Architecture and Implementation of Database Systems (Winter 2013/14)

Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2013/14

Part VIII

Online Analytical Processing (OLAP)

Motivation

Scenario: A bookstore chain collects sales data:

Sales			
Book	City	Month	Units Sold
Arlington Road Atlas	Arlington	January	134
Arlington Road Atlas	Arlington	February	327
⋮	⋮	⋮	⋮
Arlington Road Atlas	Springfield	December	193
Gone With the Wind	Arlington	January	9
⋮	⋮	⋮	⋮
Tropical Food	Springfield	December	374

Motivation

Goal: Spread sheet-style analyses ($\rightsquigarrow$ "Pivot Table")

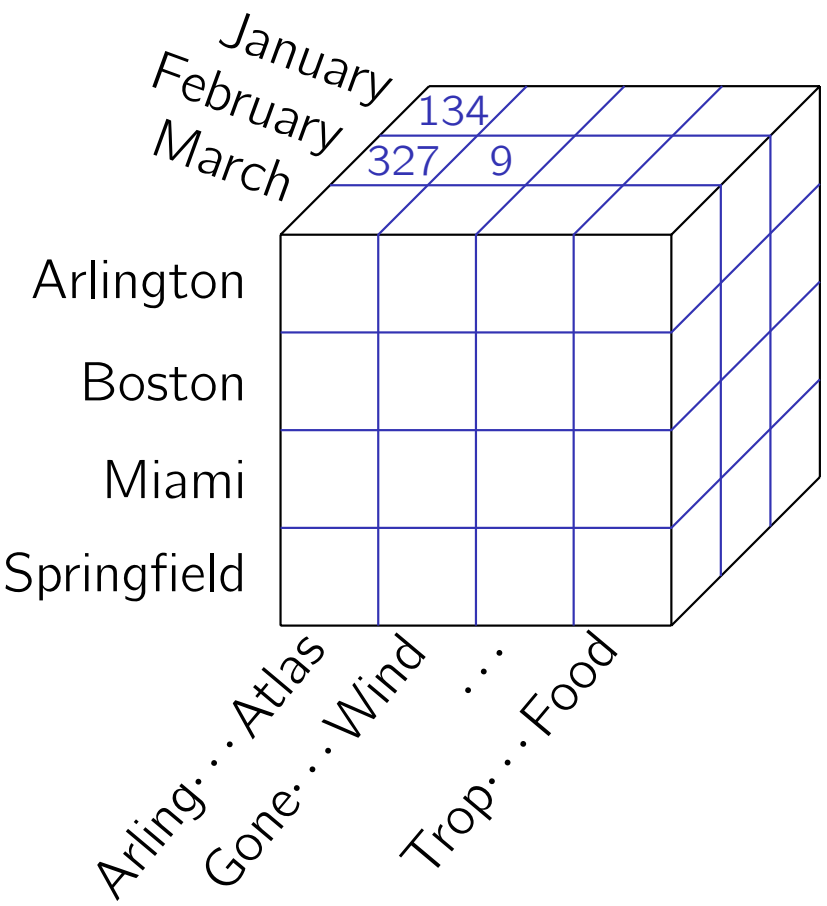
	January	February	···	Grand Total
Arlington	198	449	···	1022
Boston	226	212	···	707
Miami	152	130	···	467
Springfield	304	498	···	1303
Grand Total	880	1289	···	3499

Challenge: Large data volumes

- $\rightarrow$ How do we **model** such data (*e.g.*, in a relational system)?
- $\rightarrow$ How can we **implement** pivot tables efficiently?
- $\rightarrow$ What about *k*-**dimensional data**?

Data Cubes

Idea: Model data as a multi-dimensional **cube**



Data cube:

- **Facts** are stored as **cells** of the cube.
- Facts have **measures** associated with them (here: sales counts).
- Cells may be empty.

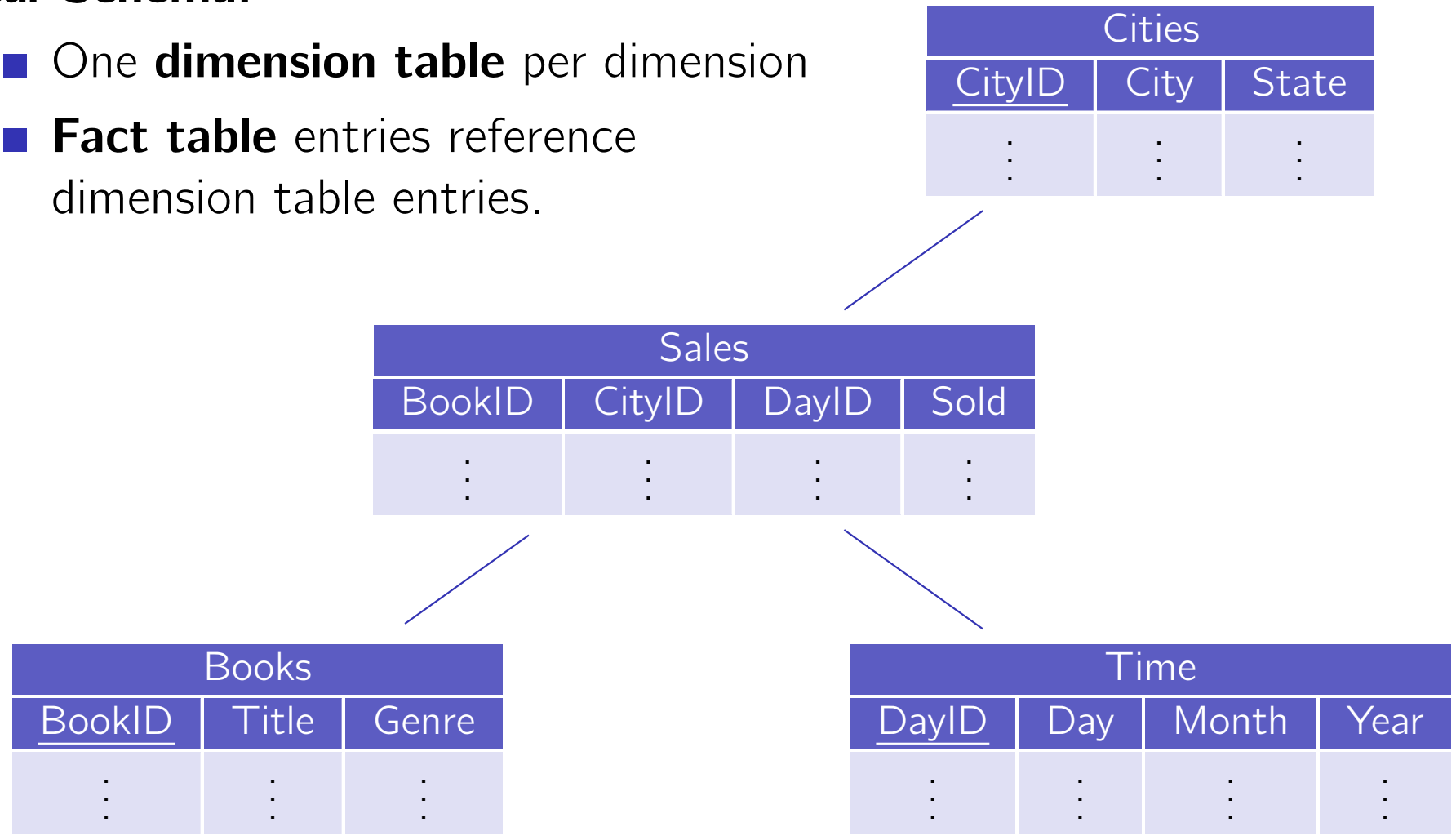
Real-world:

- 4–12 dimensions
- **Project** to 2 or 3 for analysis/viewing

Relational Representation: Star Schema

Star Schema:

- One **dimension table** per dimension
- **Fact table** entries reference dimension table entries.



