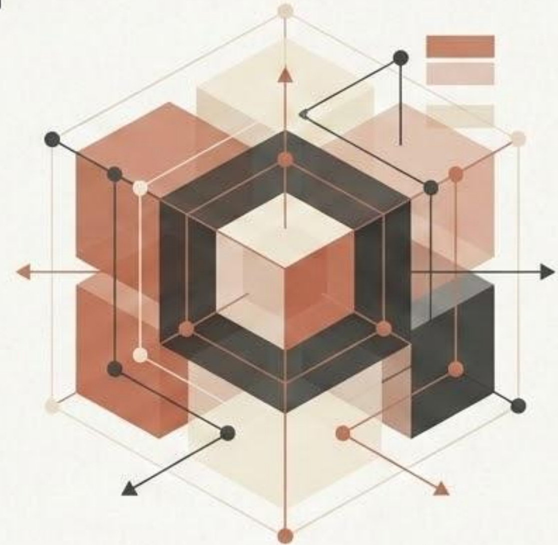
Efficient Data Cube Computation & OLAP Query Processing

Duration: 1 Hour

Prerequisites:

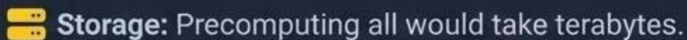
- Understanding of data cubes
- OLAP operations
- Multidimensional modeling
- Basic algorithms



THE AMAZON PROBLEM

*"Amazon has sales data across **10 dimensions**: Product × Time × Region × Customer Segment × Sales Channel × Payment Method × Device Type × ..."*

The Math: 2^{10} = 1,024 cuboids.


Storage: Precomputing all would take terabytes.

Speed: Querying on-the-fly is painfully slow.

Today: The algorithms that solve this.

LEARNING OBJECTIVES

1. Understand the **computational challenges** of data cube construction.
2. Master **Multiway Array Aggregation** for full cube computation.
3. Implement **BUC Algorithm** for iceberg cubes.
4. Design **Shell Fragments** for high-dimensional OLAP.
5. **Optimize OLAP queries** using precomputed cuboids.

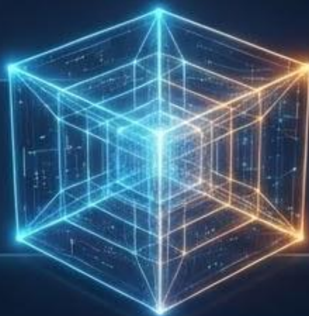
THE CURSE OF DIMENSIONALITY

EXPONENTIAL GROWTH

The number of cuboids grows exponentially with the number of dimensions.

$T = \Pi (L_i + 1)$

For 10 dimensions:
2^{10} = 1,024 cubeids



The Trade-off:

Full Materialization:

Instant Query

Huge Storage

No Materialization:

Zero Storage

Slow Query

MULTIWAY ARRAY AGGREGATION

Optimized for computing full cubes using array-based storage.

PARTITIONING

Arrays are partitioned into smaller chunks (chunks) that fit into memory.

ORDERING

Dimension order matters. Minimize memory requirements by processing smaller planes first.

SIMULTANEOUS AGGREGATION

Compute aggregates for multiple cuboids in a single pass of the data.

SHELL FRAGMENTS

The Problem: With 50+ dimensions, even BUC fails. The cube is too sparse.

The Solution: Vertical Partitioning.

FRAGMENT

Break high-dimensional cube into smaller, independent "shells" (e.g., 3 dimensions each).

COMPUTE

Compute the full cube only for these small fragments.

ASSEMBLE

For a query, join the relevant fragments (inverted indices) at runtime.

SUMMARY STRATEGY

Dense, Small Dimensions?

Use Multiway Array Aggregation.

Sparse, Iceberg Cubes?

Use BUC (Bottom-Up Computation) with Pruning.

High Dimensional (>100)?

Use Shell Fragments (Vertical Partitioning).

STRATEGIES FOR DATA CUBE COMPUTATION

Managing the Curse of Dimensionality

Section 3.4.4

THE FUNDAMENTAL PROBLEM

EXPONENTIAL GROWTH

For **n** dimensions, there are **2n** possible cuboids.

- 5 dims = 32 cuboids
- 10 dims = 1,024 cuboids
- 20 dims = 1,048,576 cuboids

⚠ **Explosion Problem:** Storage and computation grow exponentially.

THREE STRATEGIC APPROACHES

FULL MATERIALIZATION

Precompute **ALL** possible cuboids.

NO MATERIALIZATION

Compute everything **on-the-fly.**

PARTIAL MATERIALIZATION

Selective pre-computation of specific cuboids.

FULL MATERIALIZATION

Precompute ALL cuboids

PROS

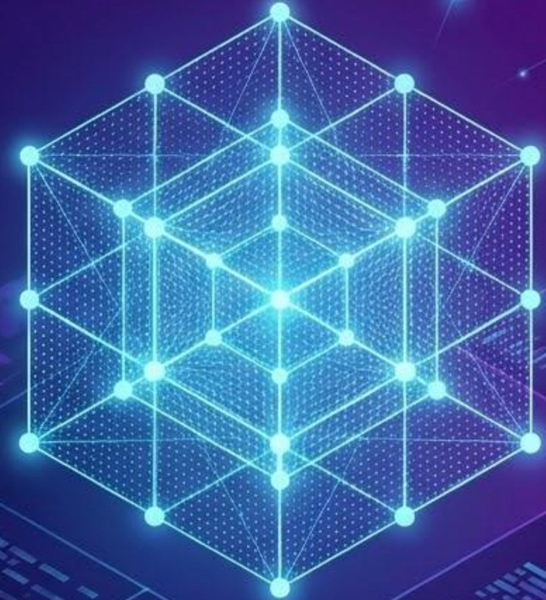
Instant query response

CONS

Exponential storage, long pre-computation time

WHEN

Small dimensions, predictable queries



Full materialization involves precalculating and storing every possible combination of dimensions in a data cube. This approach guarantees the fastest possible query performance for any aggregation but comes at the cost of enormous storage requirements and significant computational time for the initial build and updates, making it feasible only for smaller datasets or well-defined query patterns.

NO MATERIALIZATION

Compute on-the-fly

PROS

Minimal storage

CONS

Slow query response

WHEN

Ad-hoc exploration, limited storage

3. PARTIAL MATERIALIZATION

Definition: Selective pre-computation of the most valuable cuboids (e.g., Iceberg cubes).

⚖ **PROS:** Balance between storage and speed
🔍 **CONS:** Requires query pattern analysis

WHEN: Most real-world scenarios.

KEY INSIGHT

"The curse of dimensionality means we must be **smart** about what to precompute."

Optimizing Data Cubes: Storage vs. Performance

The Core Dilemma: Materialization improves query speed but consumes storage space. The goal is to find the optimal balance for the specific workload.

Full Materialization:
Storage: High (■■■■■■) – Requires massive disk space.
Query Time: Instant ( ) – All answers are pre-calculated.

No Materialization:
Storage: Minimal ( ) – Only base data is stored.
Query Time: Slow (■■■■■) – System computes answers from scratch every time.

Partial Materialization (The Sweet Spot):
Storage: Moderate (■■■)
Query Time: Fast (■■■)
Strategy: Only pre-calculate the most frequently accessed data views.

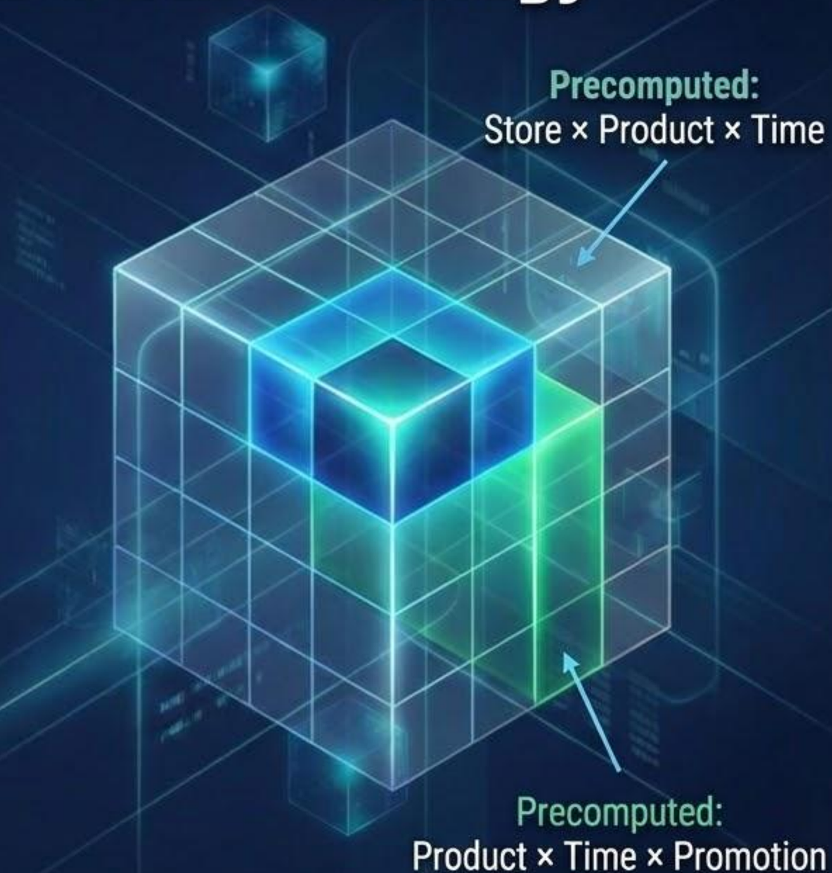
Case Study: Walmart's Data Cube Strategy

The Context:
Walmart manages massive data across 5 Key Dimensions:
Store × Product × Time × Customer × Promotion

The Strategy: Partial Materialization
Walmart cannot precompute every possible combination (data explosion). Instead, they select specific "sub-cubes" based on business value.

