INDEXING STRATEGIES FOR OPTIMIZING QUERIES ON MYSQL

ANCA MEHEDINȚU,
CERASELA PÎRVU, CRISTI ETEGAN *

ABSTRACT: *This article investigates MySQL's index capabilities. It begins by reviewing how indexes work, as well as their structure. Next, it reviews indexing features specific to each of the major MySQL data storage engines. This article then examines a broad range of situations in which indexes might help speed up your application. In addition to examining how indexes can be of assistance. In this article we present index usage type: B-trees, hash and bitmap, in order to optimize queries, although MySQL has implemented and indexes spacious R-trees. The index type corresponds to the particular kinds of internal algorithms and datastructures used to implement the index. In MySQL, support for a particular index type is dependent upon the storage engine.*

KEY WORDS: *Optimizing database structure, Database performance, Optimizing queries, Indexes, B-tree Index, Hash Index, Bitmap Index, Storage engines*

JEL CLASSIFICATION: *M15*

1. CONSIDERATIONS ON THE MYSQL DATABASE INDEXING

Database tuning is the process of improving database performance by minimizing response time (the time it takes a statement to complete) and maximizing throughput the number of statements a database can handle concurrently per second) (Schwarty, et al., 2004).

Tuning is a team exercise - collectively performed by DBAs, database designers, application designers and database users, based understanding of the database. Tuning both depends on and impacts the following: table design, relationships, index design, and other components; query design, size of data read and

* *Assoc. Prof., Ph.D., University of Craiova, Romania, ancamehedintu@yahoo.com*
Prof., Ph D., University of Craiova, Romania, ccpirvu2006@yahoo.com
Eng., Ph.D., University of Craiova, Romania, cristi.etegan@gmail.com

retrieved, and order of execution; nature and frequency of read and insert/update/delete operations; partitioning of the work between the Database server and the client; timing, events and effect of loading tables, indexes or parts there of into memory; concurrency characteristics of statements.

Database performance becomes an important issue in the presence of large amounts of data, complex queries, queries manipulating large amounts of data, long running queries, queries that lock every one else out, large number of simultaneous transactions, large numbers of users and limited bandwidth. In general, most database systems are designed for good performance. The best improvements can be achieved in the initial design phase but sometimes not enough information is available about the characteristics of a database. Later, altering a large database in production use can be expensive and practical considerations put constraints on what can be changed. Tuning can make the difference between a query taking milliseconds or minutes or even more to execute.

Database system is the core of management information systems, database-based online transaction processing (OLTP) and online analytical processing (OLAP) is a banking, business, government and other departments of the most important one of computer applications (Williams & Lane, 2007). From the application of most systems, the query operation in a variety of database operations in the largest occupied, and the query operation is based on the SELECT statement in the SQL statement is a statement of the cost of the largest. For example, if the amount of data accumulated to a certain extent, such as a bank account to the database table of information on the accumulation of millions or even tens of millions of records, full table scan often requires tens of minutes time, and even a few hours. If better than the full table scan query strategy can often be reduced to a few minutes to make inquiries, we can see the importance of query optimization technology.

Many programmers think that query optimization is a DBMS (database management system) tasks, prepared with the programmer has little to do with SQL statement, which is wrong. A good query plan performance often can improve the number of times. Query plan is submitted by users a collection of SQL statements, query plan is optimized to deal with the statement after the collection of produce. DBMS query plan to deal with the process is as follows: in the query after the lexical, syntax check, the statement will be submitted to the DBMS's query optimizer, optimizer after algebraic optimization and optimization of access to the path followed by pre-compiled modules processing of statements and generate inquiries, planning, and then at the right time to the system implementation, the final results will be returned to the user. In the actual database products (such as Oracle, Sybase, etc.) are all versions of the high cost-based optimization method, this optimization of the dictionary from the system based on the information table to estimate the different costs of planning inquiries, and then select a better planning (Opell, 2006). While it is in the database query optimization has been done better, but by the user of the SQL statement submitted to the system based on optimization, it is difficult to imagine a worse original query plan after the system has become efficient after optimization, so written statement of the advantages and disadvantages of users is essential. System we did not discuss query optimization, focusing on the following plan to improve the

user's query solution. Like beauty, the most attractive indexing strategy is very much in the eye of the beholder. After indexes are in place for primary, join, and filter keys (a universal standard of indexing beauty, perhaps?), what works for application A might be the wrong approach for application B.