Star Schema

Fact Table:

- One row per multidimensional fact.
- This table will hold the lion's share of the entire database.

Dimension Tables:

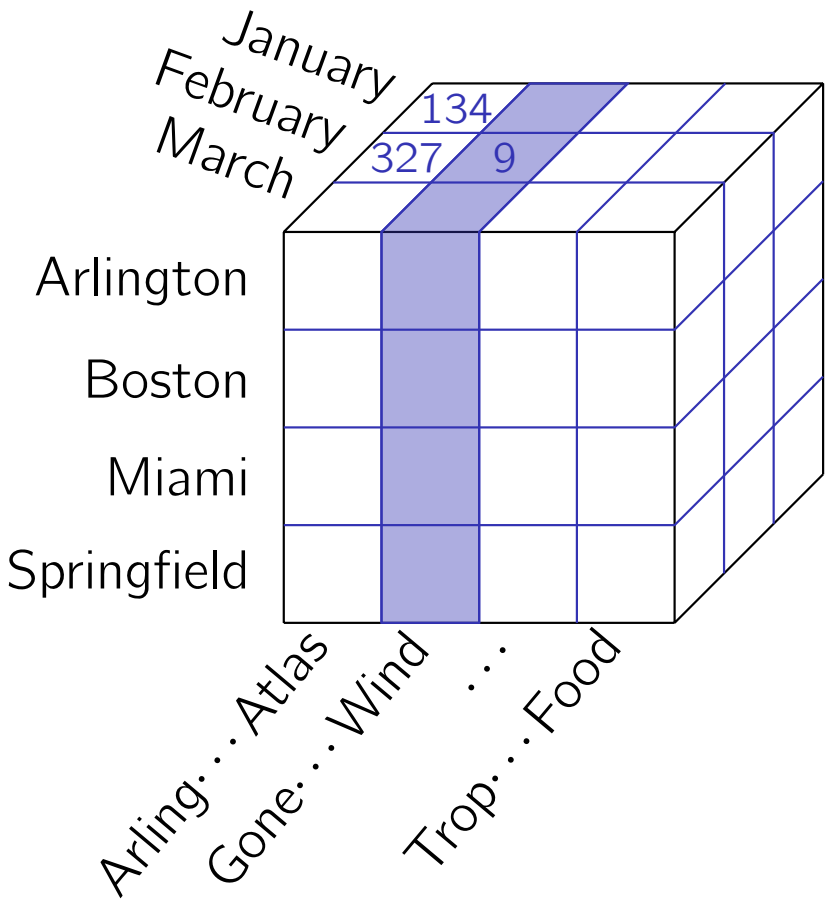
- **Key:** Artificial key (usually an integer number)
- Typically: One column per level if dimension is **hierarchical**
  $\rightarrow$ **Redundancy**

OLAP is ran on data **extracted** from transactional system.

- Load data in **batches**; most of it goes into **fact table**.
- Fact table ends up approximately **ordered by date**.

“Slicing and Dicing”

Typical queries: **aggregate** over **sub-ranges** of the full cube.




```
SELECT SUM(Sold)
  FROM Sales AS s, Books AS b
 WHERE s.BookID = b.BookID
   AND b.Title = "Gone..."
```

Roll-Up, Drill-Down, Pivot Tables

Analysts will want to look at aggregates from many different angles.

Roll-Up / Drill-Down:

→ For hierarchical dimensions, move up or down the hierarchy

→ See more or less details, "zoom" in or out

Pivot Tables:

→ Visualize roll-up/drill-down (⇝ dedicated OLAP tools)

	January	February	···	Grand Total
Arlington	198	449	···	1022
Boston	226	212	···	707
Fiction	*121*	*98*	···	*346*
Cooking	*105*	*214*	···	*361*
Miami	152	130	···	467
Springfield	304	498	···	1303
Grand Total	880	1289	···	3499

SQL OLAP Extensions

A number of **SQL extensions** ease these tasks.

E.g., multi-dimensional grouping ($\rightsquigarrow$ Pivot Table):

```
SELECT c.City, t.Month, SUM(s.Sold)
  FROM Sales AS s, Cities AS c, Time AS t
 WHERE s.DayID = t.DayID AND s.CityID = c.CityID
 GROUP BY CUBE(City, Month)
```

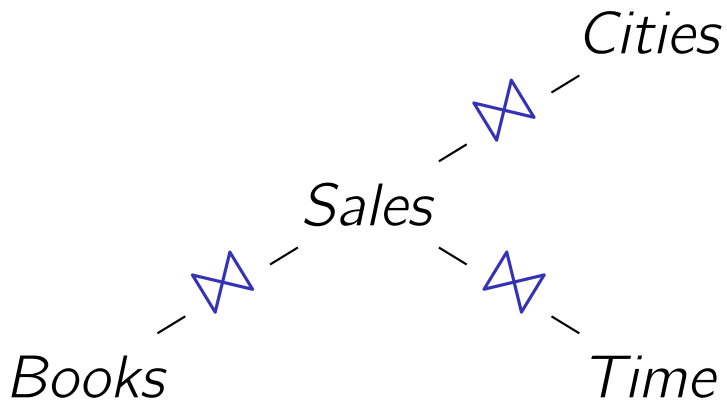
$\rightarrow$ Likewise: `GROUP BY ROLLUP (·)`

E.g., ranking, partitioning

```
SELECT c.City, t.Day,
       RANK() OVER (PARTITION BY City ORDER BY Sold)
  FROM Sales AS s, Cities AS c, Time AS t, Books AS b
 WHERE s.DayID = t.DayID AND s.CityID = c.CityID
   AND s.BookID = b.BookID AND b.Title = "Gone..."
```

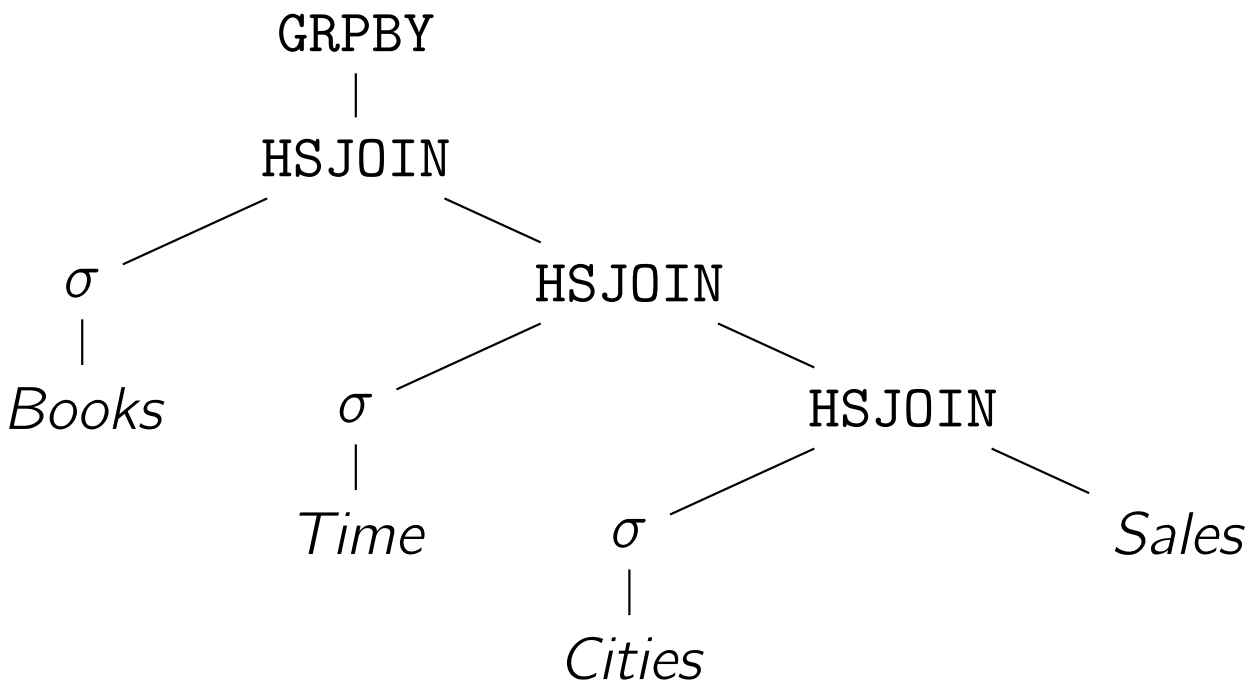
Star Join

The common query pattern is the **star join**.