Implementation:
Precomputed (Fast Access):
- Store × Product × Time: Supports daily inventory and sales tracking (High frequency).
- Product × Time × Promotion: Supports marketing effectiveness analysis.

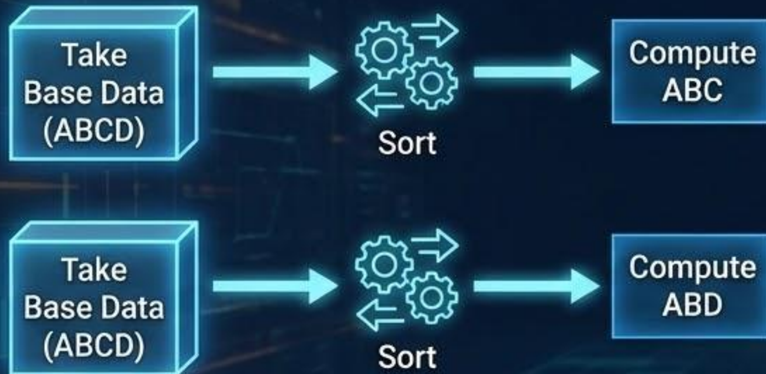
Computed On-the-Fly:
- Combinations involving Customer or complex ad-hoc queries (Rare frequency).



Multiway Array Aggregation for Full Cube Computation

The Problem with Traditional Computation

Traditional Approach (Pipesort):



Inefficiency: Requires repeated sorting and scanning of the same dataset for every different view.

The Solution (MAA):

Stores data as a **Sparse Array** instead of a table. Computes all cuboids simultaneously in a single pass.

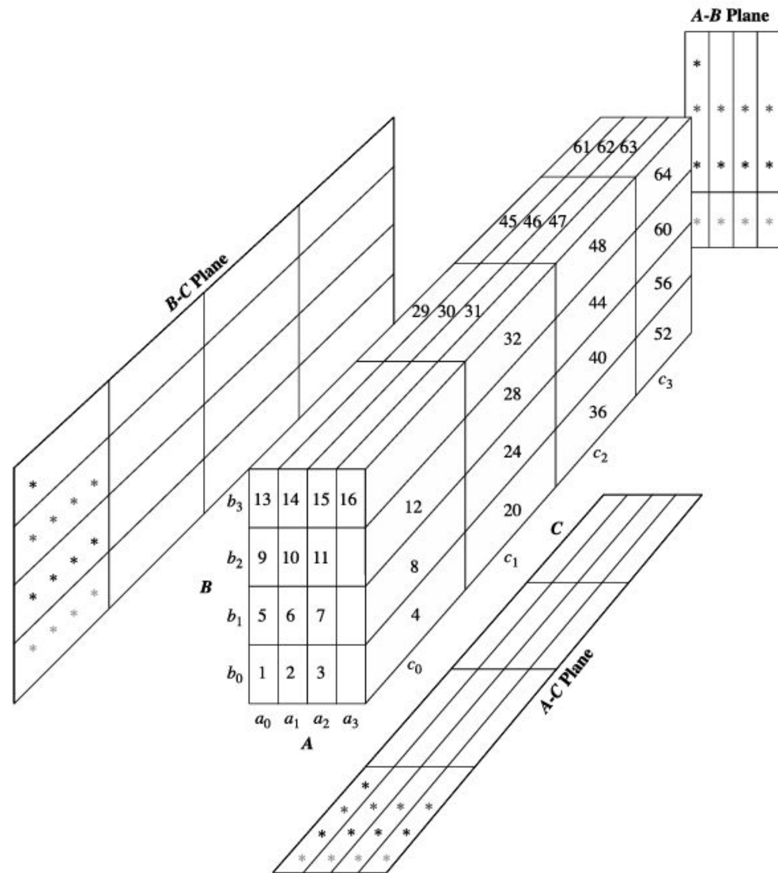


FIGURE 3.20

A 3-D array for the dimensions A, B, and C, organized into 64 *chunks*. Each chunk is small enough to fit into the memory available for cube computation. The $*$'s indicate the chunks from 1 to 13 that have been aggregated so far in the process.

Example 3.13. Multiway array cube computation. Consider a 3-D data array containing three dimensions A, B, and C. The 3-D array is partitioned into small, memory-based chunks. In this example, the array is partitioned into 64 chunks as shown in Fig. 3.20. Dimension A is organized into four equal-sized 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. Since the number of partitions of each dimension is 4, the size of each partition in A, B, and C is therefore 10, 100, and 1000, respectively. Full materialization of the corresponding data cube involves the computation of all the cuboids defining this cube. 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 need to be computed.
- The 1-D cuboids, A, B, and C, which respectively correspond to the group-by's A, B, and C. These cuboids need to be computed.
- The 0-D (apex) cuboid, denoted by all, which corresponds to the group-by (); that is, there is no group-by here. These cuboids need to be computed. It consists of only one value. If, say, the data cube measure is `count`, then the value to be computed is simply the total count of all the tuples in ABC.

Let's look at how the multiway array aggregation technique is used in this computation. There are many possible orderings with which chunks can be read into memory for use in cube computation. Consider the ordering labeled from 1 to 64, shown in Fig. 3.20. Suppose we want to compute the b_0c_0 chunk of the BC cuboid. We allocate space for this chunk in *chunk memory*. By scanning ABC chunks

1 through 4, the b_0c_0 chunk is computed. That is, the cells for b_0c_0 are aggregated over a_0 to a_3. The chunk memory can then be assigned to the next chunk, b_1c_0, which completes its aggregation after the scanning of the next four ABC chunks: 5 through 8. Continuing in this way, the entire BC cuboid can be computed. Therefore only *one* BC chunk needs to be in memory at a time for the computation of all the BC chunks.

In computing the BC cuboid, we will have scanned each of the 64 chunks. *"Is there a way to avoid having to rescan all of these chunks for the computation of other cuboids, such as AC and AB?"* The answer is, most definitely, *yes*. This is where the "multiway computation" or "simultaneous aggregation" idea comes in. For example, when chunk 1 (i.e., $a_0b_0c_0$) is being scanned (say, for the computation of the 2-D chunk b_0c_0 of BC, as described previously), all of the other 2-D chunks related to $a_0b_0c_0$ can be simultaneously computed. That is, when $a_0b_0c_0$ is being scanned, each of the three chunks (b_0c_0, a_0c_0, and a_0b_0) on the three 2-D aggregation planes (BC, AC, and AB) should be computed then as well. In other words, multiway computation simultaneously aggregates to each of the 2-D planes while a 3-D chunk is in memory.

Now let's look at how different orderings of chunk scanning and of cuboid computation can affect the overall data cube computation efficiency. Recall that the size of the dimensions A, B, and C is 40, 400, and 4000, respectively. Therefore the largest 2-D plane is BC (of size $400 \times 4000 = 1{,}600{,}000$). The second largest 2-D plane is AC (of size $40 \times 4000 = 160{,}000$). AB is the smallest 2-D plane (of size $40 \times 400 = 16{,}000$).

Suppose that the chunks are scanned in the order shown, from chunks 1 to 64. As previously mentioned, b_0c_0 is fully aggregated after scanning the row containing chunks 1 through 4; b_1c_0 is fully aggregated after scanning chunks 5 through 8, and so on. Thus we need to scan four chunks of the 3-D array to *fully* compute one chunk of the BC cuboid (where BC is the largest of the 2-D planes). In other words, by scanning in this order, one BC chunk is fully computed for each row scanned. In comparison, the complete computation of one chunk of the second largest 2-D plane, AC, requires scanning 13 chunks, given the ordering from 1 to 64. That is, a_0c_0 is fully aggregated only after the scanning of chunks 1, 5, 9, and 13.

Finally, the complete computation of one chunk of the smallest 2-D plane, AB, requires scanning 49 chunks. For example, a_0b_0 is fully aggregated after scanning chunks 1, 17, 33, and 49. Hence, AB requires the longest scan of chunks to complete its computation. To avoid bringing a 3-D chunk into memory more than once, the minimum memory requirement for holding all relevant 2-D planes in chunk memory, according to the chunk ordering of 1 to 64, is as follows: 40×400 (for the whole AB plane) $+ 40 \times 1000$ (for one column of the AC plane) $+ 100 \times 1000$ (for one BC plane chunk) $=$ $16{,}000 + 40{,}000 + 100{,}000 = 156{,}000$ memory units.

