What is the "(scanning)" variant of a loose index scan?

Posted on Sep 26, 2018 by sreeharsha ramanavarapu
Category: Optimizer
Tags: internals, optimizer

A query plan uses loose index scan if "Using index for group-by" appears in the "Extra" column of the EXPLAIN output. In some plans though, "Using index for group-by (scanning)" appears. What does "(scanning)" mean and how is it different from the regular loose index scan?

Loose index scan is mainly used for the GROUP BY / DISTINCT optimization.

Without an index, processing a GROUP BY query would require a temporary table that stores one row per group. With an index, which provides sorting order for GROUP BY columns, all the entries of an index would normally be accessed. Loose index scan avoids accessing all the entries in an index and filters based on the prefix columns.

The optimized approach considers only a fraction of the keys in an index, so it is called a loose index scan. A loose index scan reduces query execution time (aggregate processing), by using the prefix columns in an index.

How Loose Index Scan works

Example

```
mysql> CREATE TABLE t2 (
    ->   pk_col1 INT NOT NULL,
    ->   pk_col2 INT NOT NULL,
    ->   c1 CHAR(64) NOT NULL,
    ->   c2 CHAR(64) NOT NULL,
    ->   PRIMARY KEY(pk_col1, pk_col2),
    ->   KEY c1_c2_idx (c1, c2)
    -> ) ENGINE=INNODB;
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO t2 VALUES (1,1,'a','b'), (1,2,'a','b'),
    ->                       (1,3,'a','c'), (1,4,'a','c'),
    ->                       (2,1,'a','d'), (3,1,'a','b'),
    ->                       (4,1,'d','b'), (4,2,'e','b'),
    ->                       (5,3,'f','c'), (5,4,'k','c'),
    ->                       (6,1,'y','d'), (6,2,'f','b');
Query OK, 12 rows affected (0.01 sec)
Records: 12  Duplicates: 0  Warnings: 0

mysql> EXPLAIN
    -> SELECT c1, MIN(c2)
    -> FROM t2
    -> GROUP BY c1 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
   partitions: NULL
         type: range
possible_keys: c1_c2_idx
          key: c1_c2_idx
      key_len: 256
          ref: NULL
         rows: 7
     filtered: 100.00
        Extra: Using index for group-by
1 row in set, 1 warning (0.00 sec)
mysql>
mysql> SELECT c1, MIN(c2)
    -> FROM t2
    -> GROUP BY c1;
+----+---------+
| c1 | MIN(c2) |
+----+---------+
| a  | b       |
```

Search

Categories

- Backup (3)
- Character Sets (16)
- Connectors / Languages (14)
- Data Dictionary (11)
- Document Store, JSON (32)
- GIS (15)
- High Availability / Replication (182)
- InnoDB (57)
- Miscellaneous (84)
- Monitoring (18)
- NDB (69)
- Optimizer (56)
- Performance (5)
- Router (4)
- Security (57)
- Shell (30)
- Thread Pool (1)
- Upgrading (15)
- Windows / .NET (21)
- Workbench (46)

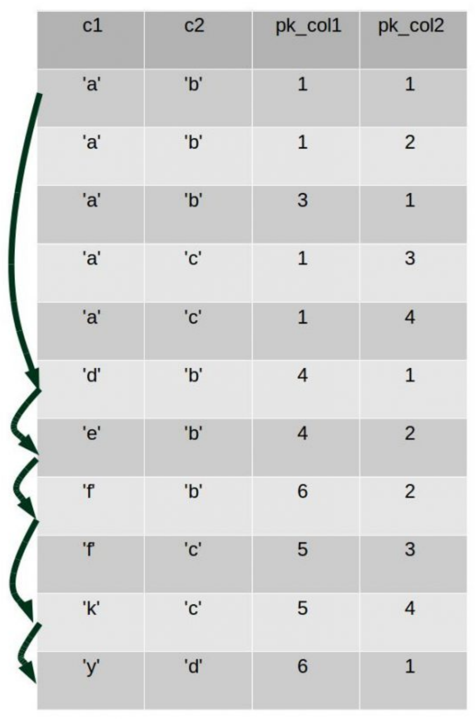
- News Announcements (200)
- Release Announcements (529)

```
| d  | b       |
| e  | b       |
| f  | b       |
| k  | c       |
| y  | d       |
+----+---------+
6 rows in set (0.00 sec)
```

What happens internally

Indexes are ordered based on their keys. Loose index scan effectively jumps from one unique value (or set of values) to the next based on the index's prefix keys. In the above query column "c1" is the prefix of the index "c1_c2_idx". Loose index scan jumps to the unique values of the column "c1" (because grouping is on c1) and picks the lowest value of "c2".

