DATA CUBES

Hari Sundaram

hs1@illinois.edu

http://sundaram.cs.illinois.edu

adapted from slides by Jiawei Han and Kevin Chang

BASIC CONCEPTS

Methods   Advanced Processing   Multidimensional   Summary

Important!

academic integrity

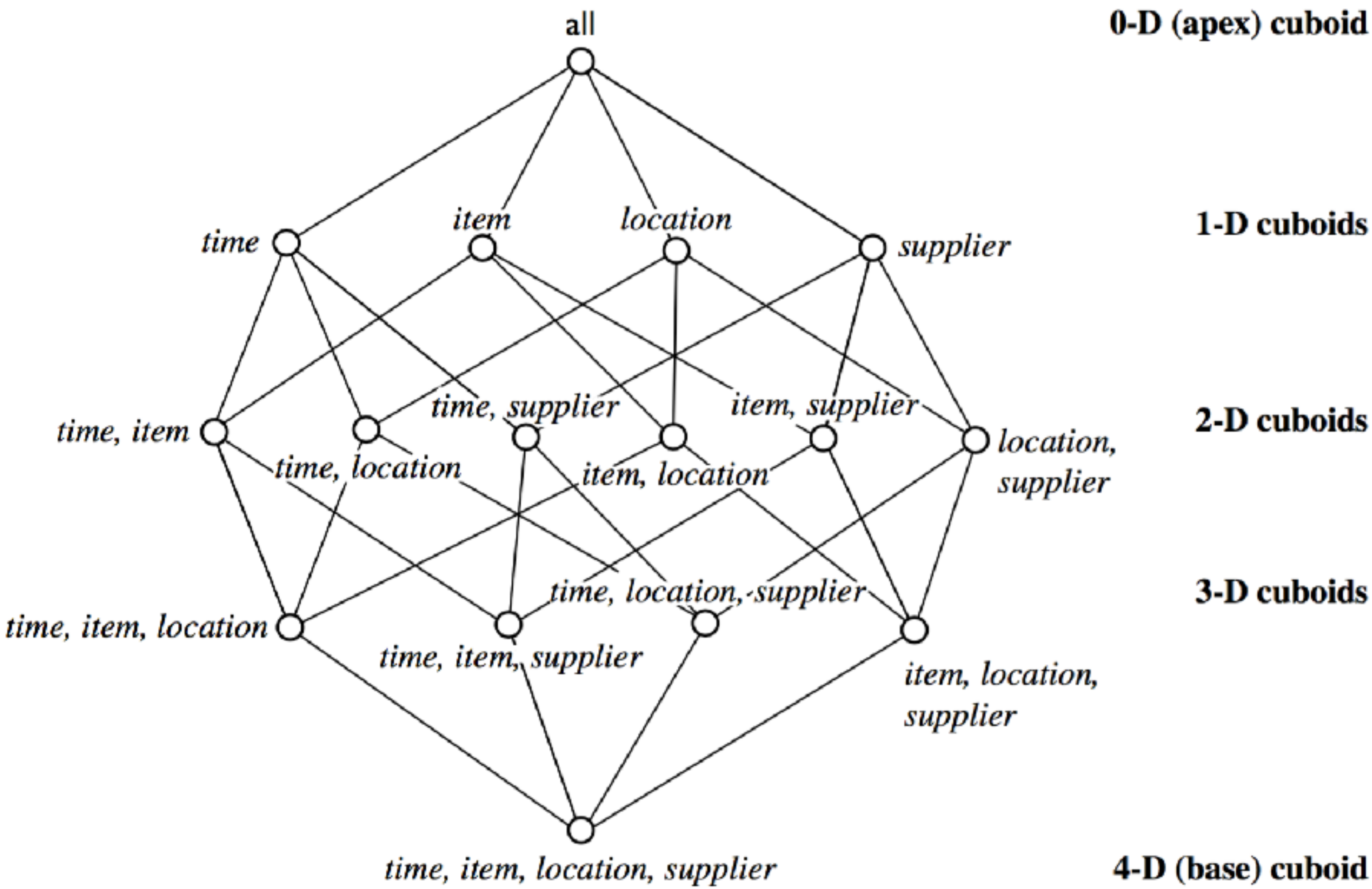
zero tolerance policy!

You are encouraged to form a study group to discuss the homework and the programming assignments but are expected to compete the homework and programming assignments **completely on your own without recourse to notes from the group discussions**.

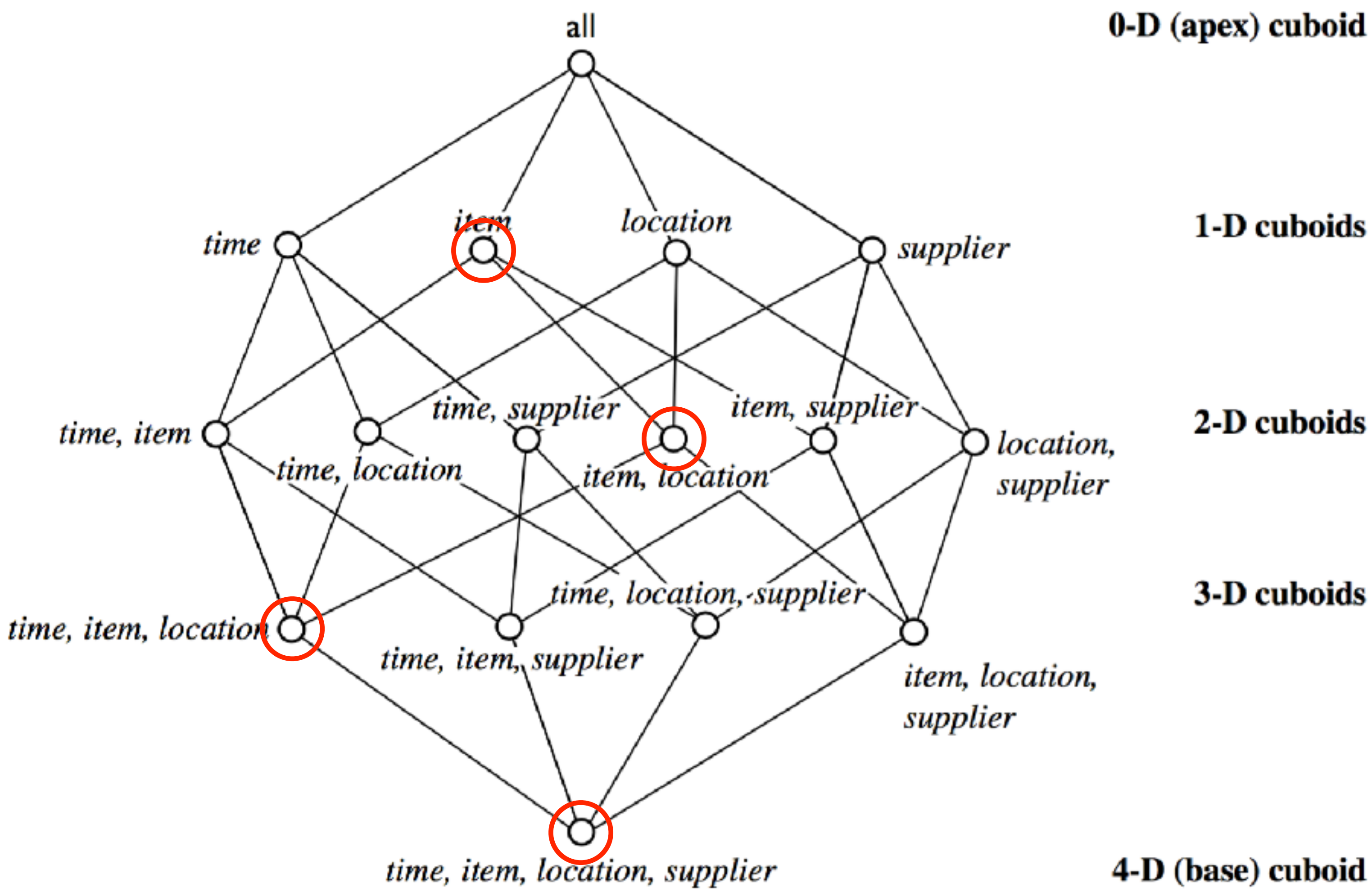
Plagiarism: **It is an academic violation to copy, to include text from other sources, including online sources, without proper citation.**

Any student found to be violating this code will be subject to **disciplinary** action.

data cube: a lattice of cuboids



DATA CUBE: A LATTICE OF CUBOIDS



Base vs. aggregate cells; ancestor vs. descendant cells; parent vs. child cells

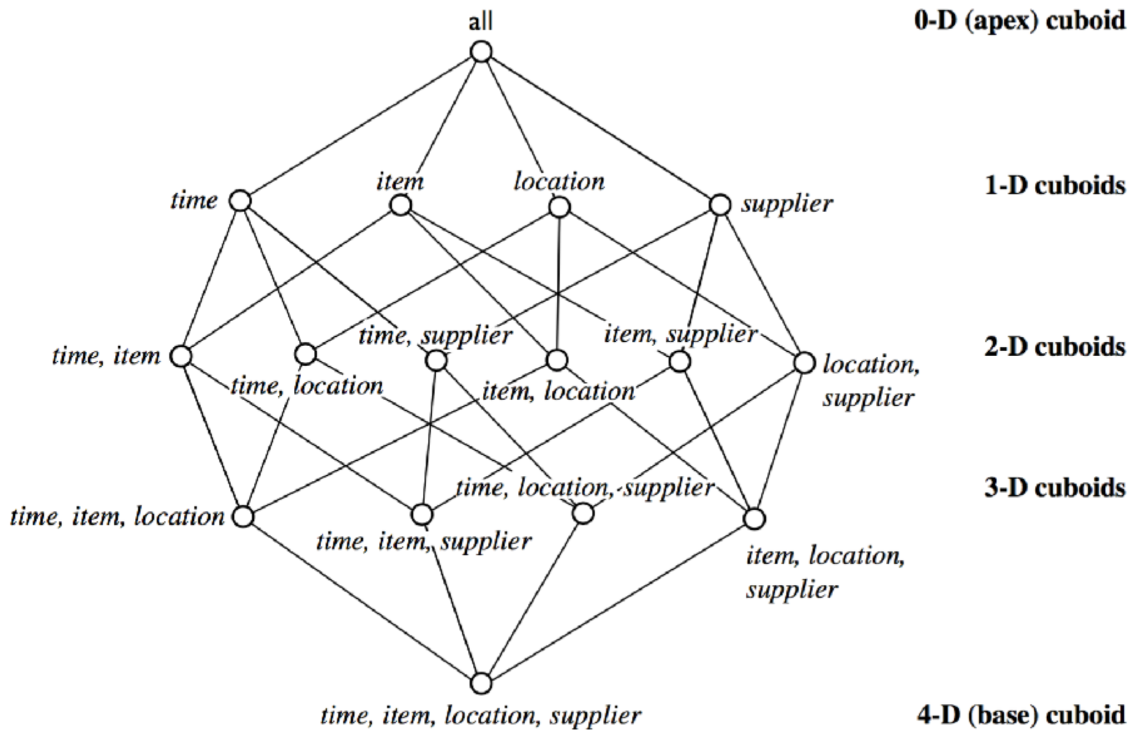
(9/15, milk, Urbana, Dairy_land)

(9/15, milk, Urbana, *)

(*, milk, Urbana, *)

(*, milk, Chicago, *)

(*, milk, *, *)



CUBE MATERIALIZATION

Full cube vs. iceberg cube

compute cube sales iceberg as

select month, city, customer group, count(*)

from salesInfo

cube by month, city, customer group

having count(*) >= min support

Computing only the cuboid cells whose measure satisfies the iceberg condition

Only a small portion of cells may be "above the water" in a sparse cube

a priori principle

How many aggregate cells
if “having count >= 1”?

	Q1	Q2	Q3	Q4
Milk	**5**	**1**	**4**	**1**
Poultry	**1**	**10**	**5**	**3**
Beer	**6**	**7**	**1**	**11**
Chocolate	**1**	**8**	**12**	**2**

base cell

	Q1	Q2	Q3	Q4
Milk	5	1	4	1
Poultry	1	10	5	3
Beer	6	7	1	11
Chocolate	1	8	12	2

base cell

how many aggregates?

(Milk, Q1)

(Beer, Q2)

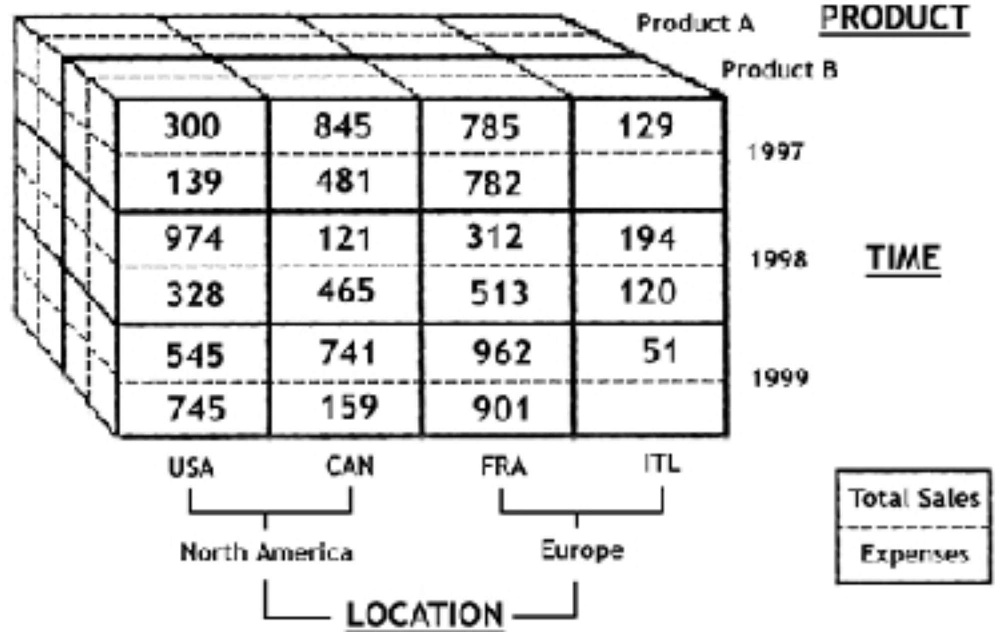
(Milk,*) (Beer,*)

(*,*) (*,*)

(*,Q1) (*,Q2)

3 + 3 - 1 = 5

Exercise!



$$\{ (a_1,a_2,a_3,\ldots, a_{100}):10, (b_1,b_2,b_3,\ldots, b_{100}):10 \}$$

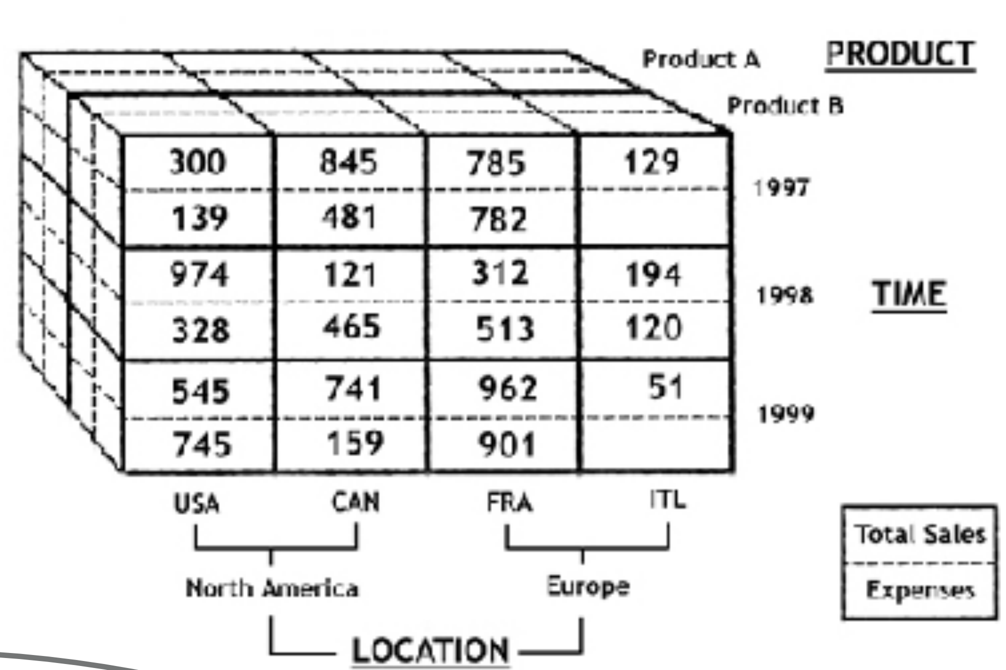
how many aggregate cells have more than or equal to 10?

$$2^{100} + 2^{100} - 1 - 2 = 2^{101} - 3$$

two base cells

one common aggregate cell (*)

Exercise!