Suppose, instead, that the chunks are scanned in the order 1, 17, 33, 49, 5, 21, 37, 53, and so on. That is, suppose the scan is in the order of first aggregating toward the AB plane and then toward the AC plane, and lastly toward the BC plane. The minimum memory requirement for holding 2-D planes in chunk memory would be as follows: 400×4000 (for the whole BC plane) $+ 10 \times 4000$ (for one AC plane row) $+ 10 \times 100$ (for one AB plane chunk) $= 1{,}600{,}000 + 40{,}000 + 1000 = 1{,}641{,}000$ memory units. Notice that this is *more than 10 times* the memory requirement of the scan ordering of 1 to 64.

Similarly, we can work out the minimum memory requirements for the multiway computation of the 1-D and 0-D cuboids. Fig. 3.21 shows the most efficient way to compute 1-D cuboids. Chunks for 1-D cuboids A and B are computed during the computation of the smallest 2-D cuboid, AB. The smallest 1-D cuboid, A, will have all of its chunks allocated in memory, whereas the larger 1-D cuboid,

B, will have only one chunk allocated in memory at a time. Similarly, chunk C is computed during the computation of the second smallest 2-D cuboid, AC, requiring only one chunk in memory at a time. Based on this analysis, we see that the most efficient ordering in this array cube computation is the chunk ordering of 1 to 64, with the stated memory allocation strategy. $\square$

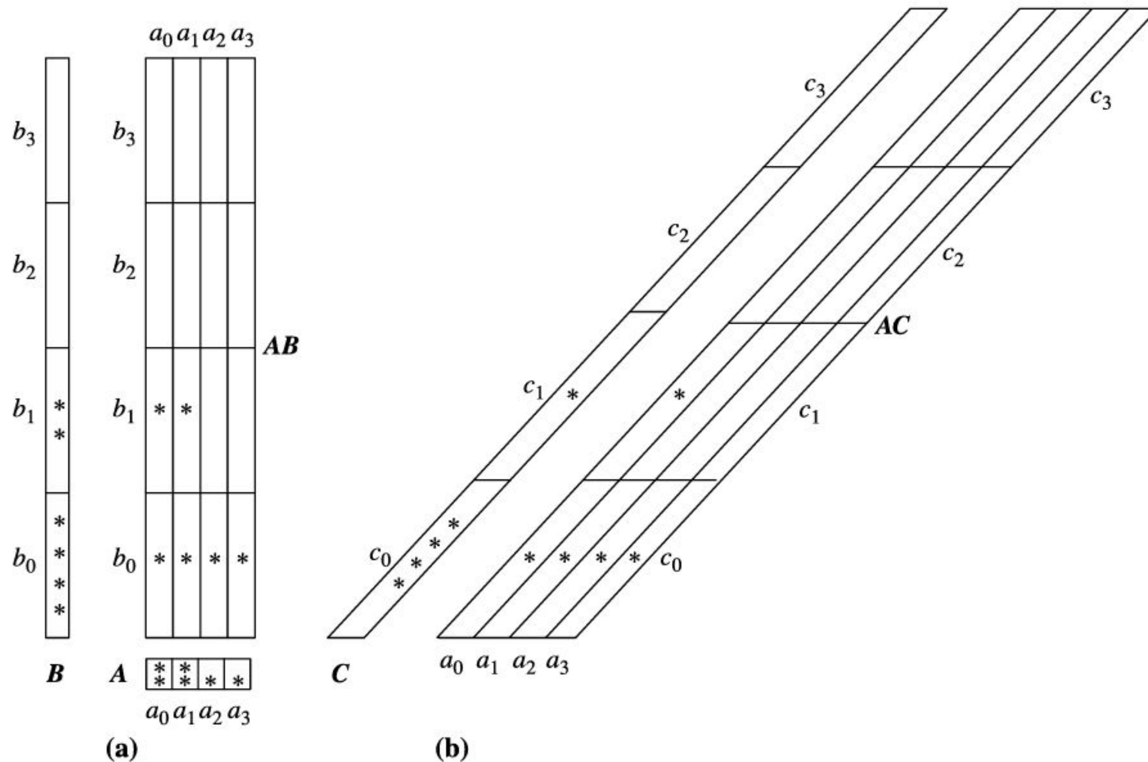


FIGURE 3.21

Memory allocation and computation order for computing Example 5.4's 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.

Problem Setup

Dimensions and Cardinalities

Dimension A: Cardinality = 40
Dimension B: Cardinality = 400
Dimension C: Cardinality = 4000

Partitioning into Chunks

Equal-Sized Partitions

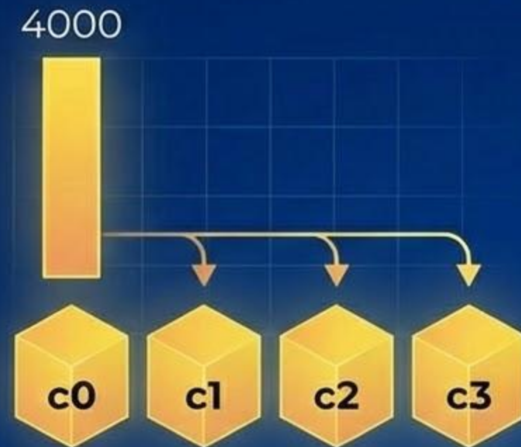
Dimension A

Partition size = 40 / 4 = 10

Dimension B

Partition size = 400 / 4 = 100

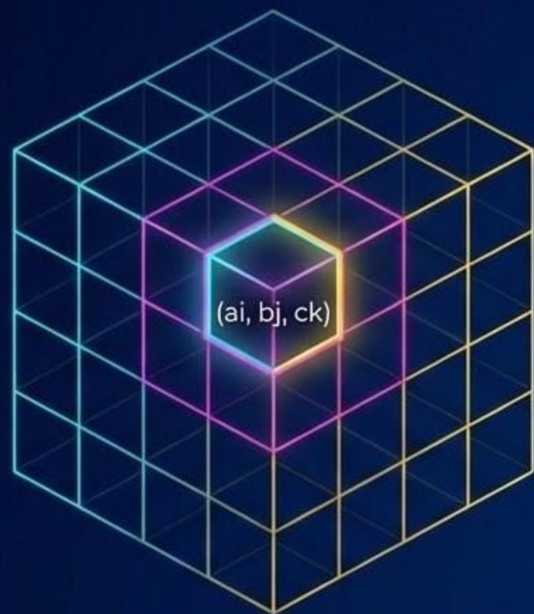
Dimension C



Partition size = 4000 / 4 = 1000

Total Number of Chunks

3-D Chunk Structure



Each chunk = subcube (ai, bj, ck)

Total chunks:

$4 \times 4 \times 4 = 64$ chunks

Examples:

a0b0c0

a1b0c0

...

a3b3c3

Chunks are numbered from 1 to 64

Sizes of 2-D Cuboid Planes

Relative Sizes of Cuboids

	Cuboid	Size	Cells
1	AB	40 × 400	16,000
2	AC	40 × 4000	160,000
3	BC	400 × 4000	1,600,000

Key Observation: BC is the **largest 2-D** plane

Chunk Scan Order

Textbook Order

B

(a3b0,c0)

a1b0,c0)

A

B

A

Scan Order: 1 to 64

Chunk sequence:
a0b0c0, a1b0c0, a2b0c0, a3b0c0
a0b1c0, a1b1c0, ...
...
a3b3c3

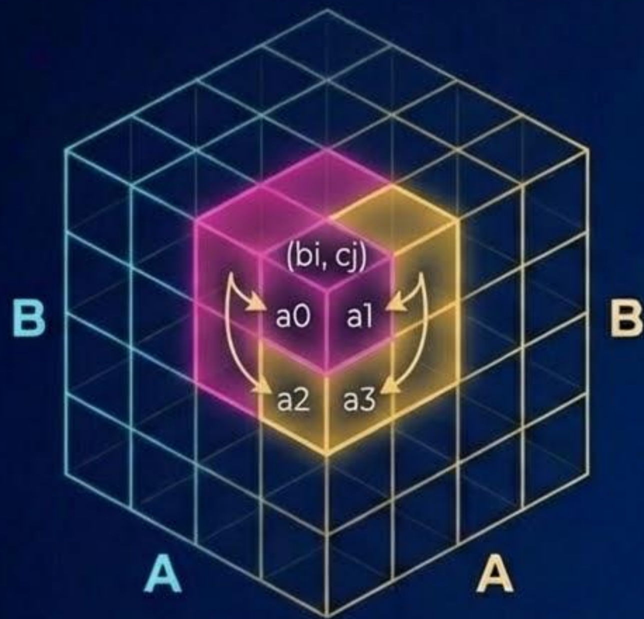
Loop structure:

```
for c:
  for b:
    for a:
      process (a,b,c)
```

👉 **A varies fastest (innermost loop)**

Effect on BC Cuboid

Largest Plane



Fast Completion of BC

To compute one BC chunk (bi, cj):