The below diagram is a representation of the index "c1_c2_idx". Note that INNODB extends each secondary index by appending the primary key columns to it. Only the first row in "group" of rows is returned to the server. The "jump" is based on the relevant prefix of the index.



c1	c2	pk_col1	pk_col2
'a'	'b'	1	1
'a'	'b'	1	2
'a'	'b'	3	1
'a'	'c'	1	3
'a'	'c'	1	4
'd'	'b'	4	1
'e'	'b'	4	2
'f'	'b'	6	2
'f'	'c'	5	3
'k'	'c'	5	4
'y'	'd'	6	1

To "jump" values in an index, we use the handler call: `ha_index_read_map(...)`. This is effectively a random disk access to read the next unique value of the index based on the prefix. The important details here is that: **"Determination of the next unique value is handled by storage engine."**

The below function calls `ha_index_read_map(...)` when `is_index_scan` is set to false.

```
static int index_next_different(bool is_index_scan, handler *file,
                                KEY_PART_INFO *key_part, uchar *record,
                                const uchar *group_prefix,
                                uint group_prefix_len, uint group_key_parts) {
  if (is_index_scan) {
    int result = 0;
    while (!key_cmp(key_part, group_prefix, group_prefix_len)) {
      result = file->ha_index_next(record);
      if (result) return (result);
    }
    return result;
  } else
    return file->ha_index_read_map(record, group_prefix,
                                   make_prev_keypart_map(group_key_parts),
                                   HA_READ_AFTER_KEY);
}
```

A special type of Loose Index Scan

Normally a query with AGG(DISTINCT ...), i.e. "SELECT [SUM|COUNT|AVG](DISTINCT...) ...", would require an index or table scan, a temporary table to filter the distinct values and then return the count of all such distinct values.

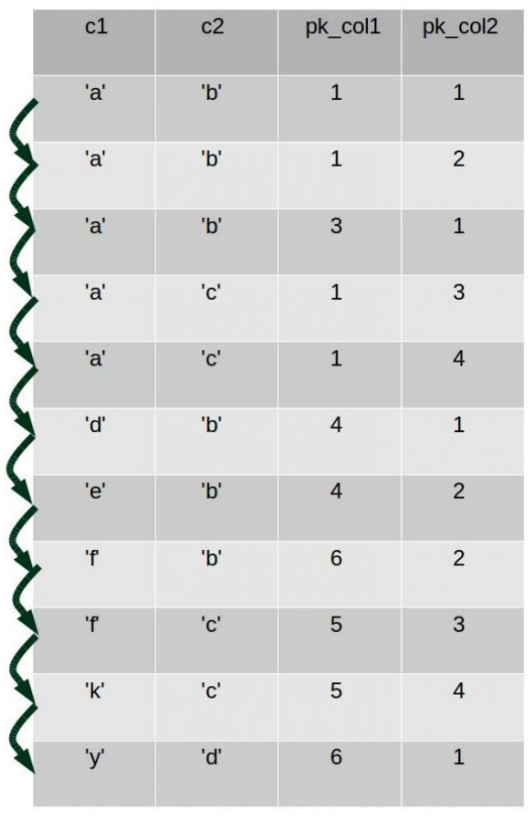
The "scanning" feature is an extension of loose index scan for faster execution of queries with AGG(DISTINCT ...). For such queries the EXPLAIN output's extra column will contain "Using index for group-by (scanning)".

"(scanning)" indicates that loose index scan will perform an index scan instead of random disk access to jump to the next distinct value (using ha_index_read_map(...)) if the cost of using loose index scan is more than that of a regular index scan. Notice in the below example that the cost for the regular index scan (cost of 1.6219) is cheaper than that of the regular loose index scan (cost of 1.75). Since the cost of filtering based on prefix in the storage engine is higher, all the rows are retrieved and filtered in the server. Hence the "(scanning)" variant.

```
mysql> EXPLAIN  SELECT COUNT(DISTINCT c1, c2) FROM t2 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
   partitions: NULL
         type: range
possible_keys: c1_c2_idx
          key: c1_c2_idx
      key_len: 512
          ref: NULL
         rows: 10
     filtered: 100.00
        Extra: Using index for group-by (scanning)
1 row in set, 1 warning (0.00 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.OPTIMIZER_TRACE;
+---------------------------------------------------------------------------------------
| EXPLAIN
SELECT COUNT(DISTINCT c1, c2)
FROM t2 | {
......
                  "best_covering_index_scan": {
                    "index": "c1_c2_idx",
                    "cost": 1.6219,
                    "chosen": true
                  },
                  "group_index_range": {
                    "potential_group_range_indexes": [
                      {
                        "index": "c1_c2_idx",
                        "covering": true,
                        "rows": 10,
                        "cost": 1.75
                      }
                    ],
                    "index_scan": true
                  },
......
      }
    }
  ]
}---------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

In contrast to the previous type of loose index scan, the index "jump" isn't done by the storage engine. See the code above, `is_index_scan` is set to true for this option. Each entry in the index is read (by calling `ha_index_next(..)`) and the executor determines the unique value of the index based on the prefix. **Determination of the next unique value is handled by executor in `key_cmp(..)`.**



c1	c2	pk_col1	pk_col2
'a'	'b'	1	1
'a'	'b'	1	2
'a'	'b'	3	1
'a'	'c'	1	3
'a'	'c'	1	4
'd'	'b'	4	1
'e'	'b'	4	2
'f'	'b'	6	2
'f'	'c'	5	3
'k'	'c'	5	4
'y'	'd'	6	1

The above diagram illustrates that all the rows in the index are brought from the storage engine to the executor.

This was added as part of WL#3220 and isn't a new feature. It has been around since Mysql-5.5.