

1 Indexes

The purpose of an index is much like the purpose of an index in a book: to help us find the important bits quickly.

1.1 Dense vs Sparse

Just as with book indexes, *sparse* database indexes don't point to individual records, but to 'pages' (chunks of memory stored on the disk). The subsequent linear (or some other) search of the page in RAM is insignificant to the time it takes to read the page from disk.

Dense indexes point directly to individual records. Sparse indexes are generally more space efficient, as they don't need to point to individual records.

Many databases allow you to tweak the data block size that is used—which has an impact (both good and bad) on indexes, disk access performance, and database memory usage.

1.2 Clustered vs Unclustered

Many indexing solutions suffer from the 'need to read the whole page to access a record' problem. This is especially true for indexes on low cardinality columns, such as gender, zip code, state, etc. For example, an index on 'state' field will likely result in an index that references all of the database pages, so looking for every customer in 'NY' wouldn't be 50 times faster than scanning the whole database outright without an index.

To overcome the above problem, we can use *clustered* indexes. The creation of a clustered index re-arranges the actual data in the table to be in the order of the index. So sequentially retrieving records only for 'NY' state wouldn't sequentially read the whole database, but only a few data blocks (all of which will mostly store data with state 'NY').

Since physical data on the disk can only have one particular order, a table can have only one clustered index defined—the rest of the indexes need to be unclustered. In many databases, by default, the primary key index is clustered.

Clustered indexes are often stored in the data file along with the relation (a clustered index essentially represents the order of data in the table). Unclustered indexes generally use separate files.

1.3 B Trees

Balanced trees are the best way of storing ordered data, as access time grows logarithmically with the number of records they store. For example, if we take a binary tree storing 1 billion records, it will only take at most 30 comparisons to find any individual record.

B Trees, either B-Trees or B+Trees, are balanced trees, usually wider than binary (3-5 way are not uncommon), and are designed for easy on-disk manipulation. B-Trees store data on every node, while B+Trees only store data on the leafs.

B Tree indexes work well on columns with large cardinality (unique, or almost unique columns). If indexed columns have few values, then we hit the 'need to read the whole page to access a record' problem (might as well not use an index).

B Trees are usually the default index type in most databases.

1.4 Bitmap Index

Bitmap indexes are designed specifically for low cardinality columns—where a column only has 50 (or 300) or so distinct values.

The index is essentially a 2D bitmap (2D array) where rows represent different values, and columns represent different records that are being indexed. If we index ‘state’ in the address table, that has 50000 records, the index will have 50 or so rows, with 50000 columns. Such indexes are often compressed, so storage requirements are relatively low (compression is usually RLE, or Run Length Encoding)

When searching for a particular value, lets say state is ‘NY’, out of this 2D bitmap, we would quickly pick out row with ‘NY’—this gives us a list of bits, where the location of each bit represents the row number. If the bit is set, we retrieve that row.

Bitmap indexes work great in conjunction with other bitmap indexes. For example, imagine we have a table with many low cardinality columns, and we wish to provide for fast queries: imagine a ‘car’ database, with columns such as ‘make’, ‘model’, ‘year’, ‘color’, ‘seats’, etc., Each of these columns would only have a few values, etc.

If we create a bitmap index on all of these columns, individually, and then select a particular ‘make’, ‘year’, and ‘color’ from the table, we would hit the ‘make’ index, ‘year’ index, and ‘color’ index, each one will retrieve a row of bits, representing record numbers. But the cool thing is that those record numbers are ordered (first bit is first record, etc.). So we simply do a simple merge of the three lists to quickly find row numbers that match all three criteria (‘make’, ‘year’, and ‘color’).

1.5 Bitmap Join Index

We can also define a bitmap index on a join condition, which will store rows from tables that join. For example, if we have a 3-way join, we may create an index:

```
create bitmap index blahidx
on (table1.somevalue, table3, somevalue)
from table1 a, table2 b, table3 c
where a.id = b.id and b.id2 = c.id2
```

Any selects that involve these three tables using this join condition will now use this index, and will be very fast (as it wouldn’t have to go through the tables to perform the join—just extract joined row numbers from the index).