✎ **How will a standard RDBMS execute such a query?**

Implementing Star Join Using Hash Joins



- (Hopefully) dimension subsets are small enough
  → Hash table(s) fit into memory.
- Here, hash joins effectively act like a **filter**.

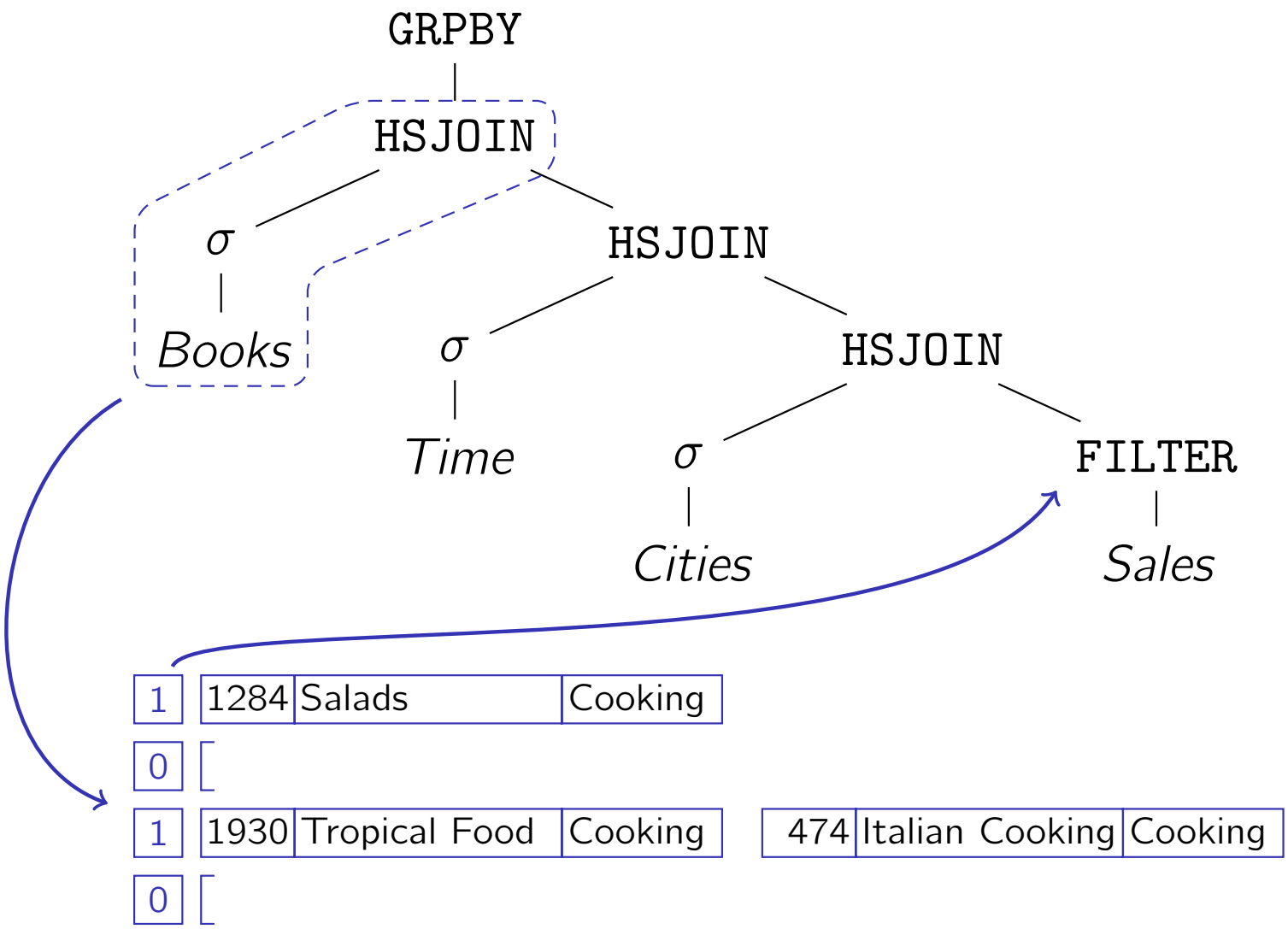
Implementing Star Join Using Hash Joins

Problems:

- Which of the filter predicates is most restrictive? — Tough optimizer task!
- A lot of processing time is invested in tuples that are eventually discarded.

- This strategy will have real trouble as soon as not **all** hash tables fit into memory.

Hash-Based Filters



→ Use compact bit vector to **pre-filter** data.

Hash-Based Filters

- Size of bit vector is independent of dimension tuple size.
  → And bit vector is **much smaller** than dimension tuples.
- Filtering may lead to **false positives**, however.
  → Must still do hash join in the end.
- Key benefit: **Discard tuples early**.

Nice side effect:

- In practice, will do pre-filtering according to all dimensions involved.
  → Can **re-arrange** filters according to actual(!) selectivity.

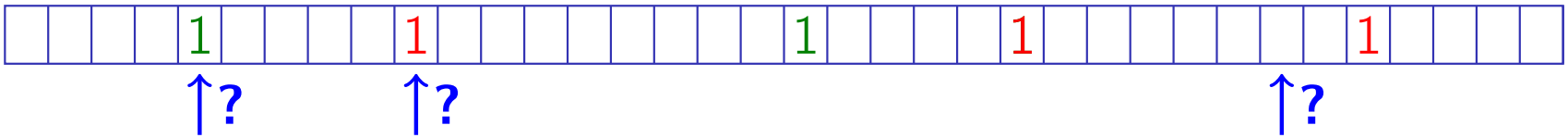
Bloom Filters

Bloom filters can improve filter efficiency.

Idea:

- Create (empty) bit field B with m bits.
- Choose k independent hash functions.
- For every dim. tuple, set k bits in B, according to hashed key values.

⟨1284, Salads, Cooking⟩
⟨1930, Tropical Food, Cooking⟩



⟨1735, Gone With the Wind, Fiction⟩

- To probe a fact tuple, check k bit positions
  → Discard tuple if any of these bits is 0.

Bloom Filters

Parameters:

