Advanced Data Warehousing: OLAP and Cube Computation

Prerequisites: Data warehouse basics, Star/Snowflake schemas

Duration: 1 Hour

Introduction & The ‘Amazon’ Scenario

The Scenario:

You are an analyst at Amazon with a 5-dimensional data cube (Products × Time × Region × Customer Segment × Sales Channel).

The CEO’s Query:

“Show me laptop sales growth from Q3 to Q4 in the Northeast, broken down by customer age group.”

Goal:

Understand how these queries work “under the hood”.

Learning Objectives:

- Understand concept hierarchies.
- Categorize measures and aggregates.
- Master OLAP operations and indexing.
- Design cube computation strategies and architectures.

Part 1 - Concept Hierarchies

Definition:

A sequence of mappings from lower-level to higher-level concepts within a dimension.

Types of Hierarchies:

Schema Hierarchies

Explicitly defined (e.g., City → State → Country)

Set-Grouping

Range-based groupings (e.g., Age 0-12 = 'Child', 13-19 = 'Teen')

Operation-Derived

Parsed from data (e.g., extracting Year from a timestamp or City from an address)

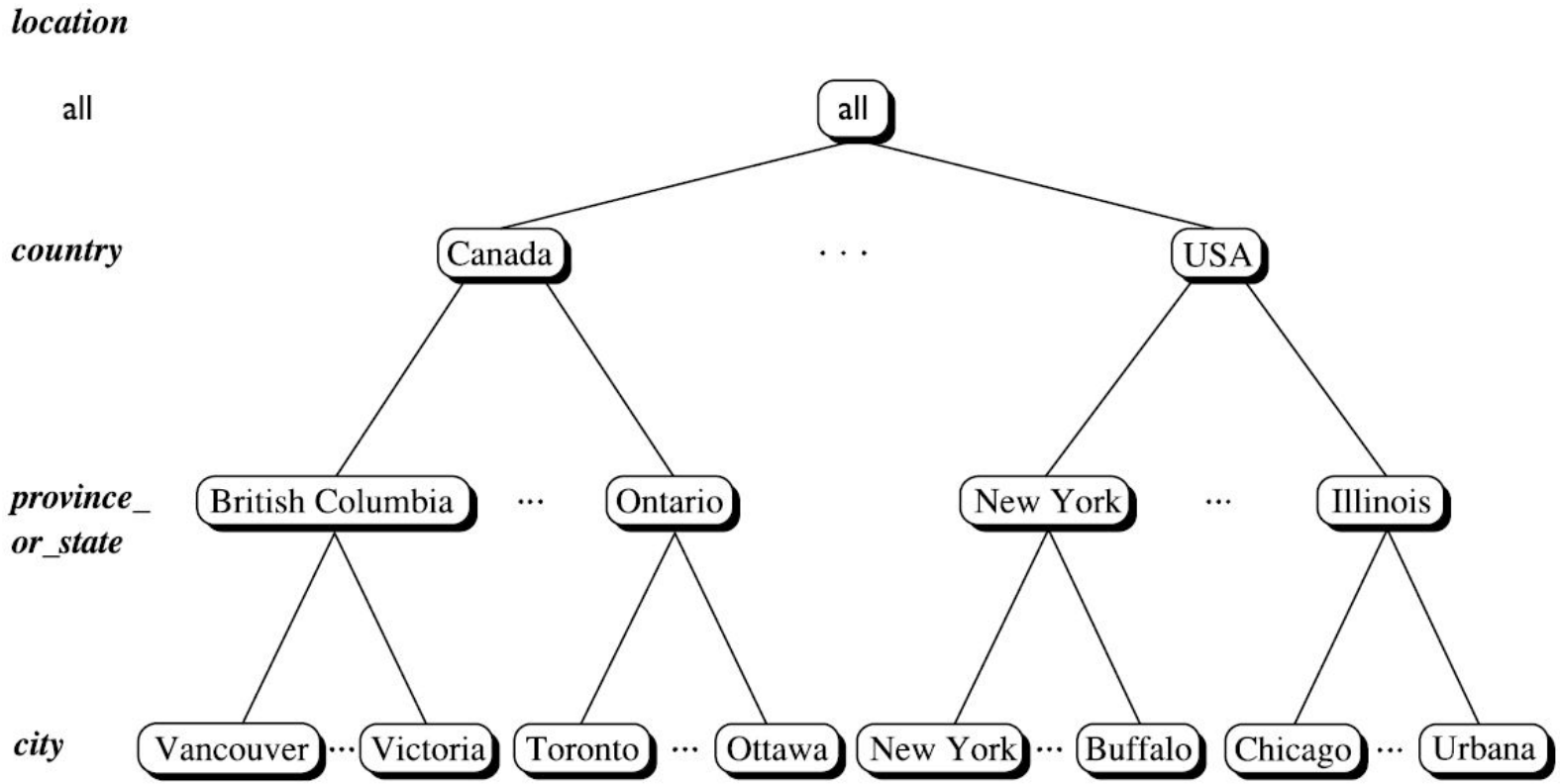


FIGURE 3.10

A concept hierarchy for *location*. Due to space limitations, not all of the hierarchy nodes are shown, indicated by ellipses between nodes.

