Advanced Data Management Technologies

Unit 14 — Bitmap Indexes

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Acknowledgements: I am indebted to M. Böhlen for providing me the lecture notes.

Outline

Outline

Indexing

- Index used in combination with pre-aggregates to improve performance
  - Index can be on dimension tables and on materialized views.
- Fact table
  - Build primary B-tree index on dim keys (primary key)?
  - Build indexes on each dimension key separately (index intersection)?
  - Indexes on combinations of dimension keys? (many!)
- Sort order is important (index-organized tables)
  - Compressing data can be possible (values not repeated).
  - Can save aggregates due to fast sequential scan.
  - Best sort order (almost) always time!
- Dimension tables
  - Build indexes on many/all individual columns.
  - Build indexes on common combinations.
- Hash indexes
  - Efficient for un-sorted data.

Bitmap Indexes/1

- A B-tree index stores a list of RowIDs for each value.
  - A RowID takes $\approx$ 8 bytes
  - Large space use for columns with low cardinality (gender, color).
    - e.g., index for 1 Bio. rows with gender takes 8 GB!
  - Not efficient to do "index intersection" for these columns.

Bitmap Indexes/2

- **Bitmap index**: make a "position bitmap" for each value of the column for which the index is created.
- **Example:** Bitmap index for gender

Female:	01110010101010...
Male:	10001101010101...

- Takes only (number of values)*(number of rows) * 1 bit
  - Bitmap index on gender with 1 bio. rows takes only 256 MB
- Very efficient to do "index intersection" (AND/OR) on bitmaps.
- Can be improved for higher cardinality using compression techniques.
- Supported by some RDBMSs, e.g, DB2, Oracle.

Using Bitmap Indexes

- Query: Find male customers in South Tyrol with blond hair and blue eyes

Male:	01010101010	
South Tyrol:	00000011111	
Blond	10110110110	
Blue	01101101111	
Result (AND)	00000000010	(only one such customer)

- Range queries can also be handled
  - Bitmap vector for ranges of values.
  - Used as regular bitmaps.
- Query: . . . and Salary BETWEEN 200,000 AND 300,000

200-250,000:	001001001
250-300,000:	010010010
OR together:	011011011

Compressed Bitmaps – Run-length Encoding

- Space use might be a problem of bitmaps
  - With m possible values and n records, $n \cdot m$ bits are required.
  - However, the probability of a 1 is $1/m \Rightarrow$ very few 1's in each vector.
- Compress bitmaps using **run-length encoding**
  - A run is composed of i 0's followed by a 1.
  - Determine
    - the binary representation of i and
    - the number j of bits in the binary representation of i.
  - If $j > 1$, the first bit of i is 1 and can be saved in the binary representation.
  - Run encoding: "⟨j-1 1's⟩" + "0" + "⟨bit 2. . . j of i in binary⟩"
    - 0 is a delimiter bit.
  - Encode next run similarly, trailing 0's not encoded.
  - Special encoding of runs of length 0 and 1.
- Concatenating length of run as binary numbers i won't work, since decoding is not unique.

Run-length Encoding Example

- Encoding of single runs

Run		Run length i	# bits j	Encoding
0 0's:	1	$i = 0 = (0)$	$j = 1$	00
1 0's:	01	$i = 1 = (1)$	$j = 1$	01
2 0's:	001	$i = 2 = (1)0$	$j = 2$	100
3 0's:	0001	$i = 3 = (1)1$	$j = 2$	101
4 0's:	00001	$i = 4 = (1)00$	$j = 3$	11000

- Bitmap 000000010000 is encoded as 11011
  - 1110111 without saving the first bit.

