Data Warehousing

Jens Teubner, TU Dortmund jens.teubner@cs.tu-dortmund.de

Winter 2015/16

© Jens Teubner · Data Warehousing · Winter 2015/16

Part V

Implementing a Data Warehouse Engine

OLAP Cubes



Query pattern:

- Specify **range(s)** along each dimension.
- 2 Collect qualifying events.
- 3 Aggregate over their measures.

Sefficient evaluation? (Assume in-memory processing for now.)

- We can prepare helper data structures if we want to.
 - ightarrow Indexes, precomputed results, sorting, data organization, ...

Our route through this chapter:

- 1 Database implementation 101
- 2 Materialized (sub)results ("materialized views")
- 3 Star Joins with B-trees
- 4 Bloom filters to accelerate Star Joins
- 5 Join indices, bitmap indices

Memory Hierarchy



fast, but expensive and small, memory close to CPU

- larger, slower memory at the periphery
- Use secondary storage for capacity, but avoid accessing it because of its slow speed.

Key Problem: Latency



- Hard drives require physical movement of disk heads.
- **Latency** dominates cost.
- **Bandwidth** less of an issue (~ parallelism).



Database Processing Model

Databases execute **query plans** using the **Volcano iterator model**:

- Operators request tuples from their input using next ().
- Data is processed **tuple at a time**.
 - $\rightarrow~$ Tuples "flow" from the base tables to the plan root.
- "pipelining"



- Simple interface to compose arbitrary query plans.
- High scalability: single tuples in flight, regardless of input size.

Note:

Some operators cannot be pipelined; they block and materialize their full input and/or output.



Indices

Index structures can be created to accelerate data access.

Usually, indices cause overhead for update operations.

Generally speaking, an index provides a lookup mechanism

attribute value(s) \mapsto record identifier(s) ,

where a *record identifier* or *rid* encodes the **physical location** of a matching tuple.



Star schema in a relational database:

- Fact table: each entry holds a set of foreign keys
- These point to dimension tables

Conceptually, a star query

- **1** joins the fact table to a number of dimension tables,
- 2 restricts the number of tuples (to obtain a "dice"), and
- **3** aggregates/groups according to some grouping criterion.

fact

$$\dim_1 \longrightarrow \sigma - \text{GRPBY}$$

 $\dim_2 \longrightarrow \sigma - \text{GRPBY}$

Expressed as relational algebra, this would look like



Pushing down selection criteria (σ) may be more efficient:



Problems:

- Fact table is huge.
- Joins will become **expensive**.

Good join implementations: $cost(R \bowtie S) \propto |R| + |S|$.

However,

we'd have plenty of time to pre-compute (partial) results.

Idea:

build materialized views over (partial) results.

"Regular" view:

```
CREATE VIEW sales_loc (date_key, city, state, qty) AS
SELECT f.date_key, loc.city, loc.state, f.qty
FROM sales_fact AS f, location_dim AS loc
WHERE f.loc_key = loc.loc_key
```

- ightarrow "Register" a query under some name (here: sales_loc)
- $ightarrow \,$ View will be accessible for querying like a real table
- → View result will be computed on the fly
 (One execution technique can be to expand a referenced view to its definition and execute the resulting, larger query.)
- $\rightarrow\,$ Mostly a **convenience feature**, plus some advantages for access control, maybe also query cost estimation

Many systems offer a mechanism to **persist** the view result instead. \rightarrow Update on modifications, access, or manually.

Such materialized views are pre-computed queries.

```
E.g., IBM DB2:
```

```
CREATE TABLE table-name (attlist) AS (
select-from-where block
)
DATA INITIALLY DEFERRED REFRESH data refresh options
```

- + Pre-computed information may speed up querying.
- Materialization requires space
- Increases update cost (less a problem for data warehouses)

Which materialized views should we create?

ightarrow Views may be as large as fact table, so **space** is an issue.

