC is an algorithm for the computation of sparse and iceberg cubes
- BUC constructs the cube from the apex cuboid toward the base cuboid. This allows BUC to share data partitioning costs.
- The BUC is viewed as lattice of cuboids in the reverse order,with the apex cuboid at the bottom and the base cuboid at the top. In that view, BUC does bottom-up construction



Efficient Methods → Buttom Up Computation (BUC)



Suppose $(a_1,*,*,*)$ satisfies the minimum support, in which case a recursive call is made on the partition for a_1.

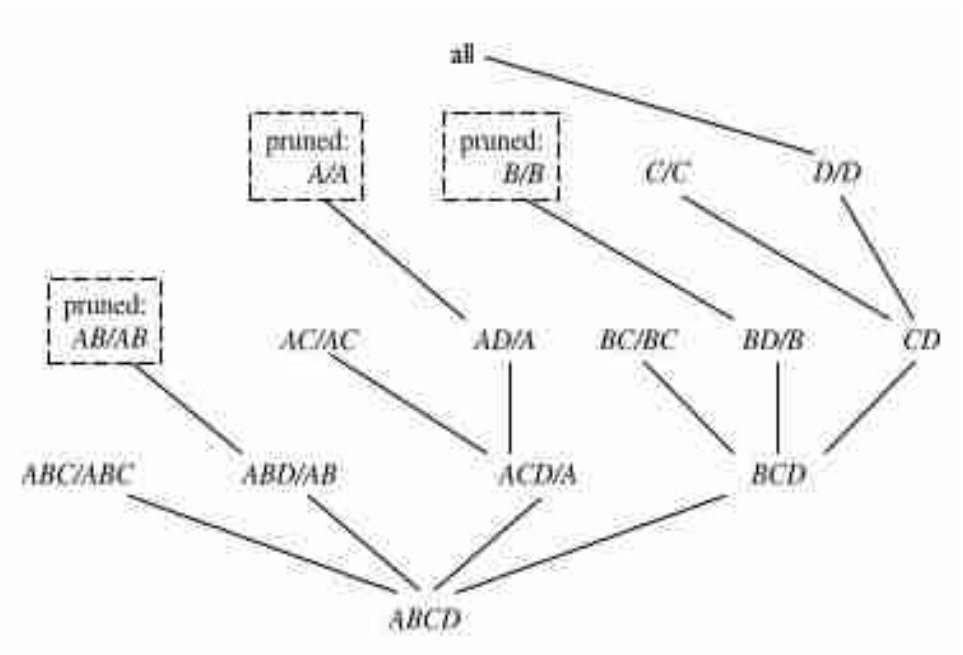
BUC partitions a_1 on the dimension B. It checks the count of $(a_1,b_1,*,*)$ to see if it satisfies the minimum support.

If it does, it outputs the aggregated tuple to the AB group-by and recurses on $(a_1,b_1,*,*)$ to partition on C, starting with c_1. Suppose the cell count for $(a_1,b_1,c_1,*)$ is 2, which does not satisfy the minimum support. According to the Apriori property, if a cell does not satisfy the minimum support, then neither can any of its descendants. Therefore, BUC prunes any further exploration of $(a_1,b_1,c_1,*)$. That is, it avoids partitioning this cell on dimension D. It backtracks to the a_1,b_1 partition and recurses on $(a_1,b_1,c_2,*)$, and so on

Efficient Methods → Star-Cubing

- Explores both the bottom-up and top-down computation models as follows:
  - On the global computation order, it uses the bottom-up model.
  - For exploring the dimension, top-down model is used.

