Lecture 2
Cube Computation

CA4010: Data Warehousing and Data Mining
2016/2017 Semester 1

Dr. Mark Roantree
Dublin City University

Agenda

1. **Efficient Computation of Data Cubes**
   - Understanding Dimensionality
   - Partial Materialization
2. **Methods for Data Cube Computation**
   - Full Cube Materialization
   - Iceberg Cube Materialization
   - Closed Cube Materialization
3. **General Strategies for Cube Computation**
4. **Multiway Array Aggregation**
5. **Bottom Up Construction**

Overview

- At the core of multidimensional data analysis is the efficient computation of aggregations across many sets of dimensions.
- In SQL terms, these aggregations are referred to as group-by's.
- Each group-by can be represented by a cuboid, where the set of group-by's forms a lattice of cuboids defining a data cube.
- It is important to understand issues relating to the efficient computation of data cubes.

The Curse of Dimensionality

- One approach to cube computation extends SQL so as to include a **compute cube** operator.
- The compute cube operator computes aggregates over all subsets of the dimensions specified in the operation.
- This can require excessive storage space, especially for large numbers of dimensions.
- We start with an intuitive look at what is involved in the efficient computation of data cubes.

Example

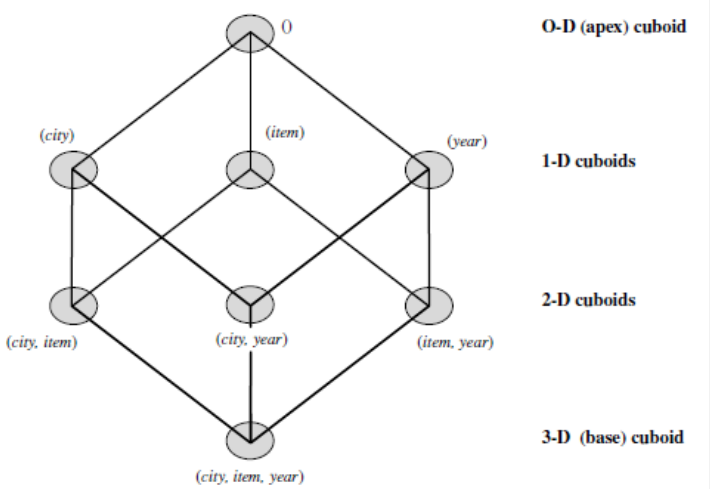
- A data cube is a lattice of cuboids.
- Suppose you need to create a data cube for *AllElectronics* sales that contains the following: *city*, *item*, *year*, and *sales* in dollars.
- This enables analyses such as:
  - Compute the sum of sales, grouping by city and item.
  - Compute the sum of sales, grouping by city.
  - Compute the sum of sales, grouping by item.

Calculating the Cuboids

- What is the total number of cuboids, or group-by's, that can be computed for this data cube?
- Taking the three attributes, *city*, *item*, and *year*, as the dimensions for the data cube, and *sales* as the **measure**, the total number of cuboids, or group by's, that can be computed for this data cube is: $2^3 = 8$.
- The possible group-by's are the following: {(city,item,year), (city,item), (city,year), (item,year), (city), (item), (year), ()}, where () means that the group-by is empty (i.e. the dimensions are not grouped).

Forming the Lattice

- These group-by's form a lattice of cuboids for the data cube, as shown in Figure 1.
- The base cuboid contains all three dimensions (*city*, *item*, and *year*) returning the total *sales* for any combination of the three dimensions.



Lattice Components

- The **apex cuboid**, or *0-D* cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales.
- The **base cuboid** is the least generalized (most specific) of the cuboids.
- The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as **all**.
- If we start at the apex cuboid and explore downward in the lattice, this is equivalent to drilling down within the data cube.
- If we start at the base cuboid and explore upward, this is the process of rolling up.

Dimensionality and Group By

- The SQL query containing no **group-by**, such as *compute the sum of total sales*, is a *zero*-dimensional operation.
- The SQL query containing one **group-by**, such as *compute the sum of sales, group by city*, is a *one*-dimensional operation.
- A cube operator on *n* dimensions is equivalent to a collection of **group by** statements, one for each subset of the *n* dimensions.
- Therefore, the **cube** operator is the *n*-dimensional generalization of the **group by** operator.