Application A might be a transactional system that supports tens of thousands of quick interactions with the database, and its data modifications must be made in milliseconds. Application B might be a decision support system in which users create an ample assortment of server-hogging queries. These two applications require very different indexing tactics.

MySQL's optimizer always tries to use the information at hand to develop the most efficient query plans. However, requirements change over time; users and applications can introduce unpredicted requests at any point. These requests might include new transactions, reports, integration, and so forth.

Indexing is the most important tool you have for speeding up queries. Other techniques are available to you, too, but generally the one thing that makes the most difference is the proper use of indexes. On the MySQL mailing list, people often ask for help in making a query run faster. In a surprisingly large number of cases, there are no indexes on the tables in question, and adding indexes often solves the problem immediately. It doesn't always work like that, because optimization isn't always simple. Nevertheless, if you don't use indexes, in many cases you're just wasting your time trying to improve performance by other means. Use indexing first to get the biggest performance boost and then see what other techniques might be helpful.

The particular details of index implementations vary for different MySQL storage engines. For example, for a MyISAM table, the table's data rows are kept in a data file, and index values are kept in an index file. You can have more than one index on a table, but they're all stored in the same index file. Each index in the index file consists of a sorted array of key records that are used for fast access into the data file.

By contrast, the BDB and InnoDB storage engines do not separate data rows and index values in the same way, although both maintain indexes as sets of sorted values. By default, the BDB engine uses a single file per table to store both data and index values. The InnoDB engine uses a single tablespace within which it manages data and index storage for all InnoDB tables. InnoDB can be configured to create each table with its own tablespace, but even so, a table's data and indexes are stored in the same tablespace file.

MySQL uses indexes in several ways. As just described, indexes are used to speed up searches for rows matching terms of a WHERE clause or rows that match rows in other tables when performing joins. For queries that use the MIN() or MAX() functions, the smallest or largest value in an indexed column can be found quickly without examining every row. MySQL can often use indexes to perform sorting and grouping operations quickly for ORDER BY and GROUP BY clauses.

Sometimes MySQL can use an index to reading all the information required for a query. Suppose that you're selecting values from an indexed numeric column in a MyISAM table, and you're not selecting other columns from the table. In this case, when MySQL reads an index value from the index file, it obtains the same value that it

would get by reading the data file. There's no reason to read values twice, so the data file need not even be consulted (Dubois, 2008).

In general, if MySQL can figure out how to use an index to process a query more quickly, it will. There are and disadvantages. There are costs both in time and in space. In practice, these drawbacks tend to be outweighed by the advantages, but you should know what they are.

First, indexes speed up retrievals but slow down inserts and deletes, as well as updates of values in indexed columns. That is, indexes slow down most operations that involve writing. This occurs because writing a record requires writing not only the data row, it requires changes to any indexes as well. The more indexes a table has, the more changes need to be made, and the greater the average performance degradation (Dubois, 2008). Second, an index takes up disk space, and multiple indexes take up correspondingly more space. This might cause to reach a table size limit more quickly than if there are no indexes: For a MyISAM table, indexing it heavily may cause the index file to reach its maximum size more quickly than the data file. For BDB tables, which store data and index values together in the same file, adding indexes causes the table to reach the maximum file size more quickly.

All InnoDB tables that are located within the InnoDB shared tablespace compete for the same common pool of space, and adding indexes depletes storage within this tablespace more quickly. However, unlike the files used for MyISAM and BDB tables, the InnoDB shared tablespace is not bound by your operating system's file-size limit, because it can be configured to use multiple files. As long as you have additional disk space, you can expand the tablespace by adding new components to it. InnoDB tables that use individual tablespaces are constrained the same way as BDB tables because data and index values are stored together in a single file.

2. INDEXING STRATEGIES. CASE STUDY

In this article we present index usage type: B-trees, hash and bitmap, in order to optimize queries, although MySQL has implemented and indexes spacious R-trees.

The index type corresponds to the particular kinds of internal algorithms and datastructures used to implement the index. In MySQL, support for a particular index type is dependent upon the storage engine.

