Oracle has a lot of index types available (IOT, Cluster, etc.), but I’ll only speak about the three main ones used in data warehouses.

**B-tree Indexes**

B-tree indexes are mostly used on unique or near-unique columns. They keep a good performance during update/insert/delete operations, and therefore are well adapted to operational environments using third normal form schemas. But they are less frequent in data warehouses, where columns often have a low [cardinality](http://en.wikipedia.org/wiki/Cardinality_(SQL_statements)). Note that B-tree is the default index type – if you have created an index without specifying anything, then it’s a B-tree index.

**Bitmap Indexes**

Bitmap indexes are best used on low-cardinality columns, and can then offer significant savings in terms of space as well as very good query performance. They are most effective on queries that contain multiple conditions in the WHERE clause.

Note that bitmap indexes are particularly [slow](http://www.clariba.com/blog/b-tree-vs-bitmap-indexes-indexing-strategy-for-your-oracle-data-warehouse/) to update.

**Bitmap Join Indexes**

A bitmap join index is a bitmap index for the join between tables (2 or more). It stores the result of the joins, and therefore can offer great performances on pre-defined joins. It is specially adapted to star schema environments.

From wikipedia: [B-Trees](http://en.wikipedia.org/wiki/B-tree) and [bitmap indexes](http://en.wikipedia.org/wiki/Bitmap_index). The use cases:

* B-Trees are the typical index type used when you do CREATE INDEX ... in a database:
  1. They are very fast when you are selecting just a small very subset of the index data (5%-10% max typically)
  2. They work better when you have a lot of distinct indexed values.
  3. Combining several B-Tree indexes can be done, but simpler approaches are often more efficient.
  4. They are not useful when there are few distinct values for the indexed data, or when you want to get a large (>10% typically) subset of the data.
  5. Each B-Tree index impose a small penalty when inserting/updating values on the indexed table. This can be a problem if you have a lot of indexes in a very busy table.

This characteristics make B-Tree indexes very useful for speeding searches in OLTP applications, when you are working with very small data sets at a time, most queries filter by ID, and you want good concurrent performance.

* Bitmap indexes are a more specialized index variant:
  1. They encode indexed values as bitmaps and so are very space efficient.
  2. They tend to work better when there are few distinct indexed values
  3. DB optimizers can combine several bitmap indexed very easily, this allows for efficient execution of complex filters in queries.
  4. They are very inefficient when inserting/updating values.

Bitmap indexes are mostly used in data warehouse applications, where the database is read only except for the ETL processes, and you usually need to execute complex queries against a [star schema](http://en.wikipedia.org/wiki/Star_schema), where bitmap indexes can speed up filtering based on conditions in your dimension tables, which do not usually have too many distinct values.

As a very short summary: use B-Tree indexes (the "default" index in most databases) unless you are a data warehouse developer and *know* you will benefit for a bitmap index.

# How b-tree database indexes work and how to tell if they are efficient (100' level)

A team member thought we should add an index on a 90 million row table to improve performance. The field on which he wanted to create this index had only four possible values. To which I replied that an index on a low cardinality field wasn't really going to help anything. My boss then asked me why wouldn't it help? I sputtered around for a response but ended up telling him that I'd get back to him with a reasonable explanation.

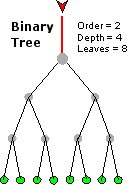
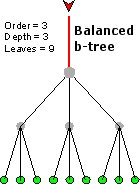
Now I'm not a DBA by any stretch but I've learned about database optimization and performance on the job from some really bright folks. I didn't have a very good grasp on how indexes worked though. So I did some research on the topic.

There are several kinds of indexes that databases use. Sybase IQ has like 20 different kinds, Oracle and DB2 appear to have two. The main type of index out there is a [**b-tree**](http://en.wikipedia.org/wiki/b-tree); this is the type that most people mean when they say database index.