2^2

$$\{\ (\underline{a_1,a_2},a_3,\ldots,a_{100}):10,\ (\underline{a_1,a_2},b_3,\ldots,b_{100}):10\ \}$$

how many aggregate cells have counts greater than or equal to 10?

$$2^{100} + 2^{100} - 4 - 2 = 2^{101} - 6$$

two base cells

four common aggregate cells

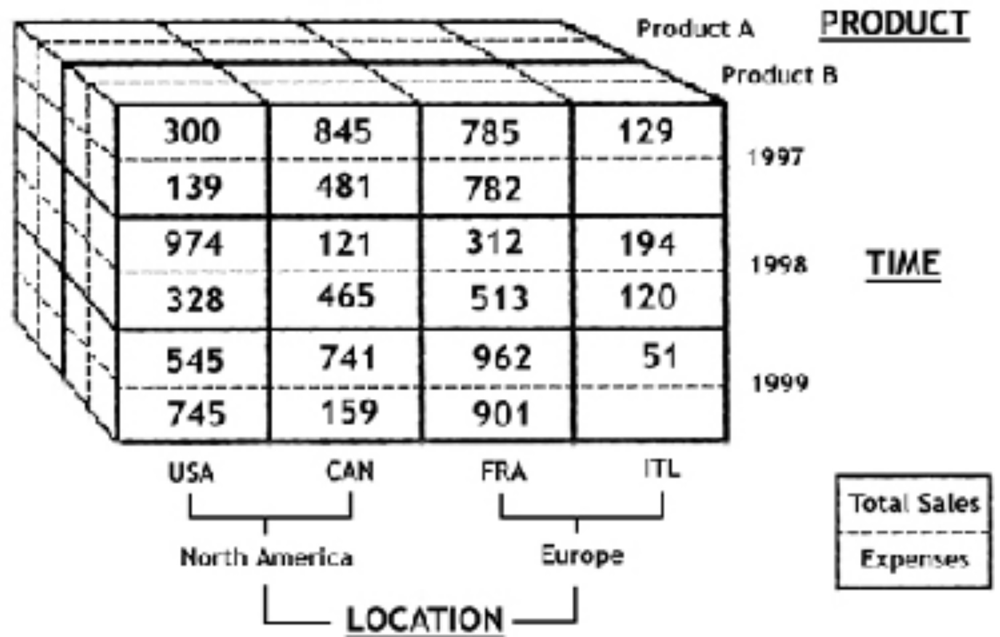
$$\begin{array}{c} c: m_1 \\ \downarrow \\ d: m_2 \end{array} \qquad m_1 \neq m_2$$

Closed cell c: if there exists <u>no</u> cell d, such that d is a descendant of c, and d has the same measure value as c.

	Q1	Q2	Q3	Q4
Milk	**5**	**1**	**4**	**1**
Poultry	**1**	**10**	**5**	**3**
Beer	**6**	**7**	**1**	**11**
Chocolate	**1**	**8**	**12**	**2**

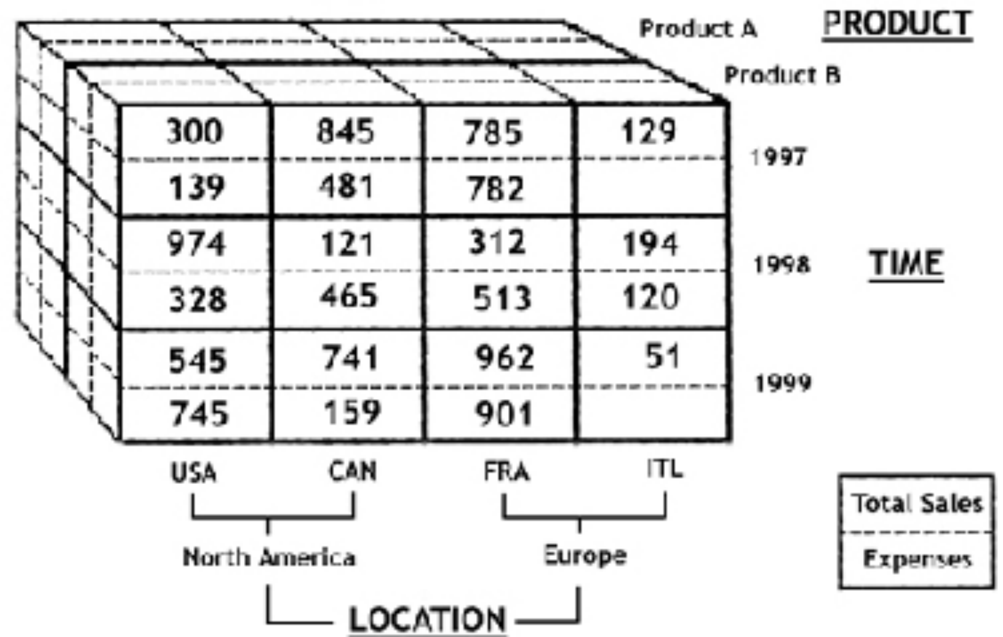
closed cube

Exercise!



$$\{\ (a_1,a_2,a_3,\ldots,a_{100}):10,\ (a_1,a_2,b_3,\ldots,b_{100}):10\ \}$$

how many closed cells?



cube shell

Pre-compute only the cuboids involving a small # of dimensions, e.g., 3

More dimension combinations will need to be computed on the fly

EFFICIENT COMPUTATION ROADMAP

General cube computation heuristics (Agarwal et al.'96)

Computing full/iceberg cubes: 3 methodologies

- Bottom-Up: Multi-Way array aggregation (Zhao, Deshpande & Naughton, SIGMOD'97)
- Top-down:
- BUC (Beyer & Ramarkrishnan, SIGMOD'99)
- H-cubing technique (Han, Pei, Dong & Wang: SIGMOD'01)

Integrating Top-Down and Bottom-Up:

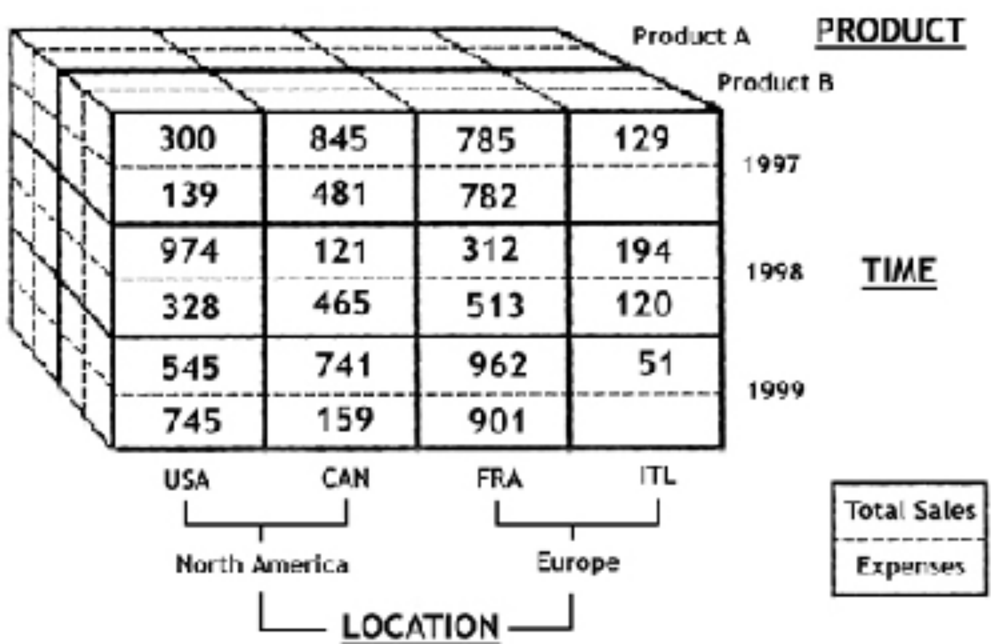
- Star-cubing algorithm (Xin, Han, Li & Wah: VLDB'03)

High-dimensional OLAP: A Minimal Cubing Approach (Li, et al. VLDB'04)

Computing alternative kinds of cubes:

- Partial cube, closed cube, approximate cube, etc.

GENERAL HEURISTICS



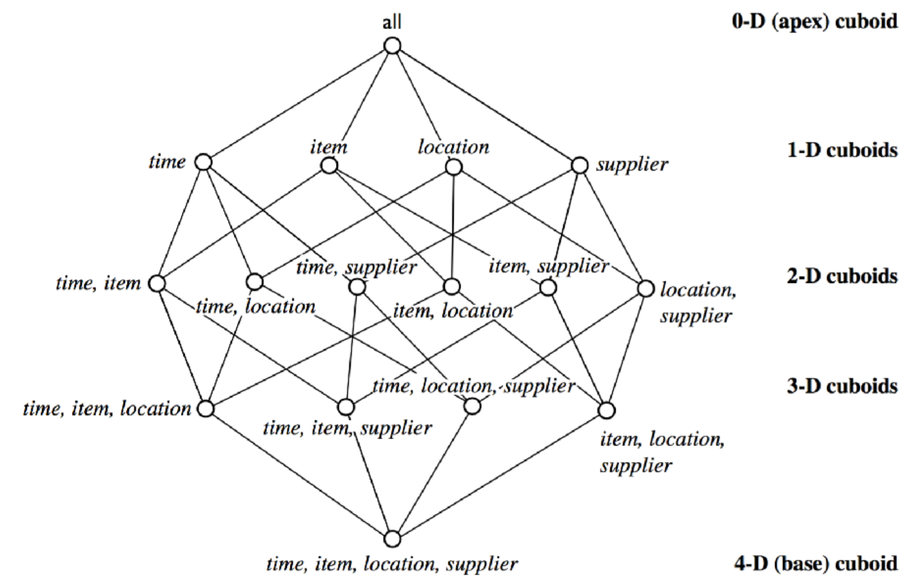
Agarwal, S., Agrawal, R., Deshpande, P. M., Gupta, A., Naughton, J. F., Ramakrishnan, R., & Sarawagi, S. (1996, September). On the computation of multidimensional aggregates. In VLDB (Vol. 96, pp. 506-521).

Sorting, hashing, and grouping operations are applied to the dimension attributes in order to reorder and cluster related tuples

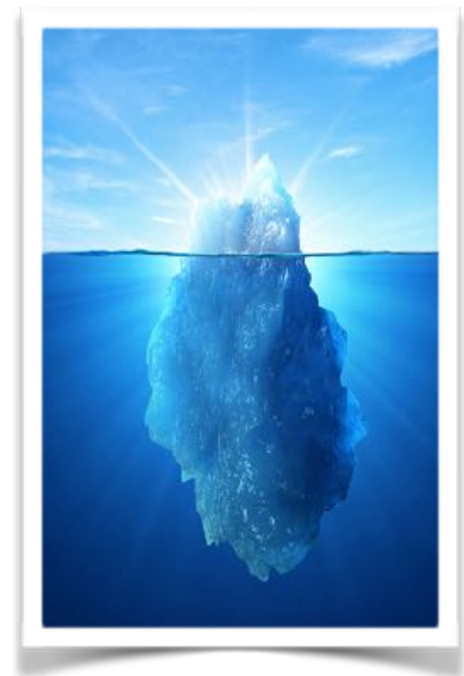
Aggregates may be computed from previously computed aggregates, rather than from the base fact table

- Smallest-child: computing a cuboid from the smallest, previously computed cuboid
- Cache-results: caching results of a cuboid from which other cuboids are computed to reduce disk I/Os
- Amortize-scans: computing as many as possible cuboids at the same time to amortize disk reads
- Share-sorts: sharing sorting costs cross multiple cuboids when sort-based method is used
- Share-partitions: sharing the partitioning cost across multiple cuboids when hash-based algorithms are used

Closed cell c: if there exists no cell d, such that d is a descendant of c, and d has the same measure value as c.