SQL code for creating the database used in the case study is:

```
create database work;
create table person (
id int( 9 ) unsigned not null auto_increment primary key ,
age tinyint( 3 ) unsigned not null ,
last_name varchar( 30 ) not null ,
hometown varchar( 30 ) not null ,
gender enum( 'female', 'male' ) not null) engine = innodb;
select count(*) as records_no
from `person`;
```

Table 1. Query result

records_no
101001

Database tables, at least conceptually, look something like in Table 2:

Table 2. A view of 'work' database

id	age	last_name	hometown	gender
...	...	...	...	...
1	10	Jianu	Slatina	Female
2	27	Stoica	Satu Mare	Male
3	15	Rosca	Pitesti	Male
4	64	Fabian	Mangalia	Female
5	13	Popescu	Slatina	Male
6	17	Stoica	Orsova	Male
...	...	...	...	...
100	49	Matei	Bucuresti	Female
101	30	Voicu	Mangalia	Male
102	18	Simion	Slatina	Female
104	6	Jianu	Slatina	Male
...	...	...	...	...
10000	4	Florescu	Mangalia	Male
10001	25	Balaci	Craiova	Female

2.1. B-tree index

They are general purpose indexes. Supported for the MyISAM, InnoDB, MEMORY and NDB storage engines. A B-Tree index is redundant if all of it's columns are also indexed in the same order by another index, and the first indexed column is the same for both these indexes. Another way of putting it is to say that the columnlist of the index is the leftmost prefix of the columnlist of the other index.

The data structure most commonly used for database indexes is B-Trees, a specific kind of self-balancing tree. A picture's worth a thousand words, so here's an example from Figure 1 wich shows the image of a B-tree index for age column in table person.

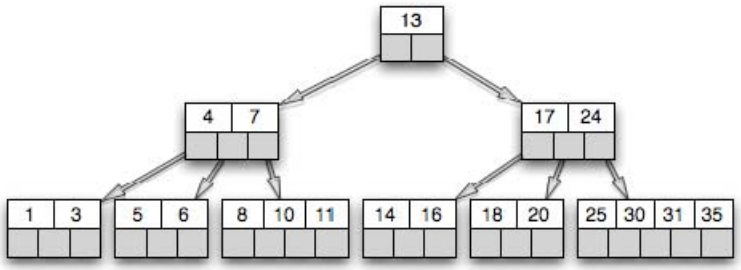

Figure 1. B-Tree index structure

A B-tree index is a data structure in the form of a tree - no surprises there - but it is a tree of database blocks, not rows. Imagine the leaf blocks of the index as the pages of a phone book. Each page in the book (leaf block in the index) contains many entries, which consist of a name (indexed column value) and an address (ROWID) that tells you the physical location of the telephone (row in the table).

The names on each page are sorted, and the pages - when sorted correctly - contain a complete sorted list of every name and address.

A sorted list in a phone book is fine for humans, because we have mastered "the flick" - the ability to fan through the book looking for the page that will contain our target without reading the entire page. When we flick through the phone book, we are just reading the first name on each page, which is usually in a larger font in the page header. If we had no thumbs, we may find it convenient to create a separate ordered list containing the first name on each page of the phone book along with the page number. This is how the branch-blocks of an index work; a reduced list that contains the first row of each block plus the address of that block. In a large phone book, this reduced list containing one entry per page will still cover many pages, so the process is repeated, creating the next level up in the index, and so on until we are left with a single page: the root of the tree.

The main benefit of a B-tree is that it allows logarithmic selections, insertions, and deletions in the worst case scenario. And unlike hash indexes it stores the data in an ordered way, allowing for faster row retrieval when the selection conditions include things like inequalities or prefixes.

B-tree indexes can be used effectively for comparisons involving exact or range-based comparisons that use the <, <=, =, >=, >, <>, !=, and BETWEEN operators. B-tree indexes can also be used for LIKE pattern matches if the pattern begins with a literal string rather than a wildcard character.

