POZNAN UNIVERSITY OF TECHNOLOGY

Data Warehouse Physical Design: Part I

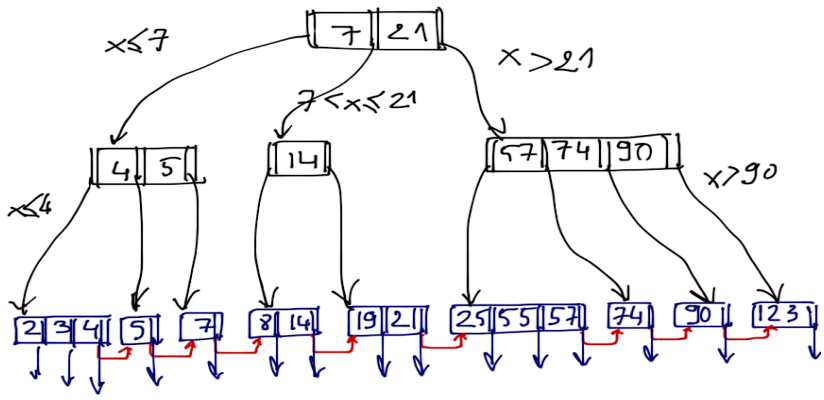
Robert Wrembel
Poznan University of Technology
Institute of Computing Science
Robert.Wrembel@cs.put.poznan.pl
www.cs.put.poznan.pl/rwrembel

Lecture outline

- **Basic index structures**
  - **B-tree**
  - **bitmap**
  - **join**
  - **bitmap join (Oracle)**
  - **clustered (DB2)**
  - **multidimensional cluster (DB2)**
  - **indexing dimensions**

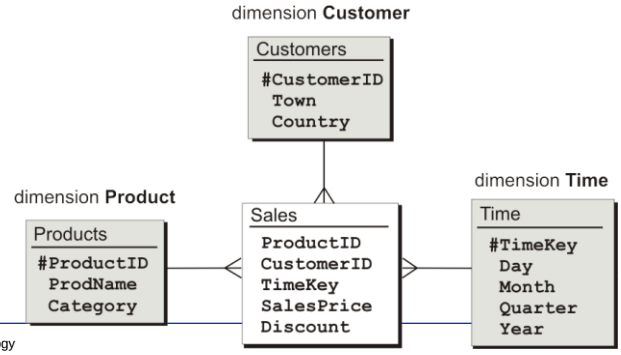
B-tree

- **Foundation for other indexes (join, bitmap, bitmap join, clustering, MDC)**



Star schema and queries

```
select sum(SalesPrice), ProdName, Country, Year
from Sales s, Products p, Customers c, Time t
where s.ProductID=p.ProductID
and s.CustomerID=c.CustomerID
and s.TimeKey=t.TimeKey
and p.Category in ('electronics')
and t.Year in (2009, 2010)
group by ProdName, Country, Year;
```



Join index

- **Materialized join of 2 tables (typically fact and dimension(s))**

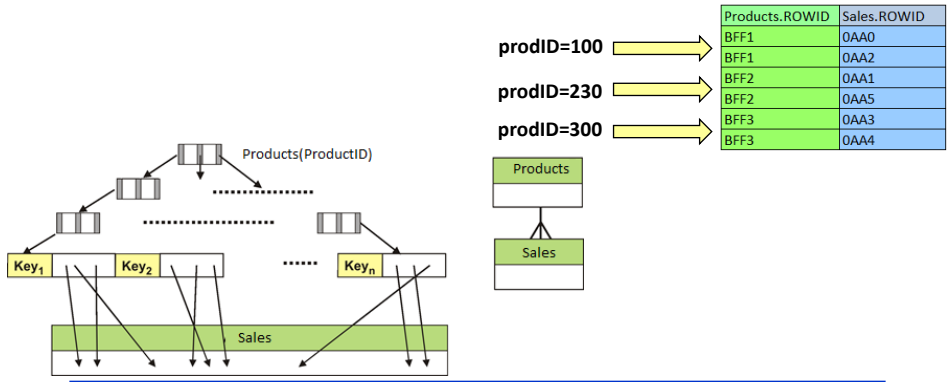
Products			
ROWID	productID	prodName	category
BFF1	100	HP Pavillon	electronics
BFF2	230	Dell Inspiron	electronics
BFF3	300	Acer Ferrari	electronics

Sales			
ROWID	salesPrice	discount	productID
0AA0	...	5	100
0AA1	...	15	230
0AA2	...	5	100
0AA3	...	10	300
0AA4	...	10	300
0AA5	...	15	230

Products.ROWID	Sales.ROWID
BFF1	0AA0
BFF1	0AA2
BFF2	0AA1
BFF2	0AA5
BFF3	0AA3
BFF3	0AA4

Join index

- **In order to make searching the join index faster, the join index is physically ordered (clustered) by one of the attributes**
- **Alternatively, the access to the join index can be organized by means of a B-tree or a hash index**



DW queries

- DW queries typically are not selective
  - select dozens % of rows in a fact table
- B-tree indexes are efficient up to 10% selectivity of a query
- Other types of indexes are needed

Bitmap index - definitions

- Attribute cardinality ⇨ domain size
  - card(make)=4, card(color)=4
- Bitmap ⇨ vector of bits
  - a bit corresponds to a row in a table

B: red
1
0
0
0
0
0
1
1
0
0
1
0

CarSales

make	...	color
Subaru		red
Mercedes		green
Mercedes		blue
Subaru		blue
BMW		blue
BMW		green
BMW		red
Audi		red
Audi		black
BMW		black
Subaru		red
Mercedes		green

Bitmap index - definitions

- **Bitmap index**
  - **collection of bitmaps**
  - **one bitmap for one value from attribute domain**
- **Organizing bitmaps**
  - **2-dimensional table**
  - **B-tree index**
  - **...**

CarSales

make	...	color
Subaru		red
Mercedes		green
Mercedes		blue
Subaru		blue
BMW		blue
BMW		green
BMW		red
Audi		red
Audi		black
BMW		black
Subaru		red
Mercedes		green

B: red	B: green	B: blue	B: black
1	0	0	0
0	1	0	0
0	0	1	0
0	0	1	0
0	0	1	0
0	1	0	0
1	0	0	0
1	0	0	0
0	0	0	1
0	0	0	1
1	0	0	0
0	1	0	0

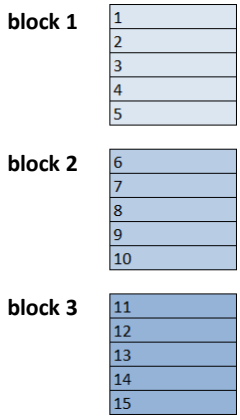
Bitmap index in queries

```
select count(*) from CarSales
where  make in ('Audi', 'Mercedes', 'BMW')
and    type = 'sport'
and    color ='red';
```

B:Audi	OR	B:Mercedes	OR	B: BMW	AND	B:sport	AND	B: red	=	
0		0		0		1		1		0
0		1		0		0		0		0
0		1		0		0		0		0
0		0		0		0		0		0
0		0		1		1		0		0
0		0		1		1		0		0
0		0		1		1		1		1
1		0		0		1		1		1
1		0		0		0		0		0
0		0		1		0		0		0
0		0		0		1		1		0
0		1		0		1		0		0

Mapping bits to ROWIDs

➲ **Easy solution: fixed number of rows per DB block - rpb**



$$pgNo = \left\lceil \frac{bitNo}{rpb} \right\rceil$$

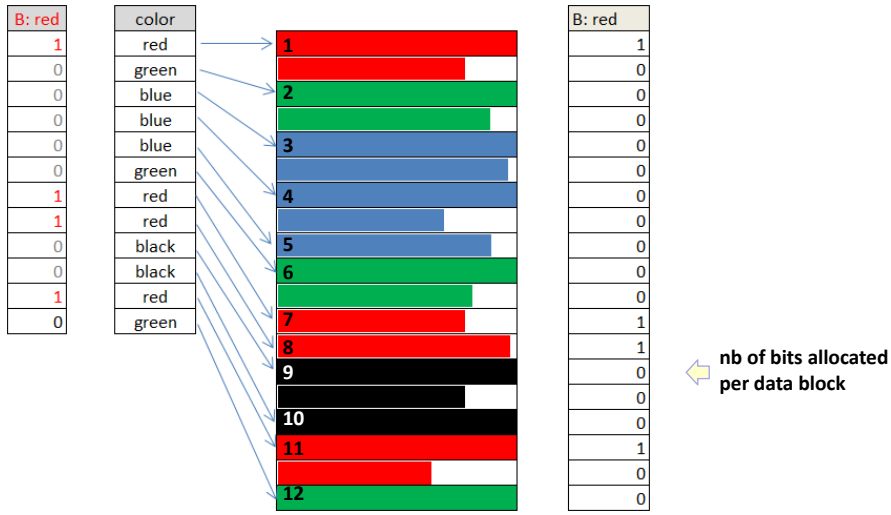
if bitNo MOD rpb !=0
then slotNo = bitNo MOD rpb
else slotNo=bitNo/pgNo

