1 Bitmap Indexes

Introduction

Bitmap indexes have traditionally been considered to work well for data such as gender, which has a small number of distinct values, for example male and female, but many occurrences of those values.

This would happen if, for example, you had gender data for each resident in a city. Bitmap indexes have a significant space and performance advantage over other structures for such data. Some researchers argue that Bitmap indexes are also useful for unique valued data which is not updated frequently. Bitmap indexes use bit arrays (commonly called bitmaps) and answer queries by performing bitwise logical operations on these bitmaps. (AND , OR , NOT , XOR).

Bitmap Indexes And Null Values

Unlike most other types of indexes, bitmap indexes include rows that have NULL values. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function COUNT.

Types of Bitmap Indexes

Base Data		Bitmap Index			
RID	X	B0	B1	B2	B3
0	2	0	0	1	0
1	1	0	1	0	0
2	3	0	0	0	1
3	0	1	0	0	0
4	3	0	0	0	1
5	1	0	1	0	0
6	0	1	0	0	0
7	0	1	0	0	0
8	2	0	0	1	0

Fig 1. Bitmap Index on Attribute X

In its simplest form a bitmap index on an indexed attribute consists of one vector of bits (ie. bitmap) per attribute value, where the size of each bitmap is equal to the number of records in the table. The bitmaps are encoded such that " the i'th record has a value of v in the indexed attribute if and only if the

i'th bit in the bitmap associated with the attribute value v is set to 1 and the i'th bit in each of the other bitmaps is set to 0. " This is called the Value-List index.

Can compression be achieved?

Yes, we can compress 0s and 1s which are consecutive. We can think of compressing seven 0s as one with a special header saying that this 0 is equivalent to seven 0s.

But this has its own disadvantage. For any bit-wise operation, we have to first decompress bitmap indexes in question and do the "and"/"or" operation, which outweighs any savings in storage costs.

Characteristic of Bitmap Indexes

- For columns with very few unique values (low cardinality)
- Columns that have low cardinality are good candidates (if the cardinality of a column is <= 0.1 % then the column is an ideal candidate, consider also 0.2% – 1%)
- Tables that have no or little insert/update are good candidates (static data in warehouse)
- Stream of bits: each bit relates to a column value in a single row of table

SQL Queries for Bitmap Indexes

create bitmap index person_region on person (region);

will create a bitmap index '**person_region'** for the table 'person' on the attribute 'region' which is expected to have low cardinality.

Advantages of Bitmap Indexes

- Not helpful when searching on one search key , main use comes when querying on multiple search keys. In that case (i.e select from r where attr1 = value1 and attr2 = value2), we grab

the entries for value1 and value2 , respectively , and do a bitwise-and on the bits - and we're done.

- Having a **or** in the predicate would be a bitwise or and having a **not** would be a bitwise not on the bits and we're done.
- Also useful for count queries - just count the bits, and we don't even need to go to the database records.
- To perform the bitwise operations, we take the advantage of native bitwise operations, and process things one word at a time - which today is 32 or 64 bits.

Insertion And Deletion

Inserts and deletes would modify the number of bits in the bitmap entries. Inserts aren't so bad since we just tack on more bits , but deletes are harder.

For performance related issues related to deletion, we can store a separate **existence bitmap**, with a 0 in the i'th bit if the i'th record has been deleted; then the queries would do a bitwise-and with the existence bitmap as their last step.

Combining Bitmaps and B+ Trees

When certain values appear very frequently in a database, the bitmap technique can be used in B+ Tree to save space. Recall that a B+ Tree leaf node points to the list of records whose attribute take over a particular value. Instead of one word-per-record (typical way to store the record list) do one bit-per-record the way it is done with a bit-map index.

Cooperativity of Bitmap Indexes

If two or more selection conditions are given in a query, say **a = v** AND **b = w** , separate simple bitmap indexes on a and b can efficiently work together to fetch the desired data by simply performing a logical operation, AND , on the corresponding bitmap vectors. In contrast, separate B-trees on attribute a and attribute b cannot efficiently cooperate with each other to fetch the data. We need to build another B-tree on the composite key (a,b).

The multidimensionality is simply a byproduct of the high cooperativity of bitmap indexing. On the other hand, if B-trees on composite keys are built, in order to cover all possible permutations of selection

conditions among given n attributes, we need at least **n!** B-Trees. The cost of maintaining so many B-trees would be unacceptable. Therefore, with respect to index co-operativity, simple bitmap indexes have dominating advantages over B-trees.

Further Advantages of Bitmap Indexes

In addition, the result of bitmap processing is also a bitmap that preserves the address information encapsulated in the bit positions. In other words, the result of bitmap processing is still a bitmap index which can be further used by successive query operations, such as Group Bys and aggregations. In contrast, value-list indexes are used by query processors mostly for selection evaluation. Successive query operations must be evaluated on the non-indexed intermediate result of selections.

Since the physical addresses of tuples are implicitly encapsulated in the bit positions and they are automatically stored in ordinal order, the address information can be easily converted to physical page numbers.