Need to aggregate over all **A:**

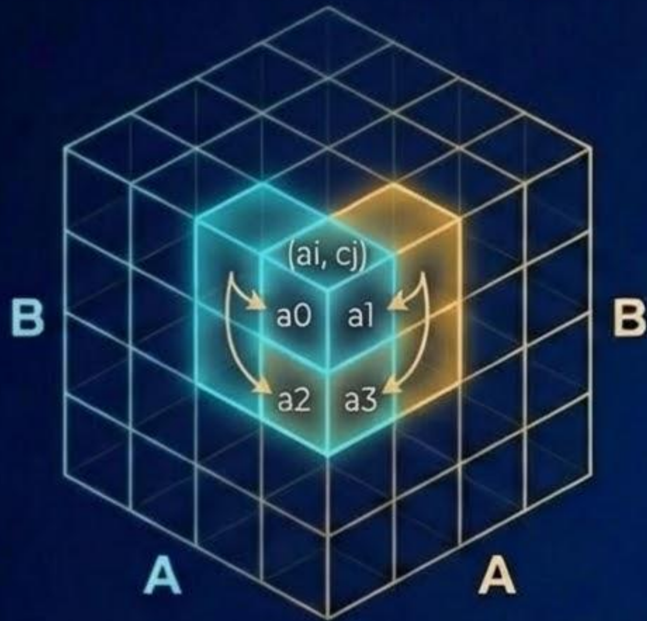
a0bi cj
a1bi cj
a2bi cj
a3bi cj

So: One BC chunk completes after scanning 4 chunks

This is very efficient for BC.

Effect on AC Cuboid

Completion of AC Chunks



To compute one AC chunk (ai, cj):

Need to scan multiple B and A partitions.

Example:

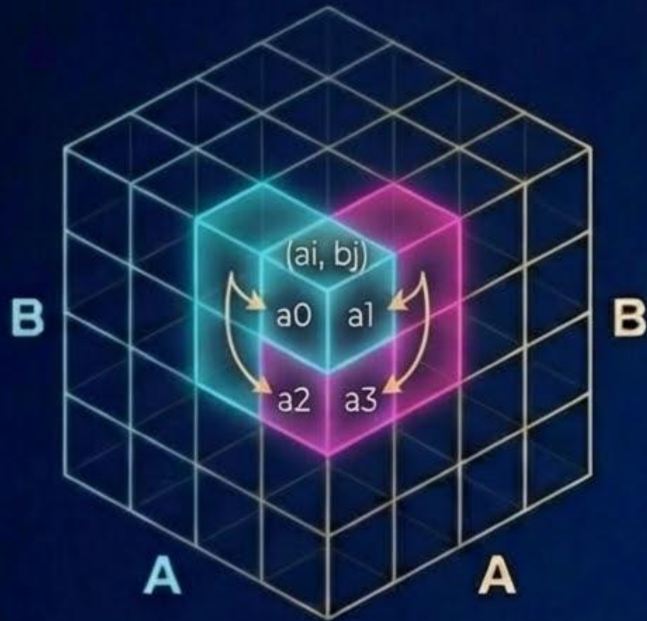
a0c0 completes after scanning: chunks 1, 5, 9, 13

So: One AC chunk requires scanning 13 chunks

This is relatively efficient for AC.

Effect on AB Cuboid

Smallest Plane



Slowest Completion of AB

To compute one AB chunk **(ai, bj)**:
Need to scan across all C partitions.

Example:

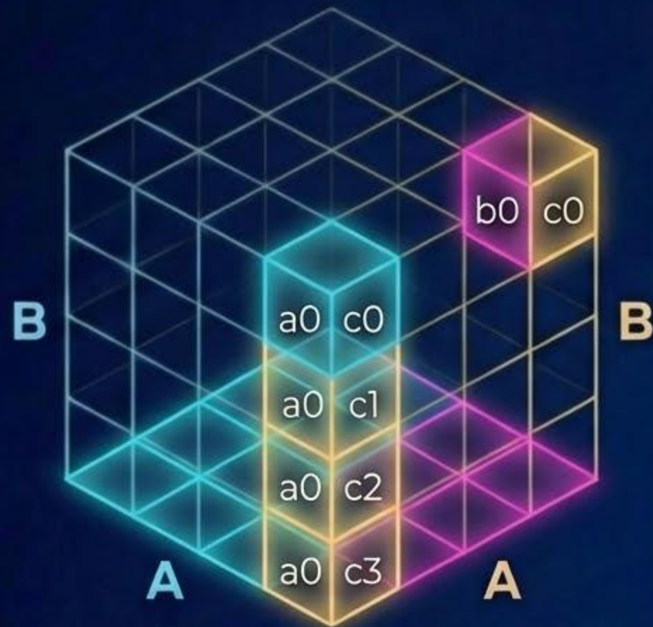
a0b0 completes after scanning:
chunks 1, 17, 33, 49

So: One AB chunk requires
scanning 49 chunks

AB takes longest — but AB is the smallest plane.

Memory Requirement (Good Ordering)

Minimum Memory (Textbook Ordering)



Memory needed to hold:

- Whole AB plane:
  40 × 400 = **16,000**
- One column of AC:
  40 × 1000 = **40,000**
- One BC plane chunk:
  100 × 1000 = **100,000**

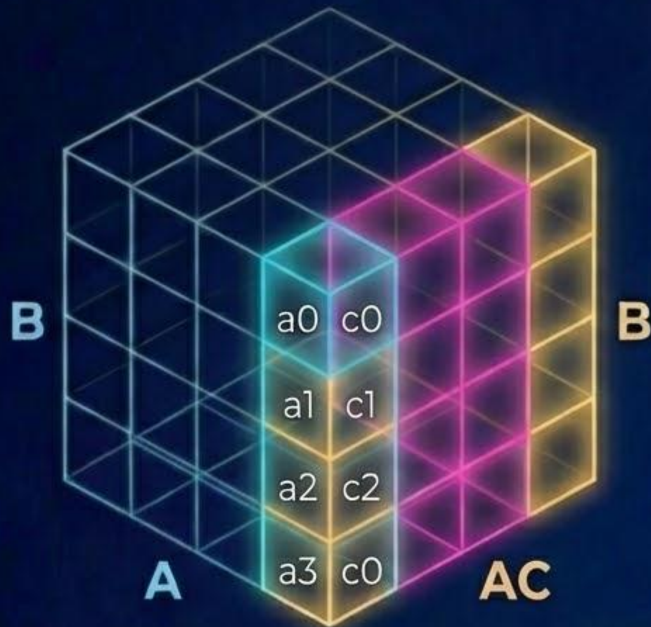
Total Memory:

16,000 + 40,000 + 100,000 =

156,000

Memory Requirement (Bad Ordering)

Memory Needed



Must hold:

- Whole BC plane: 400 × 4000 = 1,600,000
- One AC row: 10 × 4000 = 40,000
- One AB chunk: 10 × 100 = 1,000

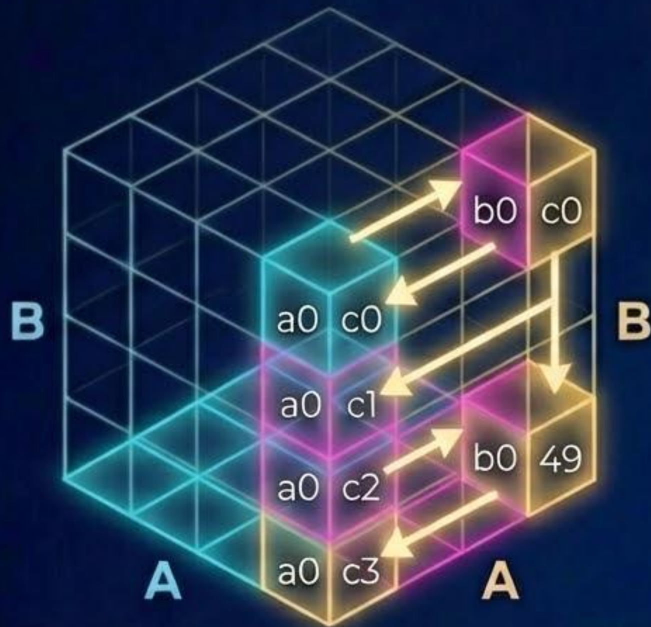
Total Memory:

1,600,000 + 40,000 + 1,000 =

1,641,000

Alternative Scan Order (Bad Ordering)

Prioritize AB, then AC, then BC



Example scan order:

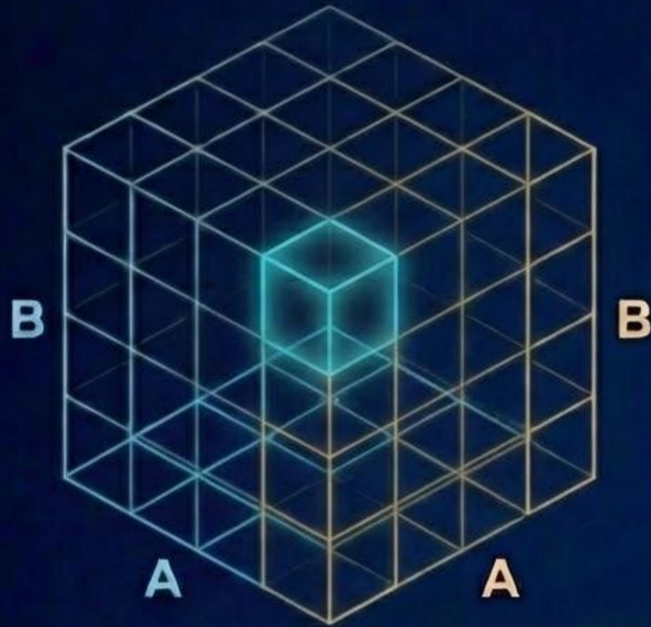
1, 17, 33, 49, 5, 21, 37, 53, ...