Mapping bits to ROWIDs

➲ **Real approach**

- **estimate the average length L of a row in a table**
- **compute the number of slots in a DB block: ⌊BSize/L⌋**
- **allocate more slots than ⌊BSize/L⌋**
- **real row length differs from L ⇨ some slots are wasted ⇨ a bitmap index is larger than it could be**

Mapping bits to ROWIDs



Mapping bits to ROWIDs: Oracle

```
create table TEST_BI1
(id number(7),
 no number(7),
 txt1 varchar2(20),
 txt2 varchar2(20),
 txt3 varchar2(20),
 txt4 varchar2(20),
 txt5 varchar2(20));
```

```
create table TEST_BI2
(id number(7) not null,
 no number(7) not null,
 txt1 varchar2(20) not null,
 txt2 varchar2(20) not null,
 txt3 varchar2(20) not null,
 txt4 varchar2(20) not null,
 txt5 varchar2(20) not null);
```

```
select o.name, o.obj#, t.spare1
from sys.obj$ o, sys.tab$ t
where o.obj#=t.obj# and o.name in ('TEST_BI1', 'TEST_BI2')

NAME                                 OBJ#     SPARE1
------------------------------ ---------- ----------
TEST_BI1                            73450        736
TEST_BI2                            73452        506
```

max number of rows per data block

Mapping bits to ROWIDs: Oracle

- Populating table TEST_BI1 with 10^6 of rows of lenght 113B
- Computing column statistics

```
create bitmap index NO_BI_INDX on TEST_BI1(NO) pctfree 0;
```

```
select index_name, leaf_blocks
from dba_indexes
where index_name = 'NO_BI_INDX';

INDEX_NAME      LEAF_BLOCKS
--------------- -----------
NO_BI_INDX              350
```


Mapping bits to ROWIDs: Oracle

```
drop index NO_BI_INDX;
```

find the actual maximum number of rows in a data block in TEST_BI1

```
alter table TEST_BI1 minimize records_per_block;
```

```
create bitmap index NO_BI_INDX on TEST_BI1(NO) pctfree 0;
```

```
select index_name, leaf_blocks
from dba_indexes
where index_name = 'NO_BI_INDX';

INDEX_NAME      LEAF_BLOCKS
--------------- -----------
NO_BI_INDX              150
```

BI characteristics (1)

- **Reasonably small size for attributes of low cardinality**
- **Example**
  - **#rows = 1 000 000**
  - **card(A) = 4**
  - **ROWID = 10B (Oracle)**
  - **bitmap index on attribute A**
    - **4 bitmaps: 4 x (1 000 000 / 8) = 4 x 125kB = 500kB**
  - **B$^+$-tree on attribute A**
    - **1 000 000 x 10B = 10MB**

BI characteristics (2)

- **Efficient processing of bitmaps**
  - **logical operations AND, OR, NOT, COUNT**
  - **64 bits processed in one CPU clock cycle**
- **Size**
  - **small index ⇨ small #I/O**
  - **processing in RAM**
- **Not applicable to LIKE**

BI characteristics (3)

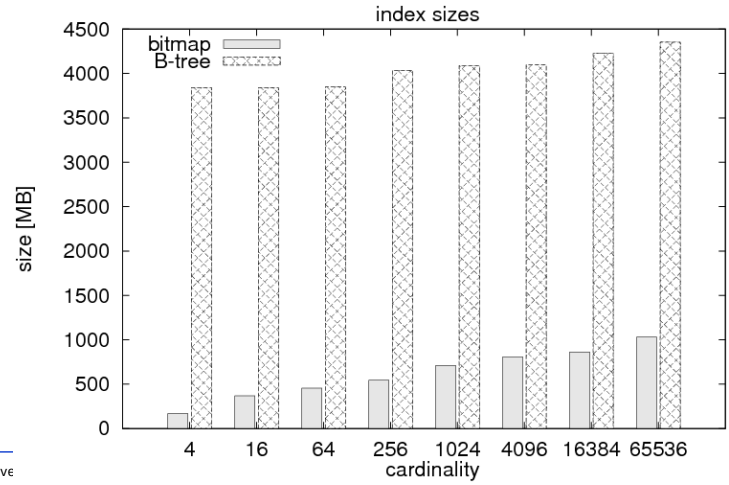
- Large size for attributes of large cardinality
- Example
  - #rows = 1 000 000
  - card(A) = 1024
  - ROWID = 10B (Oracle)
  - bitmap index on attribute A
    - 1024 bitmaps: 1024 x (1 000 000 / 8) = 1024 x 125kB = 128MB
  - B^+-tree on A
    - 1 000 000 x 10B = 10MB

BI characteristics (3)

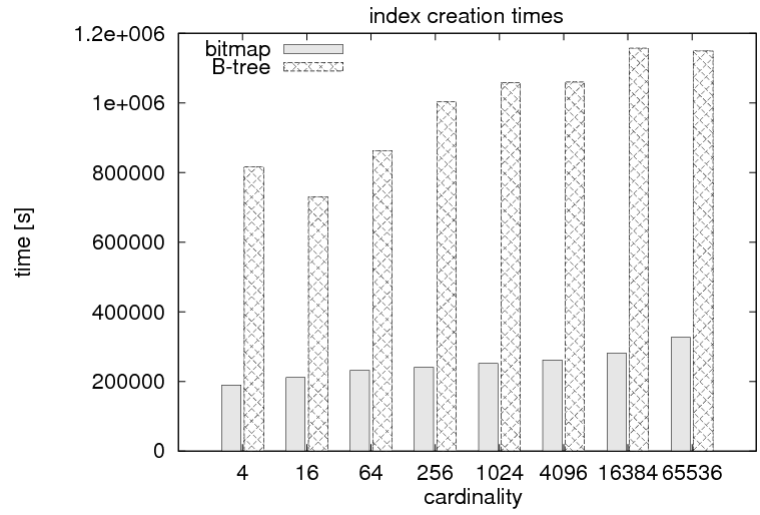
- Index maintenance
  - inserting rows ⇨ increasing length of bitmaps
  - updating rows ⇨ updating 2 bitmaps
  - deleting rows ⇨
    - decreasing length of bitmaps
    - OR bitmap of deleted rows
  - locking contiguous segments of bitmaps ⇨ concurrency decreasing

Experiment (1)

- **Oracle11g**
  - **data cache: 1.7GB, SGA: 3.4GB**
  - **#rows: 250 000 000 (DB size: 10.8GB)**

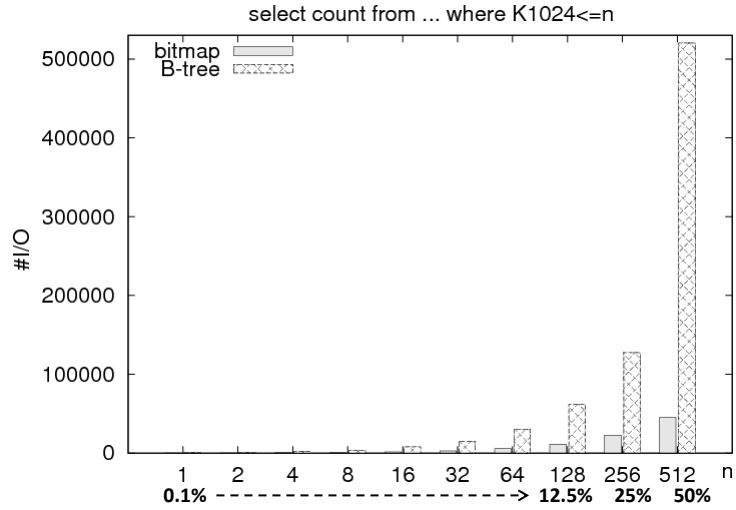


Experiment (2)