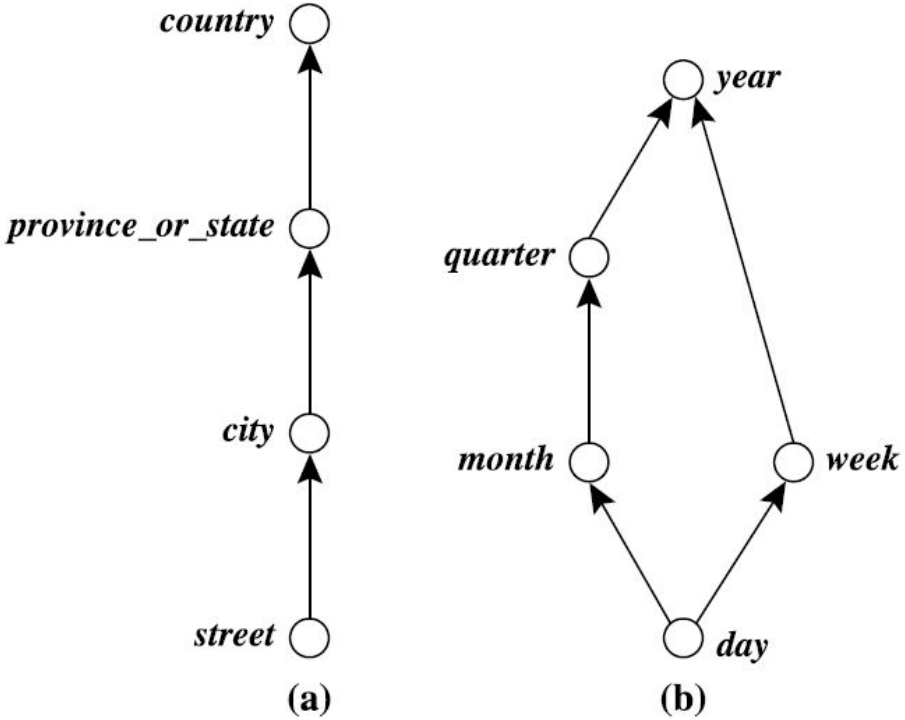


FIGURE 3.11

Hierarchical and lattice structures of attributes in warehouse dimensions: (a) a hierarchy for *location* and (b) a lattice for *time*.

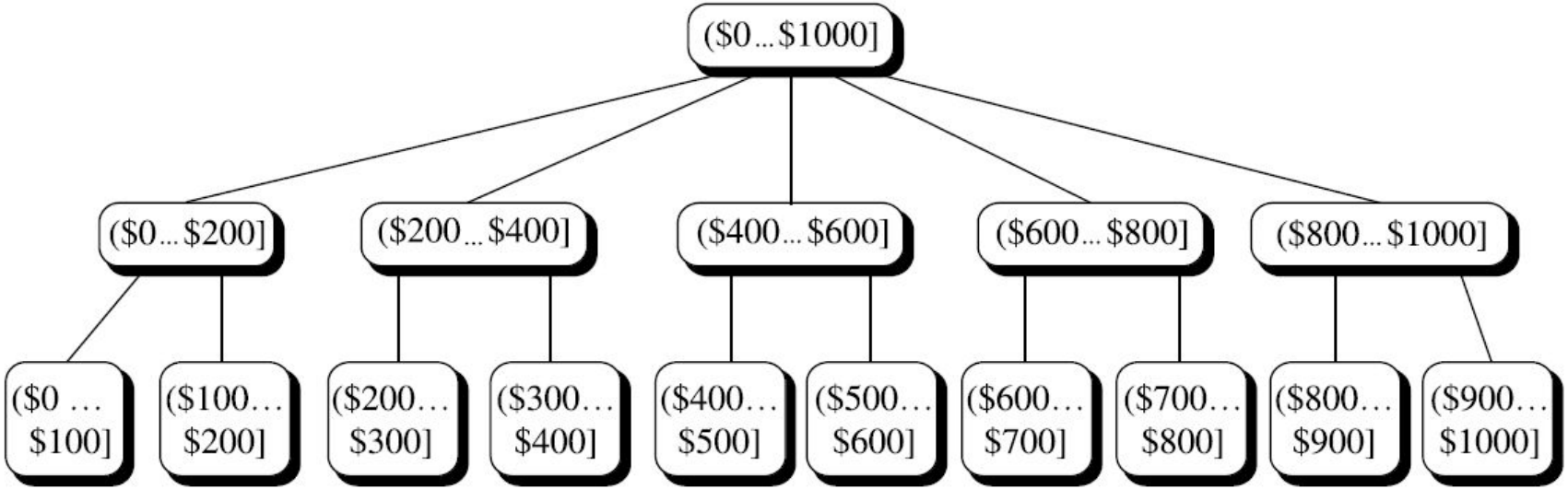


FIGURE 3.12

A concept hierarchy for *price*.

```
Product Dimension Hierarchy:
Level 1: SKU (individual item: "iPhone 14 Pro 256GB Black")
Level 2: Product ("iPhone 14 Pro")
Level 3: Product Line ("iPhone 14 Series")
Level 4: Category ("Smartphones")
Level 5: Department ("Electronics")

Level 6: Division ("Consumer Goods")
```