Define Cube

The data cube for this example could be defined as:

$$define\ cube\ sales\_cube[city, item, year] : sum(sales) \quad (1)$$

For a cube with n dimensions, there are a total of 2^n cuboids, including the base cuboid. A statement such as

$$compute\ cube\ sales\_cube \quad (2)$$

would explicitly instruct the system to compute the sales aggregate **cuboids** for all of the eight subsets of the set *{city,item,year}*, including the empty subset.

Precomputing Cubes

- OLAP analyses may need to access different cuboids for different queries.
- Therefore, it may seem like a good idea to compute all or at least some of the cuboids in a data cube in advance.
- Precomputation leads to fast response times and avoids some redundant computation.
- Most, if not all, OLAP products resort to some degree of precomputation of multidimensional aggregates.

Curse of Dimensionality

- A major challenge related to this precomputation is the required storage space: if all of the cuboids in a data cube are precomputed, especially when the cube has many dimensions.
- The storage requirements are even more excessive when many of the dimensions have associated concept hierarchies, each with multiple levels.
- This problem is referred to as the **curse of dimensionality**.

Cuboid count in an *n*-dimensional data cube

- If there were no hierarchies associated with each dimension, then the total number of cuboids for an *n*-dimensional data cube is 2^n.
- However, many dimensions do have hierarchies: time is usually explored at multiple conceptual levels, such as in the hierarchy "day < month < quarter < year".
- 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

$$Total\ Cuboids = \prod_{i=1}^{n}(L_i+1) \tag{3}$$

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

Understanding Total Cuboids

- 1 is added to L_i in Equation 3 to include the virtual top level, **all**.
- This formula is based on the fact that, at most, one abstraction level in each dimension will appear in a cuboid.
- For example, the time dimension as specified above has 4 conceptual levels, or 5 if we include the virtual level **all**.
- If the cube has 10 dimensions and each dimension has 5 levels (including **all**), the total number of cuboids that can be generated is $5^{10} \approx 9.8 \times 10^6$.

Issues of Scale

- The size of each cuboid also depends on the **cardinality** (number of distinct values) of each dimension.
- For example, if the *AllElectronics* branch in each city sold every item, there would be $| \textit{city} | \times | \textit{item} |$ tuples in the *city-item* group-by alone.
- As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by's will grossly exceed the (fixed) size of the input relation.

Choices for Data Cube Materialization

1. **No materialization**. Do not precompute any of the *nonbase* cuboids. This leads to computing expensive multidimensional aggregates on the fly, which can be extremely slow.
2. **Full materialization**. Precompute all of the cuboids. The resulting lattice of computed cuboids is referred to as the full cube and requires huge amounts of memory space.
3. **Partial materialization**. Selectively compute a proper subset of the whole set of possible cuboids. Alternatively, we may compute a subset of the cube, which contains only those cells that satisfy some user-specified criterion. We will use the term **subcube** to refer to this case where only some of the cells may be precomputed for various cuboids.

3-D Sample Cube

- Figure 2 shows a 3-D data cube for the dimensions *A*, *B*, and *C*, and an aggregate measure *M*.
- *ABC* is the base cuboid, containing all three of the dimensions.
- The aggregate measure *M*, is computed for each possible combination of the three dimensions.
- We will always use the term data **cube** to refer to a *lattice of cuboids* rather than an individual cuboid.

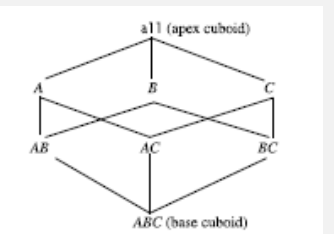


Figure 2: ABC Lattice of Cuboids

Cube Structure

- A cell in the base cuboid is a **base cell**.
- A cell from a non-base cuboid is an **aggregate cell**.
- An aggregate cell aggregates over one or more dimensions, where each aggregated dimension is indicated by a "*" in cell notation.
- Assume we have an n-dimensional data cube and let a=(a_1,a_2,...,a_n measures) be a cell from one of the cuboids making up the data cube.
- We say that a is an m-**dimensional cell** (from an m-dimensional cuboid) if exactly $m (m \leq n)$ values among $\{a_1,a_2,\ldots,a_n\}$ are not "*".
- If $m = n$, then a is a **base** cell; otherwise, it is an **aggregate** cell (where $m < n$).