Experiment (3)

➲ **cardinality of indexed attribute: 1024**



Experiment (4)

WHERE K1024 <= 128

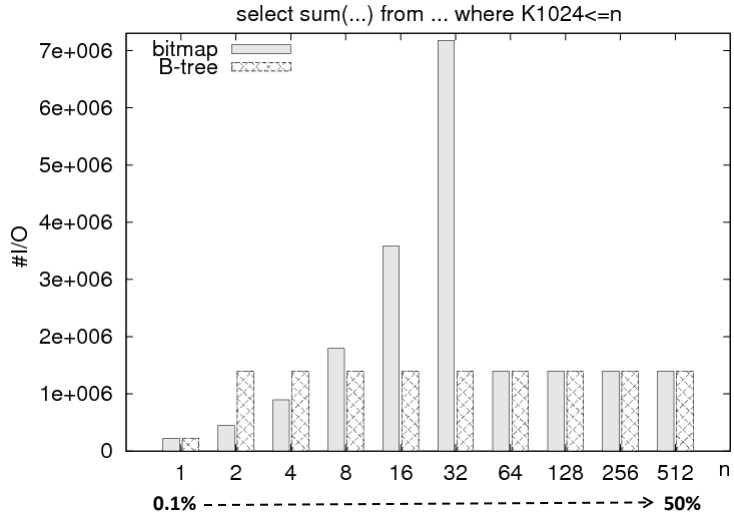
bitmap index

```
-------------------------------------------------------------------------------------------
| Id  | Operation                 | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |             |     1 |     4 | 11315   (1)| 00:02:16 |
|   1 |  SORT AGGREGATE           |             |     1 |     4 |            |          |
|   2 |   BITMAP CONVERSION COUNT |             |    31M|   119M| 11315   (1)| 00:02:16 |
|*  3 |    BITMAP INDEX RANGE SCAN| BMP_5K1024  |       |       |            |          |
-------------------------------------------------------------------------------------------
```

B*-tree index

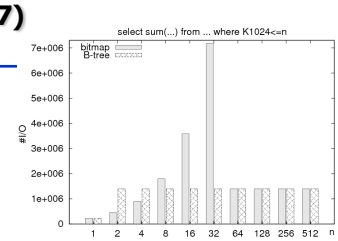
```
-------------------------------------------------------------------------------
| Id  | Operation         | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |             |     1 |     4 | 65441   (1)| 00:13:06 |
|   1 |  SORT AGGREGATE   |             |     1 |     4 |            |          |
|*  2 |   INDEX RANGE SCAN| BMP_5K1024  |    31M|   119M| 65441   (1)| 00:13:06 |
-------------------------------------------------------------------------------
```

Experiment (5)



Experiment (7)

WHERE K1024 <= 4



bitmap index

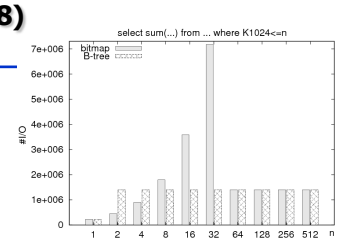
```
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |     1 |    10 |   149K  (1)| 00:29:51 |
|   1 |  SORT AGGREGATE               |             |     1 |    10 |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID | BMTEST      |   977K|  9543K|   149K  (1)| 00:29:51 |
|   3 |    BITMAP CONVERSION TO ROWIDS|             |       |       |            |          |
|*  4 |     BITMAP INDEX RANGE SCAN   | BMP_5K1024  |       |       |            |          |
-------------------------------------------------------------------------------------------
```

B*-tree index

```
-----------------------------------------------------------------------------------
| Id  | Operation          | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |         |     1 |    10 |   387K  (2)| 01:17:26 |
|   1 |  SORT AGGREGATE    |         |     1 |    10 |            |          |
|*  2 |   TABLE ACCESS FULL| BMTEST  |   977K|  9543K|   387K  (2)| 01:17:26 |
-----------------------------------------------------------------------------------
```

Experiment (8)

WHERE K1024 <= 32



bitmap index

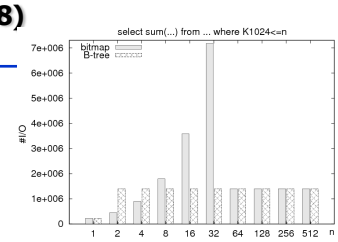
Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	344K (1)	01:08:57
1	SORT AGGREGATE		1	10		
2	TABLE ACCESS BY INDEX ROWID	BMTEST	7819K	74M	344K (1)	01:08:57
3	BITMAP CONVERSION TO ROWIDS					
* 4	BITMAP INDEX RANGE SCAN	BMP_5K1024				

B*-tree index

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	387K (2)	01:17:26
1	SORT AGGREGATE		1	10		
* 2	TABLE ACCESS FULL	BMTEST	7819K	74M	387K (2)	01:17:26

Experiment (8)

WHERE K1024 <= 64



bitmap index

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	387K (2)	01:17:26
1	SORT AGGREGATE		1	10		
* 2	TABLE ACCESS FULL	BMTEST	15M	149M	387K (2)	01:17:26

B*-tree index

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	387K (2)	01:17:26
1	SORT AGGREGATE		1	10		
* 2	TABLE ACCESS FULL	BMTEST	15M	149M	387K (2)	01:17:26

Decreasing size of BI

- Range-based bitmap index
- Encoding
- Compression

Range-based BI (1)

- Domain of indexed attribute is divided into ranges
  - e.g., temperature: <0, 20), <20, 40), <40, 60), <60, 80), <80, 100)

indexed attribute

tempC
21
39.6
51.3
12
98.8
71
68.8
50.4
40

B4	B3	B2	B1	B0
(100, 80>	(80, 60>	(60, 40>	(40, 20>	(20, 0>
0	0	0	1	0
0	0	0	1	0
0	0	1	0	0
0	0	0	0	1
1	0	0	0	0
0	1	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	1	0	0

bitmap No
bitmap range

- query: count records for which 10<=temp<45

Range-based BI (2)

- Bitmaps can represent also sets of values
  - e.g., B1: {yellow, orange, red}, B2: {light blue, blue, navy blue}
- Characteristics
  - the number of bitmaps depends less on the attribute cardinality ⇨ depends on the range/set width
  - border bitmaps may point to rows that do not fulfill selection criteria ⇨ additional row filtering after fetching

Encoding (1)

- Replacing the value of an indexed attribute by another value whose bitmap representation is more compact
- Example
  - card(productName): 50000 ⇨ typical number of products in a supermarket
  - standard bitmap index ⇨ 50000 bitmaps
  - 50000 distinct values can be encoded on 16 bits
    - $\lceil log_2 50000 \rceil = 16$
  - a mapping data structure is required for mapping the encoded values into their real values

Encoding (2)

- query: select * from T where product = 'pecorino d'Abruzzo'
- apply mask: 00...1000 (¬B15 ¬B14 ... B3 ¬B2 ¬B1 ¬B0)

indexed attribute

product
queso Manchengo
queso de Burgos
queso Cerrato
queso Serrat
tupi
queso de Urbasa
pecorino baccellone
pecorino d'Abruzzo
pecorino dei Berici
pecorino di Farindola
pecorino lucano
pecorino rosso
pecorino sardo
pecorino sense
...