Hierarchy Structure & Ordering

Retail Example:

Division → Department → Category → Product Line → Product → SKU

Ordering Types:

Total Order

Strict hierarchy where every child has exactly one parent (e.g., City → State)

Partial Order

Multiple parent possibilities (e.g., Employees might belong to a Department and a Location)

Part 2 - Measures & Computation

Additive Measures

Can be summed across ALL dimensions (e.g., Sales Amount, Quantity Sold).

Semi-Additive Measures

Can be summed across SOME dimensions (e.g., Bank Balance sums across accounts, but not across time).

Computation: SUM() for spatial dims, AVG()/LAST() for temporal dims.

Non-Additive Measures

Cannot be summed (e.g., Unit Price, Temperature, Profit Margin).

Computation: Requires AVG, MIN, MAX, or derived formulas.

Sales Amount: $100(NY) + $150(CA) = $250(Total)

Quantity Sold: 50 units + 30 units = 80 units

- Sum across accounts: Acc1($1000) + Acc2($2000) = $3000 ✓
- Sum across time: Jan1($1000) + Jan2($1100) ≠ $2100 ✗

Inventory Count:

- Sum across warehouses: WH1(500) + WH2(300) = 800 ✓
- Sum across time: Day1(500) + Day2(480) ≠ 980 ✗

Unit Price: $10/item + $15/item ≠ $25/item

Temperature: 20°C + 25°C ≠ 45°C

Profit Margin: 15% + 20% ≠ 35%

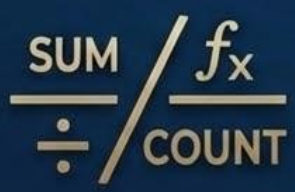
Aggregate Functions Types

Distributive

Can be computed in partitions and combined (e.g., SUM, COUNT, MAX).

SUM([1,2]) + *SUM*([3,4,5]) = *SUM*(Total)

Algebraic



Computed from distributive functions (e.g., AVG = SUM/COUNT)

Holistic

Must process the entire dataset at once; cannot use partial results (e.g., MEDIAN, MODE, RANK)

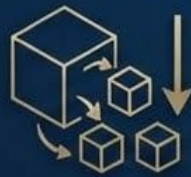
```
Sales Data:
Region  Product  Sales
NY      Laptop   $1000
NY      Phone    $500
CA      Laptop   $800
CA      Phone    $700

Measures:
- Total Sales (additive): SUM = $3000
- Average Sale (non-additive): AVG = $3000/4 = $750

- Median Sale (holistic): Sorted [$500, $700, $800, $1000] → Median = $750
```

Part 3 - OLAP Operations (The Cube)

Roll-up (Drill-up)

Increase aggregation (e.g., Day → Month)

Drill-down



Decrease aggregation for detail (e.g., Region → City)

Slice

Select one specific dimension member (e.g., Region = 'Northeast')

Dice

Select a sub-cube with multiple conditions (e.g., Laptop OR Tablet AND Q1 OR Q2)

Pivot (Rotate)

Reorient the cube axes (e.g., swapping Rows and Columns)

Drill-through

Access underlying transaction details

Example 3.4. OLAP operations. Let us look at some typical OLAP operations for multidimensional data. Each of the following operations is illustrated in Fig. 3.13. At the center of the figure is a data cube for sales in a company. The cube contains three dimensions, *location, time*, and *item*, where *location* is aggregated with respect to city values, *time* is aggregated with respect to quarters, and *item* is aggregated with respect to item types. To aid in our explanation, we refer to this cube as the central cube. The measure displayed is *dollars_sold* (in thousands). (For the sake of readability, only some cell values in the cubes are shown.) The data examined are for the cities Chicago, New York, Toronto, and Vancouver.