Decoding Compressed Bitmaps

- **Decoding**
  - Scan bits to find j: count 1s till first delimiter 0 and add 1;
  - Scan next $j-1$ bits to find i binary: add leading '1' to $j-1$ bits
  - Find next delimiter 0, etc.
  - Add trailing 0's.
- **Example:** Bitmap encoding: 11011; bitmap length $= 12$
  - $j = 2 + 1 = 3$
  - $i = 7$ (11 + leading 1 $\rightarrow$ 111)
  - Add trailing 0's $\Rightarrow$ bitmap $=$ 00000001 + 0000

Encoding/Decoding Bitmaps Example

- Bitmap `0000001 01 1 00001 000...0` (n=40)
- Encode:
  - `0000001` ⇒ `11010` ($i = 6 =$ `'110'`, $j = 3$)
  - `01` ⇒ `01` ($i = 1 =$ `'1'`, $j = 1$)
  - `1` ⇒ `00` ($i = 0 =$ `'0'`, $j = 1$)
  - `00001` ⇒ `11000` ($i = 4 =$ `'100'`, $j = 3$)
  - `Final encoding:  11010010011000`
- Decode:
  - `11010` ⇒ `0000001` ($j = 3$, $i = 6 =$ `'(1)10'`)
  - `01` ⇒ `01` ($j = 1$, $i = 1 =$ `'1'`)
  - `00` ⇒ `1` ($j = 1$, $i = 0 =$ `'0'`)
  - `11000` ⇒ `00001` ($j = 3$, $i = 4 =$ `'(1)00'`)
  - Fill up remaining 0's
  - Final bitmap: `0000001 01 1 00001 000...0`

Managing Bitmaps

- Compression factor
  - Assume $m = n$ (i.e., unique values)
  - Each value has just one run of length $i < n$
  - Each run takes at most $2 \log_2 n$ bits ($j \leq \log_2 n$)
  - Total space consumption: $2n \log_2 n$ bits (compared to n^2)
- Operations on compressed bitmaps
  - Decompress one run at a time and produce relevant 1's in output.
- Storing bit vectors
  - Index with B-trees + store in blocks/block chains
- Handling modifications
  - Deletion: "retire" record number + update bitmaps with 1's
  - Insertion: add new record to file + update bitmaps with 1's (trail 0's)
  - Updates: update bitmaps with old and new 1's

Outline

Bit-Sliced Index/1

- A **bit-sliced index** for a numeric attribute C of a relation R consists of a bit matrix B with n columns $B_0, \ldots, B_{n-1}$ and as many rows as tuples in R.
  - Row i represents the binary representation of the C-value of tuple i.
  - n is the number of bits needed by the binary representation of the maximum value of C, i.e., $\log_2 MaxVal$.
- Each column (slice) is stored separately.

- **Example:** Bit-sliced index for *Quantity* with values ranging from 1-100
  - $\lceil \log_2 100 \rceil = 7$ bits are needed.

B

RID	B_6	B_5	B_4	B_3	B_2	B_1	B_0
1	0	1	0	1	1	1	1
2	0	1	0	0	0	0	0
3	1	0	1	1	0	0	1
4	0	1	1	0	1	1	0
5	0	0	1	0	0	0	0

Sales

...	Quantity	...
...	47	...
...	32	...
...	89	...
...	54	...
...	16	...

Bit-Sliced Index/2

- Bit-sliced indexes are possible for attributes with large domains
- Standard bitmap indexes grow linearly with the number of distinct attribute values.
  - 1 column for each value
- Bit-sliced indexes have only a logarithmic grow in the size of the domain.
- Boolean operators can still be applied.
- To get all tuples with *quantity* > 63, retrieve all RIDs with $B_6 = 1$.

B

RID	B_6	B_5	B_4	B_3	B_2	B_1	B_0
1	0	1	0	1	1	1	1
2	0	1	0	0	0	0	0
3	1	0	1	1	0	0	1
4	0	1	1	0	1	1	0
5	0	0	1	0	0	0	0

Sales

...	Quantity	...
...	47	...
...	32	...
...	89	...
...	54	...
...	16	...