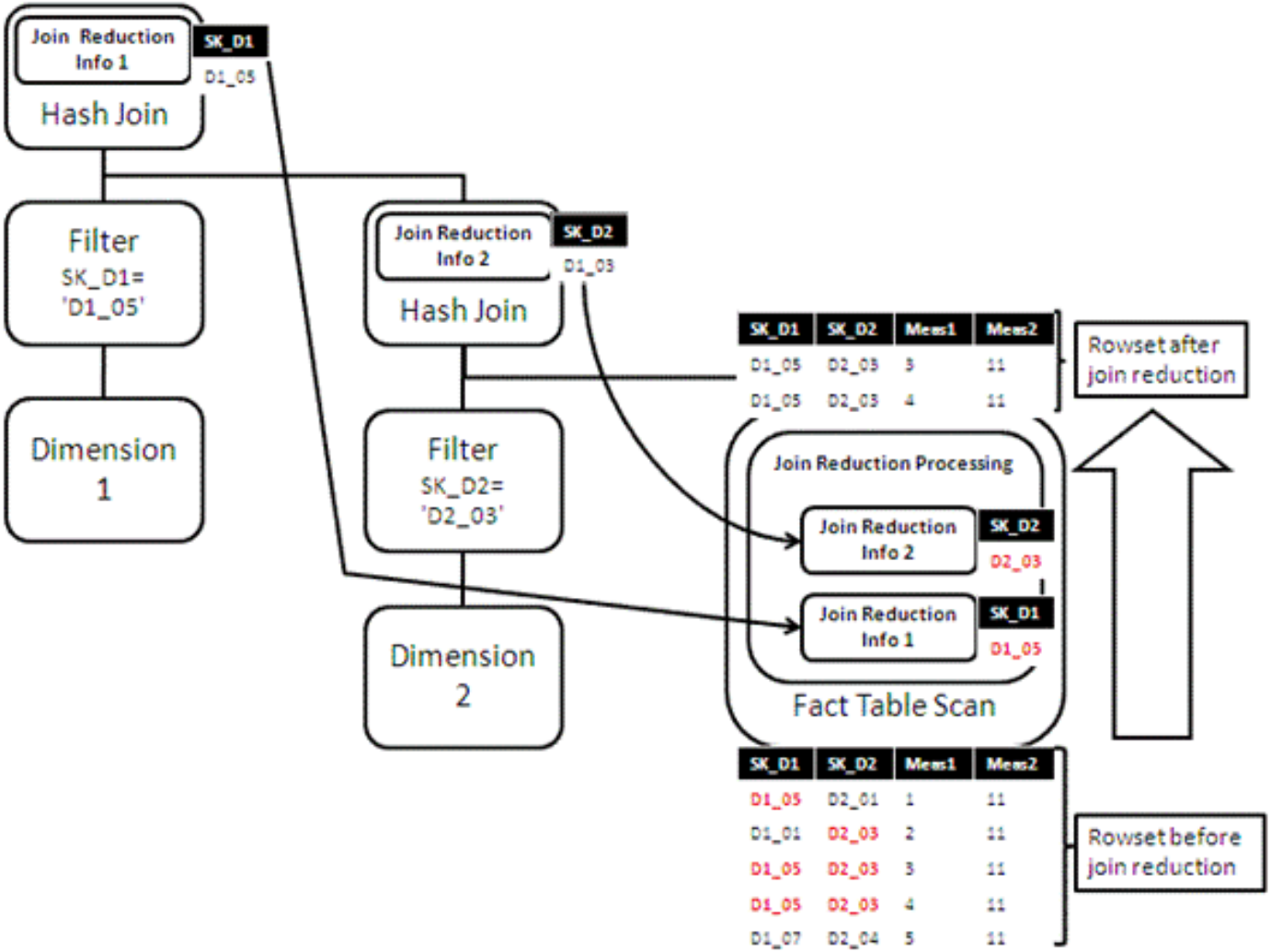
- Number of bits in B: m
  - $\rightarrow$ Typically measured in "bits per stored entry"
- Number of hash functions: k
  - $\rightarrow$ Optimal: about 0.7 times number of bits per entry.
  - $\rightarrow$ Too many hash functions may lead to high CPU load!

Example:

- 10 bits per stored entry lead to a filter accuracy of about 1 %.

Example: MS SQL Server

Microsoft SQL Server uses hash-based pre-filtering since version 2008.



Join Indices

To reduce join cost, we could **pre-compute** (partial) join results.

- ⇝ Database terminology: "materialize"
- ⇝ More generally: "materialized views"

Pre-computed join results are also called **join indices**.

Example: $\mathit{Cities} \bowtie \mathit{Sales}$

RID lists

- **Type 1:** $\mathit{join\ key} \rightarrow \langle \{\mathit{rid}_{\mathit{Cities}}\}, \{\mathit{rid}_{\mathit{Sales}}\} \rangle$
  (Record ids from *Cities* and *Sales* that contain given join key value.)
- **Type 2:** $\mathit{rid}_{\mathit{Cities}} \rightarrow \{\mathit{rid}_{\mathit{Sales}}\}$
  (Record ids from *Sales* that match given record in *Cities*.)
- **Type 3:** $\mathit{dim\ value} \rightarrow \{\mathit{rid}_{\mathit{Sales}}\}$
  (Record ids from *Sales* that join with *Cities* tuples that have given dimension value.)

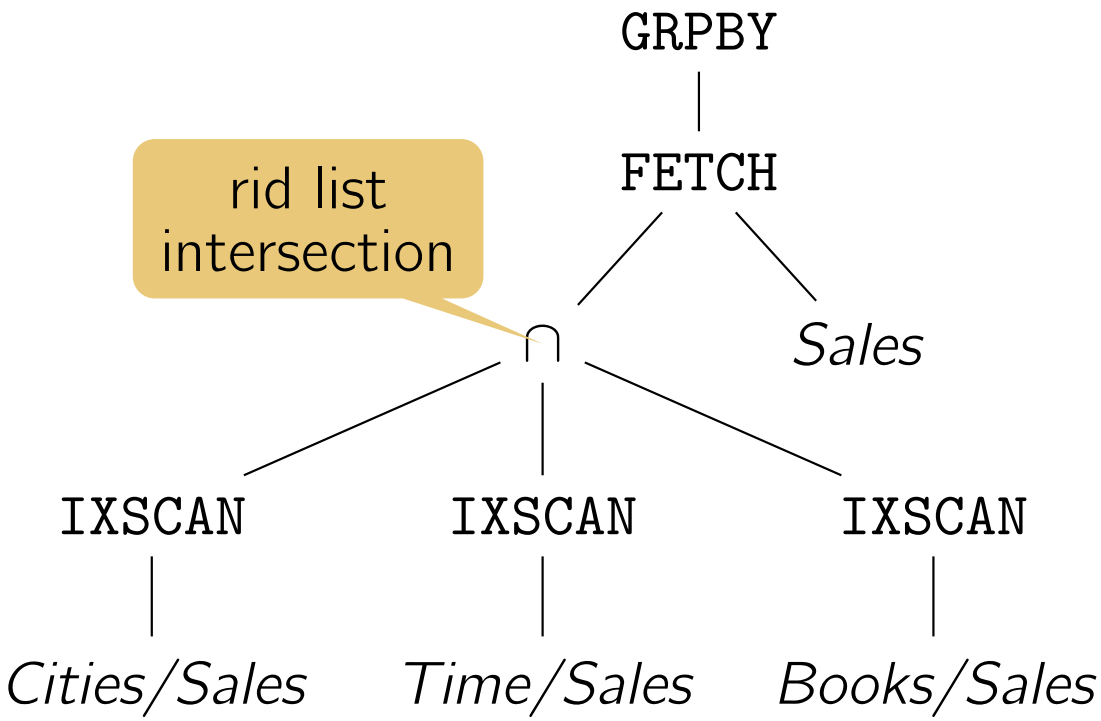
(Conventional B^+-trees are often $\mathit{value} \rightarrow \{\mathit{rid}\}$ mappings; cf. slide 76.)

Example: *Cities* ⋈ *Sales* Join Index

Cities			
rid	CtyID	City	State
c_1	6371	Arlington	VA
c_2	6590	Boston	MA
c_3	7882	Miami	FL
c_4	7372	Springfield	MA
⋮	⋮	⋮	⋮

Sales				
rid	BkID	CtyID	DayID	Sold
s_1	372	6371	95638	17
s_2	372	6590	95638	39
s_3	1930	6371	95638	21
s_4	2204	6371	95638	29
s_5	2204	6590	95638	13
s_6	1930	7372	95638	9
s_7	372	7882	65748	53
⋮	⋮	⋮	⋮	⋮

Star Join with Join Indices



1. For each of the dimensions, find matching *Sales* rids.
2. Intersect rid lists to determine relevant *Sales*.

Star Join with Join Indices

The strategy makes **rid list intersection** a critical operation.

→ Rid lists may be **sorted**.

→ Efficient implementation is (still) active research topic.

Down side:

- Rid list sorted only for (per-dimension) point lookups.