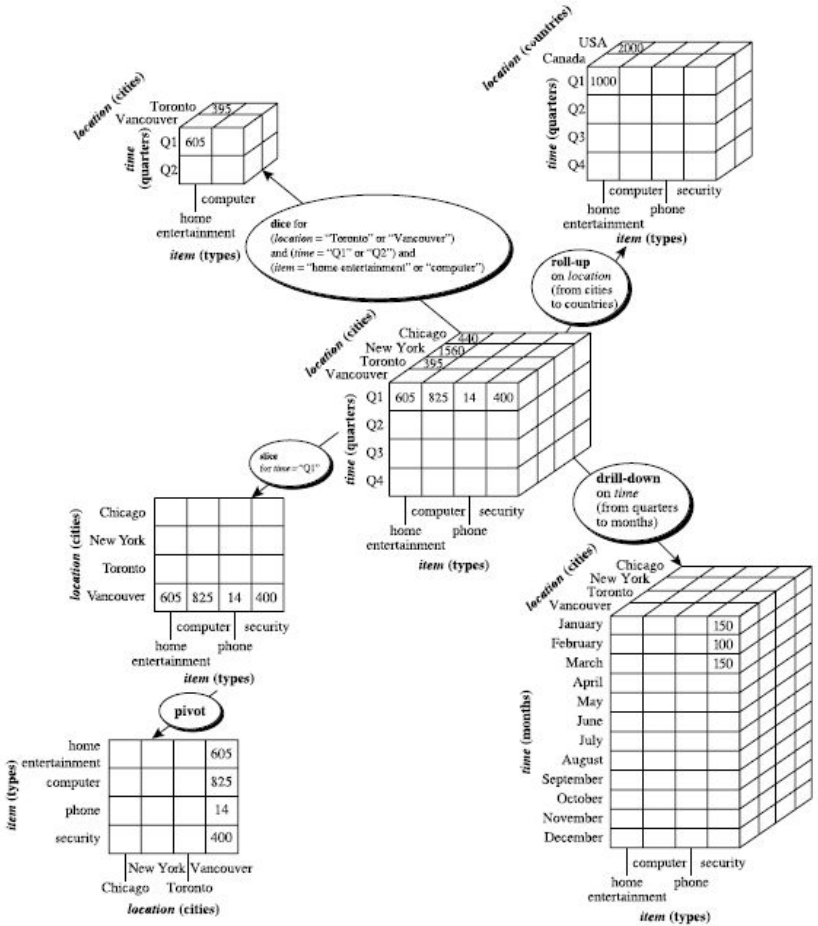


FIGURE 3.13

Examples of typical OLAP operations on multidimensional data.

SQL Equivalents for OLAP

Roll-up

Removing columns from GROUP BY to increase granularity.
GROUP BY region, year (Broad) vs GROUP BY store, month (Detailed).

Drill-down

Adding columns to GROUP BY.

Slice

Using a simple WHERE clause.

Dice

Using multiple AND / IN conditions in WHERE.

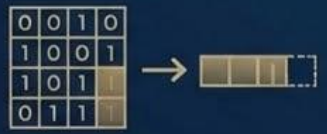
```sql
-- Roll-up: Increase GROUP BY granularity
SELECT region, year, SUM(sales) FROM fact
GROUP BY region, year;  -- Was: store, month

-- Drill-down: Decrease GROUP BY granularity
SELECT store, month, product, SUM(sales) FROM fact
GROUP BY store, month, product;  -- Was: region, quarter

-- Slice: WHERE clause
SELECT * FROM fact WHERE region = 'Northeast';

-- Dice: Multiple WHERE conditions
SELECT * FROM fact
WHERE product IN ('Laptop','Tablet')
  AND quarter IN ('Q1','Q2')

  AND region IN ('NE','MW');
```

Indexing for OLAP

Problem

Traditional B-trees are inefficient for aggregations and many WHERE clauses.

Bitmap Indexing



- Uses bit vectors (0s and 1s) for low-cardinality columns (e.g., Gender, Category).
- Allows fast logical operations (AND/OR) and high compression.

Join Indexing

- Pre-computes joins between Fact and Dimension tables to avoid runtime joins.
- Can be Traditional or Bitmap Join Index.

Example 3.5. Bitmap indexing. Consider a customer information table shown in Fig. 3.14, where there is an attribute `gender`. To keep our discussion simple, assume there are two possible values on attribute `gender`. We may use one character, that is, 8 bits, for each record to represent the `gender` value, such as *F* for female and *M* for male. Bitmap index represents the `gender` value using one bit, such as 0 for female and 1 for male. This representation immediately brings in an eight-fold saving in storage.

Product Dimension:

Product_ID	Laptop	Phone	Tablet	Accessory
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

```
"Sales of Laptops in Q1"
Bitmap(Laptop) AND Bitmap(Q1) → Result bitmap

COUNT(result) → Number of sales
```

Customer information

Name	...	Gender	...
Ada	...	Female	...
Bob	...	Male	...
Cathy	...	Female	...
Dan	...	Male	...
Elsa	...	Female	...
Flora	...	Female	...
George	...	Male	...
Hogan	...	Male	...
...	...	...	...

Bitmap index

Gender
0
1
0
1
0
0
1
1
...