B15	B14	...	B3	B2	B1	B0
0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	0	1	1
0	0	0	0	1	0	0
0	0	0	0	1	0	1
0	0	0	0	1	1	0
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	0	1	0	0	1
0	0	0	1	0	1	0
0	0	0	1	0	1	1
0	0	0	1	1	0	0
0	0	0	1	1	0	1
0	0	0	1	1	1	0
0	0	0	1	1	1	1

mapping table

Compression (1)

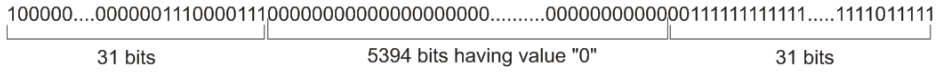
- Byte-aligned Bitmap Compression (BBC)
- Word-Aligned Hybrid (WAH)
- Run Length Huffman
- Based on the run-length encoding
  - homogeneous vectors of bits are replaced with a bit value (0 or 1) and the vector length
  - `0000000 1111111111 000` ⇨ `07 110 03`
- A bitmap is divided into words
  - BBC uses 8-bit words
  - WAH uses 31-bit words
  - RLH uses n-bit words (n - parameter)

Compression (2)

- **WAH-compressed bitmaps are larger than BBC-compressed ones**
- **Operations on WAH-compressed bitmaps are faster than on BBC-compressed ones**
  - Wu, K. and Otoo, E. J. and Shoshani, A.: Compressing Bitmap Indexes for Faster Search Operations, SSBDM, 2002
  - Wu, K. and Otoo, E. J. and Shoshani, A.: On the performance of bitmap indices for high cardinality attributes, 2004, VLDB
- **Types of words in BBC and WAH**
  - **fill word ⇨ represents a compressed segment of a bitmap (composed either of all 0s or all 1s)**
  - **tail word ⇨ represents non-compressable segment of a bitmap (composed of interchanged 0 and 1 bits)**

WAH (1)

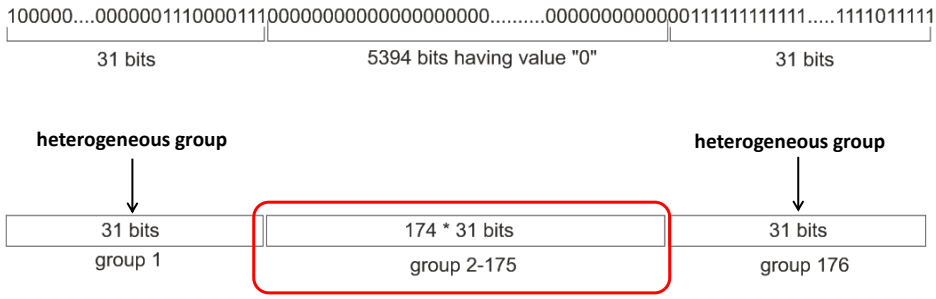
- **Example: 32-bit processor, bitmap composed of 5456 bits**
  - **taken from** Stockinger K., Wu K.: Bitmap Indices for Data Warehouses. In Wrembel R. and Koncilia C. (eds.): Data Warehouses and OLAP: Concepts, Architectures and Solutions. IGI Global, 2007



- **Step 1: divide the bitmap into groups including 31 bits each**

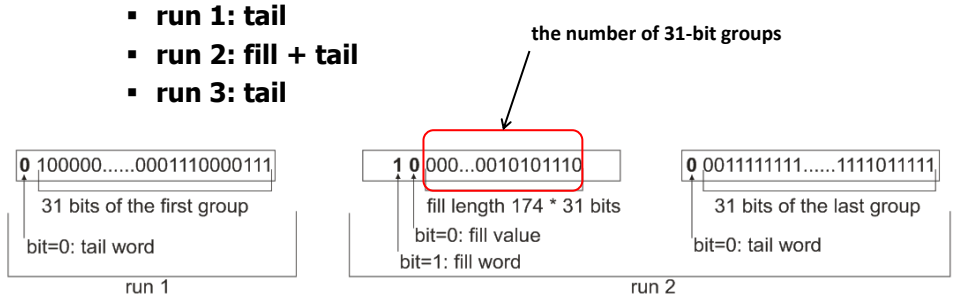
WAH (2)

- **Step 2: merge adjacent homogeneous groups (having the same values of all bits, i.e., groups 2-175)**



WAH (3)

- **Step 3: group encoding**
  - **run: fill + tail**
  - **run 1: tail**
  - **run 2: fill + tail**
  - **run 3: tail**



WAH (4)

- Unsorted data
- For low cardinality attributes bitmaps are dense
  - many homogeneous 31-bit words filled with 1
- For high cardinality attributes bitmaps are sparse
  - many homogeneous 31-bit words filled with 0
- For medium cardinality attributes
  - the number of homogeneous 31-bit words is lower

RLH

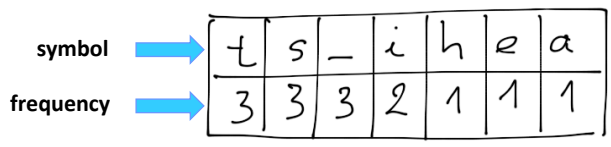
- **RLH - the Run-Length Huffman Compression** (M. Stabno and R. Wrembel. Information Systems, 34(4-5), 2009)
- Based on
  - the Huffman encoding
  - a modified run-length encoding

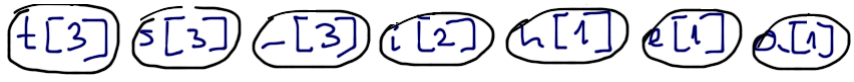
Huffman Encoding (1)

- Concept
  - original symbols from a compressed file are replaced with bit strings
  - the more frequently a given symbol appears in the compressed file the shorter bit string for representing the symbol
  - encoded symbols and their corresponding bit strings are represented as a Huffman tree
  - the Huffman tree is used for both compressing and decompressing

Huffman Encoding (2)

- Example: encoding text "this_is_a_test"
- Step 1: frequencies of the symbols in the encoded string

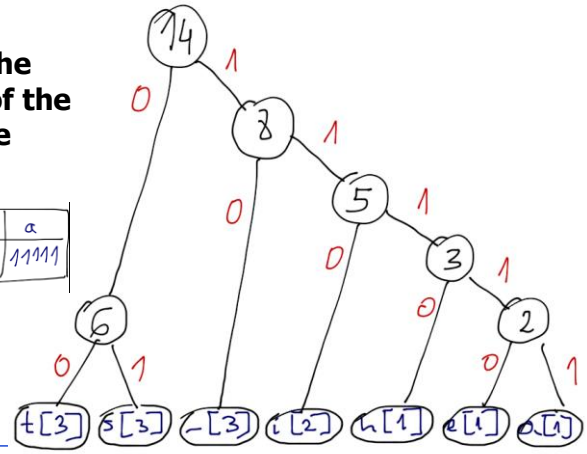




Huffman Encoding (3)

- Step 2: building Huffman tree
  - start with nodes of the lowest frequency
- Step 3: getting the Huffman codes of the symbols from the tree

t	s	_	i	h	e	a
00	01	10	110	1110	11110	11111



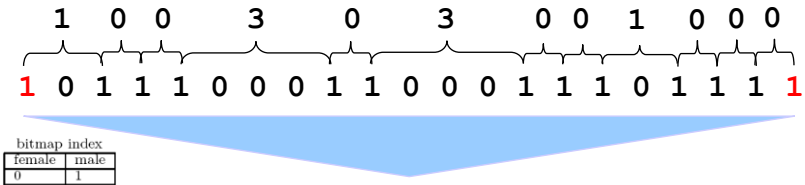
Huffman Encoding (4)

- Step 4: replacing original symbols with their Huffman codes
  - orignal text: 14B
  - compressed text: 37b ⇨ 5B

t	h	i	s	_	i	s	_	a	_	t	e	s	t
00	1110	110	01	10	110	01	10	11111	10	00	11110	01	00

RLH (1)

- Modified run-length encoding
  - encodes distances between bits of value 1



Clients

ID	sex
1	male
2	female
3	female
4	female
5	male
6	male
7	male
8	female
9	female
10	male
11	male
12	male
13	female
14	female
15	female
16	male
17	female
18	female
19	female

bitmap index

female	male
0	1
1	0
1	0
1	0
0	1
0	1
0	1
1	0
1	0
0	1
0	1
0	1
1	0
1	0
1	0
0	1
1	0
1	0
1	0

```
female: 100303001000

male:   030020033
```


RLH (2)

- Huffman encoding
  - step1: computing frequencies of symbols (distances) in encoded bitmaps

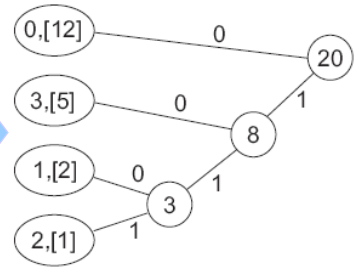
```
female: 100303001000
male:   030020033
```

distance	frequency
0	12
3	5
1	2
2	1

RLH (3)

- Huffman encoding
  - step2: building a Huffman tree

distance	frequency
0	12
3	5
1	2
2	1



distance	code
0	0
3	10
1	110
2	111

  - an encoded symbol is represented by a path from the root to a leaf

RLH (4)

- Huffman encoding
  - step3: replacing distances with their Huffman codes

original bitmap gender='female'