Challenge:

- Efficient **rid list implementation**.

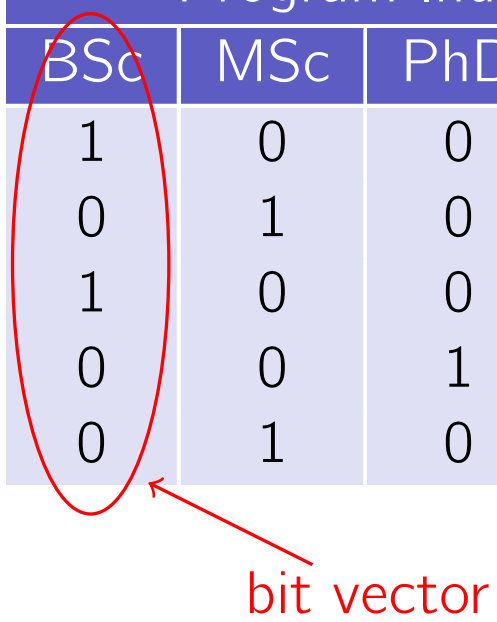
Bitmap Indices

Idea: Create **bit vector** for each possible column value.

Example: Relation that holds information about students:

Students		
LegiNo	Name	Program
1234	John Smith	Bachelor
2345	Marc Johnson	Master
3456	Rob Mercer	Bachelor
4567	Dave Miller	PhD
5678	Chuck Myers	Master

Program Index			
BSc	MSc	PhD	Dipl
1	0	0	0
0	1	0	0
1	0	0	0
0	0	1	0
0	1	0	0



Query Processing with Bitmap Indexes

Benefit of bitmap indexes:

- Boolean query operations (**and**, **or**, etc.) can be performed directly on bit vectors.

```
SELECT ···
  FROM Cities
 WHERE State = 'MA'
   AND (City = 'Boston' OR City = 'Springfield')
```

$$\downarrow$$

$$B_{\text{MA}} \wedge \left(B_{\text{Boston}} \vee B_{\text{Springfield}}\right)$$

- Bit operations are well-supported by modern computing hardware (↗ SIMD).

Equality vs. Range Encoding

Alternative encoding for **ordered domains**:

Students		
LegiNo	Name	Semester
1234	John Smith	3
2345	Marc Johnson	2
3456	Rob Mercer	4
4567	Dave Miller	1

Semester Index				
1	2	3	4	5
1	1	1	0	0
1	1	0	0	0
1	1	1	1	0
1	0	0	0	0

(set $B_{c_i}[k] = 1$ for all c_i smaller or equal than the attribute value $a[k]$).

✎ **Why would this be useful?**

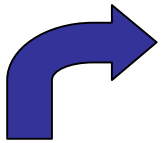
Range predicates can be evaluated more efficiently:

$$c_i < a[k] \leq c_j \;\leftrightarrow\; (\neg B_{c_i}[k]) \wedge B_{c_j}[k] \;\; .$$

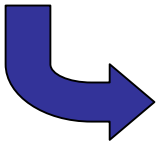
(but equality predicates become more expensive).

Data Warehousing Example

Index: D4.brand -> {RID}



RID	D4.id	D4.product	D4.brand	D4.group
0	1	Latitude E6400	Dell	Computers
1	2	Lenovo T61	Lenovo	Computers
2	3	SGH-i600	Samsung	Handheld
3	4	Axim X5	Dell	Handheld
4	5	i900 OMNIA	Samsung	Mobile
5	6	XPERIA X1	Sony	Mobile



Index: D4.group -> {RID}

B_{Dell}	B_{Len}	B_{Sam}	B_{Sony}
1	0	0	0
0	1	0	0
0	0	1	0
1	0	0	0
0	0	1	0
0	0	0	1

Bitmap Index: D4.brand

B_{Com}	B_{Hand}	B_{Mob}
1	0	0
1	0	0
0	1	0
0	1	0
0	0	1
0	0	1

Bitmap Index: D4.group

Query Processing: Example

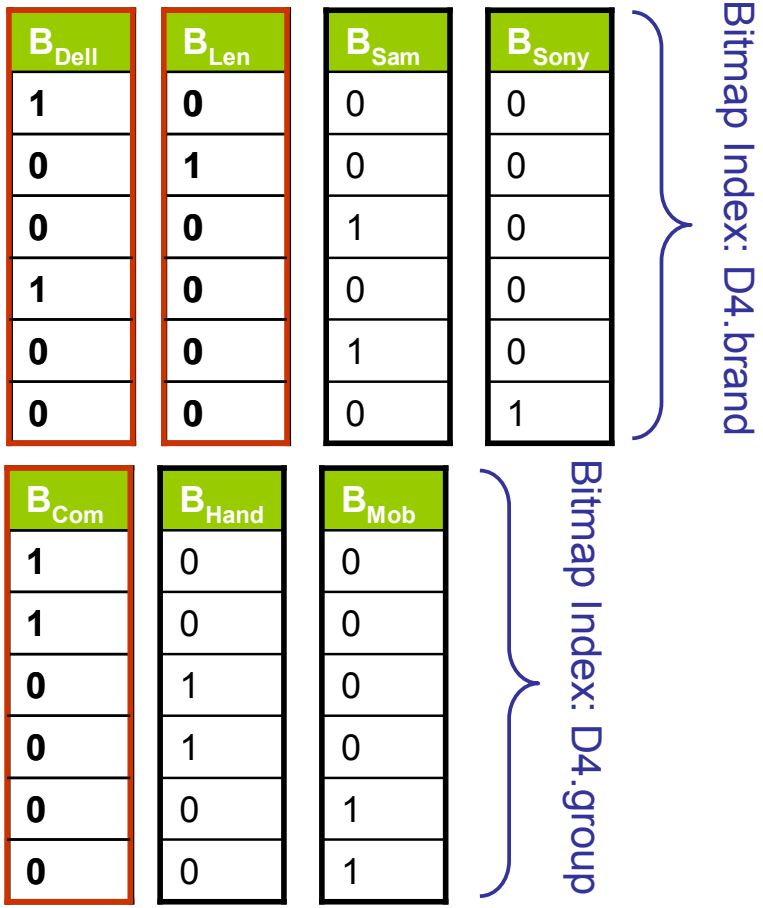
Sales in group ‘Computers’ for brands ‘Dell’, ‘Lenovo’?

```
SELECT SUM(F.price)
  FROM D4
 WHERE group='Computer'
   AND (brand='Dell'
        OR brand='Lenovo')
```

→ Calculate bit-wise operation

$$B_{Com} \wedge (B_{Dell} \vee B_{Len})$$

to find matching records.



Bitmap Indices for Star Joins

Bitmap indices are useful to implement **join indices**.

Here: Type 2 index for *Cities* ⋈ *Sales*

Cities			
rid	CtyID	City	State
c_1	6371	Arlington	VA
c_2	6590	Boston	MA
c_3	7882	Miami	FL
c_4	7372	Springfield	MA
⋮	⋮	⋮	⋮

Sales				
rid	BkID	CtyID	DayID	Sold
s_1	372	6371	95638	17
s_2	372	6590	95638	39
s_3	1930	6371	95638	21
s_4	2204	6371	95638	29
s_5	2204	6590	95638	13
s_6	1930	7372	95638	9
s_7	372	7882	65748	53
⋮	⋮	⋮	⋮	⋮

Idx		
c_1	c_2	⋯
1	0	⋯
0	1	⋯
1	0	⋯
1	0	⋯
0	1	⋯
0	0	⋯
0	0	⋯
⋮	⋮	⋱

→ One bit vector per RID in *Cities*.

→ Length of bit vector ≡ length of fact table (*Sales*).