Example: Base and Aggregate Cells

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

Example: Base and aggregate cells (2)

- An ancestor-descendant relationship may exist between cells.
- In an *n*-dimensional data cube, an *i*-D cell a=(a_1,a_2,...,a_n measures$_a$) is an **ancestor** of a *j*-D cell b=(b_1,b_2,...,b_n measures$_b$)
  and *b* is a **descendant** of *a* if and only if
  (1) i < j, and
  (2) for $1 \leq m \leq n$, $a_m = b_m$ whenever $a_m \neq$ "*".
- In particular, cell *a* is called a **parent** of cell *b*, and *b* is a **child** of *a*, if and only if $j = i + 1$ and *b* is a descendant of *a*.

Example: Ancestor and Descendant cells

Referring to our previous example,

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

Materialization and Sparse Cubes

- In many cases, a substantial amount of the cube's space is consumed by a large number of cells with very low measure values.
- This is because the cube cells are often quite sparsely distributed within a multiple dimensional space.
- For example, a customer may only buy a few items in a store at a time, generating only a few non-empty cells and leaving most other cells empty.
- If a cube contains many sparse cuboids, the cube is **sparse**.
- In such situations, it is useful to materialize only those cells in a cuboid (group-by) whose measure value is above some minimum threshold.

Iceberg Cubes

- In a data cube for *sales*, we may wish to materialize only those cells for which *count* ≥ 10 (where at least 10 tuples exist for the cell's given combination of dimensions),
  or only those cells representing *sales* $\geq$ €100.
- This not only saves processing time and disk space, but also leads to a more focused analysis.
- The cells that cannot pass the threshold are likely to be too trivial to warrant further analysis.
- Such partially materialized cubes are known as **iceberg cubes**.

Sample Iceberg using min_sup

- The minimum threshold is called the **minimum support threshold**, or *minimum support (min_sup)*.
- By materializing only a fraction of the cells in a data cube, the result is seen as the *tip of the iceberg*, where the **iceberg** contains *all cells*.
- An iceberg cube can be specified with the SQL query shown in the following example.

```
1 compute cube sales_iceberg as
2 select month, city, customer group, count(*)
3 from salesInfo
4 cube by month, city, customer group
5 having count(*) >= min_sup
```

Sample Iceberg Cube

Compute Cube

- The **compute cube** statement specifies the precomputation of the iceberg cube *sales_iceberg*, with the 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.
- When computing the full cube, each group-by corresponds to a **cuboid** in the data cube lattice.

Iceberg Condition

- The constraint specified in the **having** clause is known as the **iceberg condition**.
- Here, the iceberg measure is *count*.
- The iceberg cube computed here can also be used to answer group-by queries on *any combination* of the specified dimensions of the form **having count(*)** $\geq$ v, where v $\geq$ min sup.
- Instead of *count*, the iceberg condition could specify more complex measures, such as *average*.

Full Computation

- If we omit the **having** clause, we compute the full cube.
- We will call this full cube *sales_cube*.
- The iceberg cube *sales_iceberg*, excludes all the cells of *sales_cube* whose count is less than *min_sup*.
- Note: if we set the minimum support to 1 in *sales_iceberg*, the resulting cube would be the full cube, *sales_cube*.