0 1 1 1 0 0 0 1 1 0 0 0 1 1 1 0 1 1 1

distance	code
0	0
3	10
1	110
2	111

run-length encoded bitmap gender='female'

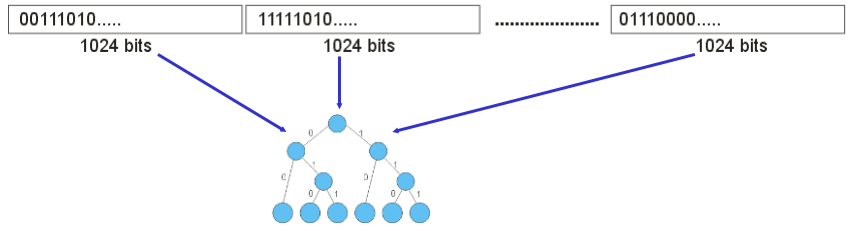
1 0 0 3 0 3 0 0 1 0 0

110 0 0 10 0 10 0 0 110 0 0

RLH-compressed bitmap gender='female'

RLH-N

- **Dividing a bitmap into N-bit sections**
  - **constructing one Huffman tree based on frequencies of distances from all N-bit sections**



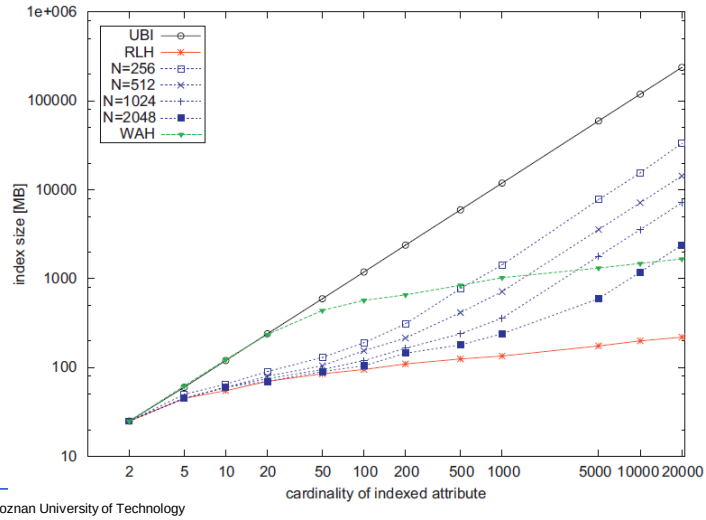
- **Including in the HT all possible distances that may appear in a N-bit section**
  - **non-existing distances have assigned the frequency of 1**

Experimental Evaluation

- **Comparing RLH, WAH, and uncompressed bitmaps (UBI) with respect to**
  - **bitmap sizes**
  - **query response times**
- **Implementation in Java**
  - **data and bitmap indexes stored on disk in OS files**
- **Experiments run on**
  - **PC, AMD Athlon XP 2500+; 768 MB RAM; Windows XP**
- **Data**
  - **100 000 000 indexed rows**
  - **indexed attribute of type integer**
    - **cardinality from 2 to 20 000**
    - **randomly distributed values**

WAH and RLH: index sizes

- RLH, RLH-N, WAH, and UBI with respect to the size of a bitmap index (N = {256, 512, 1024, 2048} for RLH-N)

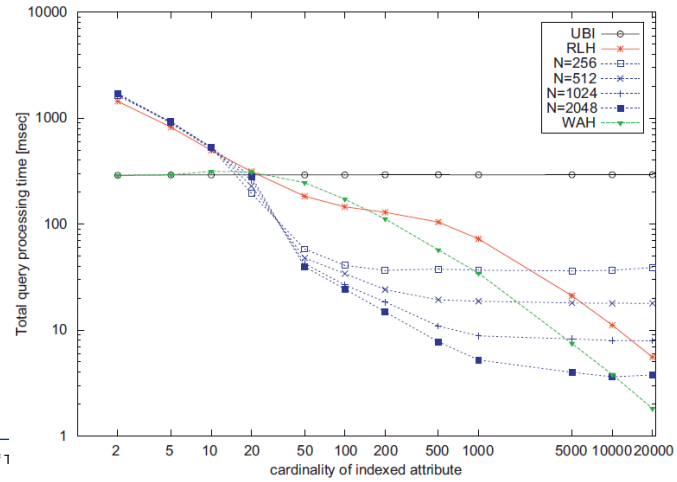


WAH and RLH: response times

- Query:

```
select ... from ...
where ind_attribute in (v1, v2, ..., v100)
```

- Randomly ordered rows wrt. the value of the indexed attribute

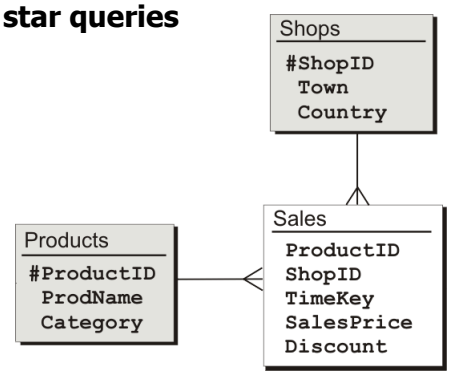


Updating RLH Bitmaps

- Costly process
  - decompressing the whole bitmap
  - modifying the bitmap
  - compressing the bitmap
  - changes frequencies of distances between 1 bits
  - creates new distances between 1 bits
- In a DW environment index structures
  - are dropped before loading a DW
  - are recreated after loading is finished

BIs in Oracle

- Defined explicitly by DBA
- Compressed automaticaly
- Bitmap join index available
- Used for optimizing star queries



Bitmap Join Index (1)

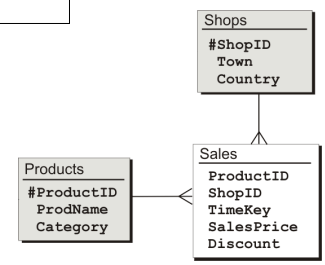
Products

ProductID	ProdName	...
100	queso Manchengo	
200	queso de Burgos	
300	queso Cerrato	
400	queso de Urbasa	
500	pecorino baccellone	

Sales

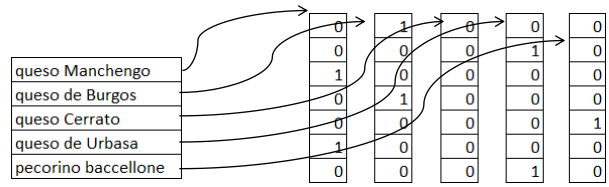
ProductID	SalesPrice	...
200	45	
400	50	
100	40	
200	55	
500	75	
100	65	
400	70	

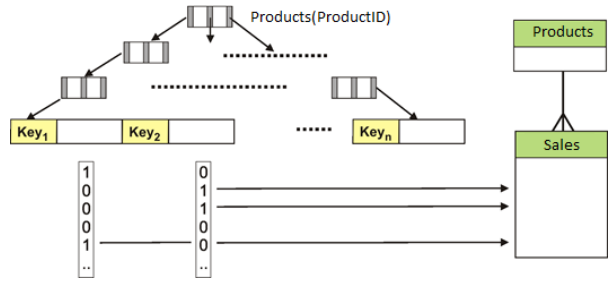
```
create bitmap index Sales_JBI
on Sales(Products.ProdName)
from Sales s, Products p
where s.ProductID=p.ProductID;
```



BJI (2)

bitmap join index on Products.ProdName





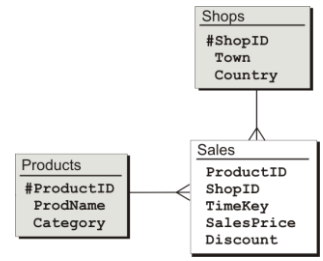
BJI (3)