A table counting 0s in bytes

Byte	Number of 0s
00000000	8
00000001	7
00000010	7
00000011	6
...	...
01010011	4
...	...

FIGURE 3.14

Indexing OLAP data using bitmap indices.

Example 3.6. Bit-sliced indexing. Suppose we want to compute the sum of the amount attribute in the fact table in Fig. 3.15. We can write an amount into an integer number of pennies and then represent it as a binary number of n bits. If we represent an amount using 32 bits, that is, 4 bytes, it is good for amounts up to \$42,949,672.96 and sufficient for many application scenarios.

After we represent all amount numbers in binary, we can build a bitmap index for every bit. To compute the sum of all amounts, we count for each bit the number of 1s. Denote by x_i $(i \geq 0)$ the number of 1s in the ith bits of the amounts from right to left, the rightmost being bit 0. Since a 1 at the ith bit carries a weight of 2^i pennies, the x_i 1s in the ith bits of all amounts represent $x_i \cdot 2^i$ pennies in the sum of the amounts. Therefore, the sum of amounts is $\sum_{i \geq 0} x_i \cdot 2^i$ pennies or $\frac{\sum_{i \geq 0} x_i \cdot 2^i}{100}$ dollars. □

A fact table

...	...	Amount
...	...	15.21
...	...	27.06
...	...	...

The bit-sliced index

Weights											
2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	1	1	1	1	0	0	0	1
1	0	1	0	1	0	0	1	0	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...

FIGURE 3.15

Indexing OLAP data using bitmap indices.

Example 3.7. Join indexing. In Example 3.1, we defined a star schema of the form "*sales_star* [*time, item, branch, location*]: *dollars_sold* = `sum` (*sales_in_dollars*)." An example of a join index between the *sales* fact table and the *locations* and *items* dimension tables is shown in Fig. 3.16. Consider the OLAP query "the total sales of smartphone and desktop in BC." If no index presents, then we have to

join the fact table and the dimension tables *locations* and *items* and select only those join results about "smartphone" and "desktop."

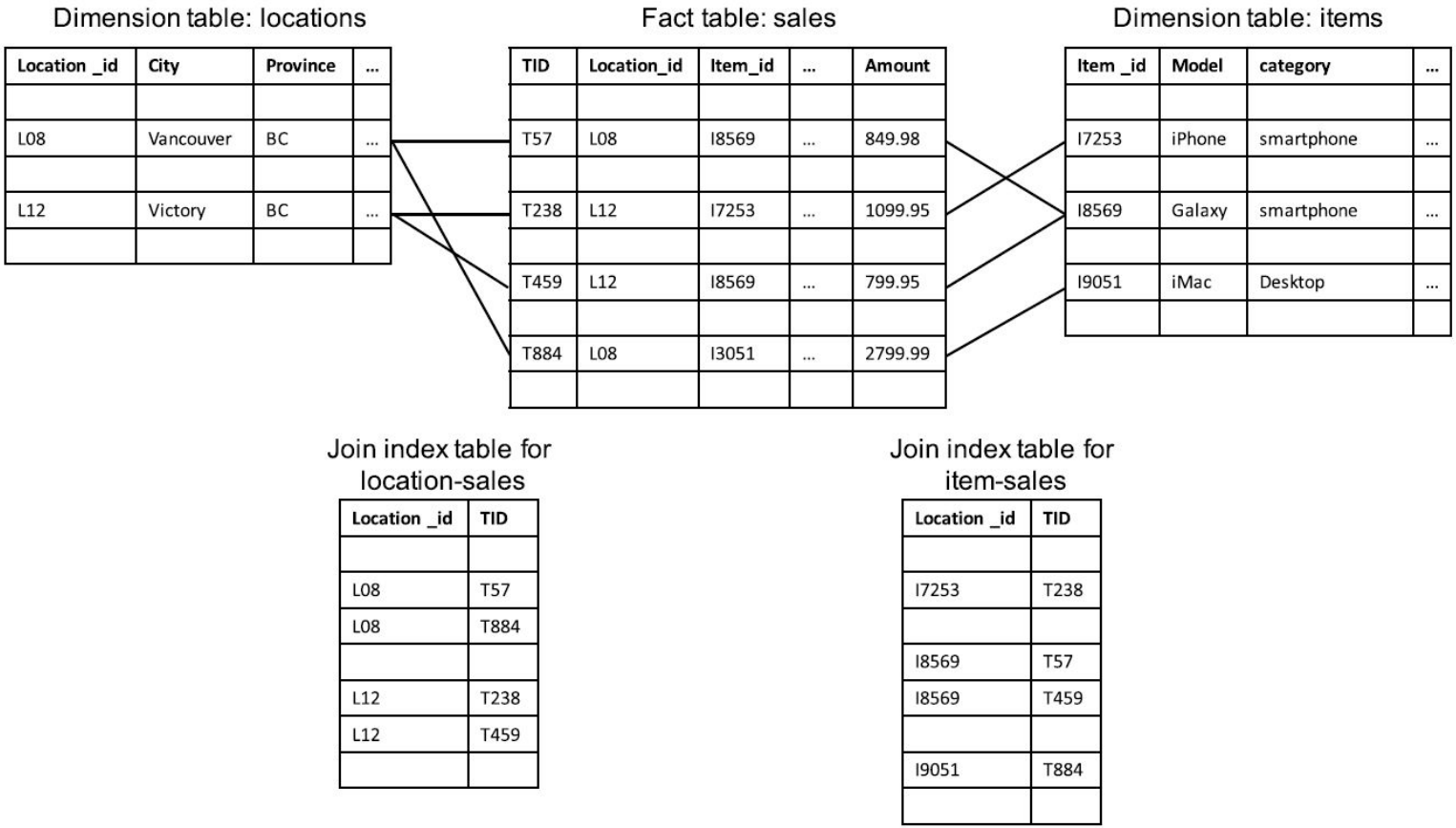
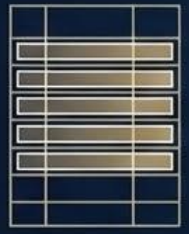