For example, using the tree above, to get the records for all people younger than 13 requires looking at only the left branch of the tree root.

```
create index age_btx using btree on
person(age);
select id, last_name, age from person where age<13;
/* Afiseaza inregistrari 0 - 29 (11,162 total, Comanda a
durat 0.0002 sec) */
```

Nodes in a B-tree contain a value and a number of pointers to children nodes. For database indexes the “value” is really a pair of values: the indexed field and a pointer to a database row. That is, rather than storing the row data right in the index, you store a pointer to the row on disk. For example, if we have an index on an age column, the value in the B-tree might be something like (34, 0×875900). 34 is the age and 0×875900 is a reference to the location of the data, rather than the data itself. This often allows indexes to be stored in memory even for tables that are so large they can only be stored on disk. Furthermore, B-tree indexes are typically designed so that each node takes up one disk block. This allows each node to be read in with a single disk operation.

MySQL database Servers support b-tree indexes, these indexes are designed to perform very well with OLTP databases with high unique values. Pages in B-tree indexes are known as index nodes. The top level node is called root node, the middle nodes are known as branch nodes / intermediate levels and the bottom level nodes are called leaf nodes (Stephens & Russell, 2004).

The root node contains node pointers to branch nodes / intermediate nodes. Branch node contains pointers to other branch nodes / intermediate nodes or leaf nodes. The bottom nodes known as leaf node contain index items and horizontal pointers to other leaf nodes.

Also, for the pedants among us, many databases use B+ trees rather than classic B-trees for generic database indexes. InnoDB's B-tree index type is closer to a B+ tree than a B-tree, for example.

A B+ tree is in the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is the same length.

Each nonleaf node in the tree has between [n/2] and n children, where n is fixed. B+ trees are good for searches, but cause some overhead issues in wasted space.

A typical node contains up to n – 1 search key values K1, K2,…, Kn-1, and n pointers P1, P2,…, Pn. The search key values are kept in sorted order.

The principal advantage of B+ trees over B trees is they allow you to in pack more pointers to other nodes by removing pointers to data, thus potentially decreasing the depth of the tree.

The disadvantage is that there are no early outs when you might have found a match in an internal node. But since both data structures have huge fanouts, the vast majority of your matches will be on leaf nodes anyway, making on average the B+ tree more efficient.

B+ trees are much easier and higher performing to do a full scan, as in look at every piece of data that the tree indexes, since the terminal nodes form a linked list. To do a full scan with a B-tree you need to do a full tree traversal to find all the data.

B-trees on the other hand can be faster when you do a seek (looking for a specific piece of data by key) especially when the tree resides in RAM or other non-block storage. Since you can elevate commonly used nodes in the tree there are less comparisons required to get to the data.

B+ trees are especially good in block-based storage (eg: hard disk). With this in mind, you get several advantages, for example (from the top of my head):

- high fanout / low depth: that means you have to get less blocks to get to the data. with data intermingled with the pointers, each read gets less pointers, so you need more seeks to get to the data;
- simple and consistent block storage: an inner node has N pointers, nothing else, a leaf node has data, nothing else. That makes it easy to parse, debug and even reconstruct;
- high key density means the top nodes are almost certainly on cache, in many cases all inner nodes get quickly cached, so only the data access has to go to disk.

One possible use of B+ tress is that it is suitable for situations where the tree grows so large that it need not fit into available memory. Thus, it'd generally expect to be doing multiple I/O's. Often it does happen that a B+ tree is used even when it in fact

fits into memory, and then your cache manager might keep it there permanently. But this is a special case, not the general one, and caching policy is a separate from B+ tree maintenance as such.

Also, in a B+ tree, the leaf pages are linked together in a linked list (or doubly-linked list), which optimizes traversals (for range searches, sorting, etc.). So the number of pointers is a function of the specific algorithm that is used.

In B+ tree, since only pointers are stored in the internal nodes, their size becomes significantly smaller than the internal nodes of B-tree (which store both data + key). Hence, the indexes of the B+ tree can be fetched from the external storage in a single disk read, processed to find the location of the target. If it has been a B tree, a disk read is required for each and every decision making process.

2.2. Hash index

They are general purpose indexes. Supported for the MEMORY and NDB storage engines. Formerly, these were known as HEAP tables. Each MEMORY table is associated with one disk file. MEMORY tables use a hashed index and are stored in memory. This makes them very fast, but if MySQL crashes you will lose all data stored in them. MEMORY is very useful for temporary tables. The MySQL internal MEMORY tables use 100% dynamic hashing without overflow areas. There is no extra space needed for free lists. MEMORY tables also don't have problems with delete + inserts, which normally is common with hashed tables.