➲ **Star query optimization with the suport of BJI**

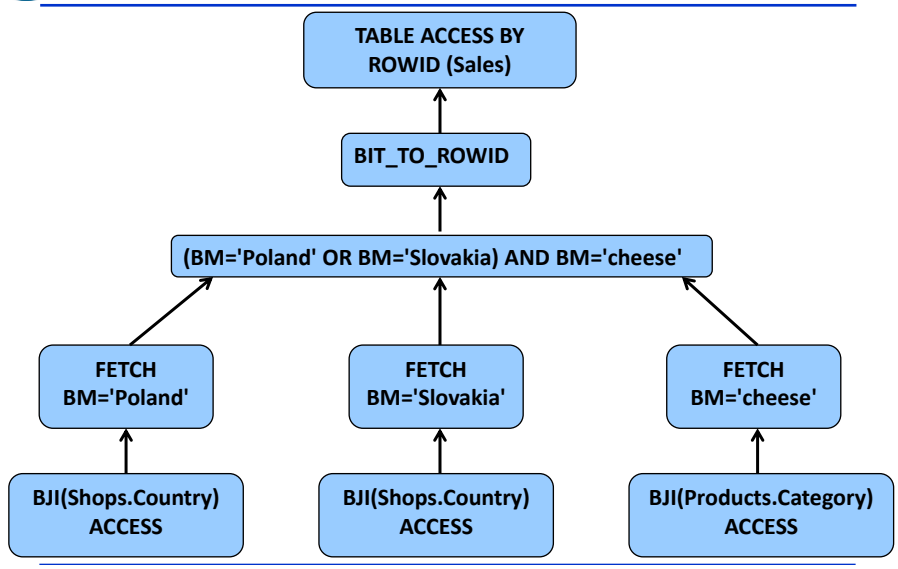
```
select sum(sa.SalesPrice), p.ProdName, sh.ShopID
from Sales sa, Shops sh, Products p
where sh.country in ('Poland', 'Slovakia')
and p.Category='cheese'
and sa.ShopID=sh.ShopID
and sa.ProductID=p.ProductID
group by p.ProdName, sh.ShopID;
```

➲ **BJIs defined on attributes**

- Shops.Country
- Products.Category

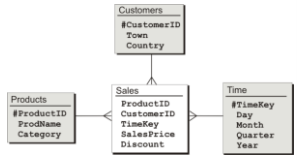


BJI (4)



BJI (5)

The Oracle case




```
select sum(SalesPrice)
from Sales, Products, Customers, Time
where Sales.ProductID=Products.ProductID
and Sales.CustomerID=Customers.CustomerID
and Sales.TimeKey=Time.TimeKey
and ProdName in
    ('ThinkPad Edge', 'Sony Vaio', 'Dell Vostro')
and Town='London'
and Year=2009;
```

```
create bitmap index BI_Pr_Sales
on Sales(Products.ProdName)
from Sales s, Products p
where s.ProductID=p.ProductID;

create bitmap index BI_Cu_Sales
on Sales(Customers.Town)
from Sales s, Customers c
where s.CustomerID=c.CustomerID;
```

```
create bitmap index BI_Ti_Sales
on Sales(Time.Year)
from Sales s, Time t
where s.TimeKey=t.TimeKey;
```


BJI (6)

```
-------------------------------------------------------------------------------------
|Id  | Operation                      |Name         |Rows |Bytes |Cost(%CPU)|Time    |
-------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT                |             |    1|    58 |   13  (8)|00:00:01|
| 1 |  SORT AGGREGATE                 |             |    1|    58 |          |        |
| 2 |   NESTED LOOPS                  |             |   21| 1218 |   13  (8)|00:00:01|
| 3 |    HASH JOIN                    |             |   22| 1012 |   12  (9)|00:00:01|
| 4 |     TABLE ACCESS FULL           |PRODUCTS     |    3|    51 |    3  (0)|00:00:01|
| 5 |     TABLE ACCESS BY INDEX ROWID |SALES        | 1155|33495 |    8  (0)|00:00:01|
| 6 |      BITMAP CONVERSION TO ROWIDS|             |     |       |          |        |
| 7 |       BITMAP AND                |             |     |       |          |        |
| 8 |        BITMAP INDEX SINGLE VALUE|BI_CU_SALES  |     |       |          |        |
| 9 |       BITMAP OR                 |             |     |       |          |        |
|10 |         BITMAP INDEX SINGLE VALUE|BI_PR_SALES  |     |       |          |        |
|11 |         BITMAP INDEX SINGLE VALUE|BI_PR_SALES  |     |       |          |        |
|12 |         BITMAP INDEX SINGLE VALUE|BI_PR_SALES  |     |       |          |        |
|13 |    TABLE ACCESS BY INDEX ROWID  |TIME         |    1|    12 |    1  (0)|00:00:01|
|14 |     INDEX UNIQUE SCAN           |PK_TIME      |    1|       |    0  (0)|00:00:01|
-------------------------------------------------------------------------------------
```

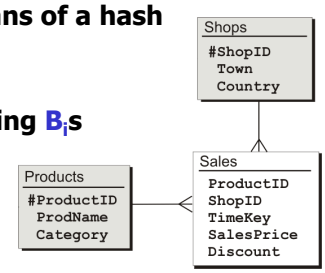
BJI (7)

```
create bitmap index BI_Pr_Cu_Ti_Sales
on Sales(Products.ProdName, Customers.Town, Time.Year)
from Sales, Products, Customers, Time
where Sales.ProductID=Products.ProductID
and Sales.CustomerID=Customers.CustomerID
and Sales.TimeKey=Time.TimeKey;
```

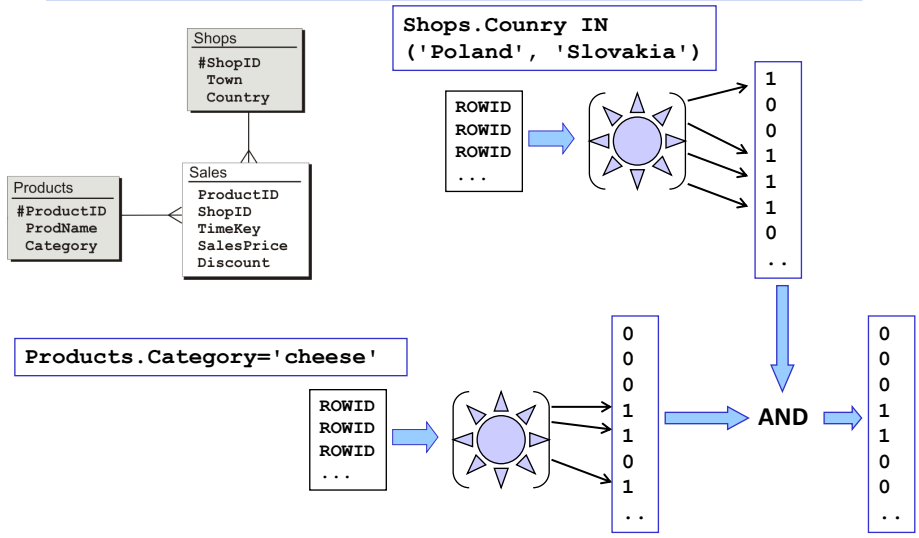
```
-------------------------------------------------------------------------------------------
|Id | Operation                      |Name             |Rows |Bytes|Cost(%CPU)|Time     |
-------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT               |                 |    1|   29|     7  (0)|00:00:01 |
| 1 |  SORT AGGREGATE                |                 |    1|   29|           |         |
| 2 |   INLIST ITERATOR              |                 |     |     |           |         |
| 3 |    TABLE ACCESS BY INDEX ROWID |SALES            |   22|  638|     7  (0)|00:00:01 |
| 4 |     BITMAP CONVERSION TO ROWIDS|                 |     |     |           |         |
| 5 |      BITMAP INDEX SINGLE VALUE |BI_PR_CU_TI_SALES|     |     |           |         |
-------------------------------------------------------------------------------------------
```


BIs in DB2 (1)

- Created and managed implicitly by the system
- Applied to join optimization
  - Every dim table is independently semi-joined with a fact table
  - The semi-joins use B-trees on foreign keys
  - ROWIDs of every semi-join result are transformed into a separate bitmap
  - Bitmaps B_i are constructed by means of a hash function on ROWID
    - the hash value points to a bit in B_i
  - Final bitmap is computed by AND-ing B_is



BIs in DB2 (2)

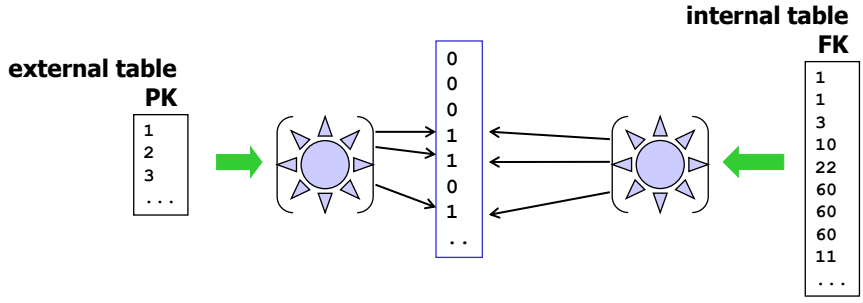


BIs in SQL Server (1)