FIGURE 3.16

Join index.

```sql
-- Traditional join index
CREATE INDEX sales_product_idx ON sales(product_id)
INCLUDE (product_name, category);

-- Bitmap join index
CREATE BITMAP INDEX sales_laptop_idx ON sales(s.product_id)
FROM sales s, products p
WHERE s.product_id = p.product_id
  AND p.category = 'Laptop';
```

Column-Based Storage (Modern Standard)



Row Store (Traditional)

Stores data record-by-record.
Good for transactions (OLTP).

Column Store (OLAP Optimized)

Stores each attribute continuously.

Why Column Store?

Compression: Similar values compress well.

Vector Processing: Process whole columns at once.

Aggregation Friendly: SUM(Amount) only reads the Amount column, ignoring others.

Example 3.8. Column-based database. Consider a fact table about customer information, which includes attributes and storage space in number of bytes `customer_id` (2), `last_name` (20), `first_name` (20), `gender` (1), `birthdate` (2), `address_street` (50), `address_city` (2), `address_province` (1), `address_country` (1), `email` (30), `registration_date` (2), and `family_income` (2). Each record occupies 133 bytes. If the fact table contains 10 million customer records, then the total space is over 1 GB.

If the data are stored row by row and we want to answer the OLAP query of the average family income of female customers by province, then we have to scan the whole table, reading all records. The I/O cost is 1 GB. At the same time, for each record, we only need to use 4 bytes among the 133 bytes, that is, attributes `gender`, `address_province`, and `family_income`. In other words, only $\frac{4}{133} = 3\%$ of the data read are useful to answer the query.

A column-based database stores the data attribute by attribute in column, as shown in Fig. 3.17. To answer the above query, a column-based database only needs to read three columns, `gender`, `address_province`, and `family_income`. It checks the values on `gender` and increments the total and count for `address_province` accordingly. Overall, the total amount of I/O incurred to a column-based database in this case is $4\times$ 10 million $=$ 40 MB. A huge saving in I/O is achieved.

In implementation, preferably a column-based database processes a column at a time and uses bitmaps to keep the intermediate results so that they can be passed to the next column. In this example, we can first process the column `gender` and use a bitmap to keep the list of female customers. That is, each customer is associated with a bit, female being 0 and male being 1. Next, we can process the column `address_province`, and form a bitmap for each province. If a customer lives in BC, for example, then the associated bit in the bitmap of BC is set to 1, otherwise, it is set to 0. Last, to calculate the average family income of customers in BC, we only need to conduct the bitwise AND operation between the bitmap for gender and the bitmap for province BC. The resulting bitmap is used to select the entries in column `family_income` to calculate the average. □

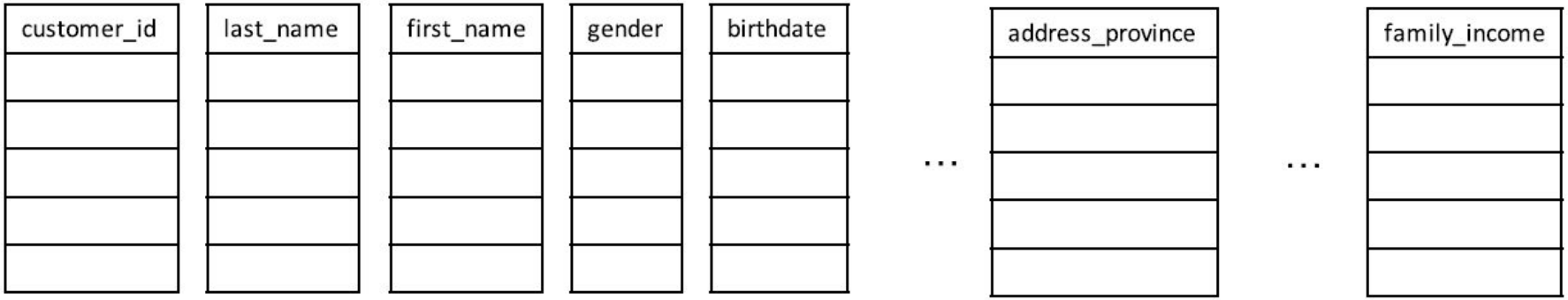


FIGURE 3.17

Column-based storage.