Bitmap Indices for Star Joins

Similarly: Type 3 index $City \rightarrow \{Sales.rid\}$

Cities			
rid	CtyID	City	State
c_1	6371	Arlington	VA
c_2	6590	Boston	MA
c_3	7882	Miami	FL
c_4	7372	Springfield	MA
⋮	⋮	⋮	⋮

Sales				
rid	BkID	CtyID	DayID	Sold
s_1	372	6371	95638	17
s_2	372	6590	95638	39
s_3	1930	6371	95638	21
s_4	2204	6371	95638	29
s_5	2204	6590	95638	13
s_6	1930	7372	95638	9
s_7	372	7882	65748	53
⋮	⋮	⋮	⋮	⋮

Idx			
VA	MA	FL	⋯
1	0	0	⋯
0	1	0	⋯
1	0	0	⋯
1	0	0	⋯
0	1	0	⋯
0	1	0	⋯
0	0	1	⋯
⋮	⋮	⋱	

→ One bit vector per *City* value in *Cities*.

→ Length of bit vector ≡ length of fact table (*Sales*).

Space Consumption

For a column with n **distinct values**, n bit vectors are required to build a bitmap index.

For a table wit N rows, this leads to a **space consumption** of

$$N \cdot n \text{ bits}$$

for the full bitmap index.

This suggests the use of bitmap indexes for **low-cardinality attributes**.

→ *e.g.*, product categories, sales regions, etc.

For comparison: A 4-byte integer column needs $N \cdot 32$ bits.

→ For $n \lesssim 32$, a bitmap index is more compact.

Reducing Space Consumption

For larger n, space consumption can be reduced by

1. **alternative bit vector representations** or
2. **compression**.

Both may be a **space/performance trade-off**.

Decomposed Bitmap Indexes:

- Express all attribute values v as a **linear combination**

$$v = v_0 + \underbrace{c_1}\, v_1 + \underbrace{c_1 c_2}\, v_2 + \cdots + \underbrace{c_1 \cdots c_k}\, v_k \quad (c_1, \ldots, c_k \text{ constants}).$$

- Create a **separate bitmap index** for each variable v_i.

Decomposed Bitmap Indexes

Example: Index column with domain $[0, \ldots, 999]$.

- Regular bitmap index would require 1000 bit vectors.
- Decomposition ($c_1 = c_2 = 10$):

$$v = 1v_1 + 10v_2 + 100v_3 \quad .$$

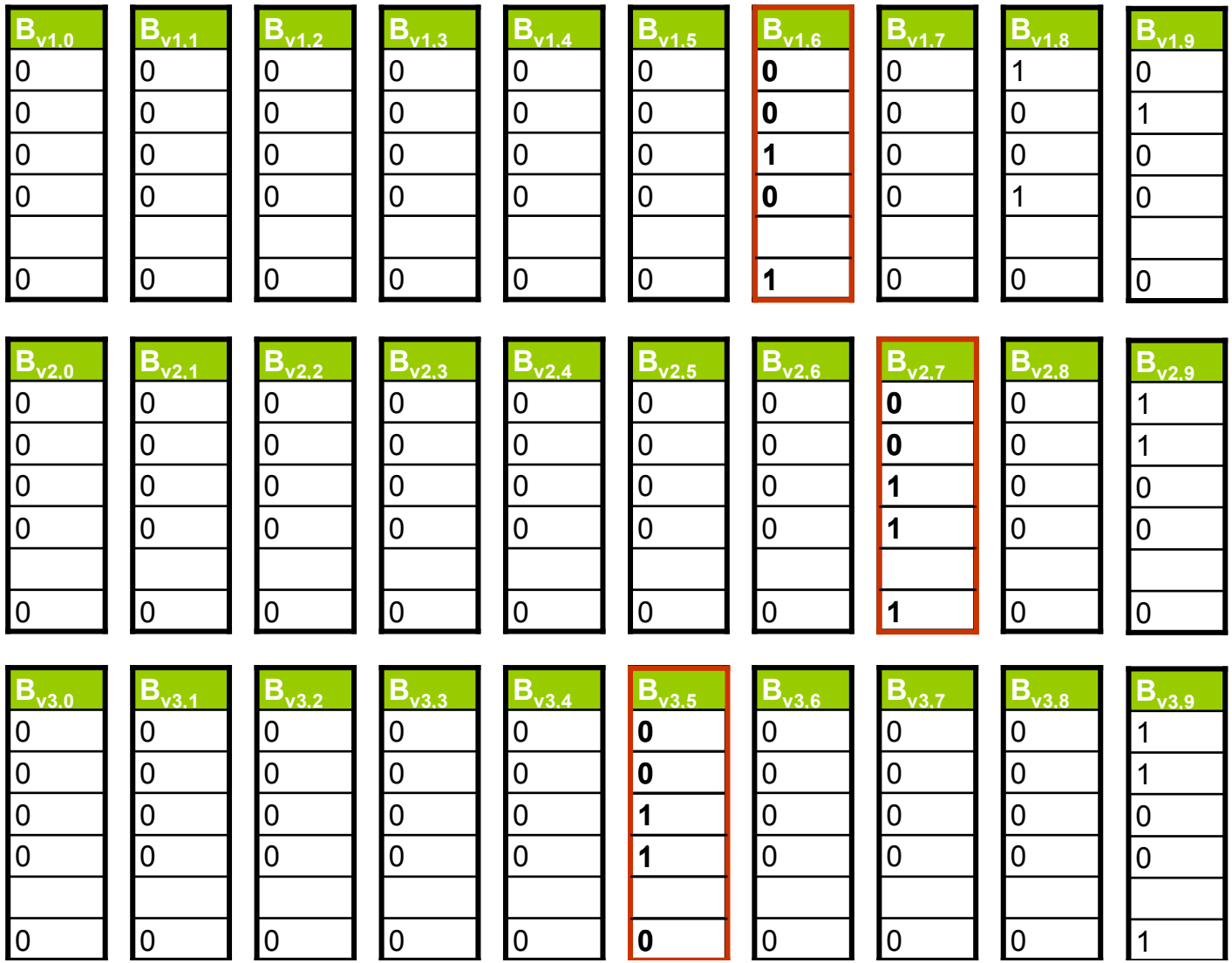
- Need to create **3 bitmap indexes** now, each for **10 different values**
  $\rightarrow$ 30 bit vectors now instead of 1000.
- However, need to **read 3 bit vectors** now (and **and** them) to answer **point query**.

Decomposed Bitmap Indexes

- **Query:**
  `a=576=5*100+7*10+6*1`
- **RIDs:**
  $B_{v3,5} \wedge B_{v2,7} \wedge B_{v1,6} =$ `[0010…0]`