A Hash index is redundant if each of its columns matches all of the columns in the same order of another Hash index. Another way of putting it is to say that the column lists of these indexes are duplicates of one another. Take the example from table person, finding all people with a last name of 'Stoica'. One solution would be to create a hash table. The keys of the hash would be based off of the last_name field and the values would be pointers to the database row. Most databases support them but they're generally not the default type. We consider a query like this: "Find all people who are younger than 45." Hashes can deal with equality but not inequality. That is, given the hashes of two fields, there's just no way for us to tell which is greater than the other, only whether they're equal or not.

For a hash index, a hash function is applied to each column value. The resulting hash values are stored in the index and used to perform lookups. (A hash function implements an algorithm that is likely to produce distinct hash values for distinct input values. The advantage of using hash values is that they can be compared more efficiently than the original values.) *Hash indexes are very fast for exact-match comparisons performed with the = or <=> operators. But they are poor for comparisons that look for a range of values, as in these expressions: id < 30; age between 10 and 20.*

If it uses a MEMORY table only for exact-value lookups, a hash index is a good choice. This is the default index type for MEMORY tables, so you need do nothing special. If it needs to perform range-based comparisons with a MEMORY table, it should use a B-tree index instead (see Figure 2). To specify this type of index, add USING BTREE to your index definition.

```
create table person_1 (
id int( 9 ) unsigned not null auto_increment primary key ,
age tinyint( 3 ) unsigned not null ,
last_name varchar( 30 ) not null ,
hometown varchar( 30 ) not null ,
gender enum( 'female', 'male' ) not null) engine = memory;

insert into person_1
(`id`,`age`,`last_name`,`hometown`,`gender`) select *
from person;

create index age_hmx using hash on person_1(age);

select id, last_name, age from person_1 where age=13;
/* Records show 0 - 29 (1,016 total, Execution time 0.0002
sec) */

select id, last_name, age from person_1 where age<13;
/* Records show 0 - 29 (11,162 total, Execution time
0.0004 sec) */

create index age_bmx using btree on person_1(age);

select id, last_name, age from person_1 where age=13;
/* Records show 0 - 29 (1,016 total, Execution time 0.0005
sec) */

select id, last_name, age from person_1 where age<13;
/* Records show 0 - 29 (11,162 total, Execution time
0.0002 sec) */
```

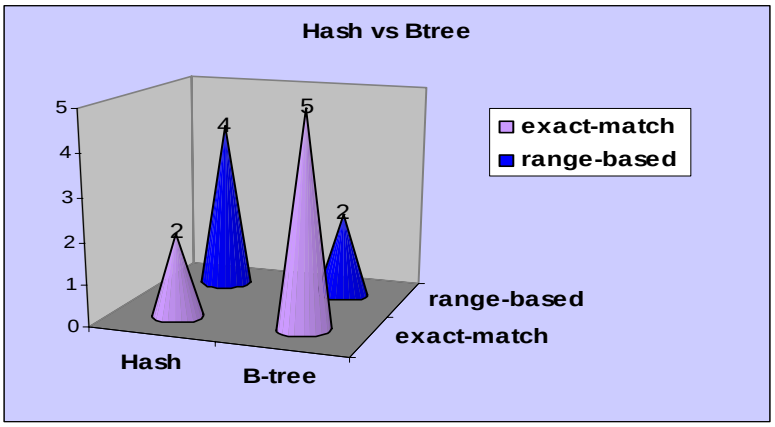

Figure 2. Comparative analysis of hash and b-tree indexes

In brief, hash indexes have somewhat different characteristics:

- They are used only for = or <=> comparisons (but are very fast).

- 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.)
- The hash-index allows you to increase access to columns.
- The hash-index is stored only in operating memory.
- Each time the indexed table is modified, the hash-index is updated automatically.
- If it have a hash index on a MEMORY table that has a high degree of key duplication (many index entries containing the same value), updates to the table that affect key values and all deletes are significantly slower. The degree of this slowdown is proportional to the degree of duplication (or, inversely proportional to the index cardinality). It can use a B-tree index to avoid this problem.