- Created and managed implicitly by the system
- Applied to join optimization
  - join of a dim table with a fact table by means of hash join
  - table with a PK (dim table) ⇨ external table
  - table with a FK (fact table) ⇨ internal table

BIs in SQL Server (2)

- Hashing PK values into a bitmap
  - HashFunction(PK) → bit no of value 1
- Hashing FK values into a bitmap
  - HashFunction(PK) → bit no of value 1
- The rows from both tables that hash to the same bit ⇨ join result

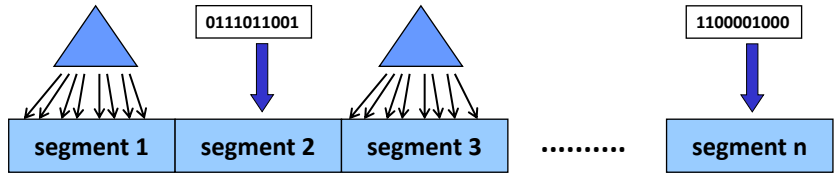


BIs in SybaseIQ

- Defined explicitly by DBA
- Low Fast ⇨ for attributes of low cardinalities
  - max cardinality: 10 000
  - the highest performance for cardinality up to 1000
- High Non Group ⇨ for attributes of high cardinalities
  - for aggregate queries with range predicates

BIs in SAS

- SAS Scalable Performance Data (SPD) Server
  - hybrid index
  - table is divided into segments (e.g., 8192 rows)
  - every segment is indexed independently by
    - bitmap index or
    - B-tree
  - index type selected automatically by the system taking into account the distribution of values in a segment



DB2: Clustering index (1)

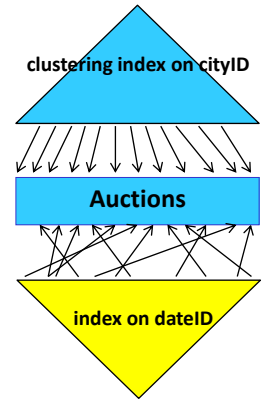
- Clustering index determines how rows are physically ordered (clustered) on disk
- After defining the index, rows are inserted in the order determined by the index
- Only one index can be a clustering index (one physical order of rows on disk)
- By default the first index created is the clustering one (unless you explicitly define another index to be the clustering index)

DB2: Clustering index (2)

```
CREATE INDEX cityID_Indx ON Auctions(cityID) CLUSTER
```

cityID	dateID	quantity	...
POZ	03-07-2013	10	
WRO	03-07-2013	13	
WAW	03-07-2013	7	
WRO	05-07-2013	8	
POZ	05-07-2013	9	
WRO	05-07-2013	21	
WRO	07-07-2013	14	
GDA	08-07-2013	17	
POZ	08-07-2013	2	
GDA	08-07-2013	7	
POZ	08-07-2013	19	
GDA	10-07-2013	17	
WRO	10-07-2013	22	
GDA	10-07-2013	11	
WRO	10-07-2013	4	

cityID	dateID	quantity	...
GDA	08-07-2013	17	
GDA	08-07-2013	7	
GDA	10-07-2013	17	
GDA	10-07-2013	11	
POZ	03-07-2013	10	
POZ	05-07-2013	9	
POZ	08-07-2013	2	
POZ	08-07-2013	19	
WAW	03-07-2013	7	
WRO	03-07-2013	13	
WRO	05-07-2013	8	
WRO	05-07-2013	21	
WRO	07-07-2013	14	
WRO	10-07-2013	22	
WRO	10-07-2013	4	



DB2: Clustering index (3)

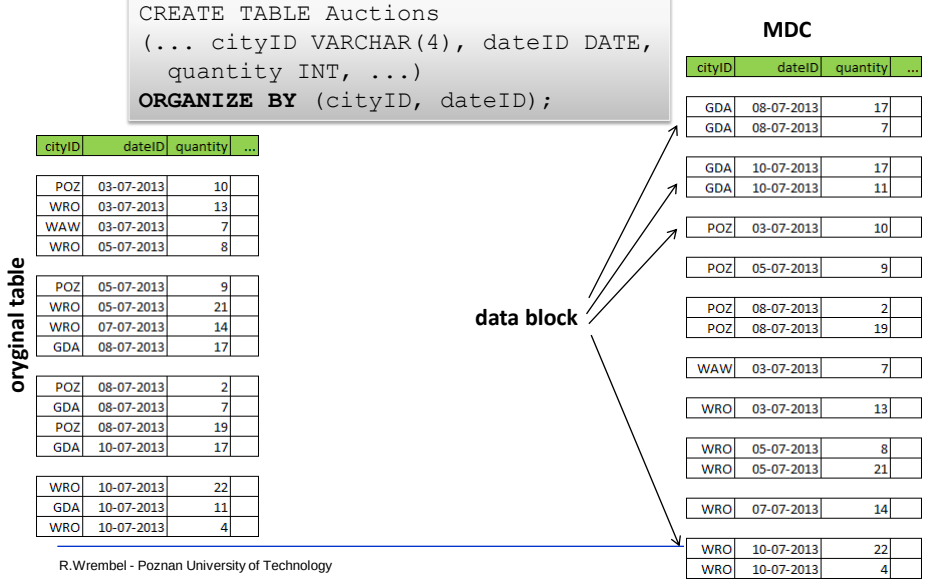
- Eliminates sorting
- Operations that benefit from clustering indexes include:
  - grouping
  - ordering
  - comparisons other than equal
  - distinct

DB2: MDC (1)

- **MultiDimensional Cluster - MDC**
  - **groups data based on values of multiple dimension attributes**
  - **a physical region (block) is associated with each unique combination of dimension attribute values**
  - **a block stores records with the same values of dimension attributes**
- **Block Map: a structure that stores information about block states** (in use, free, loaded, ...)

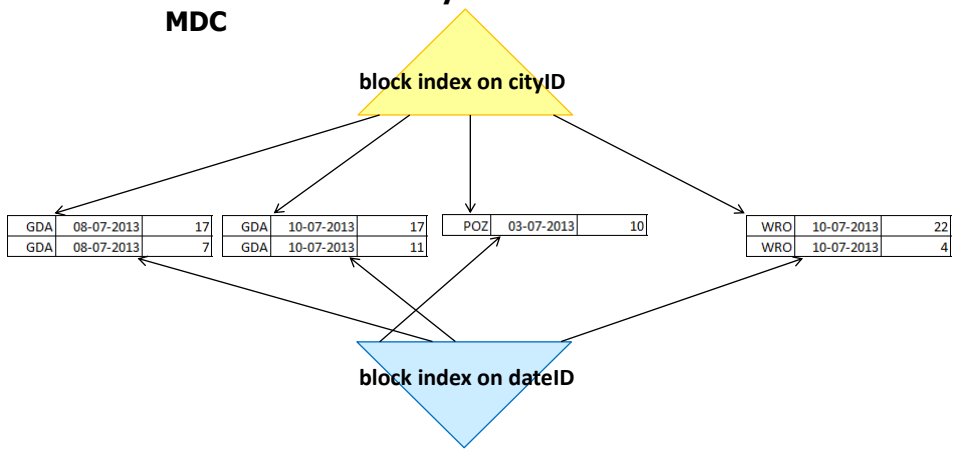
DB2: MDC (2)

```
CREATE TABLE Auctions
(... cityID VARCHAR(4), dateID DATE,
  quantity INT, ...)
ORGANIZE BY (cityID, dateID);
```



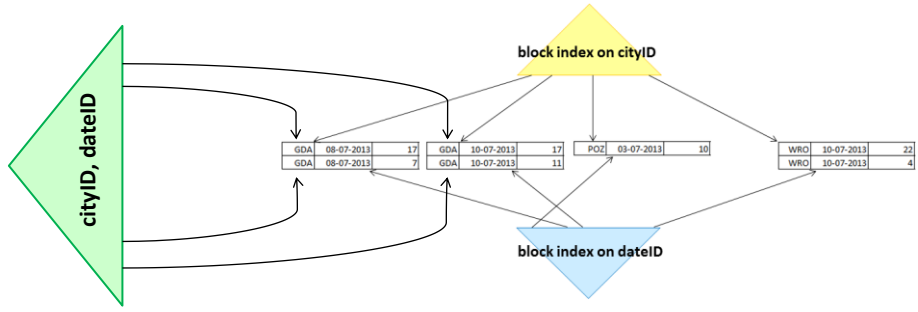
DB2: MDC (3)