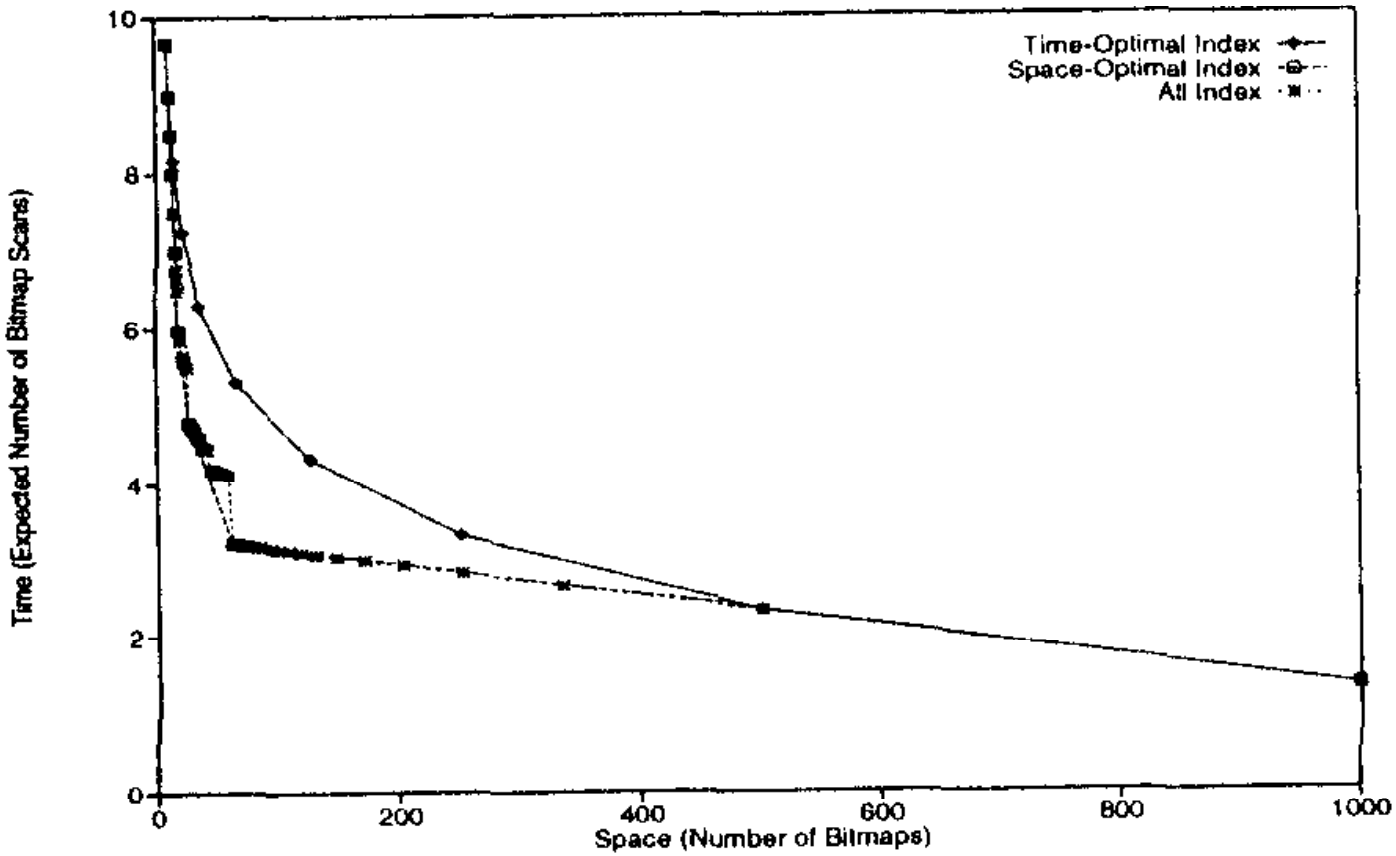
=> RID 3, ...

RID	a
0	998
1	999
2	576
3	578
1000	976



Space/Performance Trade-Offs

Setting c_i parameters allows to trade space and performance:



source: Chee-Yong Chan and Yannis Ioannidis. Bitmap Index Design and Evaluation. *SIGMOD 1998*.

Compression

Orthogonal to bitmap decomposition: Use **compression**.

- *E.g.*, straightforward equality encoding for a column with cardinality n: $1/n$ of all entries will be 0.

Which compression algorithm would you choose?

Compression

Problem: Complexity of (de)compression $\leftrightarrow$ simplicity of bit operations.

- Extraction and manipulation of **individual bits** during (de)compression can be expensive.
- Likely, this would off-set any efficiency gained from logical operations on **large CPU words**.

Thus:

- Use (rather simple) **run-length encoding**,
- but respect **system word size** in compression scheme.

↗ Wu, Otoo, and Shoshani. Optimizing Bitmap Indices with Efficient Compression. *TODS*, vol. 31(1). March 2006.

Word-Aligned Hybrid (WAH) Compression

Compress into a sequence of 32-bit words:

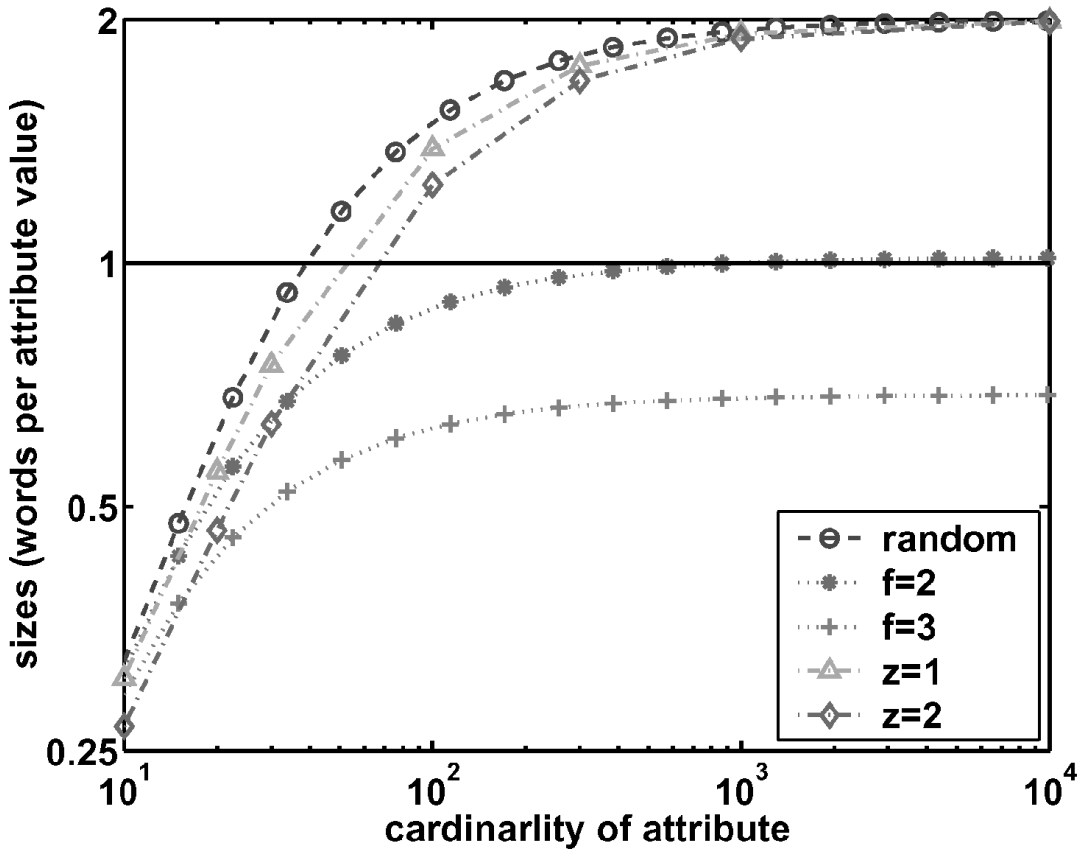
Bit ■ tells whether this is a **fill word** or a **literal word**.

- **Fill word (■ = 1):**
  - Bit ■ tells whether to fill with 1s or 0s.
  - Remaining 30 ■ bits indicate the number of fill bits.
    - → This is the number of **31-bit blocks** with only 1s or 0s.
    - → *e.g.*, ■ = 3: represents 93 1s/0s.
- **Literal word (■ = 0):**
  - Copy 31 ■ bits directly into the result.