This tries to finish:

- AB first
- AC next
- BC last

This leads to significantly higher memory usage.

Comparison of Two Orders



Scan Order	Memory Required
Textbook (1–64)	156,000
Alternative	1,641,000

Bad ordering uses more than 10× memory.

Step 1 - Sparse Array Representation

Converting Tables to Arrays

Traditional Table:

Product	Time	Region	Sales
Product A	Q1	East	$100

Sparse Array Logic:

Dimensions become indices;
Metrics become values.

Formula:
array[Product, Time, Region] = Sales

Example:
array[A, Q1, East] = 100

Sparsity:

Most cells in the multidimensional array are empty (0) because not every product sells in every region at every time.

MAA compresses this by storing only non-zero values.

Step 2 - Simultaneous Aggregation

Single-Pass Aggregation Logic

The Concept & Algorithm:

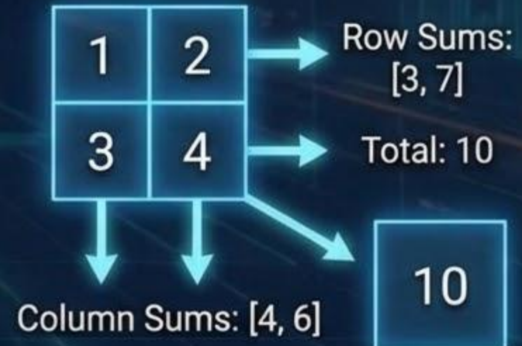
- Instead of separate scans, we traverse the array once.
- For every non-zero cell (e.g., [a,b,c,d] with value v), we immediately add v to all its ancestors.

Visualizing the Algorithm:

Visual Example (2D Array):

[[1 2
3 4]]

Simultaneous Output:



Optimization & Performance Gains: Chunking Strategy

Managing Large Data Cubes

Optimization: Chunking

Large multi-dimensional arrays often exceed RAM.

Strategy: Divide the array into small "chunks" that fit in memory.

Process chunk-by-chunk, aggregating partial results.

Performance Comparison:

Traditional (Slow)

$O(2^n \times m)$

Exponential complexity (slow for high dimensions).

MAA (Fast)

$O(m \times n)$

Linear complexity (fast for sparse cubes).

Key Points:

Key Advantage:

Avoids repeated sorting/scanning.

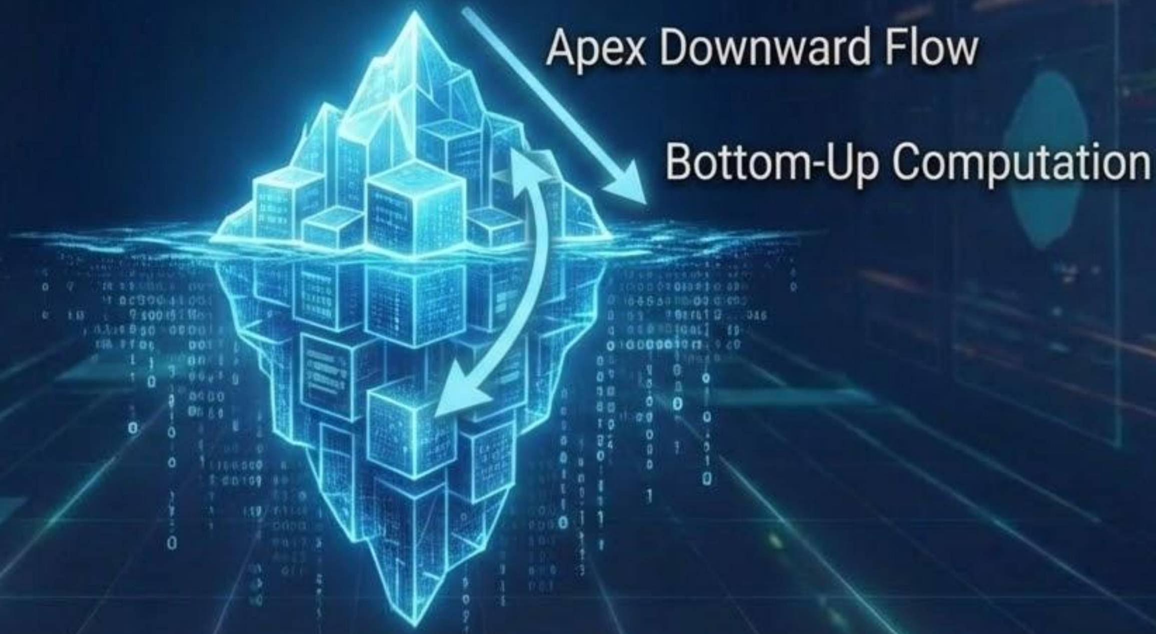
Limitation:

Performance depends on the dimension order and memory fit.

BUC: Computing Iceberg Cuboids from Apex Downward

Efficient Data Cube Computation via Pruning

Key Concept: “Bottom-Up Computation” (Processing Top-Down)



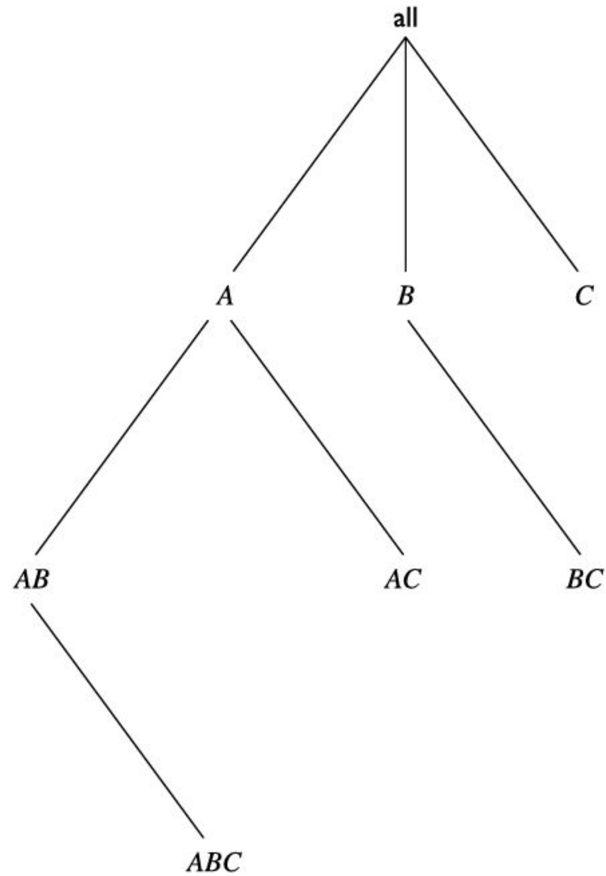


FIGURE 3.22

BUC's exploration for a 3-D data cube computation. Note that the computation starts from the apex cuboid.

Algorithm: BUC. Algorithm for the computation of sparse and iceberg cubes.

Input:

- *input*: the relation to aggregate;
- *dim*: the starting dimension for this iteration.

Globals:

- constant *numDims*: the total number of dimensions;
- constant *cardinality[numDims]*: the cardinality of each dimension;
- constant *min_sup*: the minimum number of tuples in a partition for it to be output;
- *outputRec*: the current output record;
- *dataCount[numDims]*: stores the size of each partition. *dataCount[i]* is a list of integers of size *cardinality[i]*.

Output: Recursively output the iceberg cube cells satisfying the minimum support.

Method:

```
(1)  Aggregate(input); // Scan input to compute measure, e.g., count. Place result in outputRec.
(2)  if input.count() == 1 then // Optimization
         WriteDescendants(input[0], dim); return;
     endif
(3)  write outputRec;
(4)  for (d = dim; d < numDims; d++) do //Partition each dimension
(5)      C = cardinality[d];
(6)      Partition(input, d, C, dataCount[d]); //create C partitions of data for dimension d
(7)      k = 0;
(8)      for (i = 0; i < C; i++) do // for each partition (each value of dimension d)
(9)          c = dataCount[d][i];
(10)         if c >= min_sup then // test the iceberg condition
(11)             outputRec.dim[d] = input[k].dim[d];
(12)             BUC(input[k..k + c - 1], d + 1); // aggregate on next dimension
(13)         endif
(14)         k +=c;
(15)     endfor
(16)     outputRec.dim[d] = all;
(17) endfor
```

FIGURE 3.23

BUC algorithm for sparse or iceberg cube computation. *Source:* Beyer and Ramakrishnan [BR99].