Bit-Sliced Index/3

- Bit-sliced indexes can be used to compute some aggregates without accessing the data, e.g., SUM, AVG
- Compute the sum of the binary values

```
**Algorithm:** $SUM(B_0, \ldots, B_n)$
**Input:** bit-sliced index $B$ consisting of $n$ slices built on an integer key
Sum := 0;
for $i = 0$ to $n$ do
    $Sum = Sum + 2^i * \#$ of 1's in$B_i$;
return Sum;
```

B

RID	B_6	B_5	B_4	B_3	B_2	B_1	B_0
1	0	1	0	1	1	1	1
2	0	1	0	0	0	0	0
3	1	0	1	1	0	0	1
4	0	1	1	0	1	1	0
5	0	0	1	0	0	0	0

Sales

...	Quantity	...
...	47	...
...	32	...
...	89	...
...	54	...
...	16	...

Bitmap-Encoded Index/1

- The idea of storing a binary encoding of numeric values has been applied to non-numeric domains.
- A **bitmap-encoded** index on an attribute C with k distinct values or a relation R consists of a bit matrix B and a conversion table T.
  - B contains $\log_2 k$ columns and has as many rows as tuples in R.
  - T contains k rows; the i-th row shows the binary coding of value c_i.

- **Example:** Bitmap encoded index for position attribute.

Employees

...	Position	...
...	Adm.	...
...	Prog.	...
...	Adm.	...
...	Tec.	...
...	Prog.	...
...	Ass.	...
...	Cons.	...
...	Cons.	...

B

B_2	B_1	B_0
0	0	0
1	0	0
0	0	0
1	0	1
1	0	0
0	0	1
0	1	0
0	1	0

T

Value	Coding
Adm.	000
Ass.	001
Cons.	010
Man.	011
Prog.	100
Tec.	101

Bitmap-Encoded Index/2

- Though an additional conversion table T is needed to translate the values encoded in the index, the index size can be considerably reduced (compared to a bitmap index).
- Bitmap-encoded index grows logarithmically in the size of the domain, while the bitmap index grows linearly.
- Boolean operators can be applied to bitmap-encoded indexes.
- Any selection predicate on key values can be represented by a Boolean expression, which selects intervals of valid binary values.
- To minimize the number of bitmap vectors that need to be accessed, a "good" encoding is crucial.

Coding Function

- A coding function of a bitmap-encoded index is **well defined** for a set of selection predicates if it minimizes the number of bit vectors to be accessed to check for the selection predicates.

Example: Attribute with values a, b, . . . , h

- Assume $key \in \{a, b, c, d\}$ and $key \in \{c, d, e, f\}$ are the most frequent predicates.
- The encoding is well-defined since
  - the first predicate is true if the B_1 vector is 0,
  - the second predicate is true if the B_0 vector is 1.

Value	Coding ($B_2 B_1 B_0$)
a	000
c	001
g	010
e	011
b	100
d	101
h	110
f	111

- How to verify a well-defined coding?

Verifying Well Defined Coding Functions

Example: (contd.)

- Construct a Boolean expressions for the query predicates:

$$key \in \{a, b, c, d\} \qquad \bar{B}_2\bar{B}_1\bar{B}_0 \vee B_2\bar{B}_1\bar{B}_0 \vee \bar{B}_2\bar{B}_1B_0 \vee B_2\bar{B}_1B_0$$

$$key \in \{c, d, e, f\} \qquad \bar{B}_2\bar{B}_1B_0 \vee B_2\bar{B}_1B_0 \vee \bar{B}_2B_1B_0 \vee B_2B_1B_0$$

- Using rules of Boolean algebra, these expressions can be simplified to $\bar{B}_1$ (i.e., $B_1 = 0$) and B_0; only one bit vector need to be accessed.

Value	Coding $(B_2B_1B_0)$
a	000
c	001
g	010
e	011
b	100
d	101
h	110
f	111

Bitmap-Encoded Index and Hierarchies/1

- Main OLAP operators are based on functional dependencies between dimensional attributes in hierarchies.
- Coding function for bitmap-encoded indexes allows to encode hierarchies.
- In general, the coding function allows you to encode both many-to-one and many-to-many associations.