BASIC CONCEPTS SUMMARY



iceberg cube

Methods   Advanced Processing   Multidimensional   Summary

$2^n - 1$

base cell aggregates

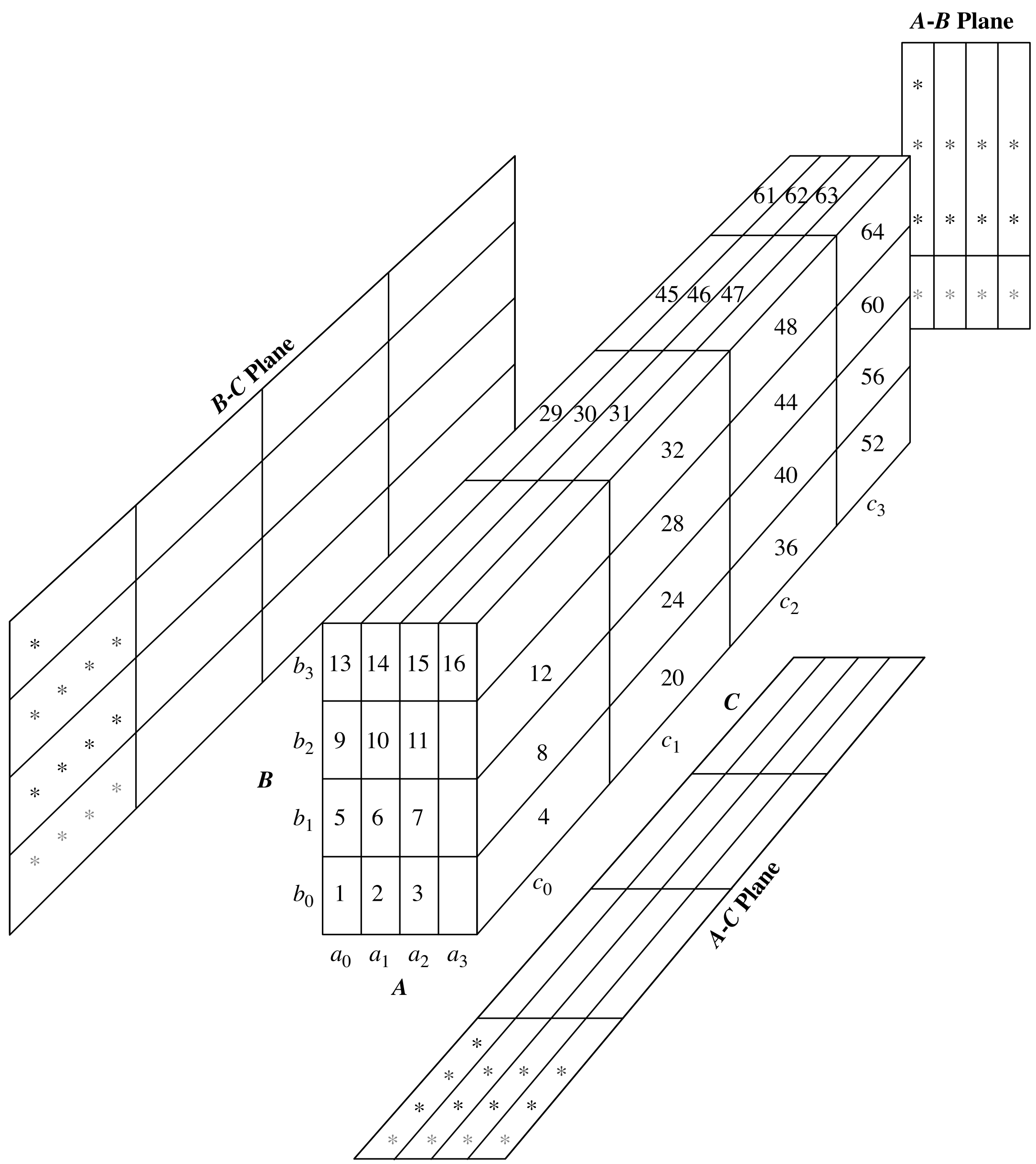
Multi-Way Array Aggregation

BUC

High-Dimensional OLAP

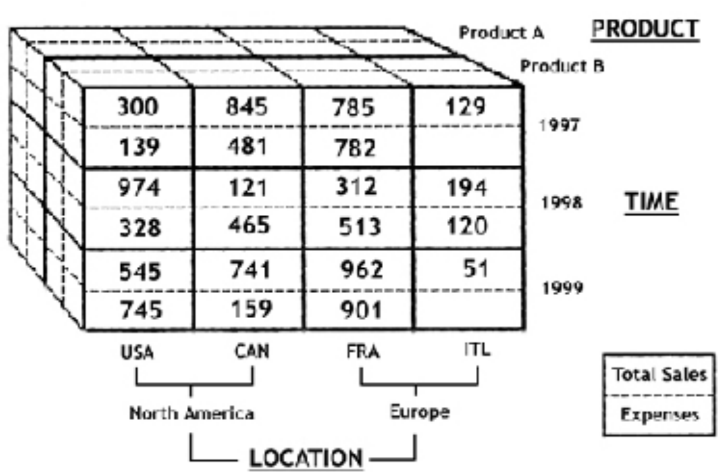
DATA CUBE METHODS

MULTI-WAY AGGREGATION



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

MOLAP

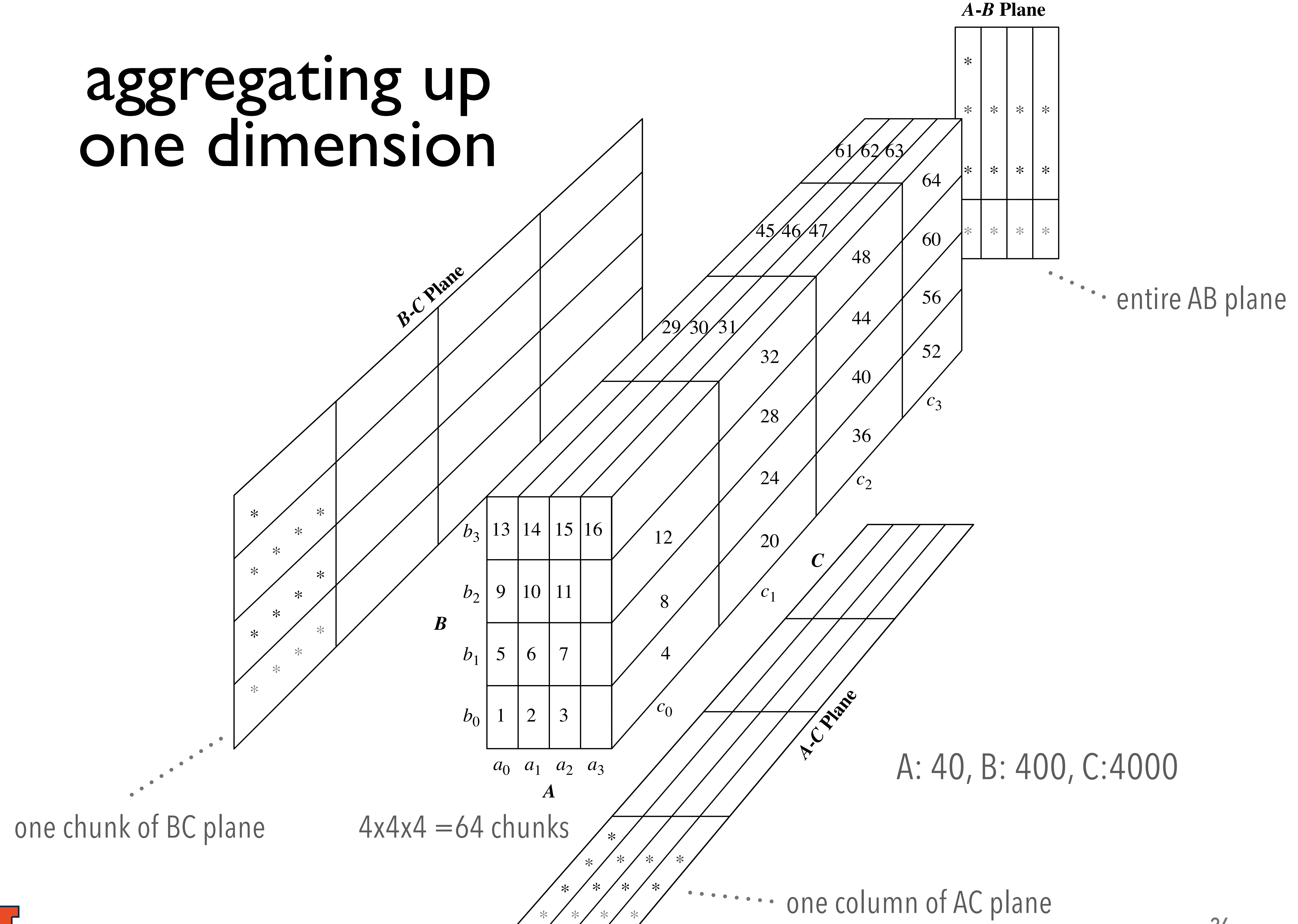


Partition arrays into chunks (a small subcube which fits in memory).

Compressed sparse array addressing: (chunk_id, offset)

Compute aggregates in "multiway" by visiting cube cells in the order which minimizes the number of cell visits, and which reduces memory access and storage cost.

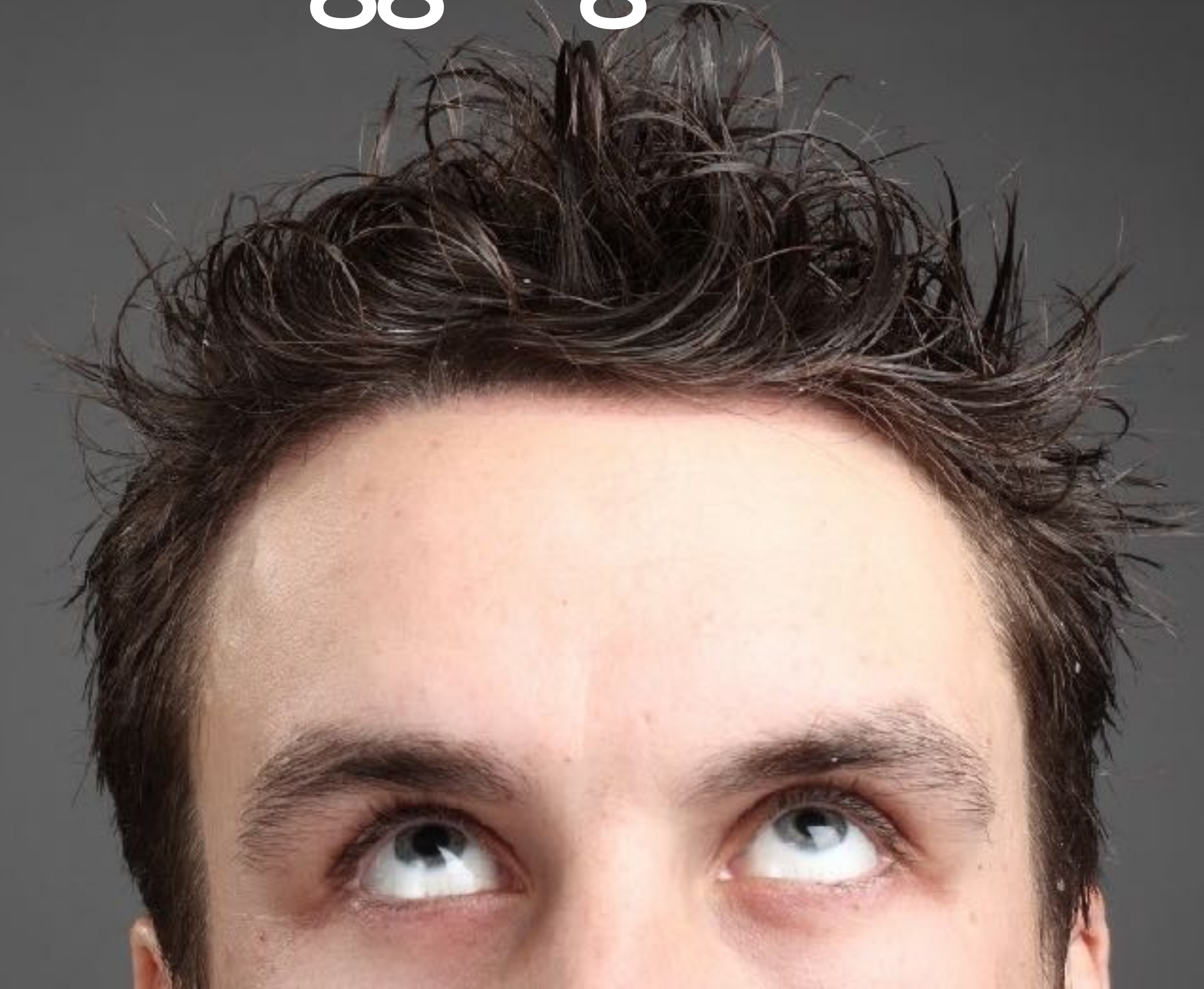
aggregating up one dimension



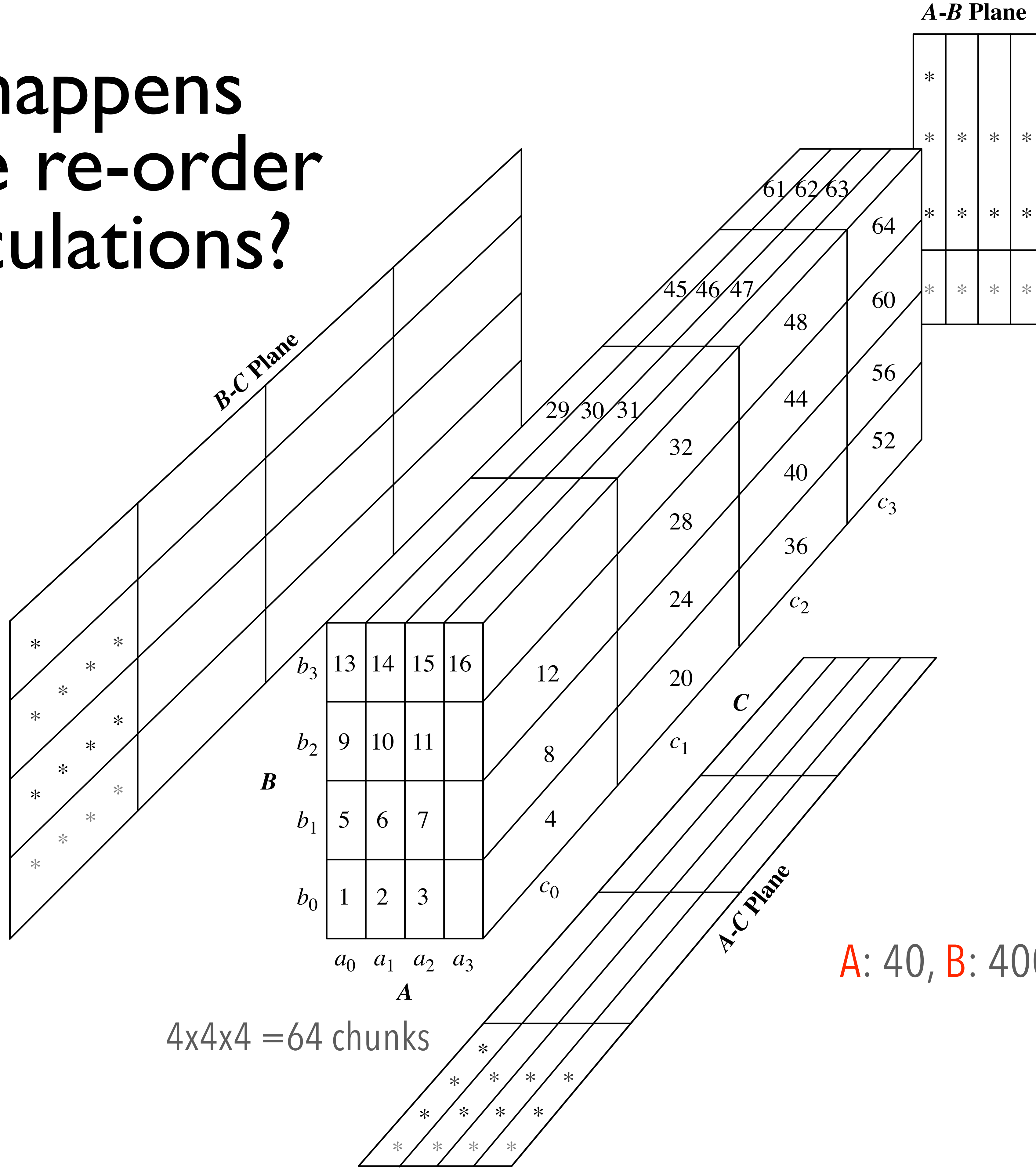
A: 40, B: 400, C:4000

4x4x4 =64 chunks

Can we avoid revisiting cells to compute different aggregates?