1.6 Function Indexes

Most databases allow users to create indexes that include functions (either built in functions, or user defined functions). For example, you may be able to create an index on the first 4 characters of a last name, via:

```
create index blahidx on customer(substr(lname,1,4))
```

1.7 Hash Indexes

Hash indexes are essentially function indexes, where the function is a ‘hash’ function; returns a distinct hash code. They’re useful when indexing wide columns (if a column has 30 characters, for example).

The limitation with hash indexes is that you cannot do comparisons nor ranges on them. Queries such as:

```
select *  
from customer  
where lname < 'blah'
```

Will not benefit from hash indexes (the query will end up doing a full table scan).

1.8 Index Tables

Some databases support storage of data in indexes. In other words, instead of creating a table, and then having an index on every column, you can simply create an index with all the columns, ie:

```
create table customer (  
    customerid int ,  
    lname varchar(30) ,  
    fname varchar(30) ,  
    email varchar(60) ,  
    primary key(customerid)  
) ORGANIZATION index;
```

The above is for Oracle, though I’m sure other databases support similar features. Whenever you query this table, the optimizer will recognize that it is an ‘index’, and retrieve it accordingly.

2 Join Types

Whenever you write database queries, the optimizer figures out the best way to perform the query. It does that by parsing the SQL statemnets, generalizing them into relational algebra, then, using huristics, transform the relational algebra expression into several other expressions, pick which one ‘seems’ like it would execute fastest (depending on past statistics, or assumed table values, or maybe after running a quick test against actual data).

Depending on the size of the tables, the database will pick which kind of joins to use.

2.1 Nested Loop Join

A nested loop join is exactly what it says. Given two tables, of size n and m respectively, the nested loop join will do $O(n \times m)$ loops. The loops look somewhat like this:

```
for (i=0;i<table1.length;i++){  
    for (j=0;j<table2.length;j++){  
        if(table1[i].joinkey == table2[j].joinkey){
```

```

        out ( table1 [ i ] , table2 [ j ] );
    }
}

```

The problem with this approach is when both tables are relatively large. If each table is 100000 elements, then this loop will run for 10 billion times, which is bad.

This join is great when one of the tables is very very small.

2.2 Hash Join

Most modern databases primarily use Hash Joins. These work by first looping through the “smaller” (as perceived by optimizer) table and storing the values in a hash. The hash key is derived from the join keys. Then looping through the larger table, and picking at the hash for matched values. ie, joining table1 and table2 on id and name, with assumption that table1 is smaller.

```

for ( i=0; i<table1.length; i++){
    hashkey = hash ( table1 [ i ].id , table1 [ i ].name );
    hash [ hashkey ] = table1 [ i ];
}
for ( j=0; j<table2.length; j++){
    hashkey = hash ( table2 [ j ].id , table2 [ j ].name );
    t1val = hash [ hashkey ]
    if ( t1val and t1val.id = table2 [ j ].id and
        t1val.name = table2 [ j ].name ){
        out ( t1val , table2 [ j ] );
    }
}

```

The assumption is that hash operations are fast (ie: constant time). If hash operations are constant, then speed is the cost of looping through the two lists independently, ie: $O(n + m)$. If hash operations are logarithmic (ie: red-black tree implementation), then cost is $O(n \lg n + m \lg m)$.

2.3 Merge Join

Merge join is the ‘old’ method of doing table joins (this is what you’d do on a mainframe). The idea is that you first sort both tables by the join keys, and then loop through both tables simultaneously, picking out records that match (the merge function is similar to mergesort).

The cost is primarily sorting the tables (which may involve disk swaps, as tables may be large), and then subsequently doing a linear join. So... $O(n \lg n + m \lg m + n + m)$ or something like that. Obviously the $O(n + m)$ are irrelevant. Note that we are assuming $O(n \lg n)$ sorting—many real world implementations used radix like sorts, which have linear running times.

2.4 Index Join

Index joins are essentially “merge joins” except instead of sorting the tables, we use the index (which is essentially the sorted view of some columns). This join type is only useful when only a handful of records are involved—if a whole table join is necessary, let the database pick a method (it will likely pick hash join, which will be faster than index join for the whole table).