Example 3.14. BUC construction of an iceberg cube. Consider the iceberg cube expressed in SQL as follows:

```
compute cube iceberg_cube as
select A, B, C, D, count(*)
from R
cube by A, B, C, D
having count(*) >= 3
```

Let's see how BUC constructs the iceberg cube for the dimensions A, B, C, and D, where 3 is the minimum support count. Suppose that dimension A has four distinct values, a_1, a_2, a_3, a_4; B has four distinct values, b_1, b_2, b_3, b_4; C has two distinct values, c_1, c_2; and D has two distinct values, d_1, d_2. If we consider each group-by to be a *partition*, then we must compute every combination of the grouping attributes that satisfy the minimum support (i.e., that have three tuples).

Fig. 3.24 illustrates how the input is partitioned first according to the different attribute values of dimension A and then B, C, and D. To do so, BUC scans the input, aggregating the tuples to obtain

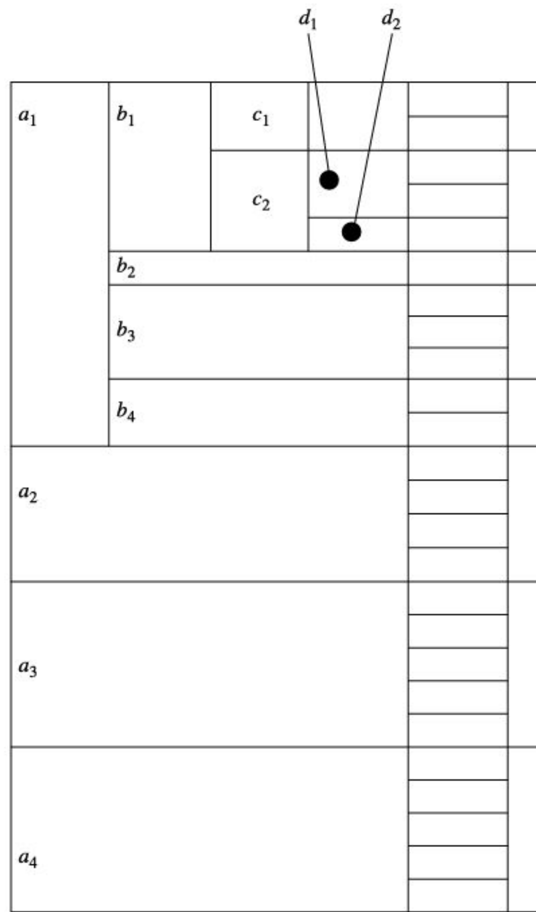


FIGURE 3.24

BUC partitioning snapshot given an example 4-D data set.

a count for all, corresponding to the cell $(*, *, *, *)$. Dimension A is used to split the input into four partitions, one for each distinct value of A. The number of tuples (counts) for each distinct value of A is recorded in *dataCount*.

BUC uses the downward antimonotonicity property to save time while searching for tuples that satisfy the iceberg condition. Starting with A dimension value, a_1, the a_1 partition is aggregated, creating one tuple for the A group-by, corresponding to the cell $(a_1, *, *, *)$. 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 downward antimonotonicity 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. By checking the iceberg condition each time before performing a recursive call, BUC saves a great deal of processing time whenever a cell's count does not satisfy the minimum support.

The partition process is facilitated by a linear sorting method, CountingSort. CountingSort is fast because it does not perform any key comparisons to find partition boundaries. For example, to sort 10,000 tuples according to an attribute A whose value is an integer in the range between 1 and 100, we can set up 100 counters and scan the data once to count the number of 1's, 2's, ..., 100's on attribute A. Suppose there are i_1 tuples having 1 on A, i_2 tuples having 2 on A, and so on. Then, in the next scan, we can move all the tuples having value 1 on attribute A to the first i_1 slots, the tuples having value 2 on attribute A to the slots $i_1 + 1, \ldots, i_1 + i_2$, and so on. After those two scans, the tuples are sorted according to A. In addition, the counts computed during the sort can be reused to compute the group-by's in BUC.

Line 2 is an optimization for partitions having a count of 1 such as $(a_1, b_2, *, *)$ in our example. To save on partitioning costs, the count is written to each of the tuple's descendant group-by's. This is particularly useful since, in practice, many partitions may have a single tuple. □

What is an Iceberg Cube?

Definition: A data cube that only stores cells (aggregates) that satisfy a specific minimum threshold.

Why "Iceberg"?

Like an iceberg, you only see the "tip" (significant data), while the vast majority of insignificant data remains hidden (uncomputed).

Examples:

Count Threshold:
Only store cells with count > 100.

Sum Threshold:
Only store cells with sum > $10,000.

BUC (Bottom-Up Computation)

Origin: Developed by Kevin Beyer and Raghu Ramakrishnan.

The Irony:

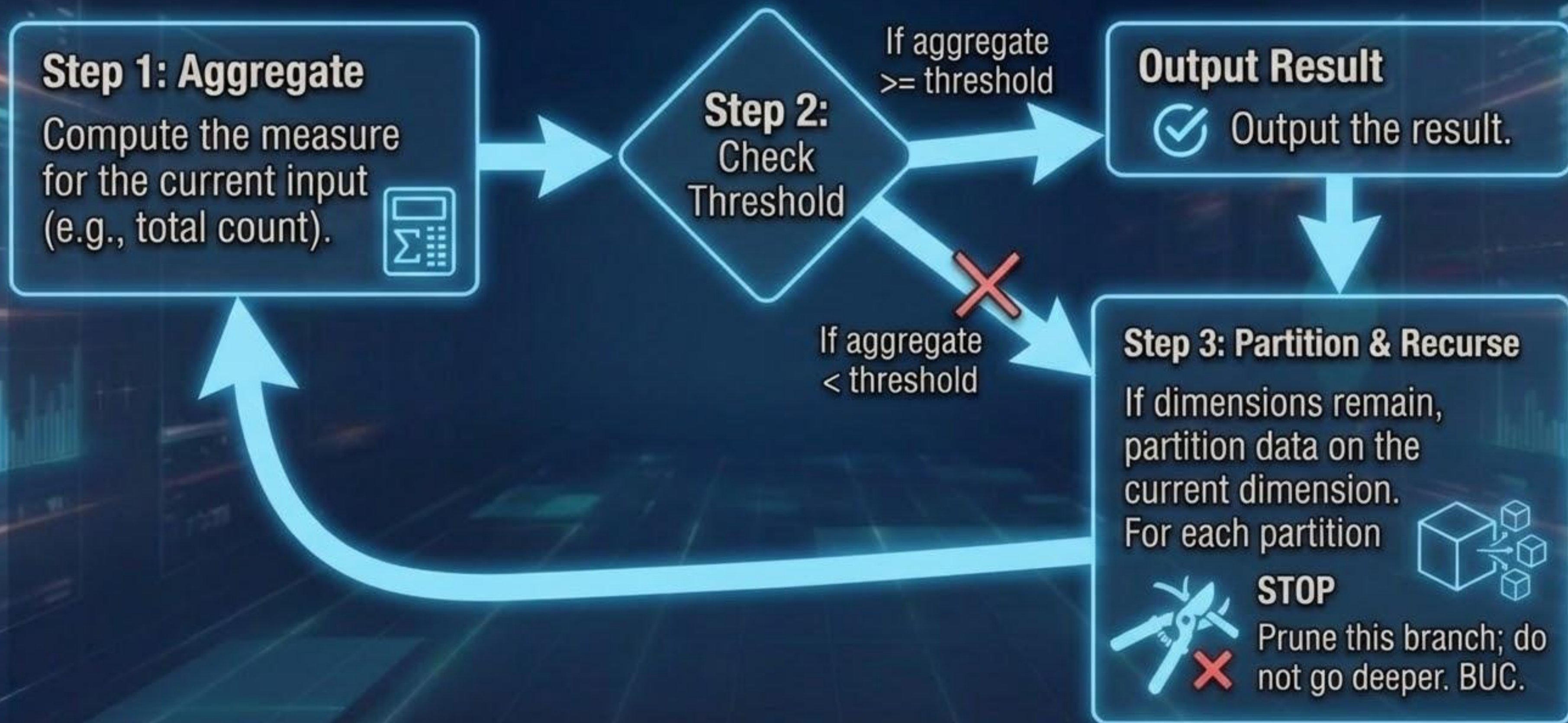
Despite standing for "Bottom-Up Computation," the processing order is strictly Top-Down (from the Apex (*,*,*) downwards).

Core Strategy:

Divide and Conquer:
Partition large datasets into smaller chunks.