- Block index: B-tree based, points to blocks
  - created automatically for each of the dimensions in MDC



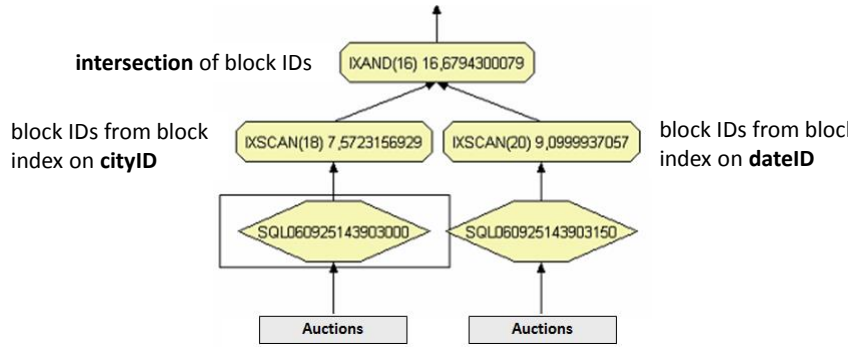
DB2: MDC (4)

- Composite block index: includes all dimension key columns
  - used for insert, update, delete



MDC in queries

```
SELECT SUM(quantity), cityID, dateID
FROM Auctions
WHERE cityID ='GDA' AND dateID='10-07-2013'
group by cityID, dateID;
```



MDC in queries

```
SELECT SUM(quantity), cityID, dateID
FROM Auctions
WHERE cityID ='GDA' OR dateID='10-07-2013'
group by cityID, dateID
```

➲ **{block IDs with cityID='GDA'} UNION {block IDs with dateID='10-07-2013'}**

MDC

- Candidates as dimensions in MDC
  - attributes used in predicates: range, =, IN
  - dimension foreign keys in fact table
  - attributes used in GROUP BY
  - attributes used in ORDER BY

- Summary
  - Data ordered on disk ⇨ less I/O
  - Block index points to a data block ⇨ inserting, updating, deleting may not affect the index structure

MDC case study

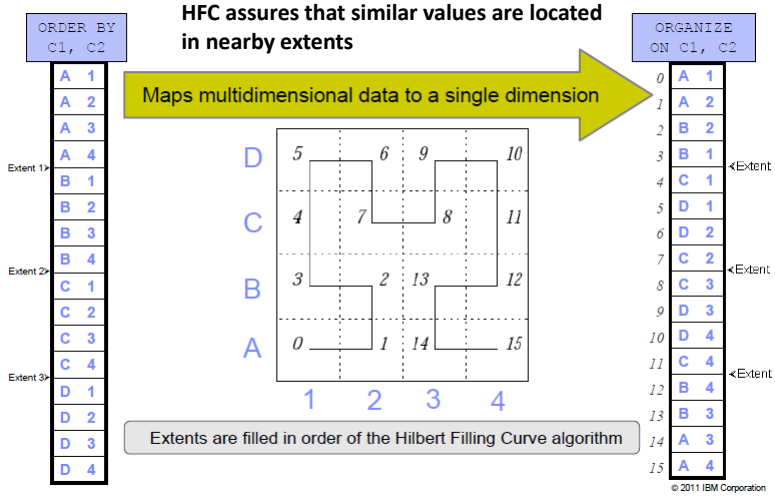
- Source: IBM presentation
- Mobile network operator in USA
- Characteristic
  - 10 billion transactions daily
  - 32 TB raw data
  - thousands concurrent users
  - up to 37000 queries daily
  - DW loading: over 1 billion rows daily (max 1.6 billion)
  - DB2 DWE, 16 x 8 CPU P5 pSeries
- MDC
  - deletion faster by 80% of time
  - I/O lower by 43%

Clustered Based Table (Netezza)

- Clustered Base Table (CBT) ⇨ data are organized by 1 to 4 attributes (organizing keys)
- Data stored in extents (zones)
  - an extent is the smallest unit of disk allocation = 3MB
- Organizing keys are used to group records within the table (store them in one or more nearby extents)
- Netezza creates zone maps for the organizing keys
- Materialized views cannot be build on CBTs

```
CREATE TABLE tab-name
(...)
[ORGANIZE ON (org-key1, ...)]
```


CBT



- Based on IBM teaching materials

References

- **Various types of indexes**
  - J. Dyke: Bitmap Index Internals. juliandyke.com
  - N. Koudas. Space efficient bitmap indexing. CIKM, 2000
  - P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. SIGMOD Record, 1995
  - P. O'Neil and D. Quass. Improved query performance with variant indexes. SIGMOD, 1997
  - R. Bouchakri, L. Bellatreche, K.W. Hidouci: Static and Incremental Selection of Multi-table Indexes for Very Large Join Queries. ADBIS, 2012
  - R. Bouchakri, L. Bellatreche: On Simplifying Integrated Physical Database Design. ADBIS, 2011
  - T. Morzy, R. Wrembel, J. Chmiel, A. Wojciechowski: Time-HOBI: Index for optimizing star queries. Information Systems, 37:(5), 2012

References

- **Indexes in DB2**
  - S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, M. Huras: Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. SIGMOD, 2003
  - P. Bates, P. Zikopoulos: Multidimensional Clustering (MDC) Tables in DB2 LUW. Talk, DB2 Night Show, Jan 2011
  - IBM Netezza System Administrator's Guide. IBM Netezza 7.0 and Later, Oct 2012
  - Clustering indexes. DB2 technical doc http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=%2Fcom.ibm.db2z10.doc.intro%2Fsrc%2Ftpc%2Fdb2z_clusteringindexes.htm

References

- Binning
  - Koudas N.: Space Efficient Bitmap Indexing. CIKM, 2000
  - Stockinger K., Wu K., Shoshani A.: Evaluation Strategies for Bitmap Indices with Binning. DEXA, 2004
  - Rotem D., Stockinger K., Wu K.: Optimizing Candidate Check Costs for Bitmap Indices. CIKM, 2005
- Encoding
  - Wu M., Buchmann A.P.: Encoded Bitmap Indexing for Data Warehouses. ICDE, 1998
  - Chan C.Y., Ioannidis Y.E.: An Efficient Bitmap Encoding Scheme for Selection Queries. SIGMOD, 1999
- Compressing
  - BBC
    - Antoshenkov G., Ziauddin M.: Query Processing and Optimization in ORACLE RDB. VLDB Journal, 1996

References

- Compressing
  - WAH
    - Stockinger K., Wu K., Shoshani A.: Strategies for Processing ad hoc Queries on Large Data Sets. DOLAP, 2002
    - Wu K., Otoo E.J., Shoshani A. (2004): On the Performance of Bitmap Indices for High Cardinality Attributes. VLDB, 2004
    - Stockinger K., Wu K.: Bitmap Indices for Data Warehouses. In Wrembel R. and Koncilia C. (eds.): Data Warehouses and OLAP: Concepts, Architectures and Solutions. IGI Global, 2007
  - PL-WAH
    - F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. EDBT, 2010
  - Approximate Compression with Bloom Filters
    - Apaydin T., Canahuate G., Ferhatosmanoglu H., Tosun, A. S.: Approximate encoding for direct access and query processing over compressed bitmaps. VLDB, 2006

References

- **Compressing**
  - **Reordering**
    - **Johnson D., Krishnan S., Chhugani J., Kumar S., Venkatasubramanian S.: Compressing Large Boolean Matrices Using Reordering Techniques. VLDB, 2004**
    - **Pinar A., Tao T., Ferhatosmanoglu H.: Compressing Bitmap Indices by Data Reorganization. ICDE, 2005**
  - **WAH vs. BBC**
    - **Stockinger K., Wu K., Shoshani A.: Strategies for processing ad hoc queries on large data warehouses. DOLAP, 2002**
    - **Wu K., Otoo E.J., Shoshani A.: Compressing bitmap indexes for faster search operations. SSDBM, 2002**
    - **Wu K., Otoo E.J., Shoshani A.: On the Performance of Bitmap Indices for High Cardinality Attributes. VLDB, 2004**
  - **RLH**
    - **M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on runlength and Hufman encoding. Information Systems, 34(4-5), 2009**