Part 4 - Data Cube Computation

Terminology

- **Base Cuboid:** Lowest level of detail (n-dimensions)

- **Apex Cuboid:** Highest aggregation (0-dimensions / “All”)

- **Lattice:** The set of all 2^n possible cuboids

Materialization Strategies

- **Full Materialization:** Pre-compute everything. Fast query, huge storage ($O(2^n)$)

- **No Materialization:** Compute on-the-fly. Slow query, zero storage

- **Partial Materialization:** The balance. Includes “Iceberg Cubes” (store only if count > threshold) or “Shell Cubes”

```
Dimensions: A, B, C
Cuboids:
- 3D: ABC (base)
- 2D: AB, AC, BC
- 1D: A, B, C
- 0D: ∅ (apex)

Total: 8 = 2³ cuboids
```

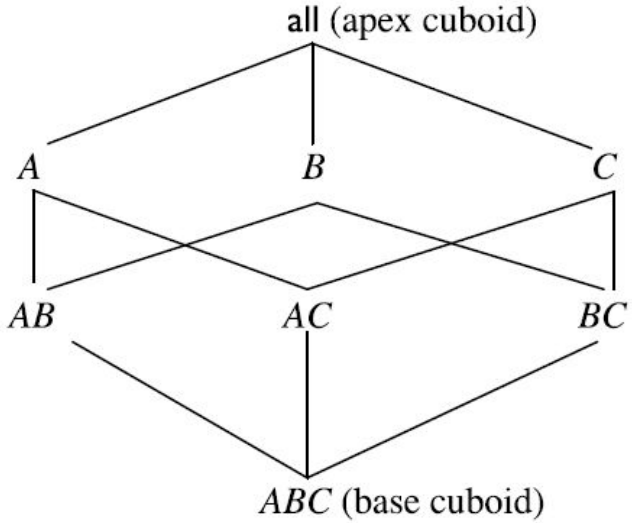


FIGURE 3.18

Lattice of cuboids making up a 3-D data cube with the dimensions A, B, and C for some aggregate measure, M.

Example 3.9. Base and aggregate cells. Consider a data cube with three 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, since the data cube has 3 dimensions, all base cells are 3-D, whereas 1-D and 2-D cells are aggregate cells.

$(month, city, *)$ is a 2-D cuboid, which contains all 2-D cells having non-$*$ values on attributes *month* and *city*. The base cuboid $(month, city, customer_group)$ contains all base cells. The apex cuboid `ALL` contains only one 0-D cell $(*, *, *)$. □

Example 3.10. Ancestor and descendant cells. Referring to Example 3.9, 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. □

Thus, for an n-dimensional data cube, the total number of cuboids that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$Total\ number\ of\ cuboids = \prod_{i=1}^{n} (L_i + 1), \tag{3.1}$$

where L_i is the number of levels associated with dimension i.