Many Cells: little new data

- Even reducing the computation of trivial aggregate cells, we could still end up with a large number of uninteresting cells to compute.
- For example, suppose that there are **2 base cells** for a database of 100 dimensions, denoted as {(a1,a2,a3,. . . ,a100):10, (a1,a2,b3,. . . ,b100) : 10}, where each has a cell count of 10.
- If the minimum support is set to 10, there are too many cells to compute and store, with most of them *not* interesting.
- For example, there are 2^{101}-6 distinct aggregate cells, like {(a_1,a_2,a_3,a_4,. . . ,$_{99}$,*):10,. . . , (a_1,a_2,*,a_4,. . . ,$_{99}$,a_{100}):10,. . . , (a_1,a_2,a_3,*,. . . ,*,*):10}, but most of them do not contain much new information.

Interesting Cells

- If we ignore all of the aggregate cells that can be obtained by replacing some constants by *'s while keeping the same measure value, there are only three distinct cells left:
  - {(a1,a2,a3,. . . ,a100):10,
  - (a1,a2,b3,. . . ,b100):10,
  - (a1,a2,*,. . . ,*):20}.
- That is, out of 2^{101}-6 distinct aggregate cells, only 3 really offer new information.

Closed Cube

- To systematically compress a data cube, we need to introduce the concept of **closed coverage**.
- A cell *c*, is a *closed cell* if there exists no cell *d*, such that *d* is a specialization (descendant) of cell *c*, and *d* has the same measure value as *c*.
- *d* is a specialization of *c* if *d* is obtained by replacing a * in *c* with a non-* value, and *d* has the same measure value as *c*.
- A **closed cube** is a data cube consisting of only closed cells.

Closed Cube Example

- For example, the three cells derived earlier are the **three closed cells** of the data cube for the data set: {(a_1,a_2,a_3,. . . ,a_{100}):10, (a_1,a_2,b_3,. . . ,b_{100}):10}.
- They form the lattice of a closed cube as shown in Figure 3.

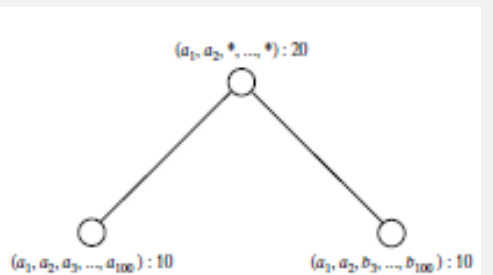


Figure 3: 3 Closed Cells forming the lattice of a closed cube

Closed Cube Example (2)

- Other **non-closed cells** can be derived from their corresponding *closed* cells in this lattice.
- For example, "(a1,*,*,. . . ,*):20" can be derived from "(a1,a2,*,. . . ,*):20"' because the former is a generalized non-closed cell of the latter.
- Similarly, we have "(a1,a2,b3,*,. . . ,*):10".

Other Methods

- Another strategy for partial materialization is to precompute only the cuboids involving a small number of dimensions, such as 3 to 5.
- These cuboids form a cube shell for the corresponding data cube.
- Queries on additional combinations of the dimensions must be computed on the fly.
- For example, we could compute all cuboids with 3 dimensions or less in an n-dimensional data cube, resulting in a cube shell of size 3.
- However, this can still result in a large number of cuboids to compute, particularly when n is large.
- Alternatively, we can choose to precompute only portions or *fragments* of the cube shell, based on cuboids of interest.

General Strategies for Cube Computation

There are a number of approaches to optimising cube computation.

1. Sorting, hashing, and grouping
2. Simultaneous aggregation
3. Aggregation from the smallest child
4. The Apriori pruning method

Optimization Technique 1: Sorting, hashing, and grouping.

- They should be applied to the dimension attributes in order to *reorder* and *cluster* related tuples.
- In cube computation, aggregation is performed on the tuples (or cells) that share the *same set of dimension values*.
- One should explore sorting, hashing, and grouping operations to access and group data together to facilitate computation of such aggregates.
- For example, to compute total sales by *branch*, *day*, and *item*, it is more efficient to sort tuples or cells by *branch*, and then by *day*, and then group them according to the *item* name.

Optimization Technique 2: Simultaneous Aggregation

- In cube computation, it is efficient to compute higher-level aggregates from previously computed *lower-level aggregates*, rather than from the base fact table.
- For example, to compute sales by *branch*, we can use the intermediate results derived from the computation of a lower-level cuboid, such as *sales by branch and day*.
- This technique can be extended to compute as many cuboids as possible simultaneously, to reduce disk reads.