Bitmap-Encoded Index and Hierarchies/2

- **Example:** Product dimension with hierarchy *category* → *type* → *product*
  - Coding table on the *product* attribute.
  - Only B_2 is needed to retrieve all products of a specific category.
    - $B_2 = 0 \rightarrow \textit{category} = \textit{Food}$
    - $B_2 = 1 \rightarrow \textit{category} = \textit{Clothes}$
  - Likewise, B_2 and B_1 are needed to retrieve a specific type
    - e.g., $B_2B_1 = 00$ represents type Cookies, $B_2B_1 = 10$ represents type Shirt.

Product Dim

Category	**Type**	**Product**
Food	Soft dring	Coca Cola
Food	Cookies	Chockly
Food	Cookies	Dippy
Clothes	Shirt	Button up
Clothes	Shirt	Classic
Clothes	Necktie	Imperial

Coding for *Product* attribute

Value	**Coding ($B_2B_1B_0$)**
Button up	100
Chockly	001
Classic	101
Coca Cola	010
Dippy	000
Imperial	110

Bitmapped Join Index/1

- A **bitmapped join index** built on the attributes C_R of a relation R and C_S of a relation S is a bit matrix B with $|R|$ rows and $|S|$ columns.
- Bit $B_{i,j}$ is 1 if the corresponding tuples satisfy the join predicate.

- **Example:** Bitmapped join index for fact table SALES and dimension table STORE.
  - e.g., Tuple 2 in SALES joins tuple 2 in STORE

STORE table RIDs $\swarrow$

SALES table RIDs $\longrightarrow$

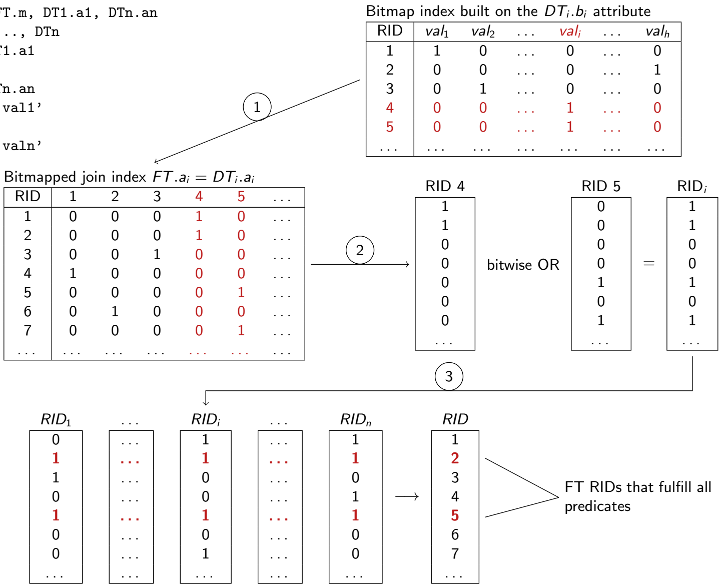
RID	1	2	3	...
1	1	0	0	...
2	0	1	0	...
3	0	0	1	...
4	0	1	0	...
5	1	0	0	...
...	...	...	...	...

Bitmapped Join Index/2

- Bitmapped join indexes can also be used to execute queries with multiple joins (star joins).
  1. Access the bitmap indexes on the dimension table(s) to identify the dimension tuples (RIDs) that fulfill the predicates on the dimensional attributes.
  2. For every bitmapped join index, load only the bit vectors corresponding to the RIDs identified in step 1. A bitwise OR yields the RID$_i$ vector that fulfills all predicates on a dimension table.
  3. Perform a bitwise AND between the *n* vectors obtained for each dimension.
- Repeat step 1 and 2 for each dimension involved in the join