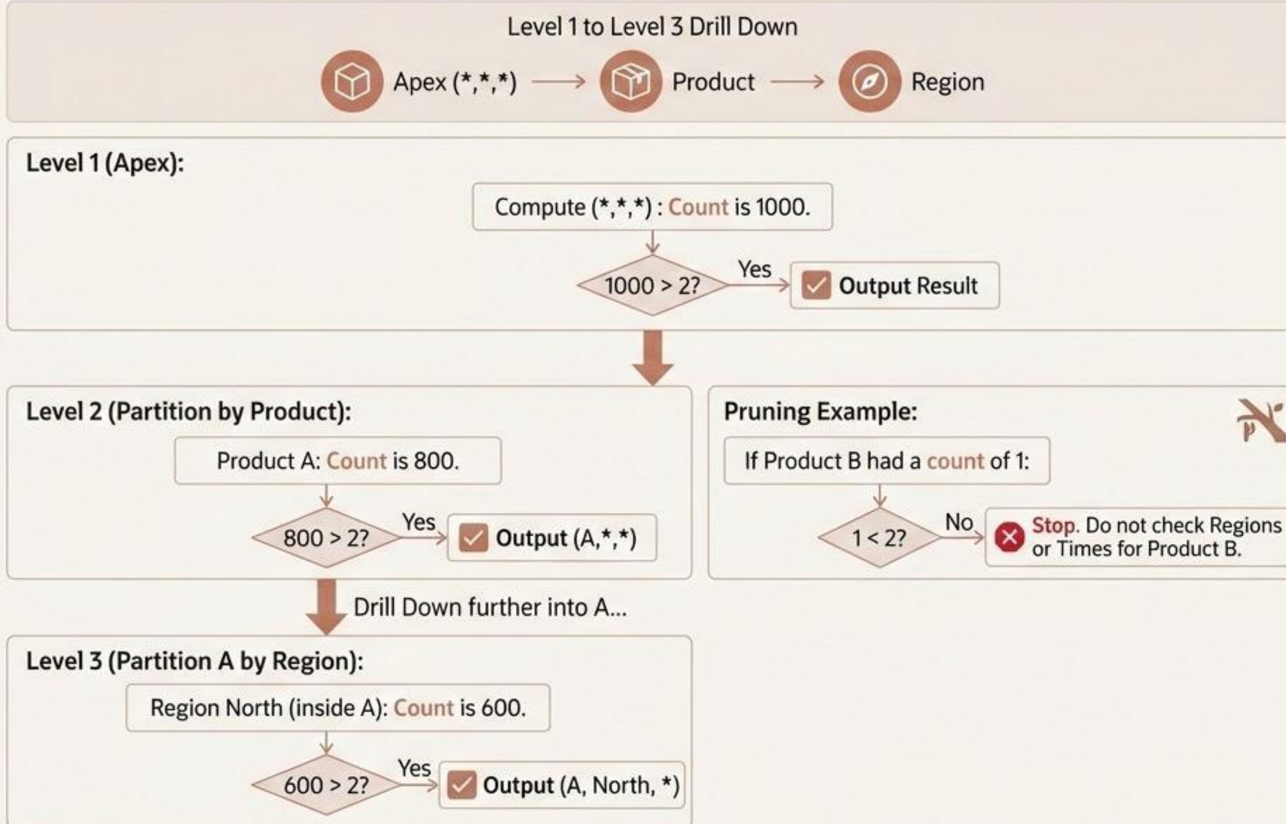
Prune Early:
Stop processing a branch immediately if it fails the threshold test (Apriori property).

BUC Algorithm Steps



Execution Example

Walkthrough: Product × Region × Time Scenario | Threshold = count > 2



Why Use BUC?

Performance

- Much faster than computing the full cube and then filtering out small numbers.
- Leverages sorting and partitioning to keep data organized in memory.

Real-World Application: Fraud Detection

Goal: Detect transaction patterns occurring > 50 times.

Benefit: Rare patterns (which might be legitimate anomalies) are pruned early.

Focus: Resources are spent only on statistically significant clusters.

Precomputing Shell Fragments for Fast High-Dimensional OLAP

Solving the “Curse of Dimensionality” in Data Cubes

Core Concept: Divide and Conquer Strategy for OLAP

Table 3.4 Original database.

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

Table 3.5 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

Table 3.6 Cuboid AB.

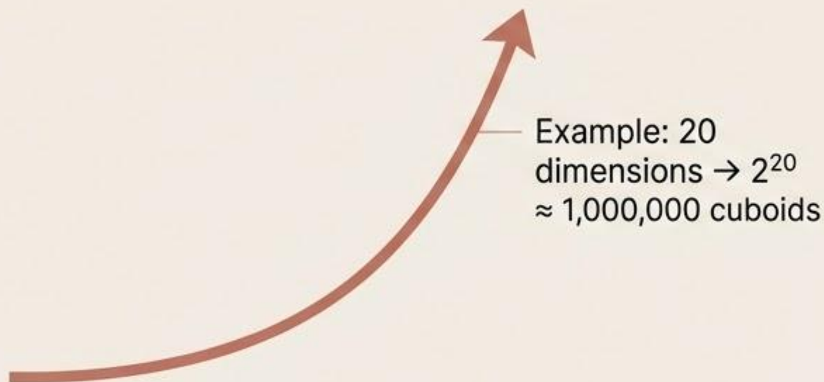
Cell	Intersection	TID List	List Size
(a_1, b_1)	$\{1, 2, 3\} \cap \{1, 4, 5\}$	$\{1\}$	1
(a_1, b_2)	$\{1, 2, 3\} \cap \{2, 3\}$	$\{2, 3\}$	2
(a_2, b_1)	$\{4, 5\} \cap \{1, 4, 5\}$	$\{4, 5\}$	2
(a_2, b_2)	$\{4, 5\} \cap \{2, 3\}$	$\{\}$	0

Table 3.7 Cuboid DE.

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

The Challenge of High Dimensions

Exponential Growth



- As dimensions increase, the number of possible cuboids explodes exponentially.

The Dilemma

Full Materialization

Impossible due to storage/computation limits (cannot precompute 1M cuboids).

No Materialization

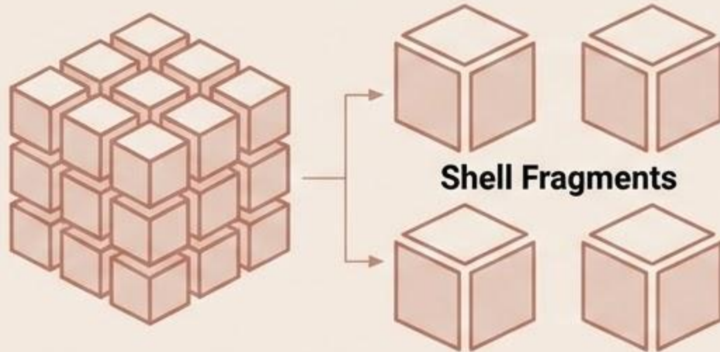
Queries are too slow for real-time analysis.

Goal

Achieve fast query speeds without computing the entire massive cube.

Shell Fragment Strategy

Concept & Algorithm



- **Divide**: Split high dimensions into smaller, disjoint groups.
- **Precompute**: Compute the full cube only for these small fragments.

The Math (Efficiency)

Scenario

20 Dimensions split into 4 fragments of 5 dimensions each.

Shell Fragments

Total storage: $4 \times (2^5$ cuboids) = 128 cuboids.

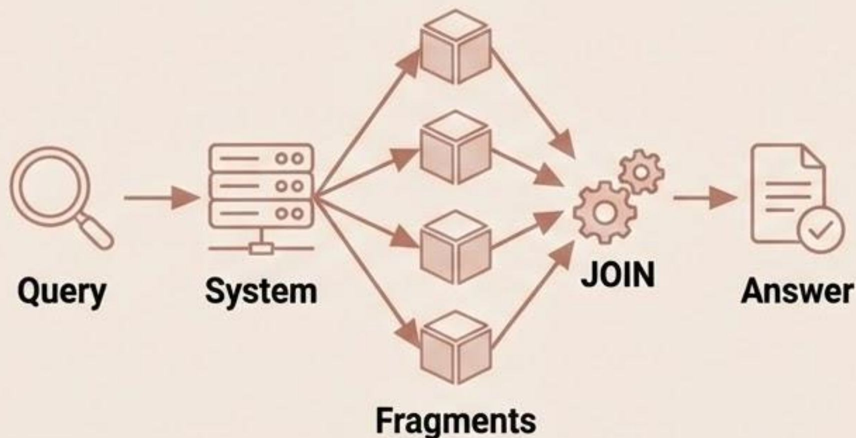
HUGE SAVINGS

Full Cube

1,048,576 cuboids

How Queries are Answered

Process



When a query arrives, the system retrieves data from relevant fragments and joins them on the fly.

Example

Setup:

Query: Requesting data for D2, D7, D12, D17.

Execution:

Efficiency & Optimization

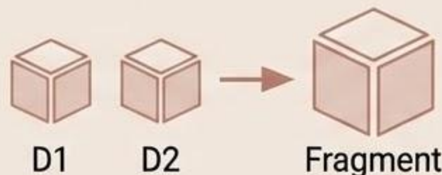
Mathematical Guarantee

$$\left\lceil \sum_{k=0} \frac{k_i}{f} \right\rceil$$

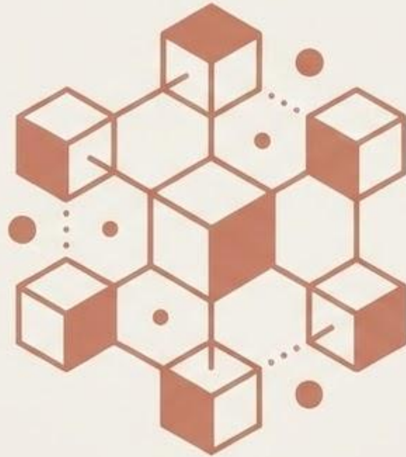
- Any k-dimensional query is answered by joining at most k fragments.
- If fragment size is f, max fragments needed = $\lceil k/f \rceil$.