2.3. Bitmap index

A bitmap index is a special kind of index that stores the bulk of its data as bit arrays (bitmaps) and answers most queries by performing bitwise logical operations on these bitmaps. The most commonly used index, such as B+trees, are most efficient if the values it indexes do not repeat or repeat a smaller number of times. In contrast, the bitmap index is designed for cases where the values of a variable repeat very frequently. For example, the gender field in a person database, from our example, usually contains two distinct values: Male or Female. For such variables, the bitmap index can have a significant performance advantage over the commonly used trees. In a Bitmap index, a 2 dimensional array is created. The array represents the index value multiplied by number of rows. One column is allotted for every row in the table being indexed. When a row is retrieved, the bitmap is decompressed into the RAM data buffer to rapidly scan for matching values. Each matching value is returned as a ROW ID which can be used to access the desired information.

In MySQL is not implemented bitmap index, but can be made an analogy with the MySQL ENUM or SET datatype wich are stored as integer value within the MySQL tables. The ENUM or SET elements are stored in the MySQL table as a bitmap: each element is represented by a single bit.

Continuing the gender example, a bitmap index may be logically viewed as in Table 3.

The SET datetype is a string type, but is often referred to as a complex type due to the increased complexity involved in implementing them. A SET datetype can hold any number of strings from a predefined list of strings specified during table creation. The SET datetype is similar to the ENUM datetype in that they both work with predefined sets of strings, but where the ENUM datetype restricts to a single member of the set of predefined strings, the SET datetype allows you to store any of the values together, from none to all of them.

Bitmap indexes might also prove useful for EXISTS and COUNT. Sometimes the user will have to use a hint to "force" bitmap index use. Bitmap indexes would be very useful in a data warehouse environment when many rows will contain the same amount of data (for aggregation), unlike b-tree indexes which are good for unique column values.

Table 3. Logical view for a bitmap index

id	gender	bitmaps	
		F	M
…	…	…	…
1	Female	1	0
2	Male	0	1
3	Male	0	1
4	Female	1	0
5	Male	0	1
6	Male	0	1
…	…	…	…
100	Female	1	0
101	Male	0	1
102	Female	1	0
104	Male	0	1
…	…	…	…
10000	Male	0	1
10001	Female	1	0

MySQL has no bitmap indexes but achieves similar functionality using its "index_merge" feature. Bitmap indexes will be introduced with the Falcon engine.

The Index Merge method is used to retrieve rows with several range scans and to merge their results into one. The merge can produce unions, intersections, or unions-of-intersections of its underlying scans. This access method merges index scans from a single table; it does not merge scans across multiple tables.

On the left, identifier refers to the unique number assigned to each person, gender is the data to be indexed, the content of the bitmap index is shown as two columns under the heading bitmaps. Each column in the above illustration is a bitmap in the bitmap index. In this case, there are two such bitmaps, one for gender Female and one for gender Male. It is easy to see that each bit in bitmap M shows whether a particular row refers to a Male. This is the simplest form of bitmap index.

In our demonstration we have worked on for is the use of indexes in simple (single table) queries. For this demonstration we used the standard schema and a couple of the provided bitmap schema on the person_1 table; that is the GENDER (GENDER_BIT) index.

Firstly, let's look at the data distributions:

```
create index gender_bit on person(gender);
select  count(*) as number_of_persons from person
group by gender;
```

Table 4. Query result

number_of_persons
50381 (Female)
50620 (Male)

There are about 51% Male persons and 49% Female persons. So what happens when we query the person_1 table on gender = Male.

```
select * from person where gender = 'Male';
explain select * from person where gender = 'Male';
/* Records show 0 - 29 (50,620 total, Execution time
0.0010 sec) */
```

Table 5. Explain plan query

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	person	ref	gender_bit	gender_bit	1	const	50620	Using where

```
select  * from person where age=13 and gender='Male'
EXPLAIN select  * from person where age=13 and
gender='Male'
/* Records show 0 - 29 (494 total, Execution time 0.0002
sec) */
```

Table 6. Explain plan query

id	select_ type	table	type	possible_ keys	key	key_ len	ref	rows	extra
1	SIMPLE	person	index_ merge	age_btx, gender_bit	age_btx, gender_bit	1,1	NULL	494	Using intersect(age _btx,gender _bit); Using where

In EXPLAIN output, the Index Merge method appears as index_merge in the type column. In this case, the key column contains a list of indexes used, and key_len contains a list of the longest key parts for those indexes.