Insight:

- Views may be helpful for query evaluation even when they contain a superset of the required information.
 - ightarrow Can refine filter criteria when querying the view
 - ightarrow Can aggregate fine grain ightarrow coarse grain

Example:

Fact table with three dimensions: part, supplier, customer

grouping attributes	# rows
part, supplier, customer	6 M
part, customer	6 M
part, supplier	0.8 M
supplier, customer	6 M
part	0.2 M
supplier	0.01 M
customer	0.1 M
– (none)	1

- ≈ 19 M rows storage needed
- Could save 12 M rows by not storing (part, customer) and (supplier, customer);
 no harm to query performance.

Can a materialized view V be used to answer a query Q?

Assumption:

V and Q are both star queries

Sufficient conditions:

- Selection predicates in Q subsumed by those in V.
- **GROUP** BY **attributes** in *Q* subsumed by those in *V*.
- Aggregation functions in *Q* compatible with those of *V*.
- All tables referenced in *V* must also be referenced in *Q*.

Problem:

Predicate subsumption **not decidable** for arbitrary predicates.

Thus:

Restrict to only simple predicates:

attribute op constant .

Convert predicates to **disjunctive normal form**.

Example:

- query predicate p_Q : year = 2008 \land quarter = 4
- view predicate *p_V* : *year* = 2008
- $ightarrow p_V$ subsumes p_Q ; can use V to answer $Q = \sqrt{2}$

Derivability: GROUP BY Lattice



Arrow $V \rightarrow V'$: V' can be derived from V.

■ Example: Create only V_{part, supplier} and V_{supplier, customer} → Can still group by {part}, {supplier}, {customer}, and {}.

For **independent dimension attributes**, the lattice becomes a **hypercube**

 $\rightarrow n$ independent dimensions $\sim 2^n$ views.

Known hierarchies simplify the lattice

- $\rightarrow~$ Can group by week, given a grouping by day
- → Can group by month, given a grouping by day; can group by year, given a grouping by month



Aggregate functions have different behaviors:

- "additive": $f(X_1 \cup X_2) = f(f(X_1), f(X_2))$ and f^{-1} exists.
 - → e.g., $sum(a_1, a_2, a_3) = sum(sum(a_1, a_2), sum(a_3))$ and $sum(a_3) = sum(a_1, a_2, a_3) sum(a_1, a_2)$
- "semi-additive": same, but *f*⁻¹ does not exist.
 - \rightarrow e.g., min(a_1, a_2, a_3) = min(min(a_1, a_2), min(a_3))
- **"additive computable:"** $f(X) = f'(f_1(X), F_2(X), ..., F_k(X))$ where f_i are (semi-)additive functions.
 - \rightarrow e.g., avg(X) = sum(X)/count(X)
- "others:" e.g., median computation

Derivability: Aggregate Functions

Behavior of aggregate functions determines

- whether a query Q can be answered based on a view V
- whether updates in the base table can be propagated to V
 - ightarrow view maintenance

In practice:

- Strict (syntactic) rules on queries that can be defined as materialized views.
- e.g., IBM DB2 (excerpt) ightarrow

- When a GROUP BY clause is specified, the following considerations apply:
 - The supported column functions are SUM, COUNT, COUNT_BIG and GROUPING (without DISTINCT). The select list must contain a COUNT(*) or COUNT_BIG(*) column. If the materialized query table select list contains SUM(X), where X is a nullable argument, the materialized query table must also have COUNT(X) in its select list. These column functions cannot be part of any expressions.
 - A HAVING clause is not allowed.
 - If in a multiple partition database partition group, the distribution key must be a subset of the GROUP BY items.
- The materialized query table must not contain duplicate rows, and the following restrictions specific to this uniqueness requirement apply, depending upon whether or not a GROUP BY clause is specified.

•

:

All tables *T* referenced by *V* must also be referenced by *Q* (and joined using the same join predicate).