Optimization Strategy

- **Smart Grouping**: Do not group dimensions randomly.
- **Query Affinity**: If D1 and D2 are frequently queried together, place them in the same fragment.



Result
Reduces the number of joins required at runtime, speeding up common queries.



Efficient Processing of OLAP Queries Using Cuboids

Optimizing Query Performance Through Smart Selection

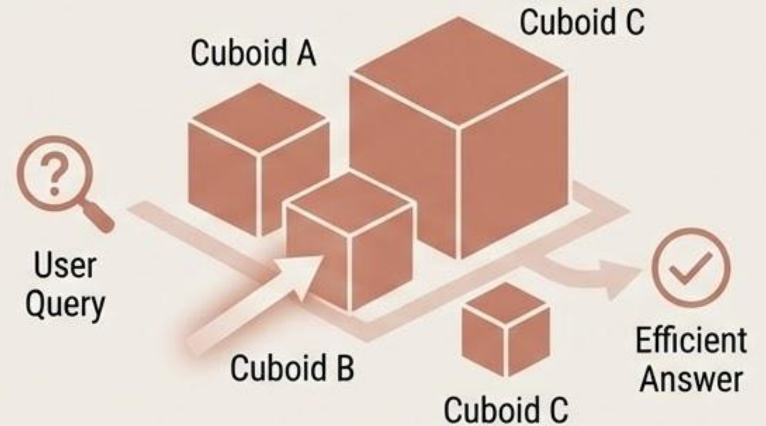
Core Concept: Using precomputed views to minimize computational cost.

The Challenge: Which Cuboid to Use?

Problem: Given a user query on specific dimensions, the system often has multiple precomputed “cuboids” (summary tables) available.

Goal: Select the single best cuboid to answer the query efficiently.

Key Principle: Use the smallest available cuboid that acts as a superset (contains all needed dimensions) for the query.



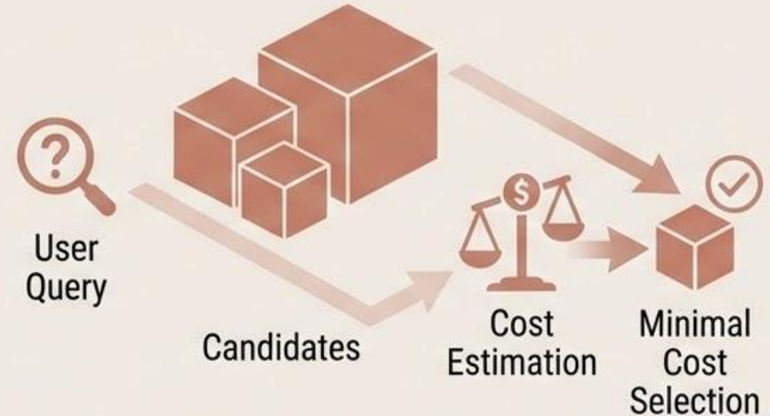
Algorithm for Cuboid Selection

Step 1: Identify Candidates
Find all precomputed cuboids that contain all the query dimensions.

Example: For a query on {A, B}, valid candidates could be {A, B}, {A, B, C}, or {A, B, C, D}.

Step 2: Estimate Cost
Formula: Cost = Size(cuboid) + Cost(additional_ aggregation)$.
Factors: Dimensionality, sparsity, and I/O requirements.

Step 3: Choose Minimal Cost
Select the candidate with the lowest estimated processing effort.



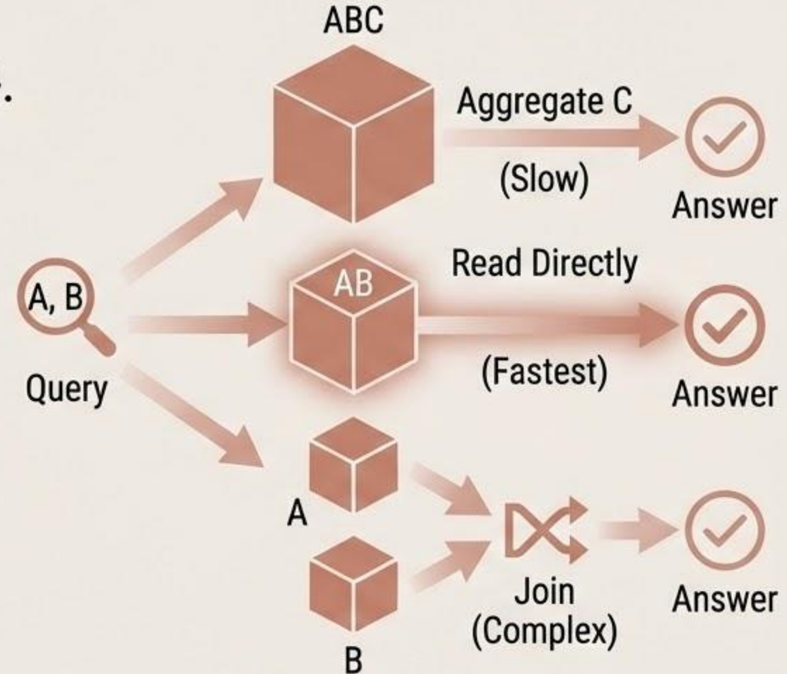
Example: Selecting the Right Cuboid

Scenario:
Precomputed Cuboids: {ABC, AB, AC, BC, A, B, C, ∅}.
Query: SELECT A, B, SUM(M) GROUP BY A, B.

Evaluation:

- Option 1 (Use ABC): Valid, but contains extra dimension C.
  Action: Must aggregate over C (Slow).
- Option 2 (Use AB): Perfect match!
  Action: Read directly (Fastest).
- Option 3 (Use A & B): Valid, but requires joining separate lists.
  Action: Join operation (Complex).

Winner: Option 2 (Cuboid AB).



At a Glance: Cube Computation Algorithms

Algorithm	Best Used For	Key Idea	Complexity
Multiway Array (MAA)	Full cubes, Sparse data	Simultaneous aggregation	$O(m \times n)$
BUC	Iceberg cubes, Skewed data	Top-down with pruning	$O(m \times \log m)$
Shell Fragments	High dimensions (>15)	Divide dimensions into fragments	$O(2^f \times \#\text{fragments})$
Cuboid Selection	Query optimization	Use smallest relevant cuboid	$O(\#\text{cuboids})$

Real-World Trade-offs & Tuning

Memory vs. Disk Trade-off

MAA: Requires significant memory for arrays; performance drops if swapping occurs.

BUC: Works efficiently with disk-based partitions, making it robust for massive datasets.

Data Skew Awareness

BUC: Excellent for skewed data (common in real-world business data) due to its partitioning logic.

MAA: Generally assumes a uniform distribution; skew can create inefficient sparse arrays.

Query Pattern Analysis

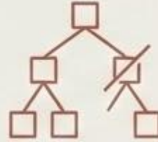
Strategy: Monitor usage logs to see which cuboids are actually queried.

Action: Adjust materialization strategy dynamically (don't compute what you don't use).

Choosing the Right Algorithm

Choose Multiway Array (MAA) when:

- Data fits into memory chunks.
- You need a Full Cube computation.
- Dimensionality is moderate (< 15 dimensions).



Choose BUC when:

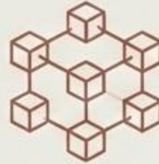
- You only need an Iceberg Cube (threshold-based).
- Data is highly skewed (e.g., zip code data).
- Pruning can significantly reduce computation.

Choose Shell Fragments when:

- Dimensionality is Very High (>15).
- Query patterns are ad-hoc/unpredictable.
- Storage space is limited.

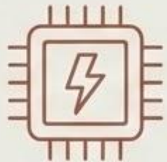
Moving Beyond Traditional Algorithms

Streaming Cubes:
Handling incremental updates in real-time without full re-computation.



Distributed Cubes:
Implementing algorithms using MapReduce/Spark for cluster computing.

Approximate Cubes:
prioritizing speed over precision for ‘good enough’ analytics (e.g., trend spotting).



GPU Acceleration:
Offloading array-based methods (like MAA) to GPUs for massive parallel speedups.

Approximate Cubes:
prioritizing speed over precision for “good enough” analytics (e.g., trend spotting).

Algorithms in Action

Google Analytics (Web Traffic):
Method: Shell Fragments
Why: Handles high dimensionality of web logs (User × Location × Device × Time × Referrer...).

Financial Risk (Fraud Detection):
Method: BUC
Why: Finds "Iceberg" patterns (e.g., specific transaction sequences occurring > 50 times).

Retail (Sales Reporting):
Method: MAA
Why: Efficient for standard sales cubes with moderate dimensions (Store × Product × Time).

Healthcare (Patient Cohorts):
Method: Cuboid Selection
Why: Quickly identifies the smallest dataset needed to analyze specific patient groups.