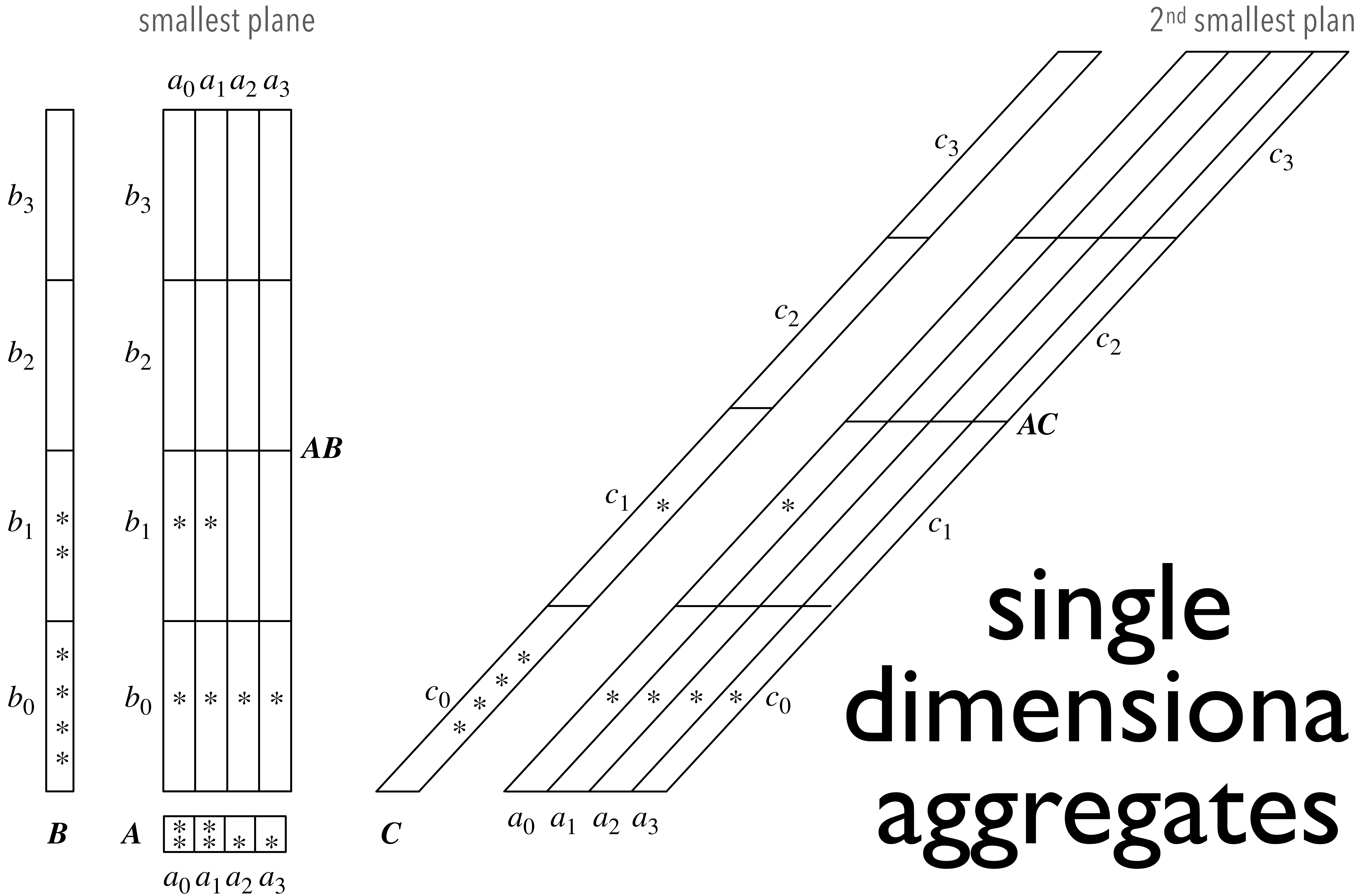


what happens when we re-order our calculations?

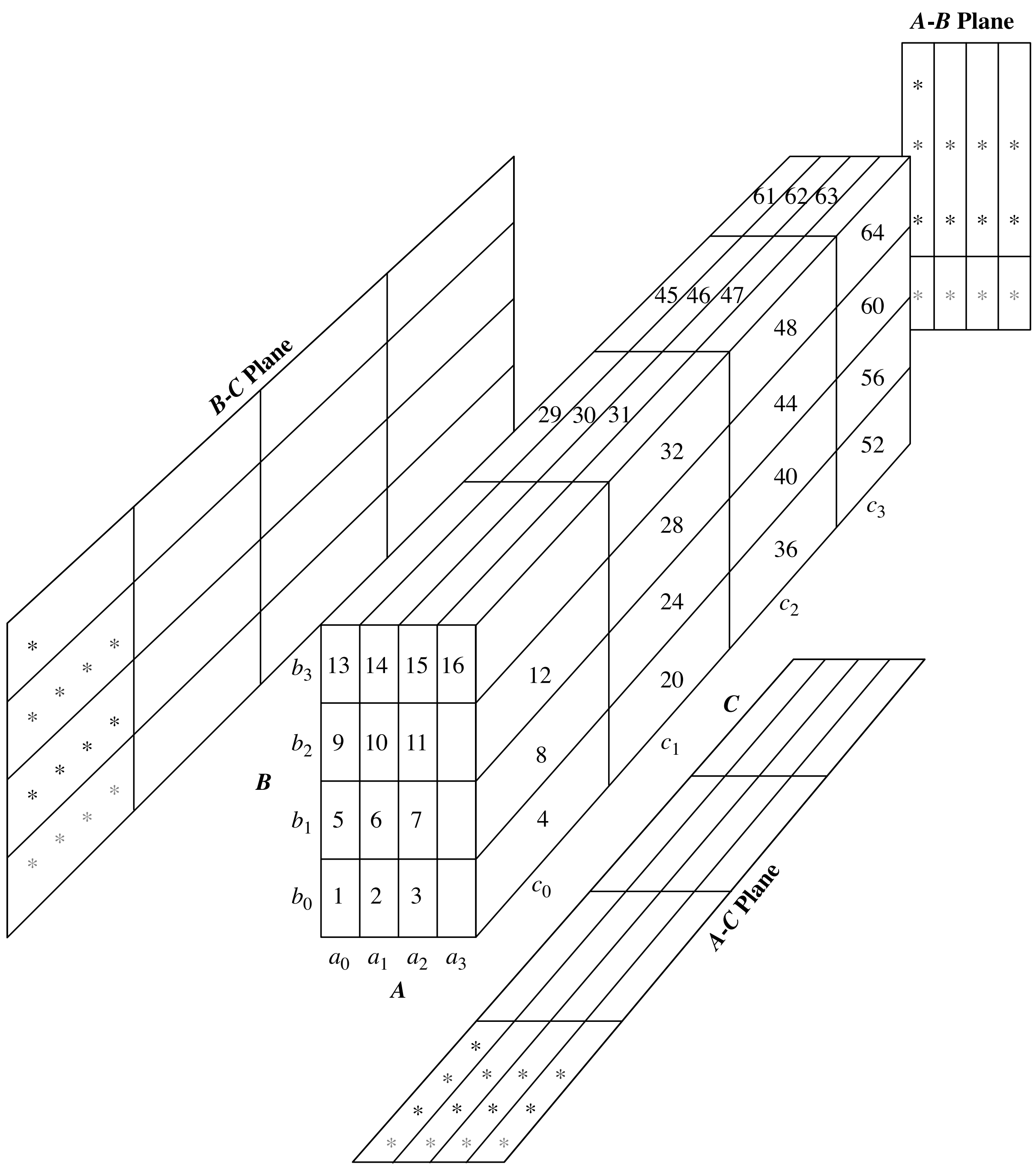


4x4x4 =64 chunks

A: 40, B: 400, C:4000



single dimensional aggregates



MOLAP SUMMARY

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

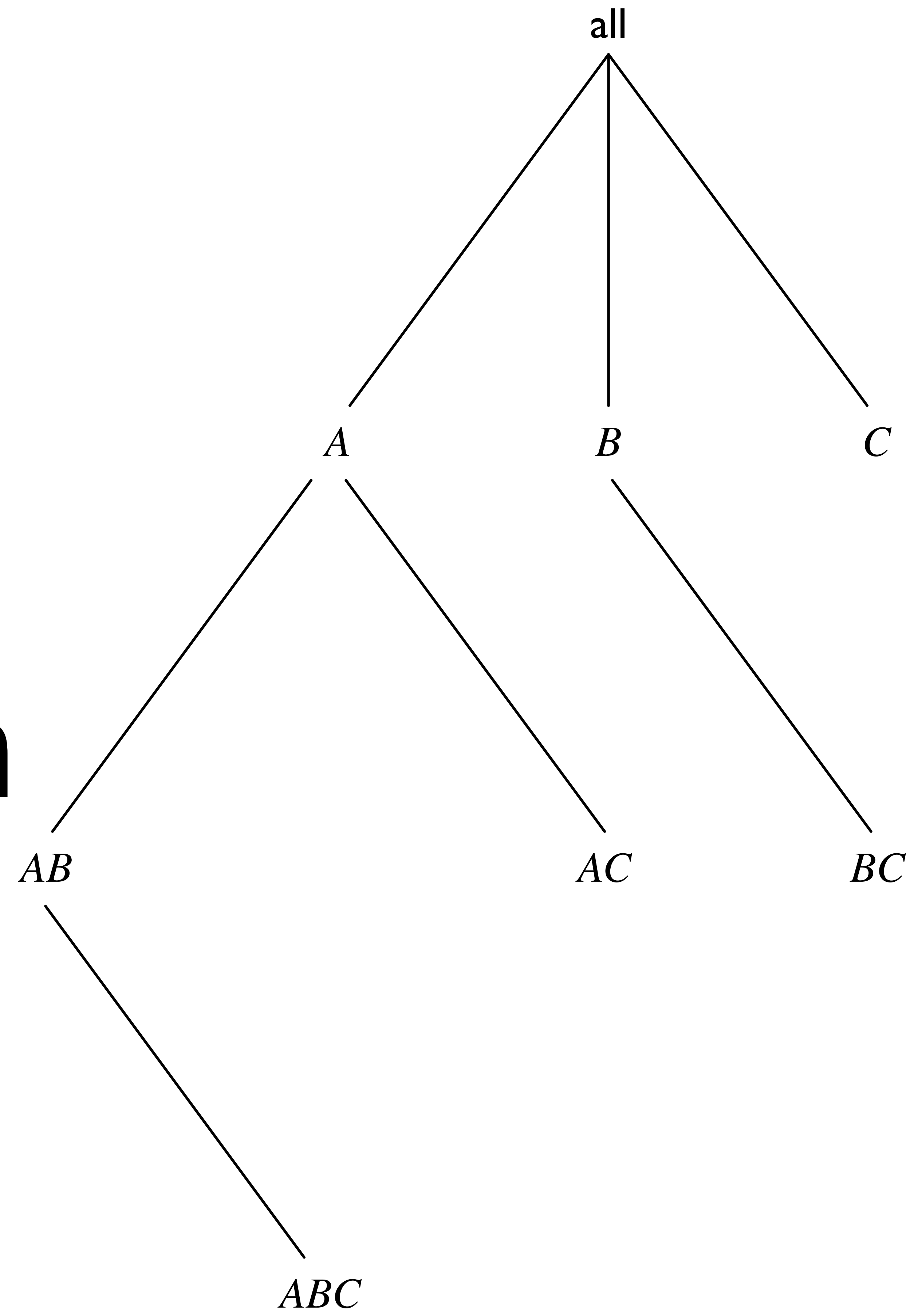
data is not sparse

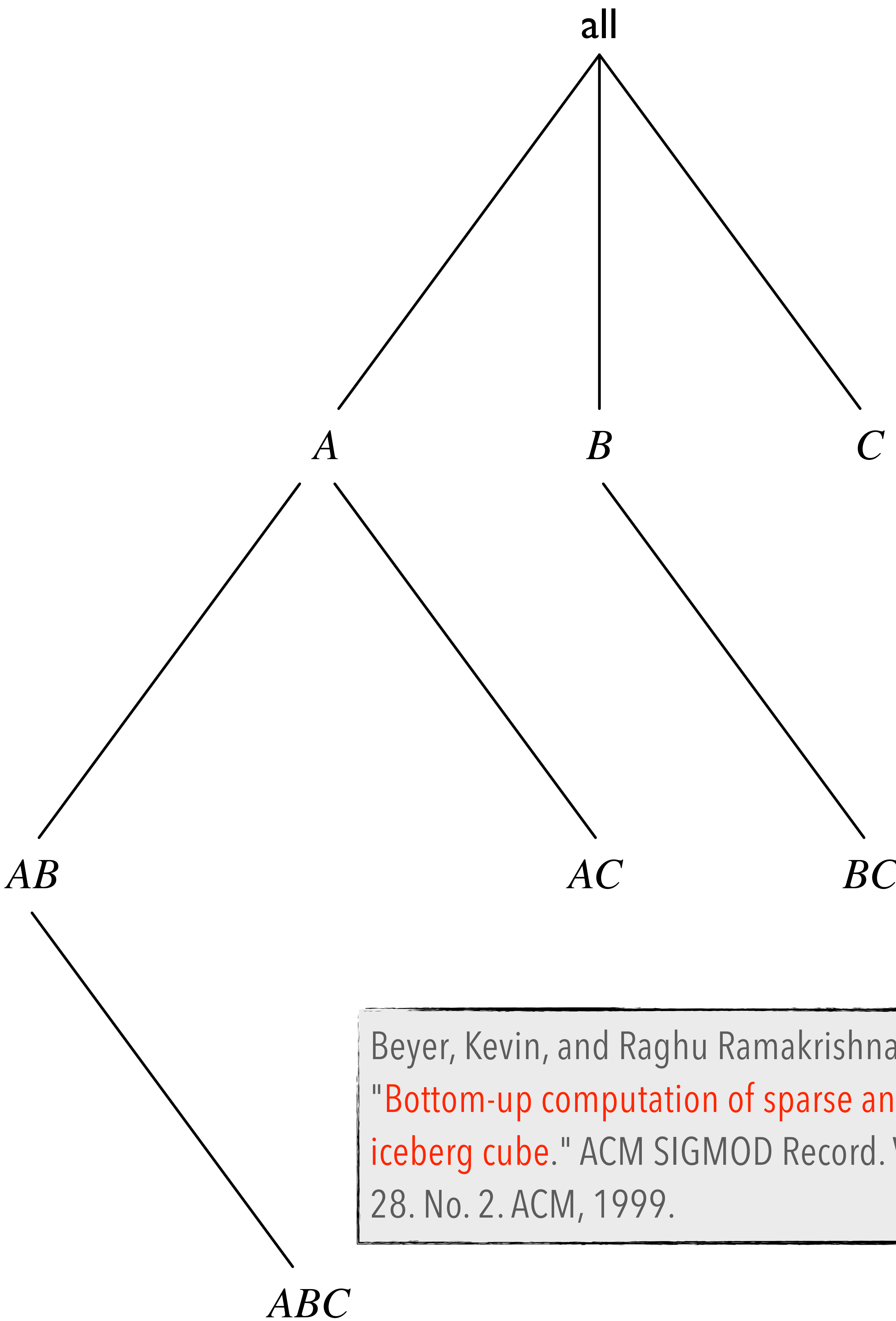
moderate dimensionality

failure cases

apriori

Bottom Up Construction





BOTTOM UP CONSTRUCTION

Bottom-up cube computation

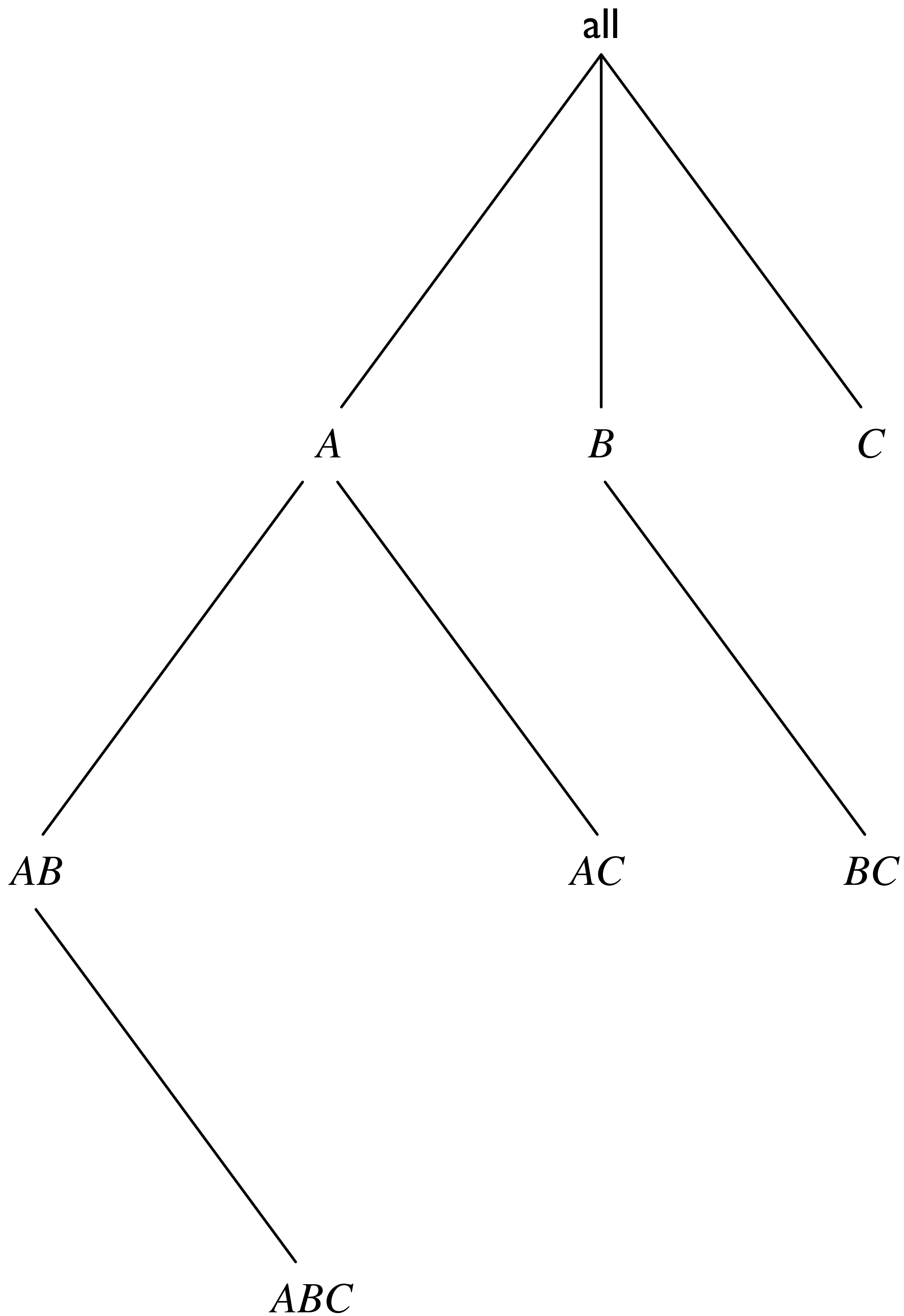
(Note: top-down in our view!)