Problem:

- Joins are not lossless if they are not equi-joins along a foreign key relationship.
 - $\, \sim \,$ "Information Systems", lossless/lossy decomposition
- If joins are lossless, not all tables of *V* must be referenced in *Q*

Which Materialized Views Should We Create?

Strategy:

- **1** Create view with GROUP BY at **finest grain** needed.
- 2 Repeatedly create new view that yields maximum benefit.
- Stop when storage budget is exceeded.
- Input: Workload description
- E.g., DB2 Design Advisor db2advis
 - Input: workload with queries and DML statements
 - Output: Recommendation for indexes and materialized views ("materialized query tables, MQTs" in DB2 speak)

A lighter-weight form of pre-computed data are **indices**.

E.g., B-tree index:



Searching records by value incurs

- **1** Traverse index using the search key
- 2 Fetch tuples from data pages.

Step **I** incurs **about one I/O per search**.

- \rightarrow Inner nodes are usually **cached**.
- ightarrow For small tables, even the full index might fit into the cache.

Step 2 requires about one I/O per result tuple.

 \rightarrow Following *rid* pointers results in **quasi-random I/O**.

(If the result set is large, the system might also decide to **sort** the list of qualifying *rids* first to improve disk access pattern.)

Two typical ways of using an index are:

Point or range conditions in the query

 \rightarrow E.g., SELECT · · · WHERE attr = constant

2 Join processing

- $\rightarrow~$ Index nested loops join:
 - 1 Function: index_nljoin(R, S, p)
 - **2 foreach** record $r \in R$ **do**
 - access index using r and p and append matching tuples to result;

How could star queries benefit from indexes?

Strategy 1: Index on value columns of dimension tables

- 1. For each **dimension table** *D_i*:
 - a. Use index to find matching dimension table rows $d_{i,j}$.
 - b. Fetch those $d_{i,j}$ to obtain key columns of D_i .
 - c. Collect a list of **fact table** *rids* that reference those dimension keys.

🇠 How?

- 2. Intersect lists of fact table *rids*.
- 3. Fetch remaining fact table rows, group, and aggregate.

Strategy 2: Index on primary key of dimension tables

1. Scan fact table

- 2. For each fact table row f:
 - a. **Fetch** corresponding dimension table row *d*.
 - b. Check "slice and dice" conditions on d; skip to next fact table row if predicate not met.
 - c. Repeat 2.a for each dimension table.
- 3. Group and aggregate all remaining fact table rows.

Problems and advantages of Strategy 1?

- + Fetch only **relevant** fact table rows (good for selective queries).
- 'Index ightarrow fetch ightarrow index ightarrow intersect ightarrow fetch' is cumbersome. $\,\star\,$
- List intersection is expensive.
 - 1. Again, lists may be large, intersection small.
 - 2. Lists are generally **not sorted**.

Problem \star can be reduced with a "trick":

- Create an index that contains value and key column of the dimension table.
 - ightarrow No **fetch** needed to obtain dimension key.
- Such indexes allow for index-only querying.
 - ightarrow Acess only index, but not data pages of a table.

E.g.,

CREATE INDEX QuarterIndex ON DateDimension (Quarter, DateKey)

 \rightarrow Will only use *Quarter* as a **search criterion** (but not *DateKey*).

础 IBM DB2:

Include columns in index, yet do **not** make them a search key.

CREATE INDEX IndexName ON TableName (col₁, col₂,..., col_n) INCLUDE (col_a, col_b,...)

(In UNIQUE indexes, it makes a difference whether a column is part of the search key or not. This is the only situation where the INCLUDE clause is allowed in DB2.)

Problems and advantages of Strategy 2?