WAH: Effectiveness of Compression

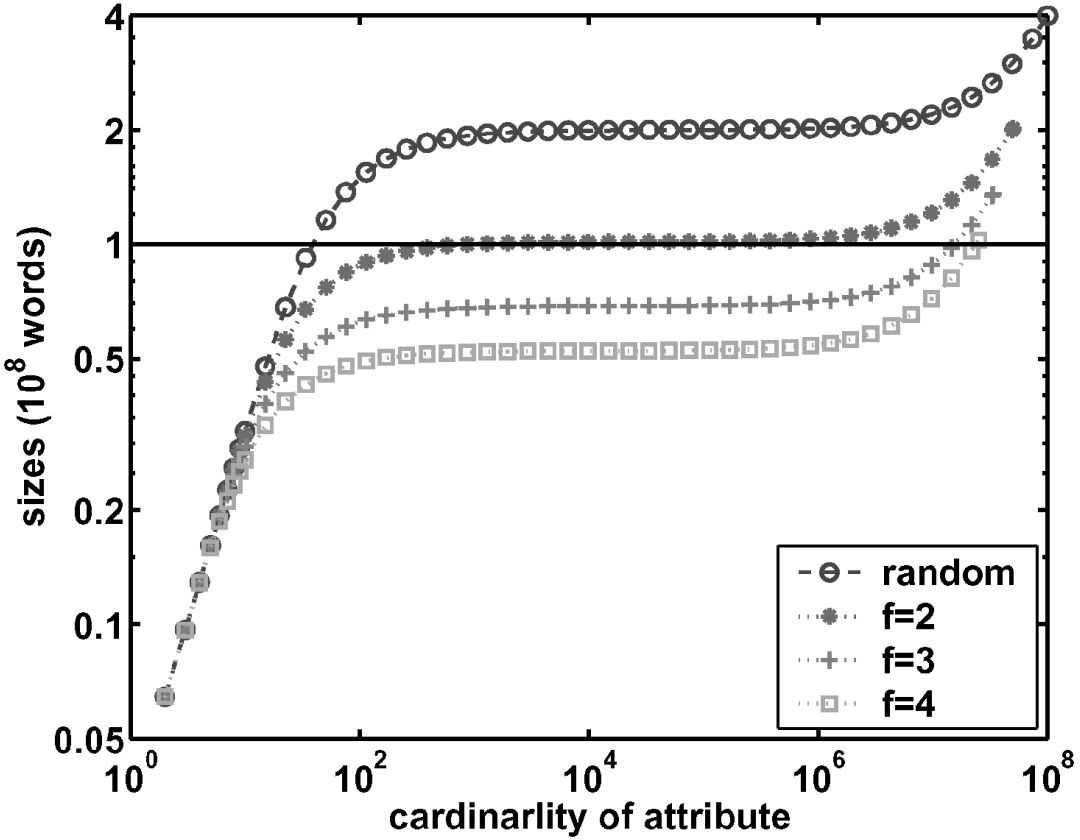
WAH is good to counter the space explosion for **high-cardinality attributes**.

- At most 2 words per '1' bit in the data set
  - $\rightsquigarrow$ At most $\approx 2 \cdot N$ words for a table with N rows, even for large n (assuming a bitmap that uses equality encoding).



Source: Wu et al. TODS 31(1), 2006.

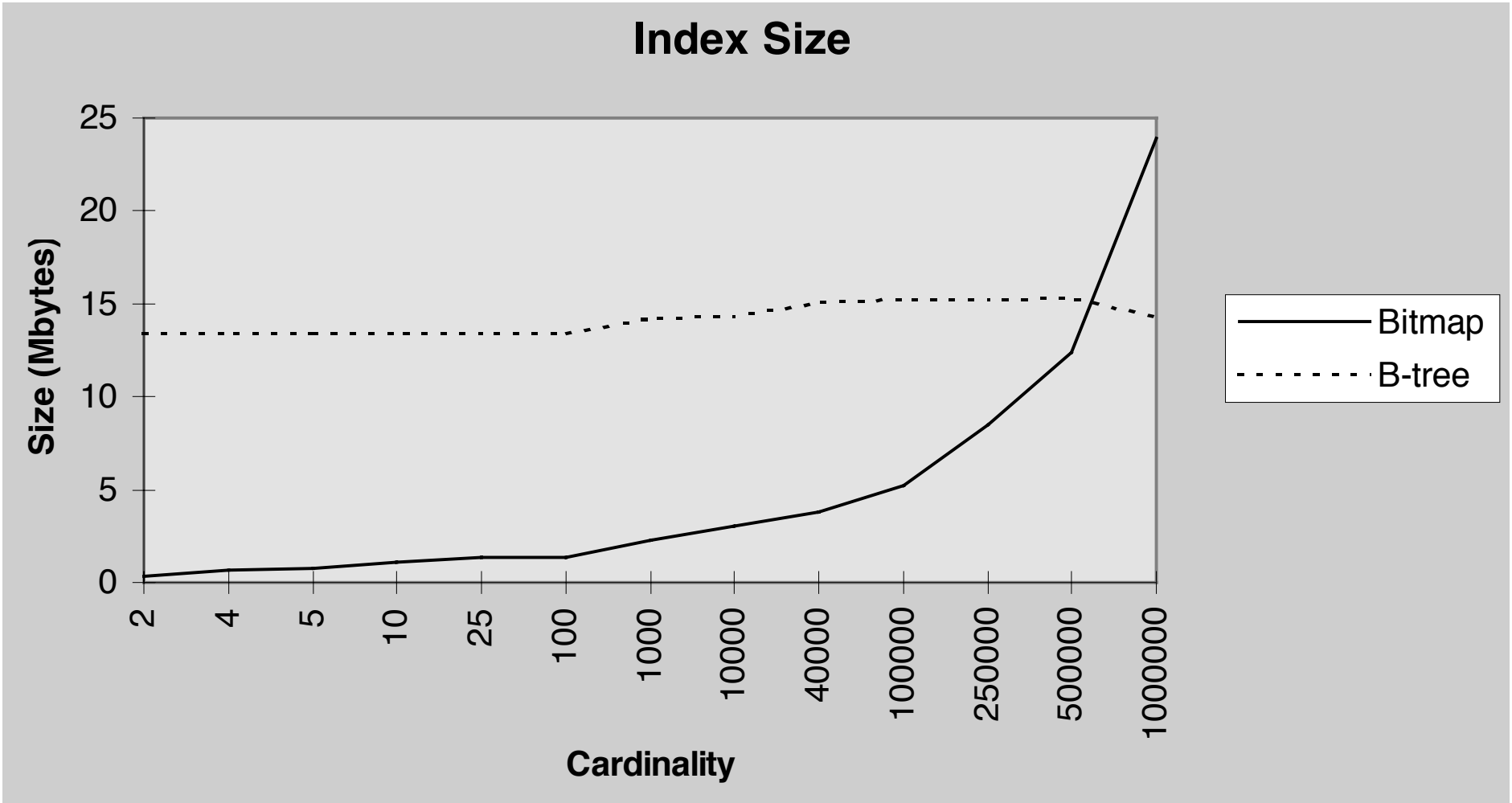
WAH: Effectiveness of Compression



(b) $n = 10^8$

- If (almost) all values are distinct, additional **bookkeeping** may need some more space ($\leadsto 4 \cdot 10^8$ bits for cardinality 10^8).

Bitmap Indexes in Oracle 8



Encoding ↔ Bitmap Sparseness/Attribute Cardinality

The most space-efficient bitmap representation depends on the **number of distinct values** (*i.e.*, the **sparseness** of the bitmap).

- **low attribute cardinality** (dense bitmap)
  → can use **un-compressed bitmap**
  WAH compression won't help much (but also won't hurt much)
- **medium attribute cardinality**
  → use (WAH-)**compressed bitmap**
- **high attribute cardinality** (many distinct values; sparse bitmap)
  → Encode "bitmap" as **list of bit positions**

In addition, compressed bitmaps may be a good choice for data with **clustered content** (this is true for many real-world data).

Bitmaps ↔ Row IDs?

Bitvectors encode a list of integer positions. But we need RIDs. What gives?

RID Lists

Conversely, bitmaps may be a good way to encode **lists of rows**.

→ Represent **RID lists** in B-tree leaves as (compressed) bit vectors.

In practice:

- Divide table into **segments** ($\approx$32,000 tuples/segment).
- Separate bitmap for each segment.
- Per segment can decide on WAH $\leftrightarrow$ RID list.

→ *E.g.*, Oracle's bitmap indexes are essentially that (though exact encoding is proprietary).

Benefits:

- May be able to **skip** over entire segments.
- Keep **update** cost reasonable.