Efficient Methods → Star-Cubing



4-D data cube computation

- Explore shared dimensions
  - Dimension A is the shared dimension of ACD and AD
  - ABD/AB means cuboid ABD has shared dimensions AB

Efficient Methods → Star-Cubing

Star Tree Construction

A	B	C	D	count
a_1	b_1	c_1	d_1	1
a_1	b_1	c_4	d_3	1
a_1	b_2	c_2	d_2	1
a_2	b_3	c_3	d_4	1
a_2	b_4	c_3	d_4	1

*A **base cuboid** table with five tuples and four dimensions. The cardinalities for dimensions A, B, C, D are 2, 4, 4, 4, respectively.*

Dimension	count = 1	count ≥ 2
A	—	$a_1(3), a_2(2)$
B	b_2, b_3, b_4	$b_1(2)$
C	c_1, c_2, c_4	$c_3(2)$
D	d_1, d_2, d_3	$d_4(2)$

The ***one dimensional aggregates*** for all attributes

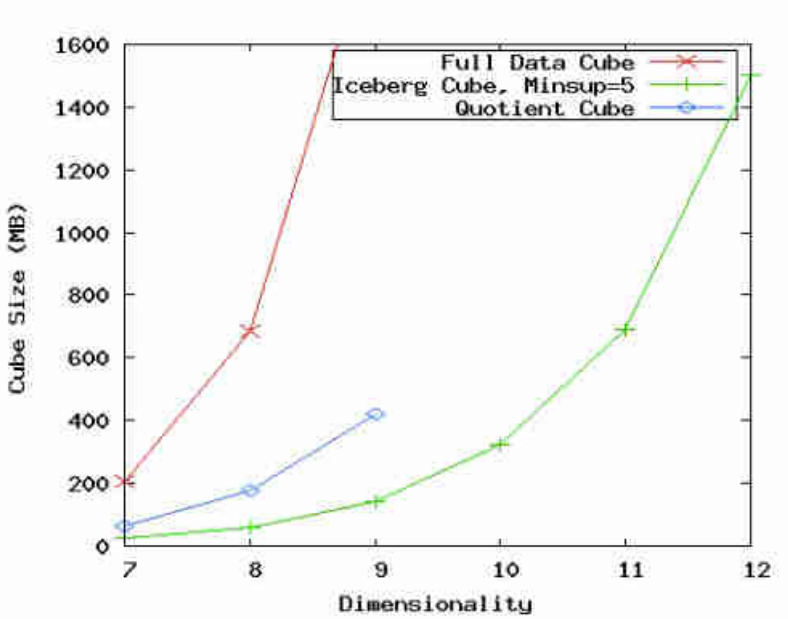
A	B	C	D	count
a_1	b_1	*	*	2
a_1	*	*	*	1
a_2	*	c_3	d_4	2

Compressed Base Table: After Star Reduction

Suppose min_{sup} = 2 in the iceberg condition. Clearly, only attribute values a1, a2, b1 ,c3 ,d4 satisfy the condition. All other values are below the threshold and thus become star-nodes.

High-Dimensional OLAP → Motivation

- High-D OLAP: Needed in many applications
  - ✗ Science and engineering analysis
  - ✗ Bio-data analysis: thousands of genes
  - ✗ Statistical surveys: hundreds of variables
- None of the previous cubing method can handle high dimensionality!
  - ✗ Iceberg cube and compressed cubes: only delay the inevitable(certain to happen) explosion
  - ✗ Full materialization: still significant overhead in accessing results on disk



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 highdimensional data cube online

Example Computation

Let the cube aggregation function be count

TID	A	B	C	D	E
1	a_1	b_1	c_1	d_1	e_1
2	a_1	b_2	c_1	d_2	e_1
3	a_1	b_2	c_1	d_1	e_2
4	a_2	b_1	c_1	d_1	e_2
5	a_2	b_1	c_1	d_1	e_3

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

Inverted Index

Attribute Value	TID List	List Size
a_1	{1, 2, 3}	3
a_2	{4, 5}	2
b_1	{1, 4, 5}	3
b_2	{2, 3}	2
c_1	{1, 2, 3, 4, 5}	5
d_1	{1, 3, 4, 5}	4
d_2	{2}	1
e_1	{1, 2}	2
e_2	{3, 4}	2
e_3	{5}	1

to compute the cell (a 1 , b 2 , *), we intersect the TID lists of a 1 and b 2 to obtain a new list of {2, 3}.