Bitmapped Join Index Example

```
SELECT DISTINCT FT.m, DT1.a1, DTn.an
FROM   FT, DT1, ..., DTn
WHERE  FT.a1 = DT1.a1
AND    ...
AND    FT.an = DTn.an
AND    DT1.b1 = 'val1'
       ...
AND    DTn.bn = 'valn'
```



Bitmap index built on the $DT_i.b_i$ attribute

RID	val_1	val_2	...	val_i	...	val_h
1	1	0	...	0	...	0
2	0	0	...	0	...	1
3	0	1	...	0	...	0
4	0	0	...	1	...	0
5	0	0	...	1	...	0
...	...	...	...	...	...	...

Bitmapped join index $FT.a_i = DT_i.a_i$

RID	1	2	3	4	5	...
1	0	0	0	1	0	...
2	0	0	0	1	0	...
3	0	0	1	0	0	...
4	1	0	0	0	0	...
5	0	0	0	0	1	...
6	0	1	0	0	0	...
7	0	0	0	0	1	...
...	...	...	...	...	...	...

RID 4	RID 5	RID_i
1	0	1
1	0	1
0	0	0
0	0	0
0	1	1
0	0	0
0	1	1
...	...	...

RID_1	...	RID_i	...	RID_n	RID
0	...	1	...	1	1
1	**...**	**1**	**...**	**1**	**2**
1	...	0	...	0	3
0	...	0	...	1	4
1	**...**	**1**	**...**	**1**	**5**
0	...	0	...	0	6
0	...	1	...	0	7
...	...	...	...	...	...

FT RIDs that fulfill all predicates

Outline

Physical Storage

- Partitioning
  - Data stored in large "lumps" (partitions)
  - Example: one partition per quarter.
  - Queries need only read the relevant partitions.
  - Can yield large performance improvements.
- Operations on partitions are independent
  - Creation, deletion, update, indexing.
  - Aggregation level can be different among partitions.
- Column storage
  - Data stored in columns, not in rows.
  - A "reverse" kind of partitioning.
  - Works well for typical DW queries (only few columns accessed).
  - Supports good compression of data.

Physical Configuration

- RAID
  - Gives (depending on level) error tolerance and improved read speed.
  - DW optimized for reads, not for writes.
  - DW well suited for, e.g., RAID5 (20% redundancy).
- Disk type
  - Small drives (many controllers) are more expensive, but faster.
  - Large drives are cheaper, store more aggregates for same price.
- Block size
  - Large sequential reads faster with large blocks (32K).
  - Scattered index reads faster with small blocks (4K).
- Memory
  - RAM is cheap: buy a lot.
  - RAM caching must be per user session.
- Monitoring user activity
  - Can give feedback to, e.g., choice of aggregates.

DBMS Functionalities

- Aggregate navigation/use
  - Oracle 9iR2, DB2 UDB, MS Analysis Services
- Aggregate choice
  - Oracle 9iR2, DB2 UDB, MS Analysis Services
- Aggregate maintenance
  - Oracle 9iR2, DB2 UDB, MS Analysis Services
- Using ordinary indexes
  - Oracle 9iR2, DB2 UDB, MS SQL Server can do "star joins"
- Bitmap indexes
  - Oracle 9iR2, DB2 UDB not yet in MS SQL Server
- Partitioning
  - Oracle 9iR2, DB2 UDB, MS SQL Server+Analysis Services
- Column storage
  - MonetDB, DB2
- MOLAP/ROLAP/HOLAP
  - Oracle 9iR2, DB2 UDB, MS SQL Server

Summary

- Bitmap indexes are haevily used by data warehouses.
- Bitmap compression can significantly reduce the size of bitmap indexes.
  - Run-length encoding is a widely used technique
- Different versions of bitmap-based indices
  - Bitmap index for categorical attributes with low cardinality.
  - Bitmap-encoded index for categorical domains with many different values.
  - Bit-sliced index for numerical attributes.
  - Bimapped join index for the efficient evaluation of joins (including star joins).