Divides dimensions into partitions and facilitates iceberg pruning

- If a partition does not satisfy **min_sup**, its descendants can be pruned
- If **min_sup** = 1 ⇒ compute full CUBE!

No simultaneous aggregation

Beyer, Kevin, and Raghu Ramakrishnan. "Bottom-up computation of sparse and iceberg cube." ACM SIGMOD Record. Vol. 28. No. 2. ACM, 1999.



BUC: PARTITIONING

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

COUNT SORT

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Count:	*0*	*2*	*2*	*0*	*1*	*1*	*0*	*1*	*0*	*0*

2) Modify the count array such that each element at each index stores the sum of previous counts.

Index:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Count:	*0*	*2*	*4*	*4*	*5*	*6*	*6*	*7*	*7*	*7*

The modified count array indicates the position of each object in the output sequence.

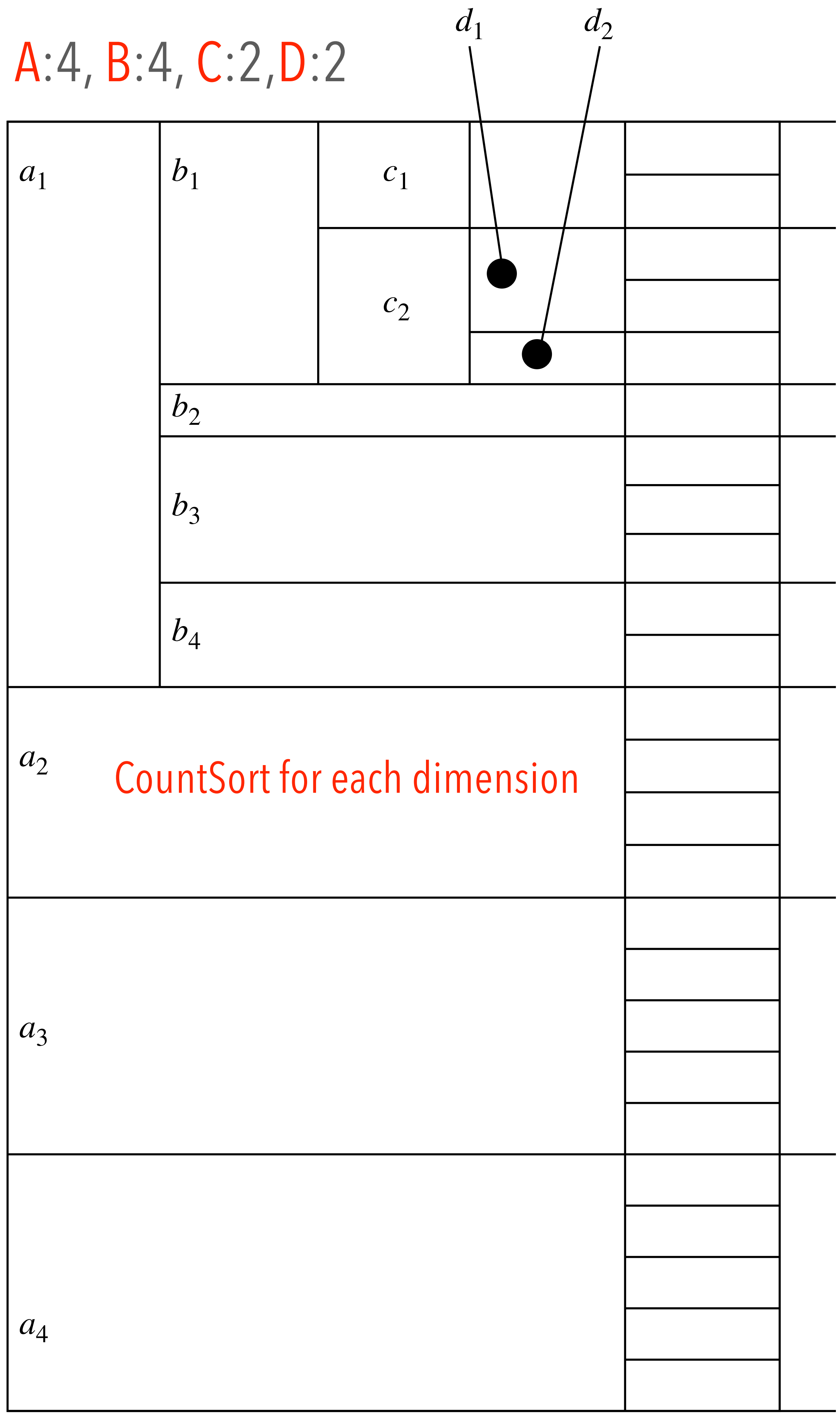
3) Output each object from the input sequence followed by

Decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place the next data value of 1 at an index one smaller than this index.

http://www.geeksforgeeks.org/counting-sort/



BUC: OVERVIEW

Start from partition = Apex (*, *, *, *)

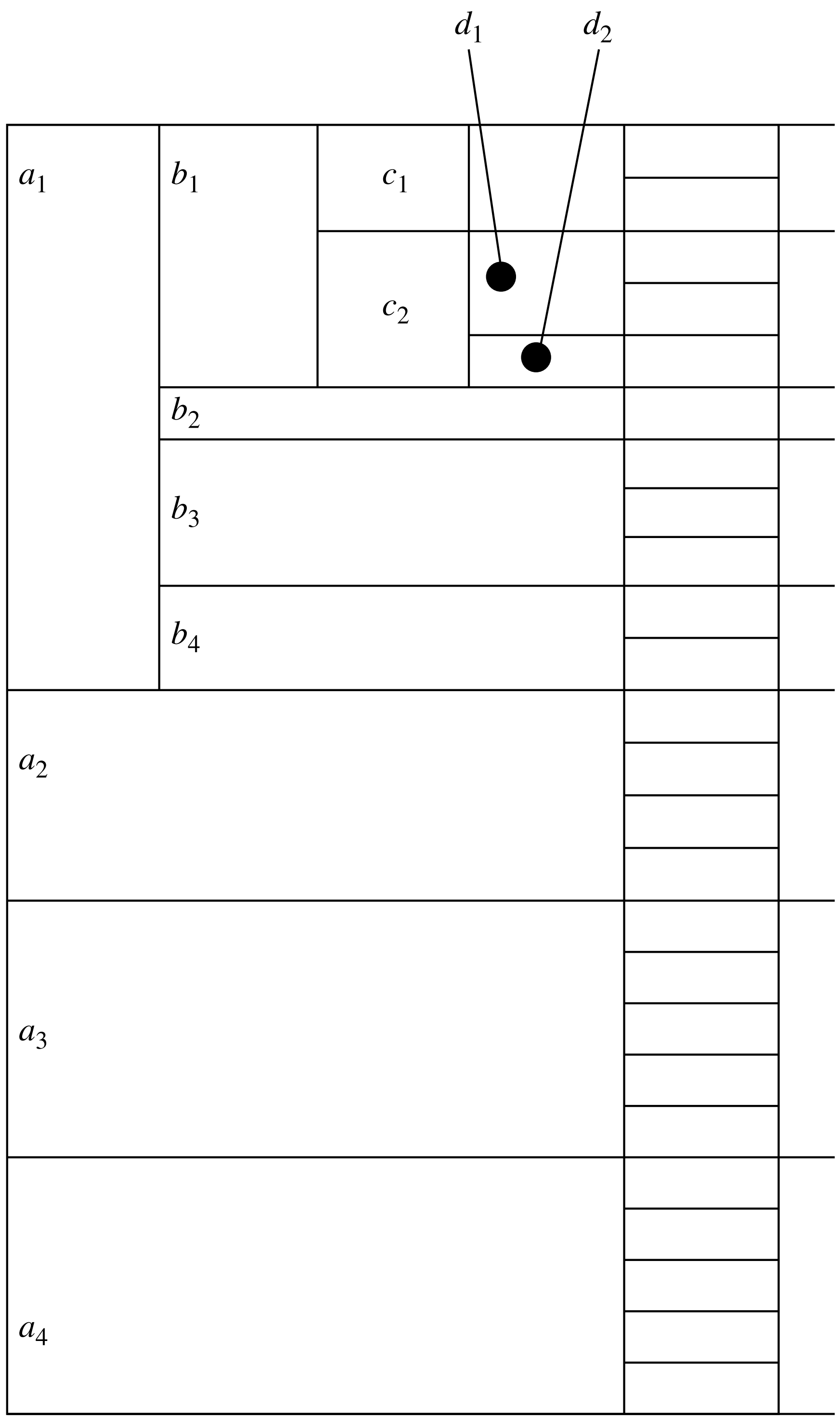
Have an order of dimensions

Count current partition

Sort distinct values in next dimension

Partition into distinct values

Recurse into each partition



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:

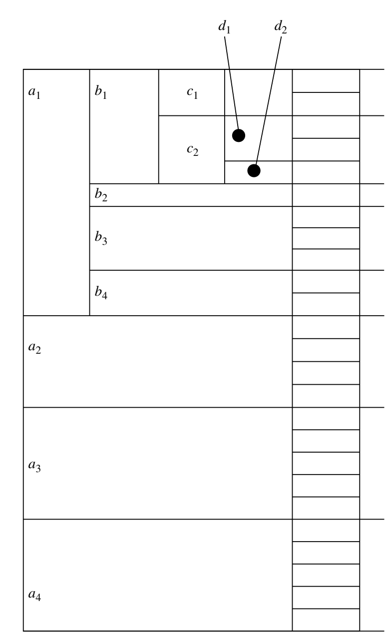
- constant *numDims*: the total number of dimensions;
- constant *cardinality[numDims]*: the cardinality of each dimension;
- constant *min_sup*: the minimum number of tuples in a partition for it to be output;
- *outputRec*: the current output record;
- *dataCount[numDims]*: stores the size of each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

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

Method:

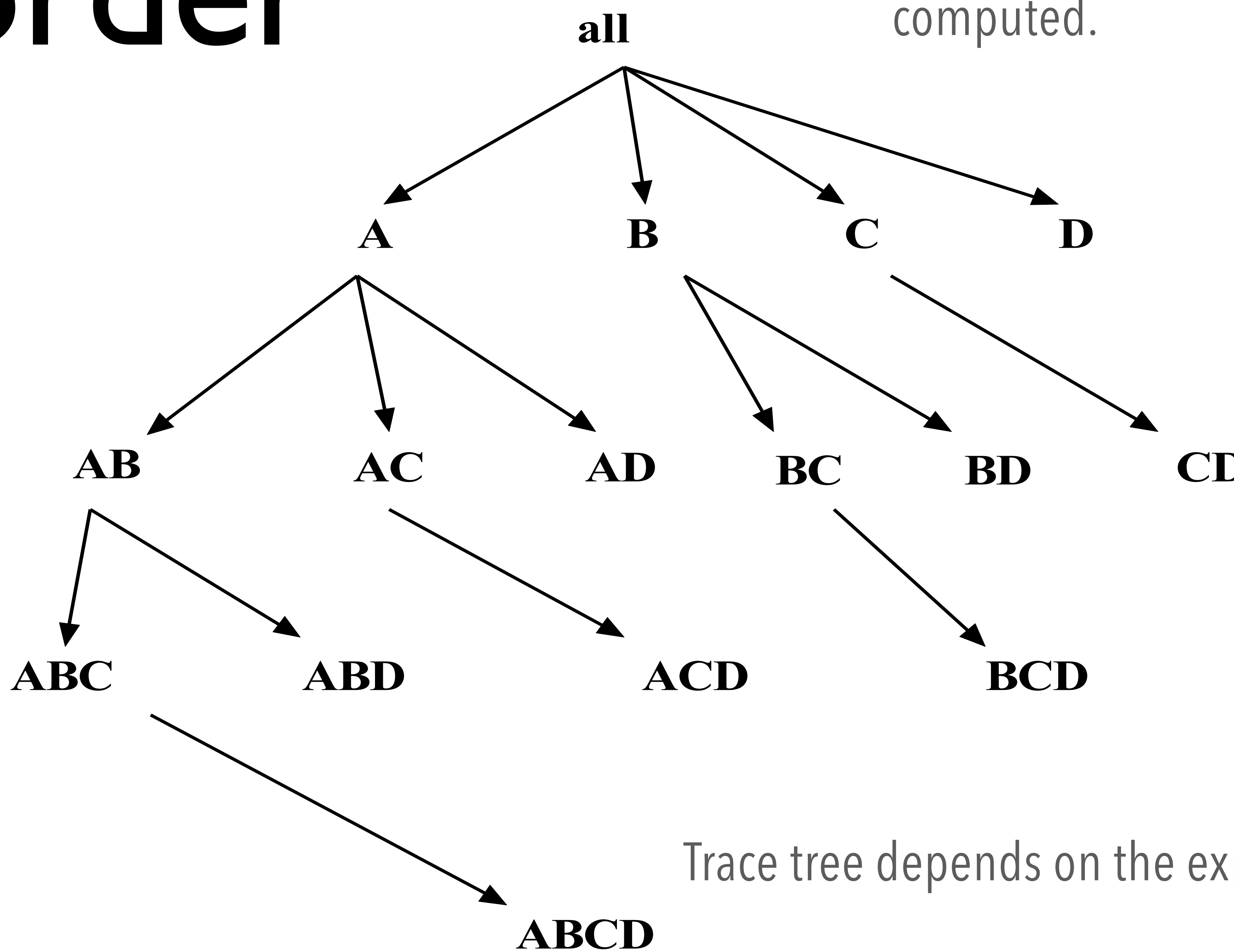
```
(1)  Aggregate(input); // Scan input to compute measure, e.g., count. Place result in outputRec.
(2)  if input.count() == 1 then // Optimization
         WriteDescendants(input[0], dim); return;
     endif
(3)  write outputRec;
(4)  for (d = dim; d < numDims; d++) do //Partition each dimension
(5)      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
```

```
(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
                                   the total number of dimensions
(3)  write outputRec;
(4)  for (d = dim; d < numDims; d++) do //Partition each dimension
(5)      C = cardinality[d];        the number of attribute values in 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];        the number of times a specific attribute value occurs
(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
                            recursively call BUC
(14)         k +=c;
(15)     endfor
(16)     outputRec.dim[d] = all;
(17) endfor
```



