Indexing the MySQL Index

-A database index is a data structure that improves the speed of data retrieval operations on a database table.

-A mechanism to locate and access data within a database.

-Without an index, MySQL must begin with the first row and then read through the entire 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. This is much faster than reading every row sequentially.

-Most of MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees.
Exceptions are that indexes on spatial data types use R-trees.

MySQL uses indexes for these operations:

- Speedy data retrieval. - SPEED of SELECTs

-Rapid random look ups.

- Efficient for Reporting, OLAP, read-intensive applications

However it is expensive for

 - Slows down writes as every indices need to be modified during INSERT,UPDATE, DELETE.

 - heavy write applications (OLTP) be careful

 - More disk space used

- Query optimizer considers indexes during every query, more indexes makes this work harder.

Properties

- can be created on one or more fields.

- Index may be unique or non-unique.

- Index may quote one or more columns and be a means of enforcing uniqueness of their values

- The maximum number of indexes per table and the maximum index length is defined per storage engine.

- All storage engines support at least 16 indexes per table and a total index length of at least 256 bytes.

Most storage engines have higher limits.

Index choice depends on

Index choice depends on...

- What tables and columns you need to query

- What JOINs you need to perform

- What GROUP BY’s and ORDER BY’s you need

- Index design is not implicit from table design.

relational schema design is based on data. index design is based on queries.

Type of indexes.

- Column Index
- Concatenated Index /Multiple-Column Indexes / composite indexes
- Covering Index
- Partial Index
- Fulltext Indexes
- Clustered/Non-clustered Index

Column index

Index on a single column

The B-tree data structure lets the index quickly find a specific value, a set of values, or a range of values, corresponding to operators such as =, >, ≤, BETWEEN, IN, and so on, in a WHERE claus

empid | lname | fname | salary | gender

In this table if we have to search for a member whose lname is patil then code will look like :

For each row in table

if(row[2] = 'patil' then

return the row.

Else

movenext

So we checking each row for condition.

Only those query will be optimized which satisfy your criteria.

Eg: SELECT lname

FROM emp

WHERE empid = 20001

By adding an index to empid, the query is optimized to only look at records that satisfy your criteria

Concated index

Index on multiple columns.

```
SELECT empid, lname
    FROM   emp
    WHERE empid = 20001
    AND lname = 'Patil';
```

- MySQL can use multiple-column indexes for queries that test all the columns in the index, or queries that test just the first column, the first two columns, the first three columns, and so on.

- If we specify the columns in the right order in the index definition, a single composite index can speed up several kinds of queries on the same table.

- If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows.

That is, if we have a three-column index on (col1, col2, col3), we have indexed search capabilities on (col1), (col1, col2), and (col1, col2, col3).

-MySQL cannot use the index to perform lookups if the columns do not form a leftmost prefix of the index.

Suppose that we have the SELECT statements shown here:

1) SELECT * FROM tbl_name WHERE col1=val1;
2) SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;
3) SELECT * FROM tbl_name WHERE col2=val2;
4) SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;

If an index exists on (col1, col2, col3),

Only the first two queries use the index. The third and fourth queries do involve indexed columns, but (col2) and (col2, col3) are not leftmost prefixes of (col1, col2, col3).

```
CREATE TABLE `emp2` (
  `empid` int(11) NOT NULL auto_increment,
  `lname` char(30) NOT NULL,
  `fname` char(30) NOT NULL,
  `salary` int(11) NOT NULL,
  `gender` enum('m','f') DEFAULT NULL,
   primary key(empid),
   INDEX name (lname,fname)
)
```

The index can be used for lookups in queries that specify values in a known range for combinations of lname and fname values.

It can also be used for queries that specify just a lname value because that column is a leftmost prefix of the index. Therefore, the name index is used for lookups in the following queries:

```
SELECT * FROM emp2 WHERE lname='penna';

SELECT * FROM emp2 WHERE lname='penna' AND fame='Ravi';

SELECT * FROM emp2 WHERE lname ='patil' AND (fname='Uma' OR fname='Mau');

SELECT * FROM emp2 WHERE lname='Joshi' AND fname >='M' AND fname < 'Q';
```

However, the name index is not used for lookups in the following queries:

```
SELECT * FROM emp2 WHERE  fname='bhu';

SELECT * FROM emp2  WHERE lname='patil' OR fname='uma';
```

Partial index /Prefix Indexes

With col_name(N) syntax in an index specification, you can create an index that uses only the first N characters of a string column.

- Subset of a column for the index.

- Use on CHAR, VARCHAR,TEXT etc

-Indexing only a prefix of column values in this way can make the index file much smaller. When you index a BLOB or TEXT column, you must specify a prefix length for the index. e.g

CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));

- Creating a partial index may greatly reduce the size of the index, and minimize the additional data lookups required.

Create table test (name char(255) , INDEX (name(15)));

Fulltext Indexes

Ability to search for text.

Earlier was available only for MyISAM

Can be created for a TEXT, CHAR or VARCHAR.

Searches are not case sensitive.

Short words are ignored, the default minimum length is 4 character.

ft_min_word_len ft_max_word_len

Words called stopwords are ignored:

ft_stopword_file= ' '

If a word is present in more than 50% of the rows it will have a weight of zero. This has advantage on large data sets

Selectivity

Selectivity of a column is the ratio between number of distinct values and number of total values.

Primary Key has selectivity 1.

eg: emp table has 500,000 users with fields empid ,fname ,lname ,salary ,gender

Our application searches for following fields:

empid | fname | fname | gender

So empid, lname, fname and gender can be candidates for indexes

Since employee id is unique its selectivity will be equal to the primary key selectivity.

In case of gender it will have two values m,f

selectivity = 2/500,000 = 0.000004 #If we drop this index , it will be more beneficial.

Selectivity above than 15% is a good index.

- 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 (the most selective index).

-To retrieve rows from other tables when performing joins. MySQL can use indexes on columns more efficiently if they are declared as the same type and size.

-In this context, VARCHAR and CHAR are considered the same if they are declared as the same size. For example, VARCHAR(10) and CHAR(10) are the same, but VARCHAR(10) and CHAR(15) are not.

For comparisons between non-binary string columns, both columns should use the same character set. For example, comparing a utf8 column with a latin1 column precludes use of an index

- Comparison of dissimilar columns (comparing a string column to a temporal or numeric column, for example) may prevent use of indexes if values cannot be compared directly without conversion.

For a given value such as 1 in the numeric column, it might compare equal to any number of values in the string column such as '1', ' 1', '00001', or '01.e1'

Displaying INDEX Information:

SHOW INDEX command

e.g

mysql> SHOW INDEX FROM emp1\G

mysql> SHOW INDEX FROM emp2\G

Thanks.