### What is a b-tree?

In a tree, records are stored in locations called leaves. The starting point is called the root. The maximum number of children per node is called the **order** of the tree. The maximum number of access operations required to reach the desired leaf (data stored on the leaf) is called the **depth** (level). Oracle indexes are balanced b-trees; the order is the same at every node and the depth is the same for every leaf.

|  |  |
| --- | --- |
| **real tree (in nature)** | **b-tree** |
| grows up | grows down |
| main trunk | root |
| branch | node |
| leaf | leaf |

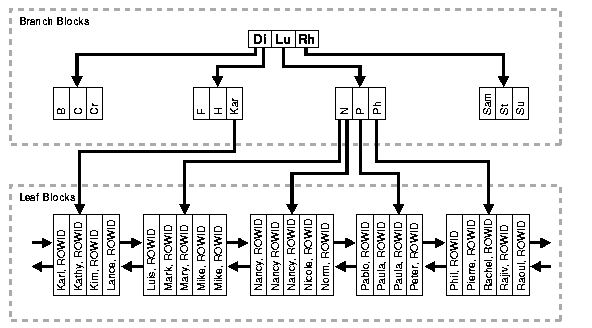
**[](http://mattfleming.com/node/193)** --------- **[](http://mattfleming.com/node/194)**

The bigger the order, the more leaves and nodes you can put at a certain depth. This means that there are fewer levels to traverse to get to the leaf (which contains the data you want). In the example above and all balanced b-trees, the number of hops to a leaf == depth.

### How does a b-tree help with database access?

Most indexes are too large to fit into memory, which means that they are going to be stored on disk. Since I/O is usually the most expensive thing you can do in a [COMPUTER[http://cdncache-a.akamaihd.net/items/it/img/arrow-10x10.png](http://mattfleming.com/node/192)](http://mattfleming.com/node/192) system, these **indexes need to be stored in an I/O efficient way**.

A b-tree is a good way to do this. If we make the nodes the size of a physical I/O block, it would take one I/O to move to a lower depth in the tree. In the example below, an index was created on a first name kind of field.

**[](http://mattfleming.com/node/196)**

If every level were an I/O it would take 3 I/Os to find Mary (or any other leaf).

### How good is the index?

Now back to the original point I was trying to make-- low cardinality fields make bad indexes. Why is this the case? The answer here is really about selectivity.

unique index values

selectivity = -----------------------

total number records

A primary key is highly selective. If there are 1000 rows, there will be 1000 unique keys in the index. Eacy unique key will return at most 1 row. The index will be 100% selective (1000/1000).. the best you can get.

Now let's say we have an index on a low cardinality thing like gender. If we had 1000 records, the selectivity is in the database is 2/1000 =.2%. Said in another way, 500 records come back per unique key (1000 records / 2 uniques).

Note: this seems to assume an even distribution of data (e.g. 500 male, 500 female). Things might be different if you had 999 males, and 1 female.

### Hand-wavy rule

**10% selectivity is the minimum selectivity** necessary for a b-tree index to be helpful.

### What to do about low cardinality columns?

I'm not going to go into it in any detail. You can either use a different kind of index (e.g. [**Bitmap index**](http://www.oracle.com/technology/pub/articles/sharma_indexes.html)) or **combine that column with another** to make a highly selective composite index.

### Oracle way to find low selectivity indexes

Run this query to get an idea of how your indexes are set up.

SELECT

INDEX\_NAME "NAME",

DISTINCT\_KEYS / NUM\_ROWS \* 100 "SELECTIVITY %",

NUM\_ROWS,

DISTINCT\_KEYS "DISTINCT",

LEAF\_BLOCKS,

CLUSTERING\_FACTOR,

BLEVEL "LEVEL",

AVG\_LEAF\_BLOCKS\_PER\_KEY "ALFBPKEY"

FROM

DBA\_INDEXES

WHERE

DISTINCT\_KEYS / NUM\_ROWS < .1 AND

NUM\_ROWS > 0

ORDER BY "SELECTIVITY %" DESC;

### Conclusion

B-tree indexes are created to decrease the amount of I/O required to find and load a set of data. A highly selective index uses least amount of I/O necessary, poorly selective indices are not much better than a table scan.