Cuboid *AB*

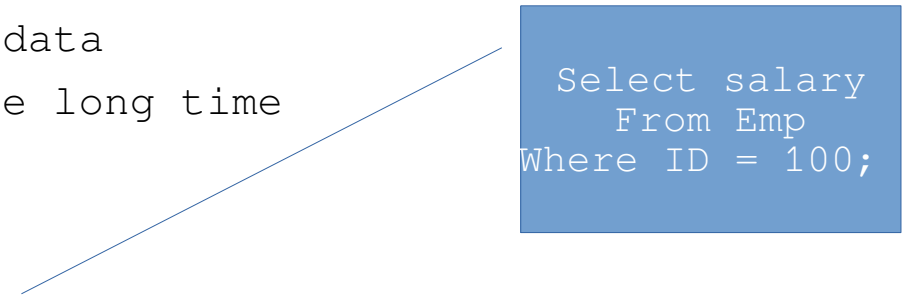
Cell	Intersection	TID List	List Size
(a_1, b_1)	{1, 2, 3} ∩ {1, 4, 5}	{1}	1
(a_1, b_2)	{1, 2, 3} ∩ {2, 3}	{2, 3}	2
(a_2, b_1)	{4, 5} ∩ {1, 4, 5}	{4, 5}	2
(a_2, b_2)	{4, 5} ∩ {2, 3}	{}	0

Cuboid *DE*

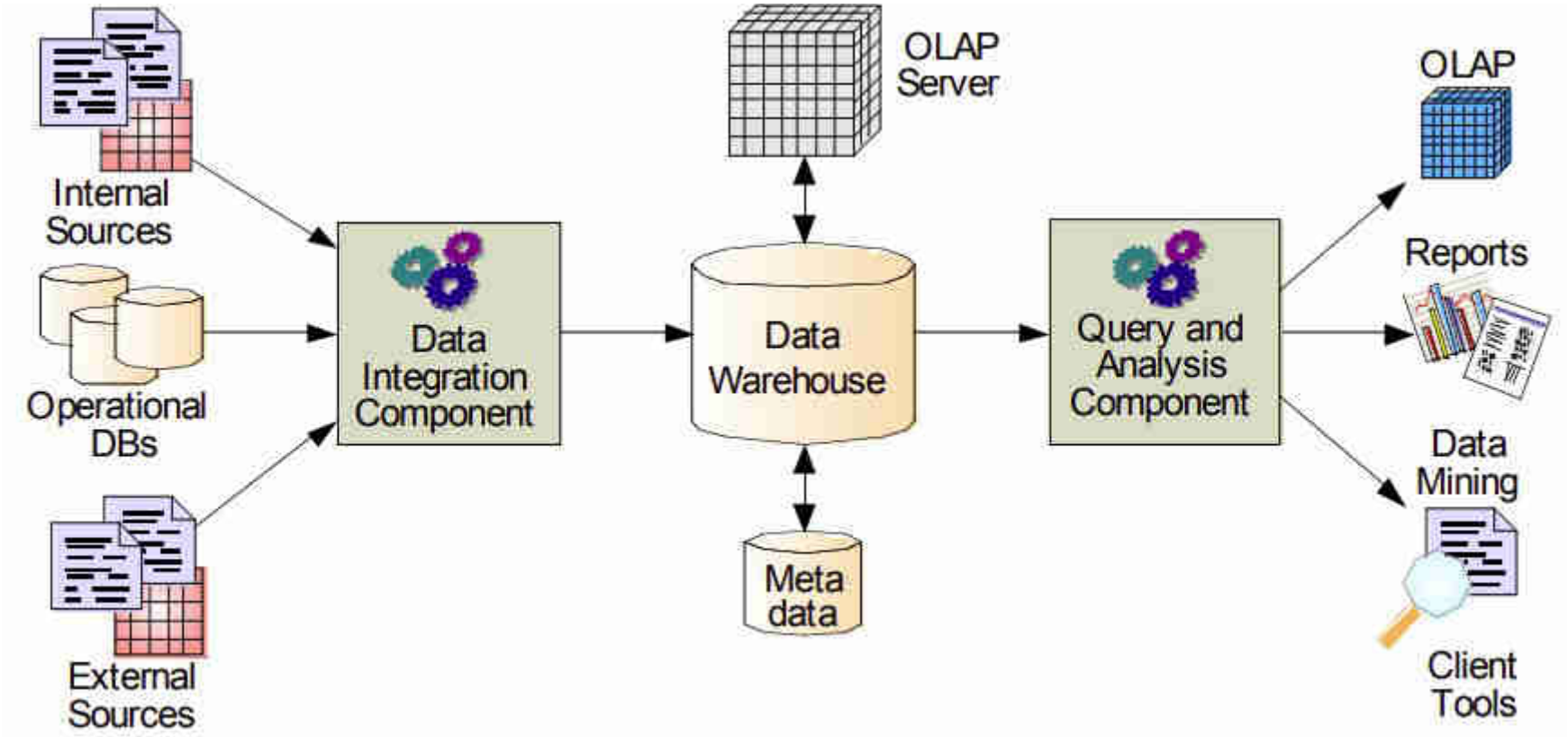
Cell	Intersection	TID List	List Size
(d_1, e_1)	{1, 3, 4, 5} ∩ {1, 2}	{1}	1
(d_1, e_2)	{1, 3, 4, 5} ∩ {3, 4}	{3, 4}	2
(d_1, e_3)	{1, 3, 4, 5} ∩ {5}	{5}	1
(d_2, e_1)	{2} ∩ {1, 2}	{2}	1