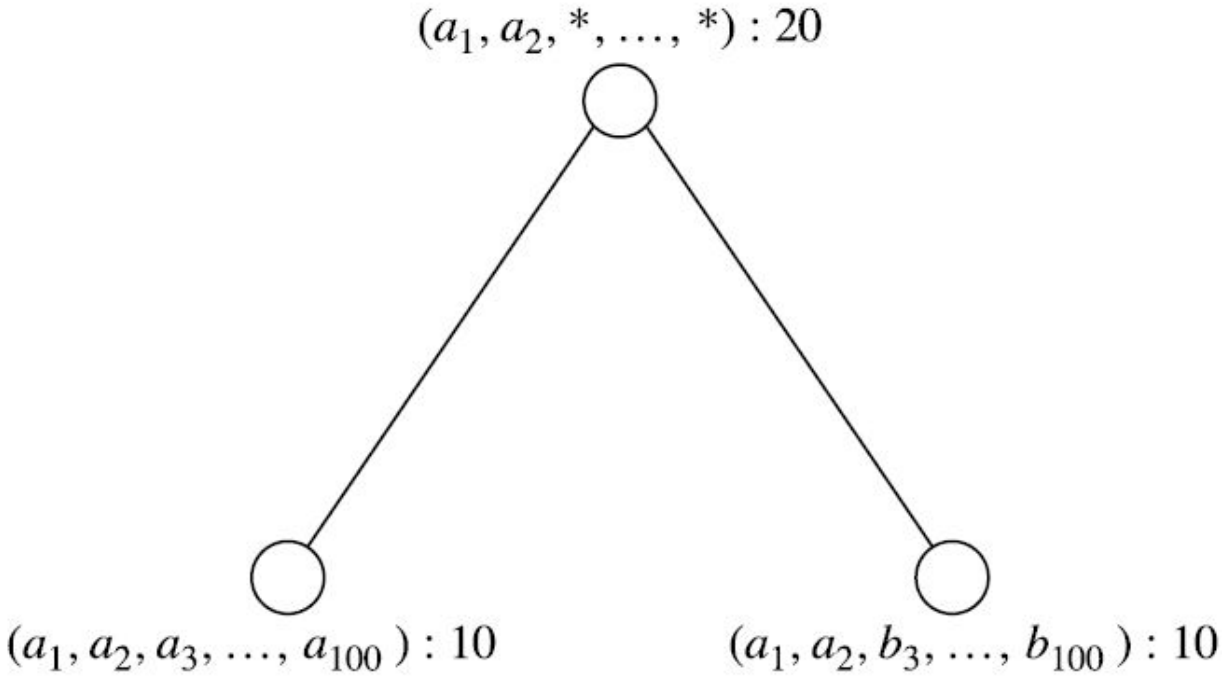


FIGURE 3.19

Three closed cells forming the lattice of a closed cube.

Example 3.11. Iceberg cube. Consider the following iceberg cube query.

```
compute cube sales_iceberg as
select month, city, customer_group, count(*)
from salesInfo
cube by month, city, customer_group
having count(*) >= min_sup
```

The compute cube statement specifies the precomputation of the iceberg cube, *sales_iceberg*, with three dimensions, *month*, *city*, and *customer_group*, and the aggregate measure count(). The input tuples are in the *salesInfo* relation. The cube by clause specifies that aggregates (group-by's) are to be formed for each of the possible subsets of the given dimensions. If we were computing the full cube, each group-by would correspond to a cuboid in the data cube lattice. The constraint specified in the having clause is known as the **iceberg condition**. Here, the iceberg measure is count(). Note that the iceberg cube computed here can be used to answer group-by queries on any combination of the specified dimensions of the form having count(*) >= v, where $v \geq min\_sup$. Instead of count(), the iceberg condition may specify more complex measures, such as average().

If we were to omit the having clause, we would end up with the full cube. Let us call this cube *sales_cube*. The iceberg cube, *sales_iceberg*, excludes all the cells of *sales_cube* with a count that is less than *min_sup*. Obviously, if we were to set the minimum support to 1 in *sales_iceberg*, the resulting cube would be the full cube, *sales_cube*. □

OLAP Server Architectures

ROLAP (Relational)

Data stays in RDBMS. Scalable but slower for complex cubes.

MOLAP (Multidimensional)

Data in proprietary arrays. Very fast, but limited scalability and requires load time.

HOLAP (Hybrid)

Details in RDBMS, Aggregates in in MOLAP. Balances speed and scale.

Example 3.12. A ROLAP data store. Table 3.3 shows a summary fact table that contains both base fact data and aggregated data. The schema is "⟨*record_identifier (RID), item, . . . , day, month, quarter, year, dollars_sold*⟩," where *day, month, quarter*, and *year* define the sales date, and *dollars_sold* is the sales amount. Consider the tuples with an *RID* of 1001 and 1002, respectively. The data of these tuples are at the base fact level, where the sales dates are October 15, 2010, and October 23, 2010, respectively. Consider the tuple with an *RID* of 5001. This tuple is at a more general level of abstraction than the tuples 1001 and 1002. The *day* value has been generalized to `all`, so that the corresponding *time* value is October 2010. That is, the *dollars_sold* amount shown is an aggregation representing the entire month of October 2010, rather than just October 15 or 23, 2010. The special value `all` is used to represent subtotals in summarized data. □

Table 3.3 Single table for base and summary facts.

RID	item	...	day	month	quarter	year	dollars_sold
1001	TV	...	15	10	Q4	2010	250.60
1002	TV	...	23	10	Q4	2010	175.00
...	...	...	...	...	...	...	...
5001	TV	...	all	10	Q4	2010	45,786.08
...	...	...	...	...	...	...	...

Architecture Comparison & Decisions

Feature	ROLAP	MOLAP	HOLAP
 Volume	High (TB+)	Medium	High
 Speed	Slow-Mod	Very Fast	Fast
 Storage	Efficient	Poor (Sparse)	Good
 Flexibility	High (Ad-hoc)	Low (Pre-defined)	Medium

Summary & Modern Trends

Modern Trends:

- In-Memory OLAP (SAP HANA)
- Cloud OLAP (Snowflake/BigQuery)
- Real-time Streaming Cubes.

Common Pitfalls:

- Over-materialization (wasted storage).
- Wrong indexes (using Bitmap for high-cardinality).
- Ignoring semi-additive measures (incorrect sums).