Optimization Technique 3: Aggregation from the smallest child

- When there exist multiple child cuboids, it is usually more efficient to compute the desired parent (more generalized) cuboid from the smallest, previously computed child cuboid.
- For example, to compute a sales cuboid C_{branch}, when there exist two previously computed cuboids, $C_{\{branch,year\}}$ and $C_{branch,item}$, it is obviously more efficient to compute C_{branch} from the former, than from the latter if there are many more distinct *items* than distinct *years*.
- Many other optimization tricks may further improve the computational efficiency.
- For example, string dimension attributes can be mapped to integers with values ranging from zero to the cardinality of the attribute.

Optimization Technique 4: The Apriori pruning method

- This plays a particularly important role in efficient iceberg cube computation.
- The **Apriori property** states that if a given cell does not satisfy minimum support, then no descendant (i.e. more specialized or detailed version) of the cell will satisfy minimum support either.
- This property can be used to substantially reduce the computation of iceberg cubes.

Apriori Pruning

- Recall that the specification of **iceberg** cubes contains an *iceberg condition*, which is a constraint on the cells to be materialized.
- A common iceberg condition is that the cells must satisfy a **minimum support threshold**, such as a minimum count or sum.
- Thus, the Apriori property can be used to prune exploration of the descendants of the cell.
- For example, if the count of a cell c, in a cuboid is less than a minimum support threshold v, then the count of any of c's descendant cells in the lower-level cuboids can never be $\geq v$ and thus, can be pruned.

Apriori Efficiency

- In other words, if the iceberg condition specified in a having clause is violated for some cell c, then every descendant of c will also violate that condition.
- For example, if $C_{branch,item}$ has a count of 10, then it is impossible for $C_{branch,year,item}$ to have a greater count.
- This form of pruning was made popular in association rule mining, yet also aids in data cube computation by cutting processing time and disk space requirements.

Multiway Array Aggregation for Full Cube Computation

- The Multiway Array Aggregation (or simply MultiWay) method computes a full data cube by using a multidimensional array as its basic data structure.
- We will examine a popular method with an example to demonstrate how it works.

Stage 1: Partition the array into chunks

- A **chunk** is a sub-cube that is small enough to fit into the memory available for cube computation.
- **Chunking** is a method for dividing an *n*-dimensional array into small *n*-dimensional chunks, where each chunk is stored as an object on disk.
- The chunks are compressed so as to remove wasted space resulting from empty array cells: those cells that do not contain any valid data, whose cell count is zero.
- For instance, "*chunkID + offset*" can be used as a cell addressing mechanism to **compress a sparse array structure** and when searching for cells within a chunk.
- Such a compression technique is powerful enough to handle sparse cubes, both on disk and in memory.

Stage 2: 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.
- The trick is to exploit this ordering so that partial aggregates can be computed simultaneously, and any unnecessary revisiting of cells is avoided.
- Because this chunking technique involves *overlapping* some of the aggregation computations, it is referred to as **multiway array aggregation**.
- It performs **simultaneous aggregation**: it computes *aggregations* simultaneously on multiple dimensions.

Example: Multiway array cube computation

- Consider a 3-D data array containing the three dimensions *A, B,* and *C*.
- The 3-D array is partitioned into 64 chunks as shown in figure 4.
- 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 $a_0b_0c_0, a_1b_0c_0, \ldots, a_3b_3c_3$, respectively.