OLAP Queries

- OLAP: Online Analytic Processing
- OLAP queries are complex queries that
  - Touch large amounts of data
  - Discover patterns and trends in the data
  - Typically expensive queries that take long time
  - Also called decision-support queries
- In Contrast :
  - OLTP: Online Transaction Processing
  - OLTP queries are simple queries, e.g., over banking or airline systems
  - OLTP queries touch small amount of data for fast transactions



OLAP AND DATA WAREHOUSE



OLAP Queries

- Typically, OLAP queries are executed over a separate copy of the working data → Over data warehouse
- Data warehouse is periodically updated, e.g., overnight → OLAP queries tolerate such out-of-date gaps
- **Why run OLAP queries over data warehouse**?
  - ✗ Warehouse collects and combines data from multiple sources
  - ✗ Warehouse may organize the data in certain formats to support OLAP queries
  - ✗ OLAP queries are complex and touch large amounts of data
    - They may lock the database for long periods of time
    - Negatively affects all other OLTP transactions

OLAP queries in SQL

Consider the following sales table

PRODUCT	QUARTER	REGION	SALES
A	Q1	Europe	10
A	Q1	America	20
A	Q2	Europe	20
A	Q2	America	50
A	Q3	America	20
A	Q4	Europe	10
A	Q4	America	30
B	Q1	Europe	40
B	Q1	America	60
B	Q2	Europe	20
B	Q2	America	10
B	Q3	America	20
B	Q4	Europe	10
B	Q4	America	40

```
SELECT QUARTER, REGION, SUM(SALES)
FROM SALESTABLE
GROUP BY CUBE (QUARTER, REGION)
```

QUARTER	REGION	SALES
Q1	Europe	50
Q1	America	80
Q2	Europe	40
Q2	America	60
Q3	Europe	NULL
Q3	America	40
Q4	Europe	20
Q4	America	80
Q1	NULL	130
Q2	NULL	100
Q3	NULL	40
Q4	NULL	90
NULL	Europe	110
NULL	America	250
NULL	NULL	360

Online Query Computation: Query

- A query has the general form : <a1,a2,…..,a_n: M>
- Each a_i has 3 possible values
  1. Instantiated value
  2. Aggregate (*) function
  3. Inquire (?) function
- For example, <3,*,1> returns a 2-D data cube.