- + For small dimension tables, all indexes might fit into memory.
 - $\rightarrow~$ On the other hand, indexes might not be worth it; can simply build a hash table on the fly.
- Fact table is large \rightarrow many index accesses.
- Individually, each dimension predicate may have low selectivity.
 - *E.g.*, four dimensions, each restricted with 10% selectivity:
 - ightarrow Overall selectivity as low as 0.01 %.
 - $\rightarrow\,$ But as many as 10 %/1 %/... of fact table tuples pass individual dimension filters (and fact table is huge).

Together, dimension predicates may still be highly selective.

• Cost is independent of predicate selecitivites.

What do you think about this query plan?

ightarrow Join dimension tables first, then fact table as last relation.



Joins between dimension tables are effectively Cartesian products.



 \rightarrow Clearly won't work if (filtered) dimension tables are large.

Hub Star Join

Idea:



- Cartesian product approximates the set of foreign key values relevant in the fact table.
- Join Cartesian product with fact table using index nested loops join (multi-column index on foreign keys).

Advantages:

- + Fetch only **relevant** fact table rows.
- + No intersection needed.
- + No **sorting** or **duplicate removal** needed.

Down Sides:

- Cartesian product overestimates foreign key combinations in the fact table.
 - \rightarrow Many key combinations won't exist in the fact table.
 - ightarrow Many unnecessary index probes.

Overall:

Hub Join works well if **Cartesian product is small**.

Zigzag Join



Hash Join

Hash join is one of the classical join algorithms.

```
To compute R \bowtie S,
```

- **1** Build a hash table on the "outer" join relation S.
- **2** Scan the "inner" relation R and probe into the hash table for each tuple $r \in R$.

```
} Build Phase
} Join Phase
```

Hash Join



✓ $\mathcal{O}(N)$ (approx.) ✓ Easy to **parallelize**

Implementing Star Join Using Hash Joins



• (Hopefully) dimension subsets are small enough \rightarrow Hash table(s) fit into memory.

Here, hash joins effectively act like a filter.

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



$\rightarrow~$ Use compact bit vector to pre-filter data.

- Size of bit vector is independent of dimension tuple size.
 - ightarrow And bit vector is **much smaller** than dimension tuples.
- Filtering may lead to **false positives**, however.
 - ightarrow 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.
 - \rightarrow 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)
```



 $\langle 1735,$ Gone With the Wind, Fiction \rangle

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

Parameters:

- Number of bits in *B*: *m*
 - $\rightarrow~$ Typically measured in "bits per stored entry"
- Number of hash functions: k
 - ightarrow Optimal: about 0.7 times number of bits per entry.
 - $ightarrow\,$ 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.



A variant of pre-computed data (similar to materialized views) are **join indices**.

Example: Cities \bowtie Sales

RID lists

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

(Conventional B⁺-trees are often value \rightarrow {rid} mappings; cf. slide 97.)

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.

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

- ightarrow Rid lists may or may not be **sorted**.
- $\rightarrow\,$ Efficient implementation is (still) active research topic.

Down side:

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

Challenge:

Efficient rid list implementation.

Idea: Create bit vector for each possible column value.

Example: Relation that holds information about students:

		\sim	Prograr	n Inde>	(
StudNo	Name	Program		ßSc	MSc	PhD	Dipl	
1234	John Smith	Bachelor		1	0	0	0	
2345	Marc Johnson	Master		0	1	0	0	
3456	Rob Mercer	Bachelor		1	0	0	0	
4567	Dave Miller	PhD		0	0	1	0	
5678	Chuck Myers	Master		\ 0 /	1	0	0	
bit vector								

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_{MA} \land (B_{Boston} \lor B_{Springfield})
```

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

Alternative encoding for ordered domains:

Students					
StudNo	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]).

Nhy would this be useful? Range predicates can be evaluated more efficiently:

$$c_i > a[k] \geq c_j \iff (\neg B_{c_i}[k]) \land B_{c_j}[k]$$
 .

(but equality predicates become more expensive).

Data Warehousing Example



Bitmap Index: D4.brand























Index: D4.brand -> {RID} B_{Dell}

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

Sales in group 'Computers' for brands 'Dell', 'Lenovo'?