3-D Array for Dimensions A,B,C in 64 chunks

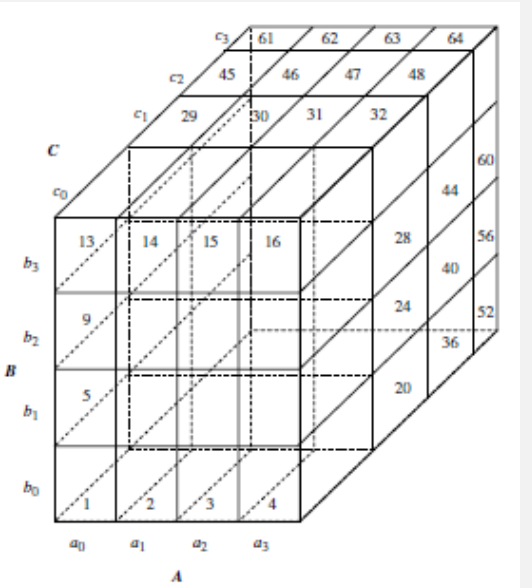


Figure 4: 64 chunk array

Initial Cardinalities and Sizes

- 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.
- 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 of the cuboids defining this cube.

Cuboids for Resulting Full Cube

- The **base cuboid**, denoted by *ABC* (from which all of 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*, corresponding 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(): there is *no group-by* here.
  - This cuboid, consisting of a single value, must be computed.
  - If the data cube measure is *count*, then the value to be computed is the total count of all of tuples in *ABC*.

Computing Random Chunks

- Consider the ordering labeled from 1 to 64, shown in figure 4 and assume we would like to compute the b_0c_0 chunk of the *BC* cuboid.
- We allocate space for this chunk in *chunk memory*.
- By scanning chunks 1 to 4 of *ABC*, the b_0c_0 chunk is computed.
- To do this, 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 chunks of *ABC*: 5 to 8.

Multiway Computation

- 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?*
- Yes: this is where the **multiway computation** or *simultaneous aggregation* idea comes in.

Simultaneous Computation

- For example, when scanning chunk 1 ($a_0b_0c_0$) perhaps for the computation of the 2-D chunk b_0c_0 of *BC*, all other 2-D chunks relating to $a_0b_0c_0$ can be simultaneously computed.
- In other words, 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 also be computed.
- In other words, multiway computation *simultaneously aggregates* to each of the 2-D planes while a 3-D chunk is in memory.

Efficiency and Chunk Scanning (1)

How can different orderings of chunk scanning and of cuboid computation 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 400x4000 = 1,600,000).
- The second largest 2-D plane is *AC* (of size 40x4000 = 160,000).
- *AB* is the smallest 2-D plane (with a size of 40x400 = 16,000).

Efficiency and Chunk Scanning (2)

- Assume the chunks are scanned from chunk 1 to 64.
- By scanning in this order, one chunk of the largest 2-D plane BC, is fully computed for each row scanned.
- Thus, b_0c_0 is fully aggregated after scanning the row containing chunks 1 to 4;
  b_1c_0 is fully aggregated after scanning chunks 5 to 8, and so on.

Efficiency and Chunk Scanning (3)

- 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.
- Thus, 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.

Algorithm Walk Through

```
1  For BC:
2  Use 1 chunk, read 1-4
3  Compute b0c0, write to disk
4  Same chunk, read 5-8
5  Compute b1c0, write to disk
6  Same chunk, read 9-16
7  Compute b2c0, write to disk
8  and thus, we can read BC using a single
       chunk
9
10 For AC:
11 Use chunk 1, read 1,5,9,13
12 Compute a0c0, write to disk
13 Use chunk 2, read 2,6,10,14
14 Compute a1c0, write to disk
15 Use chunk 3, read 3,7,11,15
16 Compute a2c0, write to disk
17 Use chunk 4, read 4,8,12,16
18 Compute a3c0, write to disk
19 and thus, we can read AC using 4 chunks (one
       row)
20
21 For AB:
```


Index Selection

- 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:
  - 40x400 (for the entire *AB* plane) +
  - 40x4000 / 4 (for one row of the *AC* plane) +
  - 400 / 4 x 4000 / 4 (for one chunk of the *BC* plane)
  - = 16,000+40,000+100,000
  - = 156,000 memory units.

22 You must read to chunk 49 for a0b0

An Alternate Ordering

- Suppose, instead, that the chunks are scanned in the order 1, 17, 33, 49, 5, 21, 37, 53, and so on.
- That assumes 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:
  - 400x4000 (for the entire *BC* plane) +
  - 40x1000 (for one row of the *AC* plane) +
  - 10x100 (for one chunk of the *AB* plane)
  - = 1,600,000 + 40,000 + 1000
  - = 1,641,000 memory units.
- This is more than 10 times the memory requirement of the scan ordering of 1 to 64!