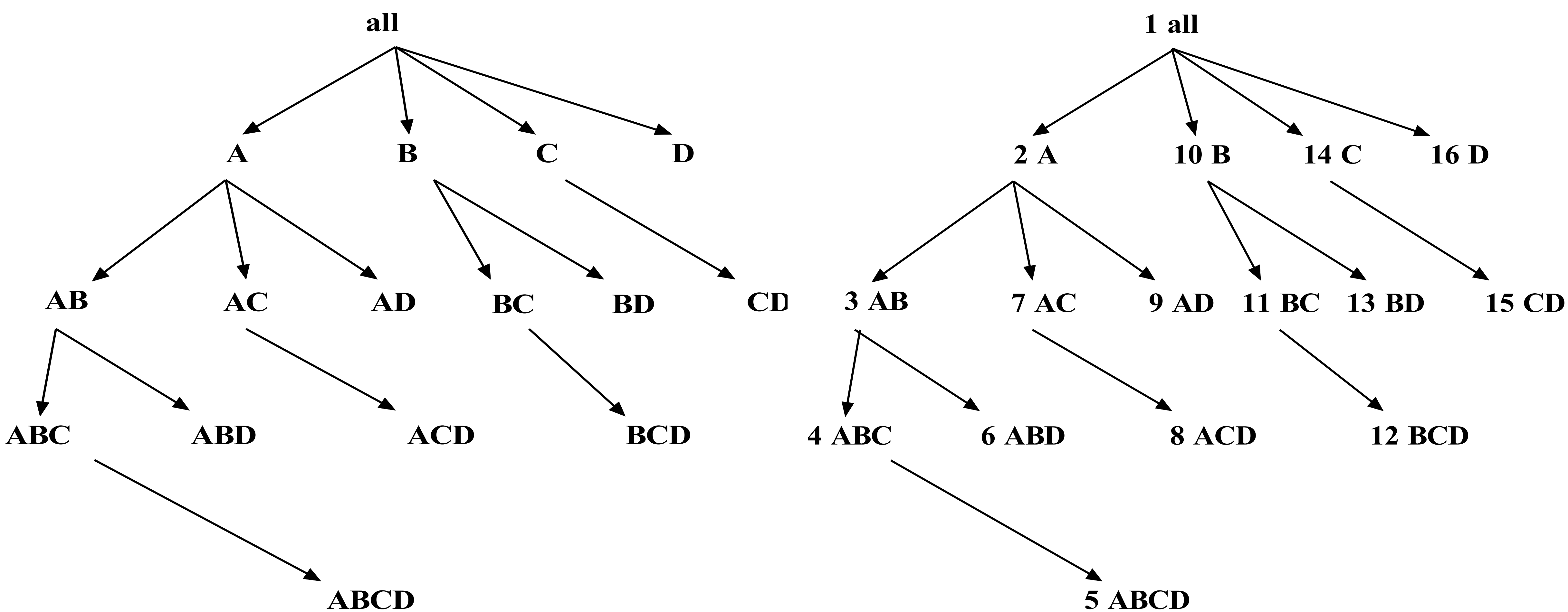
exploration order

Different orders may result in different numbers of cells to be computed.

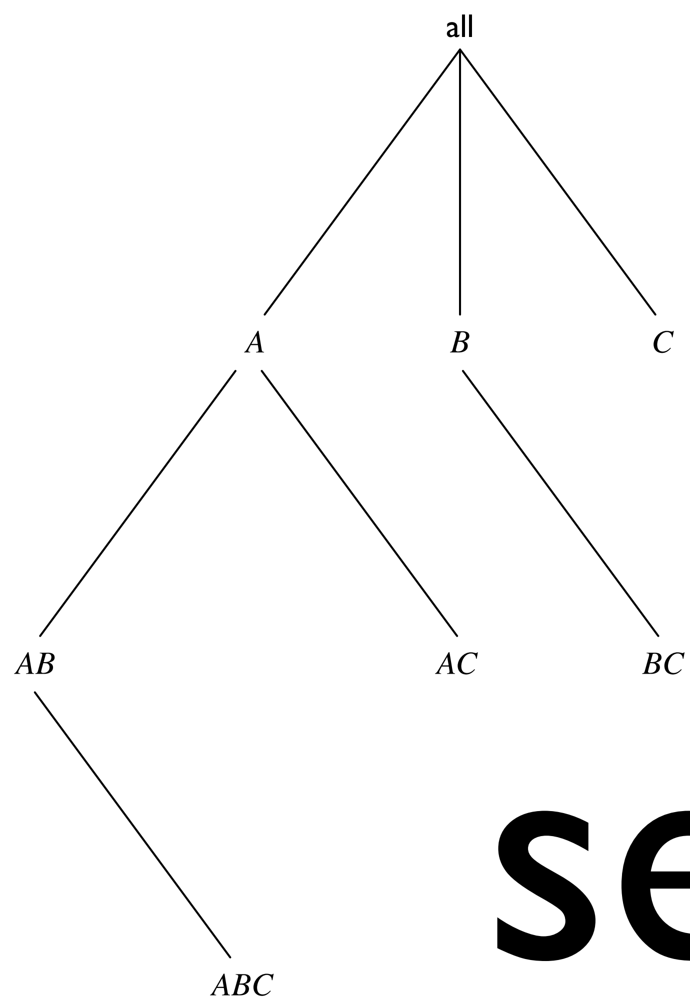


Trace tree depends on the exploration order.

A→B→C→D



all
A
B
C
D
AB
AC
AD
BC
BD
CD
ABC
ABD
ACD
BCD
ABCD
1 all
2 A
10 B
14 C
16 D
3 AB
7 AC
9 AD
11 BC
13 BD
15 CD
4 ABC
6 ABD
8 ACD
12 BCD
5 ABCD



skew

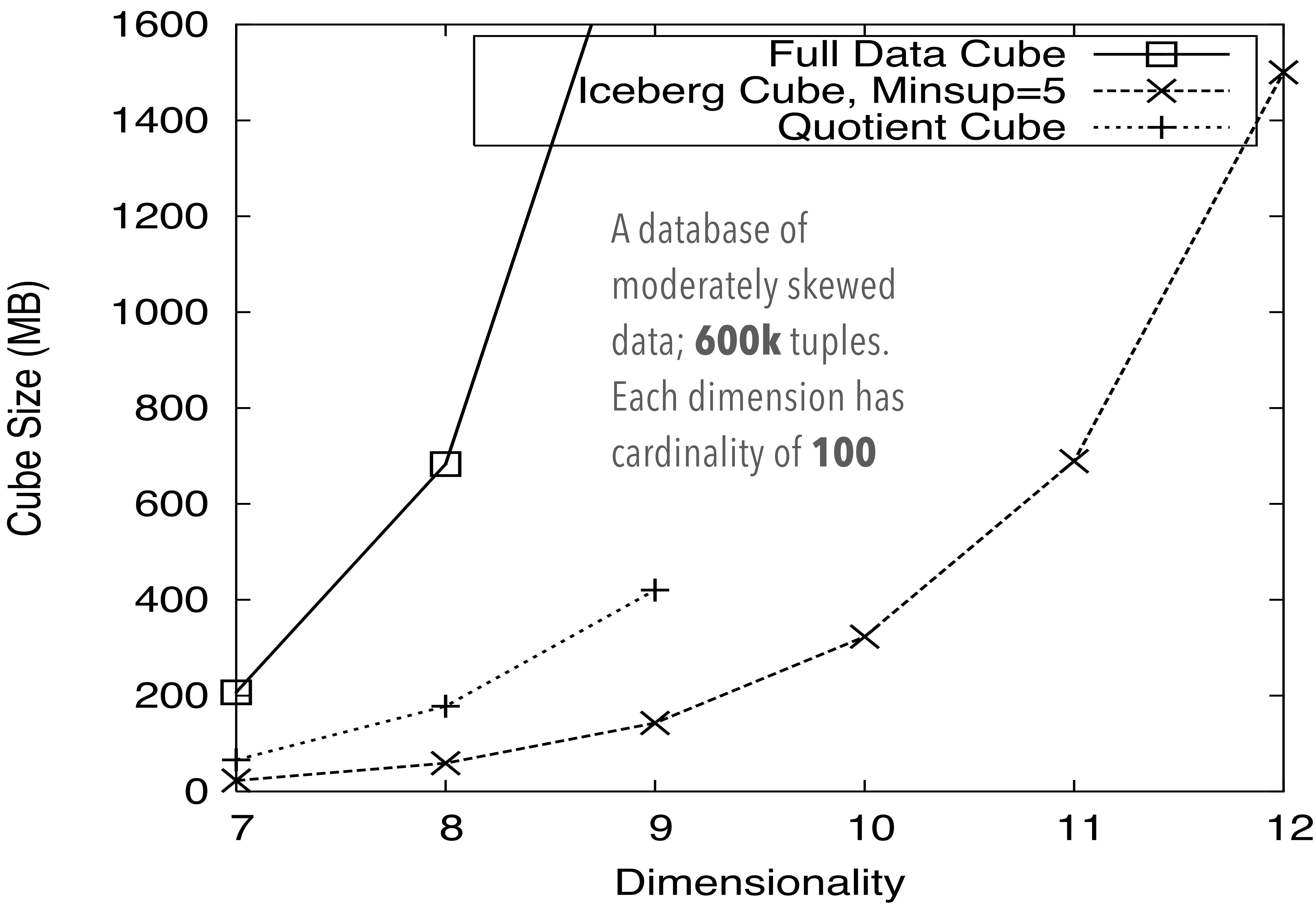
cardinality

order

sensitivities

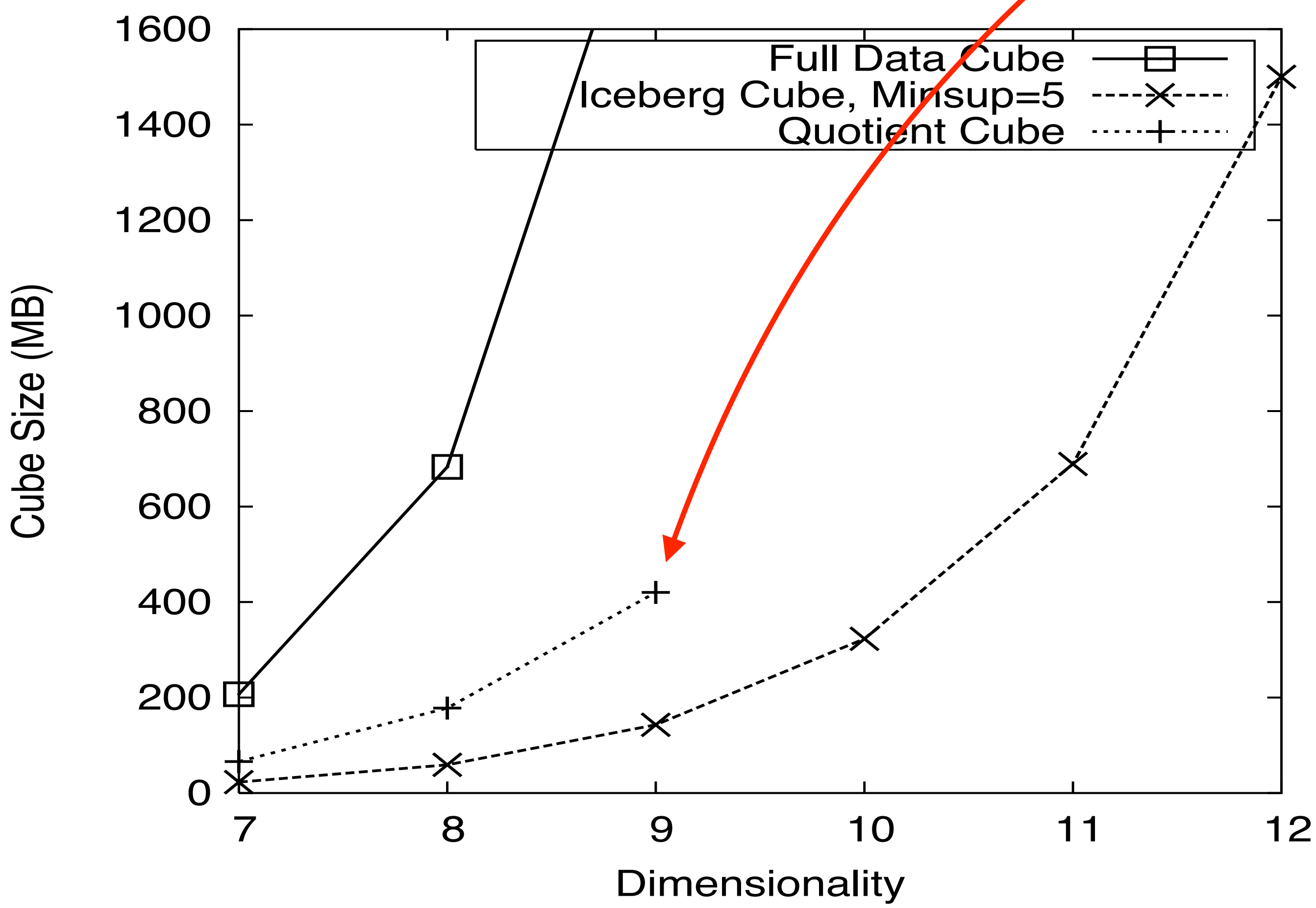
no aggregate sharing

what about high dimensions?



Li, Xiaolei, Jiawei Han, and Hector Gonzalez. "High-dimensional OLAP: a minimal cubing approach." Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. VLDB Endowment, 2004.

ICEBERG CHALLENGES

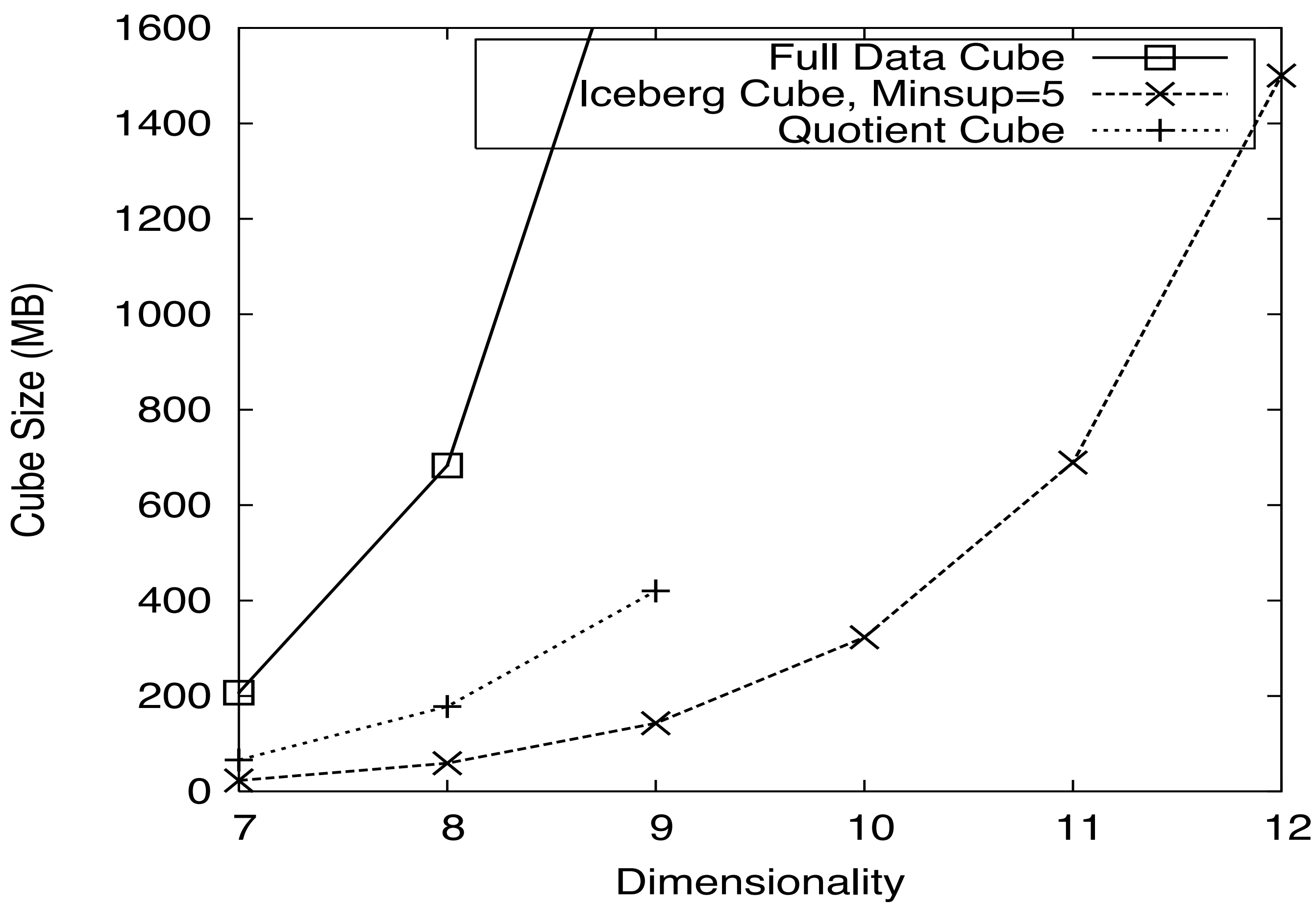


First, if a high-dimensional cell has the support already passing the iceberg threshold, it cannot be pruned by the iceberg condition and will still generate a huge number of cells. For example, a base- cuboid cell: "(a1, a2, . . . , a60): 5" (i.e., with count 5) will still generate 2^{60} iceberg cube cells.

