

Title: How MySQL Uses Indexes

Subtitle: General

Author: MySQL AB. MySQL DOC

Date: 2005/4/16

URL: <http://xoopsforge.com/modules/article/view.article.php/c4/4>

Keywords: Index, PRIMARY KEY, UNIQUE, FULLTEXT

Summary: Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions are that indexes on spatial column types use R-trees, and MEMORY (HEAP) tables support hash indexes.

Indexes are used to find rows with specific column values fast. Without an index, MySQL has to start with the first record and then read through the whole table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. If a table has 1,000 rows, this is at least 100 times faster than reading sequentially. Note that if you need to access almost all 1,000 rows, it is faster to read sequentially, because that minimizes disk seeks.

Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions are that indexes on spatial column types use R-trees, and MEMORY (HEAP) tables support hash indexes.

Strings are automatically prefix- and end-space compressed.

See [Section#160:13.2.4, “CREATE INDEX Syntax”](#).

In general, indexes are used as described in the following discussion. Characteristics specific to hash indexes (as used in MEMORY tables) are described at the end of this section.

Indexes are used for these operations:

To quickly find the rows that match a WHERE clause.

To eliminate rows from consideration. If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows.

To retrieve rows from other tables when performing joins.

To find the MIN() or MAX() value for a specific indexed column *key_col*. This is optimized by a preprocessor that checks whether you are using WHERE *key_part_# = constant* on all key parts that occur before *key_col* in the index. In this case, MySQL does a single key lookup for each MIN() or MAX() expression and replace it with a constant. If all expressions are replaced with constants, the query returns at once. For example:

```
SELECT MIN(key_part2),MAX(key_part2)
FROM tbl_name WHERE key_part1=10;
```

To sort or group a table if the sorting or grouping is done on a leftmost

prefix of a usable key (for example, ORDER BY *key_part1*, *key_part2*). If all key parts are followed by DESC, the key is read in reverse order.

See [Section 7.2.10, "How MySQL Optimizes ORDER BY"](#).

In some cases, a query can be optimized to retrieve values without consulting the data rows. If a query uses only columns from a table that are numeric and that form a leftmost prefix for some key, the selected values may be retrieved from the index tree for greater speed:

```
SELECT key_part3 FROM tbl_name WHERE key_part1=1
```

Suppose that you issue the following SELECT statement:

```
mysql> SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;
```

If a multiple-column index exists on *col1* and *col2*, the appropriate rows can be fetched directly. If separate single-column indexes exist on *col1* and *col2*, the optimizer tries to find the most restrictive index by deciding which index finds fewer rows and using that index to fetch the rows.

If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to find rows. For example, if you have a three-column index on (*col1*, *col2*, *col3*), you have indexed search capabilities on (*col1*), (*col1*, *col2*), and (*col1*, *col2*, *col3*).

MySQL can't use a partial index if the columns don't form a leftmost prefix of the index. Suppose that you have the SELECT statements shown here:

```
SELECT * FROM tbl_name WHERE col1=val1;
```

```
SELECT * FROM tbl_name WHERE col2=val2;
```

```
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
```

If an index exists on (*col1*, *col2*, *col3*), only the first of the preceding queries uses the index. The second and third queries do involve indexed columns, but (*col2*) and (*col2*, *col3*) are not leftmost prefixes of (*col1*, *col2*, *col3*).

A B-tree index can be used for column comparisons in expressions that use the =, >, >=, <, <=, or BETWEEN operators. The index also can be used for LIKE comparisons if the

argument to LIKE is a constant string that doesn't start with a wildcard character. For example, the following SELECT statements use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
```

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

In the first statement, only rows with 'Patrick' <= *key_col* < 'Patricl' are considered. In the second statement, only rows with

'Pat' <= *key_col* < 'Pau' are considered.

The following SELECT statements do not use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
```

```
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

In the first statement, the LIKE value begins with a wildcard character. In the second statement, the LIKE value is not a constant.

MySQL 4.0 and up performs an additional LIKE optimization. If you use ... LIKE '%string%' and *string* is longer than three characters,

MySQL uses the Turbo Boyer-Moore algorithm to initialize the pattern for the string and then use this pattern to perform the search quicker.

Searching using *col_name* IS NULL uses indexes if *col_name* is indexed.

Any index that doesn't span all AND levels in the WHERE clause is not used to optimize the query. In other words, to be able to use an index, a prefix of the index must be used in every AND group.

The following WHERE clauses use indexes:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3
```

```
/* index = 1 OR index = 2 */
```

```
... WHERE index=1 OR A=10 AND index=2
```

```
/* optimized like "index_part1='hello'" */
```

```
... WHERE index_part1='hello' AND index_part3=5
```

```
/* Can use index on index1 but not on index2 or index3 */
```

```
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

These WHERE clauses do *not* use indexes:

```
/* index_part1 is not used */
```

```
... WHERE index_part2=1 AND index_part3=2
```

```
/* Index is not used in both AND parts */
```

```
... WHERE index=1 OR A=10
```

```
/* No index spans all rows */
```

```
... WHERE index_part1=1 OR index_part2=10
```

Sometimes MySQL does not use an index, even if one is available. One way this occurs is when the optimizer estimates that using the index would require MySQL to access a large percentage of the rows in the table. (In this case, a table scan is probably much faster, because it requires fewer seeks.) However, if such a query uses LIMIT to only retrieve part of the rows, MySQL uses an index anyway, because it can much more quickly find the few rows to return in the result.

Hash indexes have somewhat different characteristics than those just discussed:

They are used only for equality comparisons that use the = or <=> operators (but are *very* fast). They are not used for comparison operators such as < that find a range of values.

The optimizer cannot use a hash index to speed up ORDER BY operations. (This type of index cannot be used to search for the next entry in order.)

MySQL cannot determine approximately how many rows there are between two values (this is used by the range optimizer to decide which index to use). This may affect some queries if you change a MyISAM table to a hash-indexed MEMORY table.

Only whole keys can be used to search for a row. (With a B-tree index, any leftmost prefix of the key can be used to find rows.)