Efficient Ordering

- Similarly, we can work out the minimum memory requirements for the multiway computation of the 1-D and 0-D cuboids. data cube computation.
- The most efficient ordering is the chunk ordering of 1 to 64.

Multiway and Iceberg

What would happen if we tried to use MultiWay to compute iceberg cubes?

- The Apriori property states that if a given cell does not satisfy minimum support, then neither will any of its descendants.
- Unfortunately, MultiWay's computation starts from the base cuboid and progresses upward, toward more generalized, ancestor cuboids.
- It cannot take advantage of Apriori pruning, which requires a parent node to be computed before its child nodes.
- For example, if the count of cell c in AB does not satisfy the minimum support specified in the iceberg condition, then we cannot prune computation of c's ancestors in the A or B cuboids, because the count of these cells may be greater than c.

BUC: Computing Iceberg Cubes from the Apex Cuboid Downward

- BUC is an algorithm for the computation of sparse and iceberg cubes.
- Unlike MultiWay, BUC constructs the cube *from* the **apex** cuboid toward the **base** cuboid.
- This allows BUC to share data partitioning costs.
- This order of processing also allows BUC to prune during construction, using the Apriori property.

Cuboid Lattice

- Figure 5 shows a lattice of cuboids, making up a 3-D data cube with dimensions *A*, *B*, and *C*.
- The apex (0-D) cuboid, representing the concept **all** (*,*,*), is at the top of the lattice: the most aggregated or generalized level.

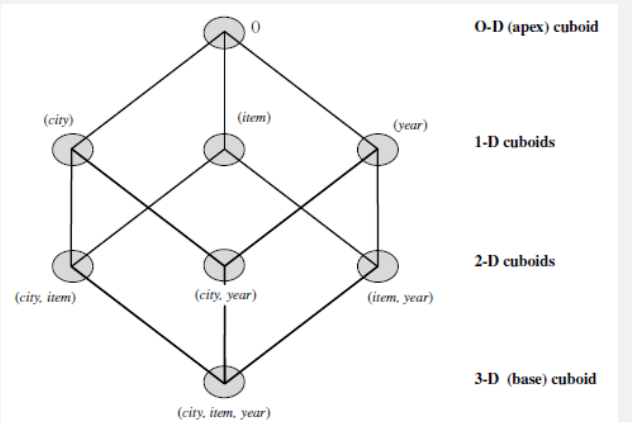


Figure 5: Lattice of Cuboids

Recap

- The 3-D base cuboid *ABC*, is at the bottom of the lattice.
- It is the least aggregated (most detailed or specialized) level.
- This representation of a lattice of cuboids, with the **apex** at the top and the **base** at the bottom, is commonly accepted in data warehousing.
- It consolidates the notions of drill-down and roll-up.

BUC: Bottom-Up Construction

- In reality, the order of processing of BUC is actually top-down!
- The creators of BUC view a 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.
- We see drill-down as drilling from the apex cuboid down toward the base cuboid and thus, the exploration process of BUC is regarded as top-down.
- BUC's exploration for the computation of a 3-D data cube is shown in Figure 6.

BUC Computation

Computation starts from the Apex Cuboid!

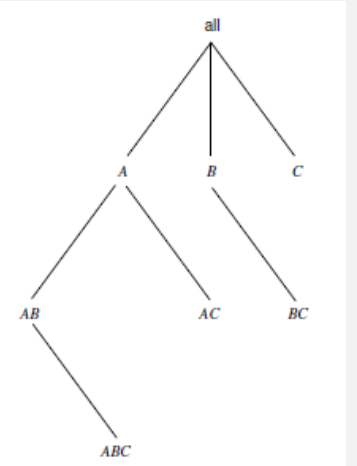


Figure 6: BUC's exploration for the computation of a 3-D Cube

BUC Algorithm (part 1)

Initially, the algorithm is called with the input set of tuples.

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 in order 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.