Second, it is difficult to set up an appropriate iceberg threshold. A too low threshold will still generate a huge cube, but a too high one may invalidate many useful applications.

Third, an iceberg cube cannot be incrementally updated. Once an aggregate cell falls below the iceberg threshold and is pruned, incremental update will not be able to recover the original measure.

CHALLENGES



The "curse of dimensionality" problem

Iceberg cube and compressed cubes: only delay the inevitable explosion

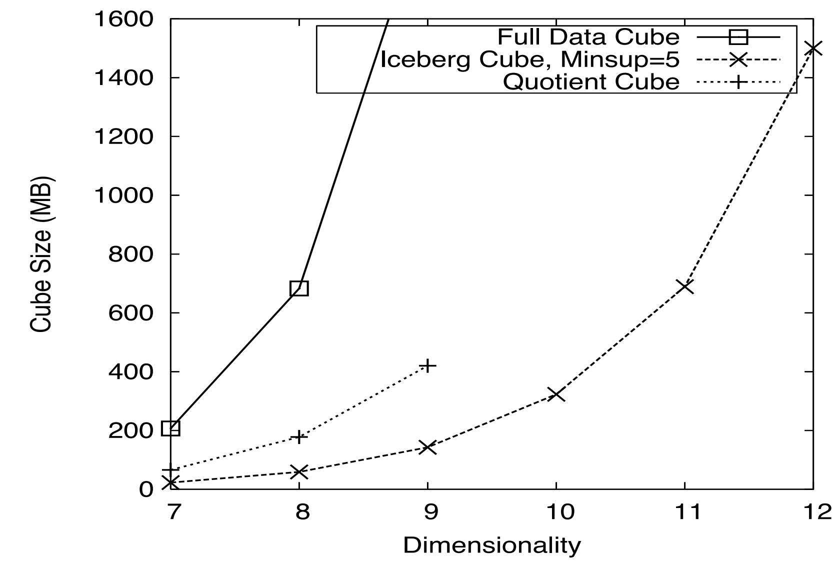
Full materialization: still significant overhead in accessing results on disk

High-D OLAP is needed in applications

- Science and engineering analysis
- Bio-data analysis: thousands of genes
- Statistical surveys: hundreds of variables

Observation: OLAP occurs only on a small subset of dimensions at a time

Semi-Online Computational Model



High Dimensional OLAP

Partition the set of dimensions into shell fragments

Compute data cubes for each shell fragment while retaining inverted indices or value-list indices

Given the pre-computed fragment cubes, dynamically compute cube cells of the high-dimensional data cube online

Partitions the data vertically

Reduces high-dimensional cube into a set of lower dimensional cubes

properties

Online re-construction of original high-dimensional space

Lossless reduction

Offers tradeoffs between the amount of pre-processing and the speed of online computation

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

tid	**A**	**B**	**C**	**D**	**E**
1	a1	b1	c1	d1	e1
2	a1	b2	c1	d2	e1
3	a1	b2	c1	d1	e2
4	a2	b1	c1	d1	e2
5	a2	b1	c1	d1	e3

measure: count

The inverted index table uses the same amount of storage space as the original database.

tid	**A**	**B**	**C**	**D**	**E**
1	a1	b1	c1	d1	e1
2	a1	b2	c1	d2	e1
3	a1	b2	c1	d1	e2
4	a2	b1	c1	d1	e2
5	a2	b1	c1	d1	e3

Attribute Value	TID List	List Size
a1	1 2 3	3
a2	4 5	2
b1	1 4 5	3
b2	2 3	2
c1	1 2 3 4 5	5
d1	1 3 4 5	4
d2	2	1
e1	1 2	2
e2	3 4	2
e3	5	1

Build traditional inverted index or RID list

SHELL FRAGMENTS

Attribute Value	TID List	List Size
a1	1 2 3	3
a2	4 5	2
b1	1 4 5	3
b2	2 3	2
c1	1 2 3 4 5	5
d1	1 3 4 5	4
d2	2	1
e1	1 2	2
e2	3 4	2
e3	5	1

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

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

Compute all cuboids for data cubes ABC and DE while retaining the inverted indices

For example, shell fragment cube ABC contains 7 cuboids:

- A, B, C
- AB, AC, BC
- ABC

This completes the offline computation stage

SHELL SIZE AND DESIGN

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

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

each tuple id will be stored in these many cuboids

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

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

tid	count	sum
1	5	70
2	3	10
3	8	20
4	5	40
5	2	30

FRAG SHELLS ALGORITHM

Partition set of dimension (A_1,…, A_n) into a set of k fragments (P_1, …,P_k).

Scan base table once and do the following:

- insert <tid, measure> into **ID_measure** table.
- for each attribute value a_i of each dimension A_i
  - build inverted index entry <a_i, tidlist>

For each fragment partition P_i

- build local fragment cube S_i by intersecting tid-lists in bottom-up fashion.

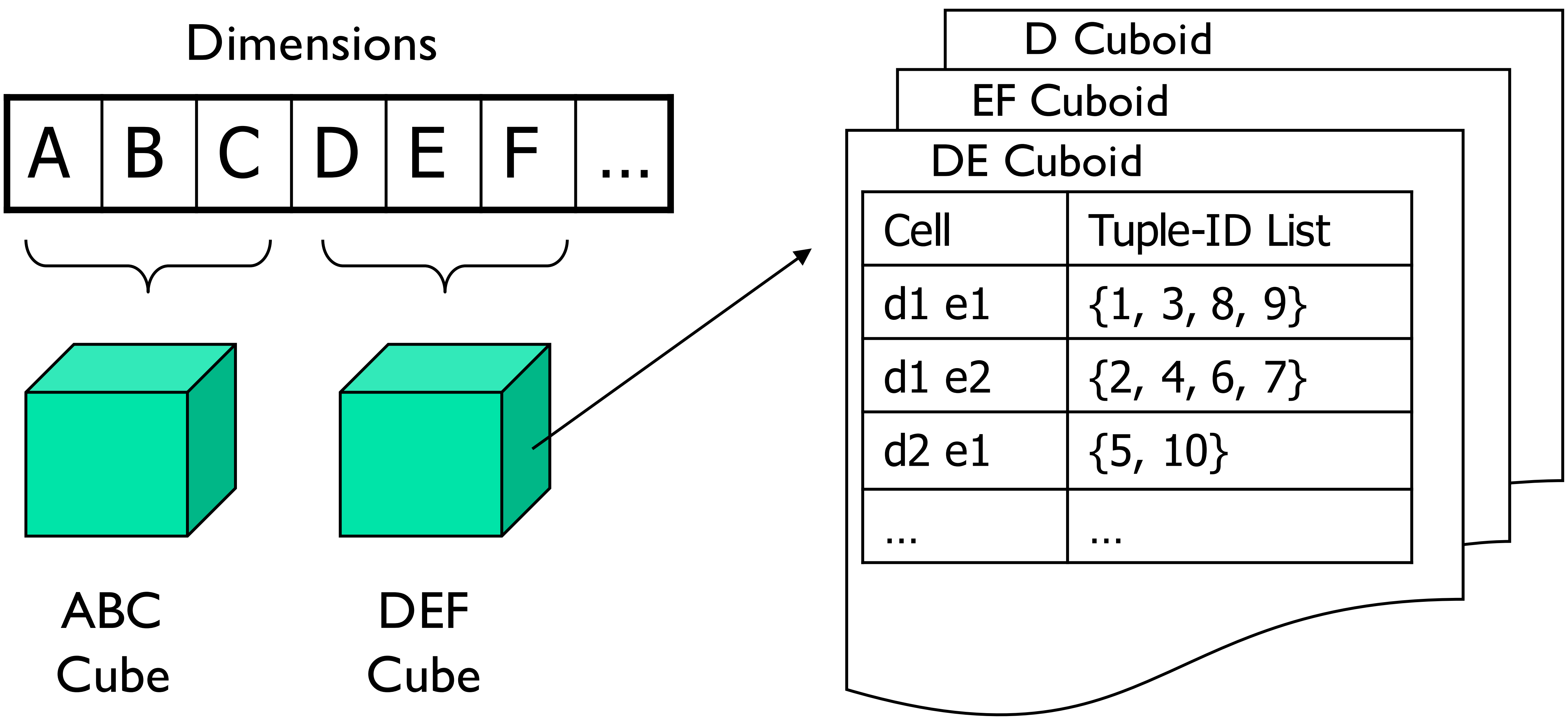
measure array

tid	count	sum
1	5	70
2	3	10
3	8	20
4	5	40
5	2	30

Attribute Value	TID List	List Size
a1	1 2 3	3
a2	4 5	2
b1	1 4 5	3
b2	2 3	2
c1	1 2 3 4 5	5
d1	1 3 4 5	4
d2	2	1
e1	1 2	2
e2	3 4	2
e3	5	1

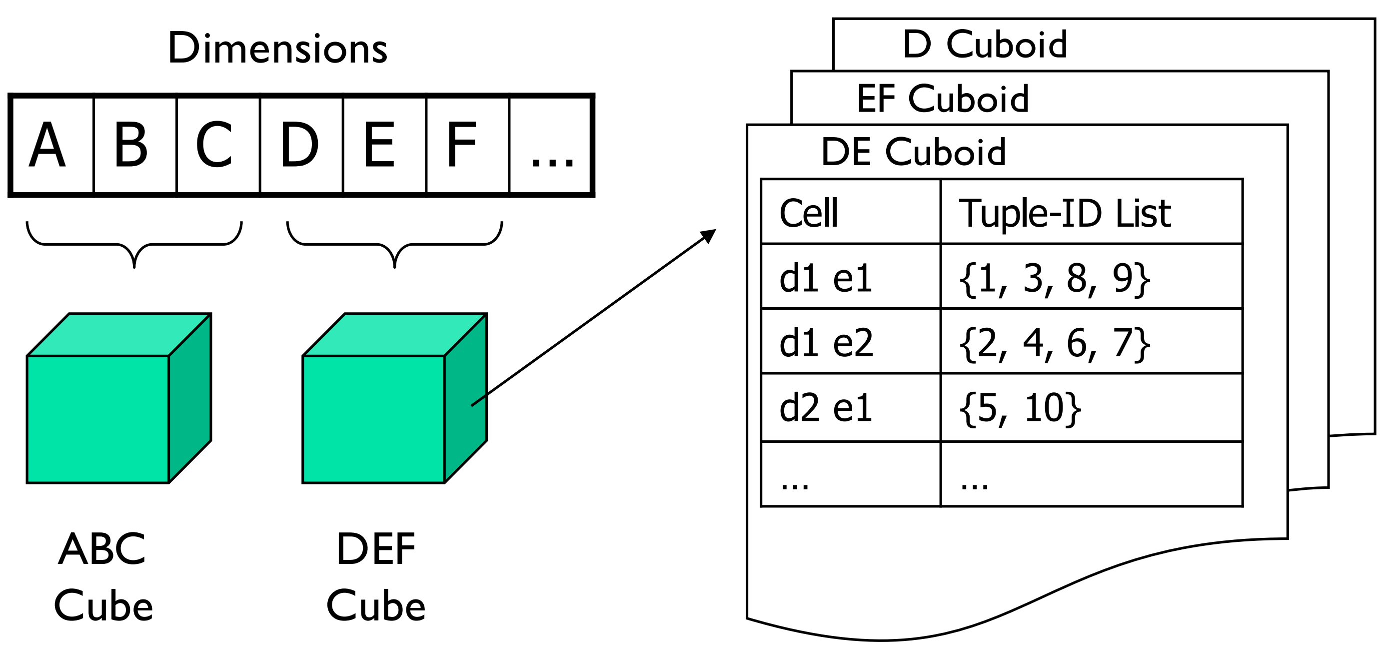
Cell	*Intersection*	*TID List*	*List Size*
(a_1, b_1)	$\{1, 2, 3\} \cap \{1, 4, 5\}$	$\{1\}$	1
(a_1, b_2)	$\{1, 2, 3\} \cap \{2, 3\}$	$\{2, 3\}$	2
(a_2, b_1)	$\{4, 5\} \cap \{1, 4, 5\}$	$\{4, 5\}$	2
(a_2, b_2)	$\{4, 5\} \cap \{2, 3\}$	$\{\}$	0

frag shells



Cell	Tuple-ID List
d1 e1	{1, 3, 8, 9}
d1 e2	{2, 4, 6, 7}
d2 e1	{5, 10}
...	...