Multidimensional OLAP queries

- OLAP queries and tools in data warehouse assist in reports and analysis processing.
- The basic OLAP query language is MDX.
- Multidimensional Expressions is a query language for getting access to multidimensional data structures.
- MDX queries work with OLAP members, dimensions and hierarchies. With MDX it is possible to query data from SQL server and get a dataset with axis and cell data.

Multidimensional OLAP queries

- **SELECT**: the principle MDX statement. SELECT points out the dimension members that will be parts of axis;
- **WHERE**: identifies the dimension or member that are used as slicer dimensions;
- **WITH**: calculates the sets named during SELECT and WHERE clauses processing;
- **FROM**: determines the source of multidimensional data to which our query is addressed to.

```
SELECT
{[Measures].[Unit Sales], [Measures].[Store Sales] } ON COLUMNS,
{[TIME].[1997], [TIME].[1998]} ON ROWS
FROM Sales
WHERE ( [Store].[USA].[CA] )
```

Data Warehouse back end tools

Data warehouse systems use back-end tools and utilities to populate and refresh their data. These tools and utilities include the following functions

- **Data extraction**, which typically gathers data from multiple, heterogeneous, and external sources
- **Data cleaning**, which detects errors in the data and rectifies them when possible
- **Data transformation**, which converts data from legacy or host format to warehouse format
- **Load**, which sorts, summarizes, consolidates, computes views, checks integrity, and builds indices and partitions
- **Refresh**, which propagates the updates from the data sources to the warehouse

Data Warehouse Tuning

- The process of applying different strategies in performing different operations of data warehouse such that performance measures will enhance is called data warehousing tuning.
- We can tune the different aspects of a data warehouse such as performance, data load, queries, etc.
- A data warehouse keeps evolving and it is unpredictable what query the user is going to post in the future. Therefore it becomes more difficult to tune a data warehouse system.

Data Warehouse Tuning

Tuning a data warehouse is a difficult procedure due to following reasons:

- Data warehouse is dynamic; it never remains constant.
- It is very difficult to predict what query the user is going to post in the future.
- Business requirements change with time.
- Users and their profiles keep changing.
- The user can switch from one group to another.
- The data load on the warehouse also changes with time.

Testing in Data warehouse

- The process of creating and running detailed test cases to ensure that data in a warehouse is trustworthy, accurate, and compatible with the organization's data structure is known as data warehouse testing.
- Because of the rising emphasis on data analytics and the way complicated business insights are recognized on the assumption that data is reliable, Testing is critical for modern firms.

Data warehouse testing process

- **Identify the various entry points** : If testing is done only at the destination, it can be confusing when errors are found as it becomes more difficult to determine the root cause.
- **Prepare the required collaterals** : Two fundamental collaterals required for the testing process are database schema representation and a mapping document.
- **Design an elastic, automated, and integrated testing framework** :
- **Adopt a comprehensive testing approach** : Application components such as ETL tools, reporting engines, or GUI applications need to be included in the testing framework. Also, it's important to design multiple testing approaches such as unit, integration, functional, and performance testing.

Testing in Data warehouse

Data Warehouse stores huge amount of data, which is typically collected from multiple heterogeneous source like files, DBMS, etc to produce statistical result that help in decision making.

Testing is very important for data warehouse systems for data validation and to make them work correctly and efficiently.

There are three basic levels of testing performed on data warehouse which are as follows:

- Unit Testing
- Integration Testing
- System Testing

Testing in Data warehouse → Unit Testing

- This type of testing is being performed at the developer's end.
- In unit testing, each unit/component of modules is separately tested.
- Each modules of the whole data warehouse, i.e. program, SQL Script, procedure,, Unix shell is validated and tested.

Testing in Data warehouse → Integration Testing

- In this type of testing the various individual units/ modules of the application are brought together or combined and then tested against the number of inputs.
- It is performed to detect the fault in integrated modules and to test whether the various components are performing well after integration.

Testing in Data warehouse → System Testing

- System testing is the form of testing that validates and tests the whole data warehouse application.
- This type of testing is being performed by technical testing team.
- This test is conducted after developer's team performs unit testing and the main purpose of this testing is to check whether the entire system is working altogether or not.