Figure 7: BUC Algorithm Paramters and Output

BUC explained

- BUC aggregates the entire input (line 1) and writes the resulting total (line 3).
- (Line 2 is an optimization feature that is discussed later in our example.)
- For each dimension *d* (line 4), the input is partitioned on *d* (line 6).
- On return from `Partition()`, *dataCount* contains the total number of tuples for each distinct value of dimension *d*.
- Each distinct value of *d* forms its own partition.

BUC Algorithm (part 2)

Method:

```
(1)  Aggregate(input); // Scan input to compute measure, e.g., count. Place result in outputRec.
(2)  if input.count() == 1 then // Optimization
         WriteAncestors(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], d+1); // aggregate on next dimension
(13)         endif
(14)         k +=c;
(15)     endfor
(16)     outputRec.dim[d] = all;
(17) endfor
```

Figure 8: BUC Algorithm

BUC explained (2)

- Line 8 iterates through each partition.
- Line 10 tests the partition for minimum support: if the number of tuples in the partition satisfies ($\geq$) the minimum support, then the partition becomes the input relation for a recursive call made to BUC, which computes the iceberg cube on the partitions for dimensions *d+1* to *numDims* (line 12).
- Note that for a full cube (where minimum support in the having clause is 1), the minimum support condition is always satisfied.
- Thus, the recursive call descends one level deeper into the lattice.
- Upon return from the recursive call, we continue with the next partition for *d*.
- After all the partitions have been processed, the entire process is repeated for each of the remaining dimensions.

Worked Example

Consider the iceberg cube expressed in SQL as follows:

```
1 compute cube iceberg cube as
2 select A, B, C, D, count(*)
3 from R
4 cube by A, B, C, D
5 having count(*) >= 3
```

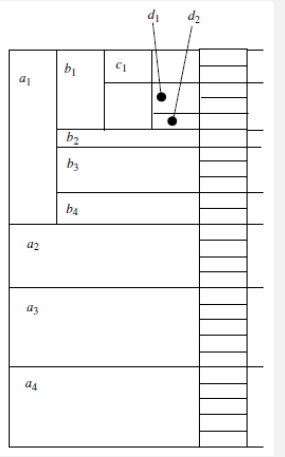
Sample Iceberg Cube

BUC Example

How does BUC constructs the iceberg cube for the dimensions A, B, C, and D, where the minimum support count is 3?

- 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 minimum support (ie. that have 3 tuples).
- Figure 9 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 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 and Apriori (1)

- BUC uses the Apriori property for optimisation.
- 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 $(a_1, *, *, *)$.
- If $(a_1, *, *, *)$ satisfies the minimum support, a recursive call is made on the partition for a_1 and BUC partitions a_1 on dimension B.
- It checks the count of $(a_1, b_1, *, *)$ to see if it satisfies 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.

Figure 9: BUC partitioning given a sample 4-D data set

BUC and Apriori (2)

- Assume the cell count for $(a_1, b_1, c_1, *)$ is 2, which does not satisfy the minimum support.
- According to Apriori: if a cell does not satisfy minimum support, then neither can any of its descendants.
- Thus, BUC prunes any further exploration of $(a_1, b_1, c_1, *)$ and avoids partitioning the 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.

Process Optimisation

- 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.
- 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 ancestor group-by's.
- This is particularly useful since, in practice, many partitions have a single tuple.

BUC and Performance

- The performance of BUC is sensitive to the order of the dimensions and to skew in the data.
- Ideally, the most discriminating dimensions should be processed first.
- Dimensions should be processed in order of decreasing cardinality.
- The higher the cardinality is, the smaller the partitions are, and thus, the more partitions there will be, thereby providing BUC with greater opportunity for pruning.
- Similarly, the more uniform the dimension (having less skew), the better it is for pruning.

Comparing BUC and Multiway

- BUC's major contribution is the idea of sharing partitioning costs.
- However, unlike MultiWay, it does not share the computation of aggregates between parent and child group-by's.
- For example, the computation of cuboid *AB* does not help that of *ABC*.
- The latter needs to be computed essentially from scratch.