QUERY



A query has the general form <a_1, a_2, …, a_n : m>

Each a_i has 3 possible values

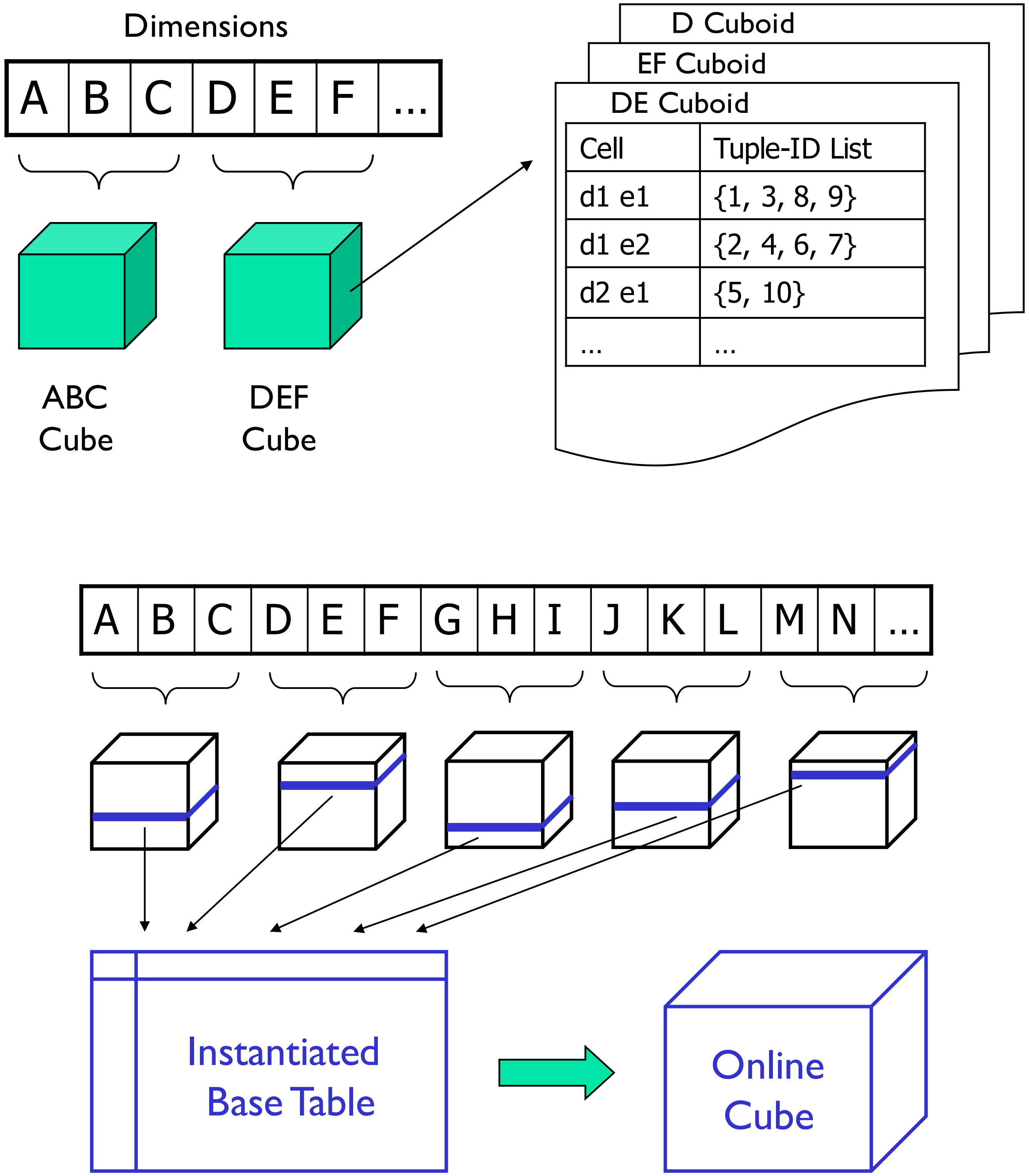
- Instantiated value (specific; e.g. a_1 = Chicago)
- Aggregate * function (i.e. don't care; aggregate over)
- Inquire ? function (results should include all values of this attribute)

For example:

<3, ?, ?, *, *, *, 1 : count>

returns a 2-D data cube.

METHOD



Given the fragment cubes, process a query as follows

- Divide the query into fragment, same as the shell
- Fetch the corresponding TID list for each fragment from the fragment cube
- Intersect the TID lists from each fragment to construct instantiated base table
- Compute the data cube using the base table with any cubing algorithm

Deep Dive!

STORAGE SIZE

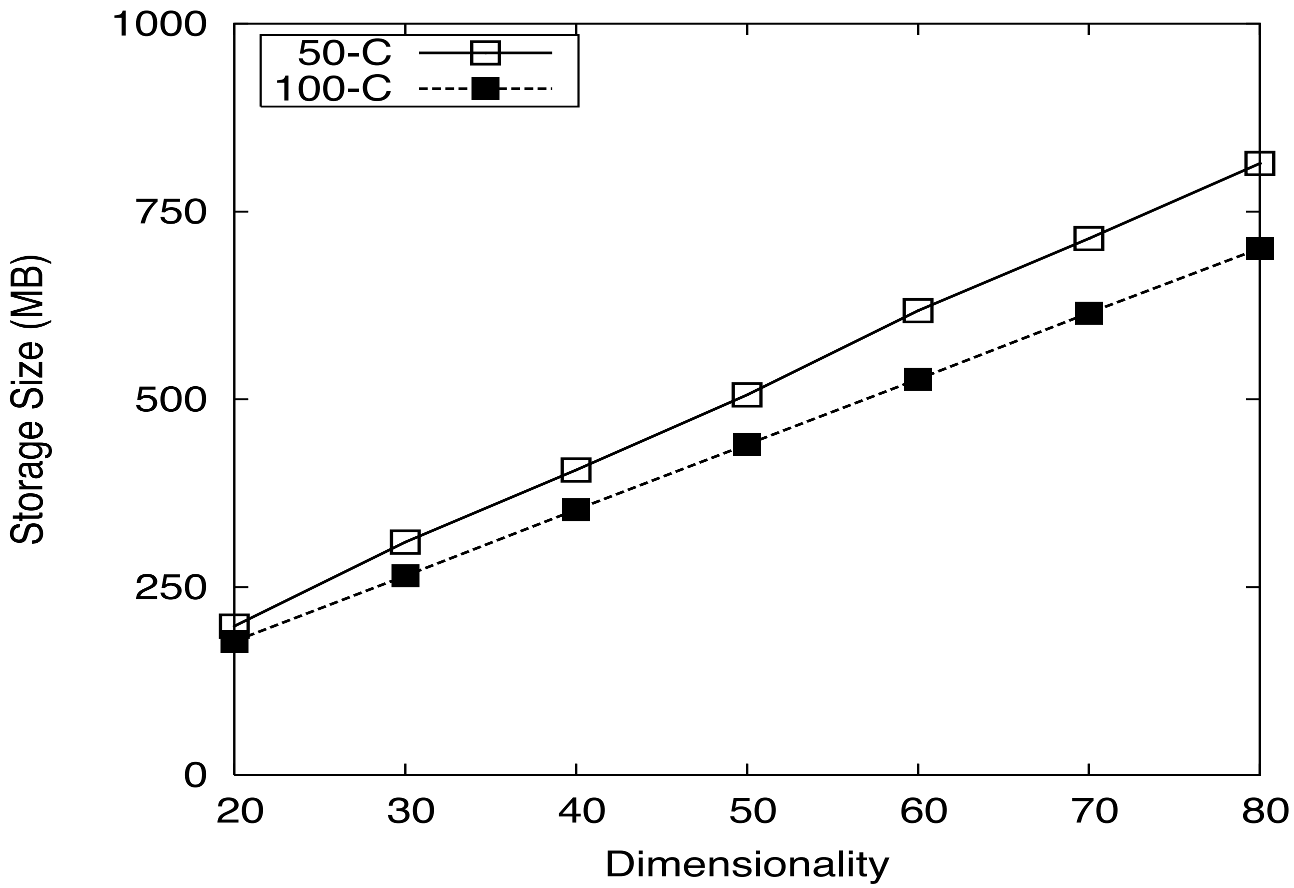


Figure 2: Storage size of shell fragments: (50-C) $\mathcal{T} = 10^6$, $\mathcal{C} = 50$, $\mathcal{S} = 0$, $\mathcal{F} = 3$. (100-C) $\mathcal{T} = 10^6$, $\mathcal{C} = 100$, $\mathcal{S} = 2$, $\mathcal{F} = 2$.

D denotes the number of dimensions, C is the cardinality of each dimension, T is the number of tuples in the database, F is the size of the shell fragment, I is the number of instantiated dimensions, Q is the number of inquired dimensions, and S is the skew or zipf of the data. **Minimum support level is 1.**

Storage grows linearly as the number of dimensions D.

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

Deep Dive!

EXPERIMENTS

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

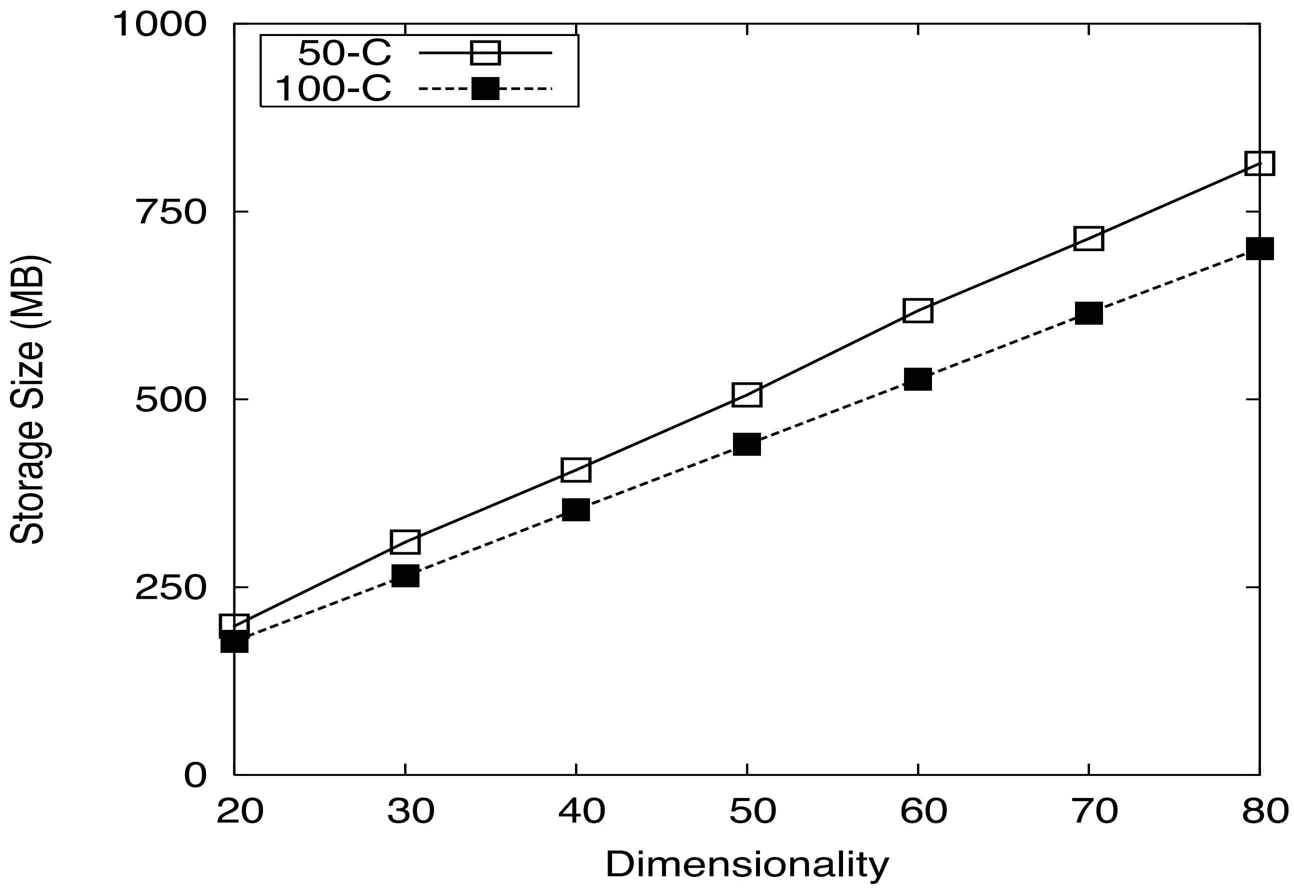


Figure 2: Storage size of shell fragments: (50-C) $\mathcal{T} = 10^6$, $\mathcal{C} = 50$, $\mathcal{S} = 0$, $\mathcal{F} = 3$. (100-C) $\mathcal{T} = 10^6$, $\mathcal{C} = 100$, $\mathcal{S} = 2$, $\mathcal{F} = 2$.

D denotes the number of dimensions, C is the cardinality of each dimension, T is the number of tuples in the database, F is the size of the shell fragment, I is the number of instantiated dimensions, Q is the number of inquired dimensions, and S is the skew or zipf of the data. **Minimum support level is 1.**

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

Deep Dive!

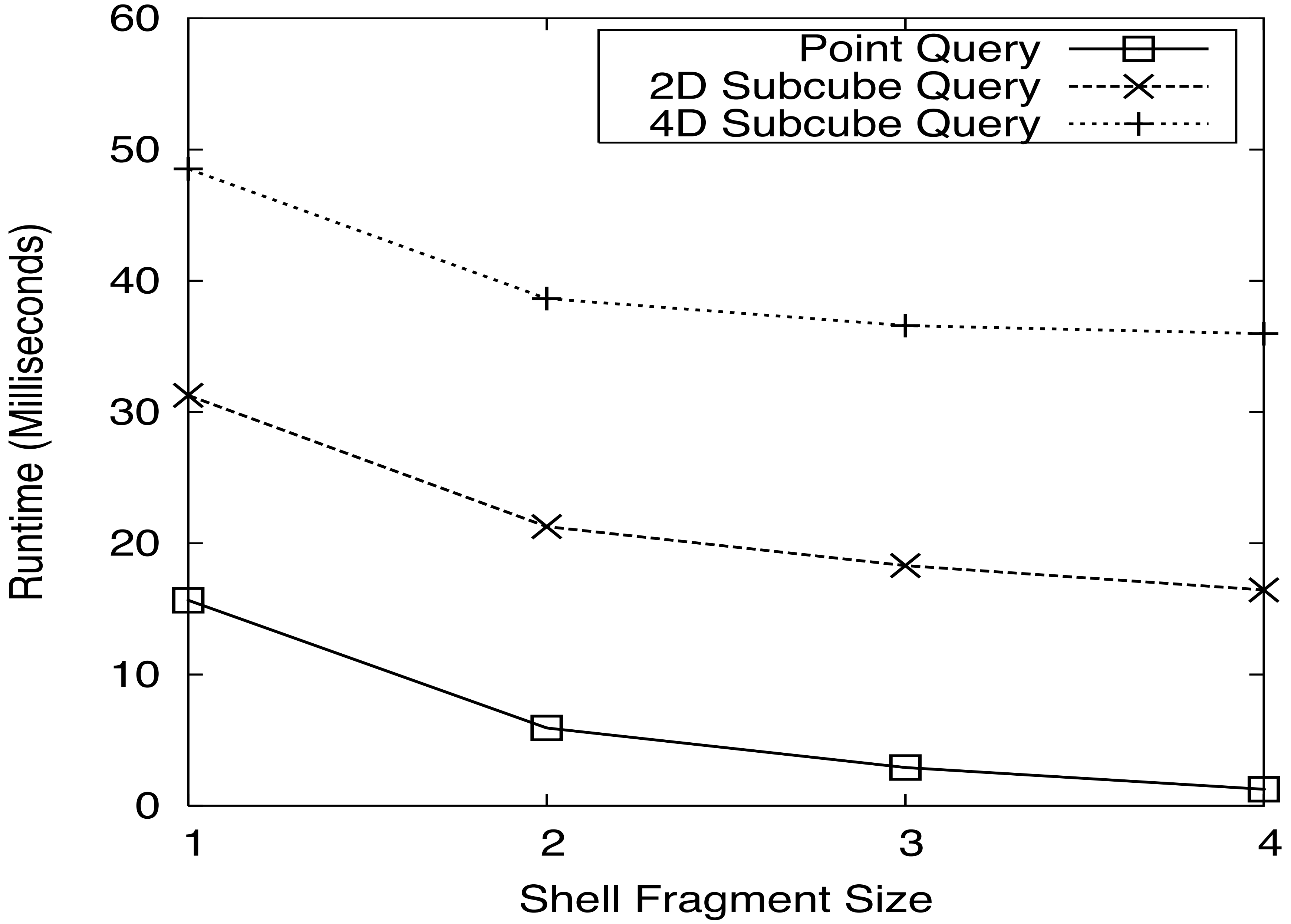


Figure 5: Average computation time per query over 1,000 trials. $\mathcal{T} = 10^6$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, $\mathcal{S} = 0$, $\mathcal{I} = 4$.

Exercise!

what is the value of F, that will lead to smallest fragment cube size?

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

Exercise!

Why don't we use the smallest value in shell cube design?

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