Now we know the basics of what indexes are and how they work. Here are few things to keep in mind when creating indexes.

- **Indexes should be created on columns with high cardinality**, the basic rule for B-Tree indexes is minimum 10% before index will be useful. If table have 1000 rows then the column you want to index should have at least 100 different values.
- Index should be created on columns used in WHERE clause, Order By, Group By, Distinct etc, **not columns you only display as output**.

- **All columns used in WHERE clause** should be included in single index to get best results.
- **Do not create indexes unless you need them**, too many indexes will slow INSERT, UPDATE and DELETE.
- **Index short values.** Use smaller data types when possible. Smaller values improve index processing in several ways: Shorter values can be compared more quickly, so index lookups are faster; Smaller values result in smaller indexes that require less disk I/O; With shorter key values, index blocks in the key cache hold more key values. MySQL can hold more keys in memory at once, which improves the likelihood of locating key values without reading additional index blocks from disk. For the InnoDB and BDB storage engines that use clustered indexes, it's especially beneficial to keep the primary key short. A clustered index is one where the data rows are stored together with (that is, clustered with) the primary key values. Other indexes are secondary indexes; these store the primary key value with the secondary index values. A lookup in a secondary index yields a primary key value, which then is used to locate the data row. The implication is that primary key values are duplicated into each secondary index, so if primary key values are longer, the extra storage is required for each secondary index as well.
- **Create indexes on column(s) which are queried frequently**.
- **Index prefixes of string values.** If you're indexing a string column, specify a prefix length whenever it's reasonable to do so. For example, if you have a CHAR(200) column, don't index the entire column if most values are unique within the first 10 or 20 characters. Indexing the first 10 or 20 characters will save a lot of space in the index, and probably will make your queries faster as well. By indexing shorter values, you gain the advantages described in the previous item relating to comparison speed and disk I/O reduction. You want to use some common sense, of course. Indexing just the first character from a column isn't likely to be that helpful because then there won't be very many distinct values in the index.
- **Take advantage of leftmost prefixes.** When it creates an *n*-column composite index, it actually creates *n* indexes that MySQL can use. A composite index serves as several indexes because any leftmost set of columns in the index can be used to match rows. Such a set is called a "leftmost prefix." (This is different from indexing a prefix of a column, which is using the first *n* characters of the column for index values.)
- **Match index types to the type of comparisons you perform.** When it create an index, most storage engines choose the index implementation they will use. For example, InnoDB always uses B-tree indexes. MySQL also uses B-tree indexes, except that it uses R-tree indexes for spatial data types. However, the MEMORY storage engine supports hash indexes and B-tree indexes, and allows you to select which one you want. To choose an index type, consider what kind of comparison operations you plan to perform on the indexed column.

Database indexes are auxiliary data structures that allow for quicker retrieval of data. The most common type of index is a B-tree index because it has very good general performance characteristics and allows a wide range of comparisons, including both equality and inequalities. The penalty for having a database index is the cost

required to update the index, which must happen any time the table is altered. There is also certain about of space overhead, although indexes will be smaller than the table they index.

For specific data types different indexes might be better suited than a B-tree. For fields with only a few possible values bitmap indexes might be appropriate.

REFERENCES:

[1]. Acu, C.I. (2003) *Web pages optimization*, Polirom Publishing House, Iaşi
[2]. Dubois, P. (2008) *MySQL*, 4th Edition, Addison Wesley Professional, Iaşi
[3]. Opell, A. (2006) *SQL fără mistere*, Rosetti Educational, Bucureşti
[4]. Schwarty, B.; Zaitsev, P.; Tkachenko, V.; Zawodny, J.; Lentz, A.; Balling, A. (2004) *High performance MySQL*, O'Reilly Media
[5]. Stephens, J.; Russell, C. (2004) *Beginning MySQL database design and optimization: From novice to professional*, Apress
[6]. Stephens, R. (2009) *Beginning database design solutions*, Wiley Publishing
[7]. Welling, L.; Thomson, L. (2005) *Dezvoltarea aplicațiilor cu PHP şi MySQL*, Teora, Bucureşti
[8]. Williams, H.E.; Lane, D. (2007) *Web database applications with PHP& MySQL*, O'Reilly Media
[9]. MySQL, Available at: http://www.mysql.com