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

ightarrow 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							ldx	
rid	BkID	CtyID	DayID	Sold		C 1	C ₂	• • •
S 1	372	6371	95638	17		1	0	• • •
S ₂	372	6590	95638	39		0	1	• • •
S 3	1930	6371	95638	21		1	0	• • •
S 4	2204	6371	95638	29		1	0	• • •
S 5	2204	6590	95638	13		0	1	• • •
S 6	1930	7372	95638	9		0	0	• • •
S 7	372	7882	65748	53		0	0	
:	÷	÷	÷	:		÷	÷	·

- \rightarrow One bit vector per RID in Cities.
- \rightarrow Length of bit vector \equiv length of fact table (Sales).

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

Cities				Sales				ldx				
rid	CtyID	City	State	rid	BkID	CtyID	DayID	Sold	VA	MA	FL	• • •
C ₁	6371	Arlington	VA	S 1	372	6371	95638	17	1	0	0	• • •
C ₂	6590	Boston	MA	s ₂	372	6590	95638	39	0	1	0	• • •
C 3	7882	Miami	FL	S 3	1930	6371	95638	21	1	0	0	
C 4	7372	Springfield	MA	S 4	2204	6371	95638	29	1	0	0	
				S 5	2204	6590	95638	13	0	1	0	
:	:		:	S 6	1930	7372	95638	9	0	1	0	
				S 7	372	7882	65748	53	0	0	1	• • •
				÷	÷	÷	÷	÷	÷	÷	۰.	

- \rightarrow One bit vector per City value in Cities.
- \rightarrow Length of bit vector \equiv length of fact table (Sales).

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

 $N \cdot n$ bits

for the full bitmap index.

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

ightarrow e.g., product categories, sales regions, etc.

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

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

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

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

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

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

- Regular bitmap index would require 1000 bit vectors.
- Decomposition (c₁ = c₂ = 10):

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

- Need to create 3 bitmap indexes now, each for 10 different values
 - ightarrow 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} ∧ B_{v2,7} ∧ $B_{v1,6} =$ [0010...0]

=> RID 3, ...

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

B _{v1,0}	B _{v1,1}	B _{v1,2}
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

	-	
B _{v2.0}	B _{v2.1}	B _{v2.2}
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

B _{v3.0}	B _{v3.1}	B _{v3.2}
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

2,2	B _{v2,3}	
	0	
	0	
	0	
	0	
	0	

۱n

3,1	B _{v3.2}	B _{v3,3}
	0	0
	0	0
	0	0
	0	0
	0	0

4	B _{v3,5}
	0
	0
	1
	1
	0

	1	0
1	P	P
	0 0	0
	0	0
	0	1
	0	1

-		
	B _{v2,8}	
	0	
	0	
	0	
	0	
	0	

B _{v2.9}
1
1
0
0
0

	_
B _{v3,7}	B _{v3,8}
0	0
0	0
0	0
0	0
0	0

	B _{v3.9}
	1
	1
	0
	0
	1

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?

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 **t**ells whether this is a **fill word** or a **literal word**.

- Bit _ tells whether to fill with 1s or 0s.
- Remaining 30 bits indicate the number of fill bits.
 - $\rightarrow~$ This is the number of **31-bit blocks** with only 1s or 0s.
 - \rightarrow 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
 - → At most $\approx 2 \cdot N$ words for table with N rows, even for large n (assuming a bitmap that uses equality encoding).



WAH: Effectiveness of Compression



If (almost) all values are distinct, additional **bookkeeping** may need some more space (→ 4 · 10⁸ bits for cardinality 10⁸).



$\textbf{Encoding} \leftrightarrow \textbf{Sparseness/Attribute Cardinality}$

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
 - ightarrow use (WAH-)compressed bitmap
- high attribute cardinality (many distinct values; sparse bitmap)
 - \rightarrow 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).