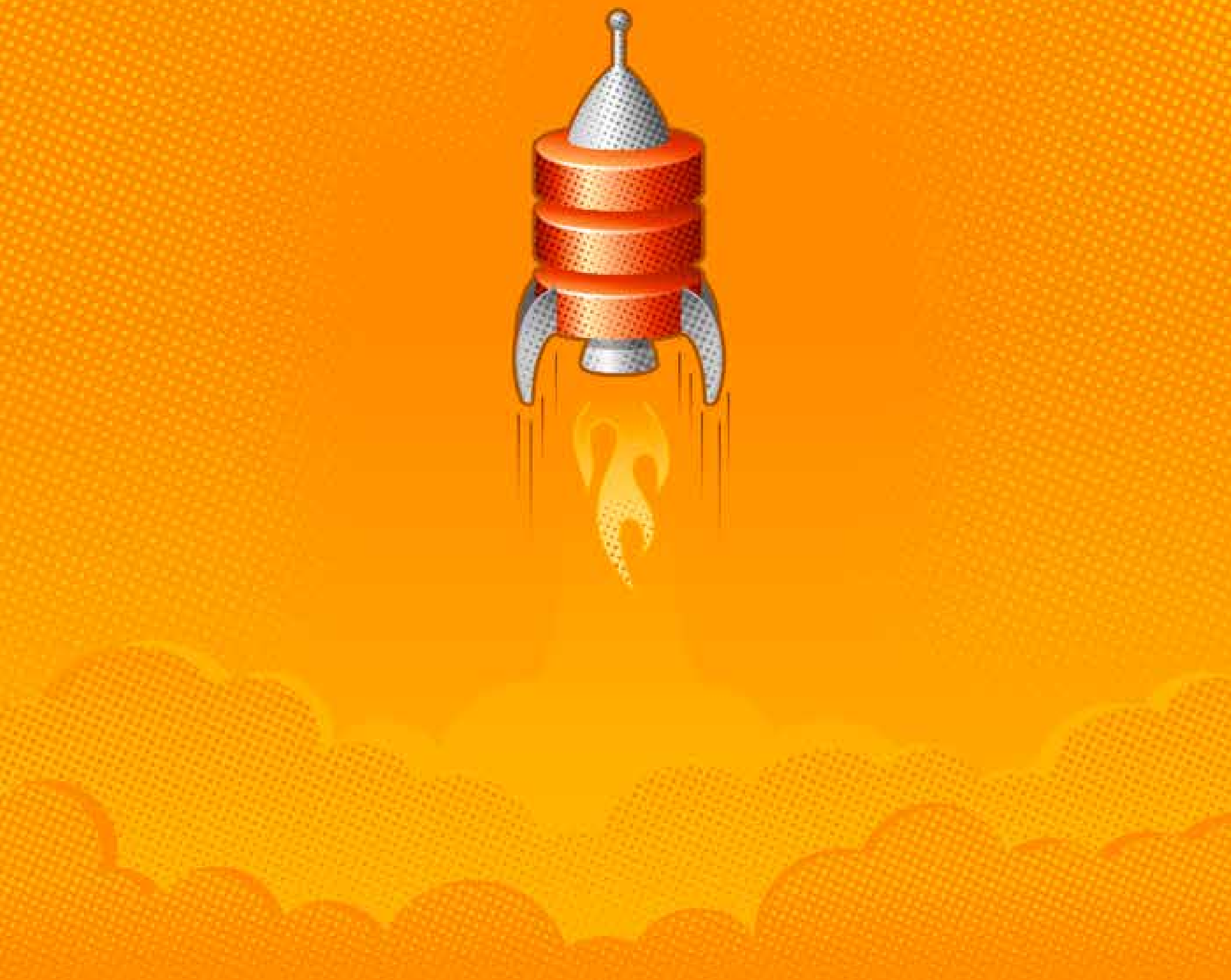


PRACTICAL

MySQL® Performance Optimization

A hands-on, business-case-driven guide to understanding MySQL® query parameter tuning and database performance optimization.

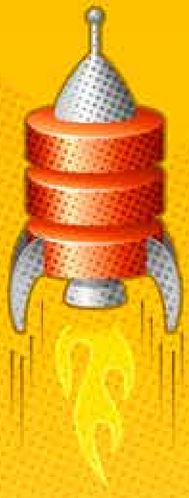
PERCONA

With the increasing importance of applications

and networks in both business and personal interconnections, performance has become one of the key metrics of successful communication. Optimizing performance is key to maintaining customers, fostering relationships, and growing business endeavors.

A central component to applications in any business system is the database, how applications query the database, and how the database responds to requests. MySQL is arguably one of the most popular ways of accessing database information. There are many methods to configuring MySQL that can help ensure your database responds to queries quickly and with a minimum amount of application performance degradation.

Percona is the only company that delivers enterprise-class software, support, consulting, and managed services solutions for both MySQL and MongoDB® across traditional and cloud-based platforms that maximize application performance while streamlining database efficiencies. Our global 24x7x365 consulting team has worked with over 3,000 clients worldwide, including the largest companies on the Internet, who use MySQL, Percona Server®, Amazon® RDS for MySQL, MariaDB® and MongoDB.



This book, co-written by Peter Zaitsev (CEO and co-founder of Percona) and Alex Rubin (Percona consultant, who has worked with MySQL since 2000 as DBA and Application Developer) provides practical hands-on technical expertise to understanding how tuning MySQL query parameters can optimize your database performance and ensure that its performance improves application response times. It also will provide a thorough grounding in the context surrounding what the business case is for optimizing performance, and how to view performance as a function of the whole system (of which the database is one part).

THIS BOOK CONTAINS THE FOLLOWING SECTIONS:

- **Section 1**: *Application Performance and User Perception*
- **Section 2:** *Why is Your Database Performance Poor?*
- **Section 3** *(this section)*: *Advanced MySQL Tuning*

For more information on Percona, and Percona's software and services, visit us at www.percona.com.

ABOUT THE AUTHORS

Peter Zaitsev, CEO

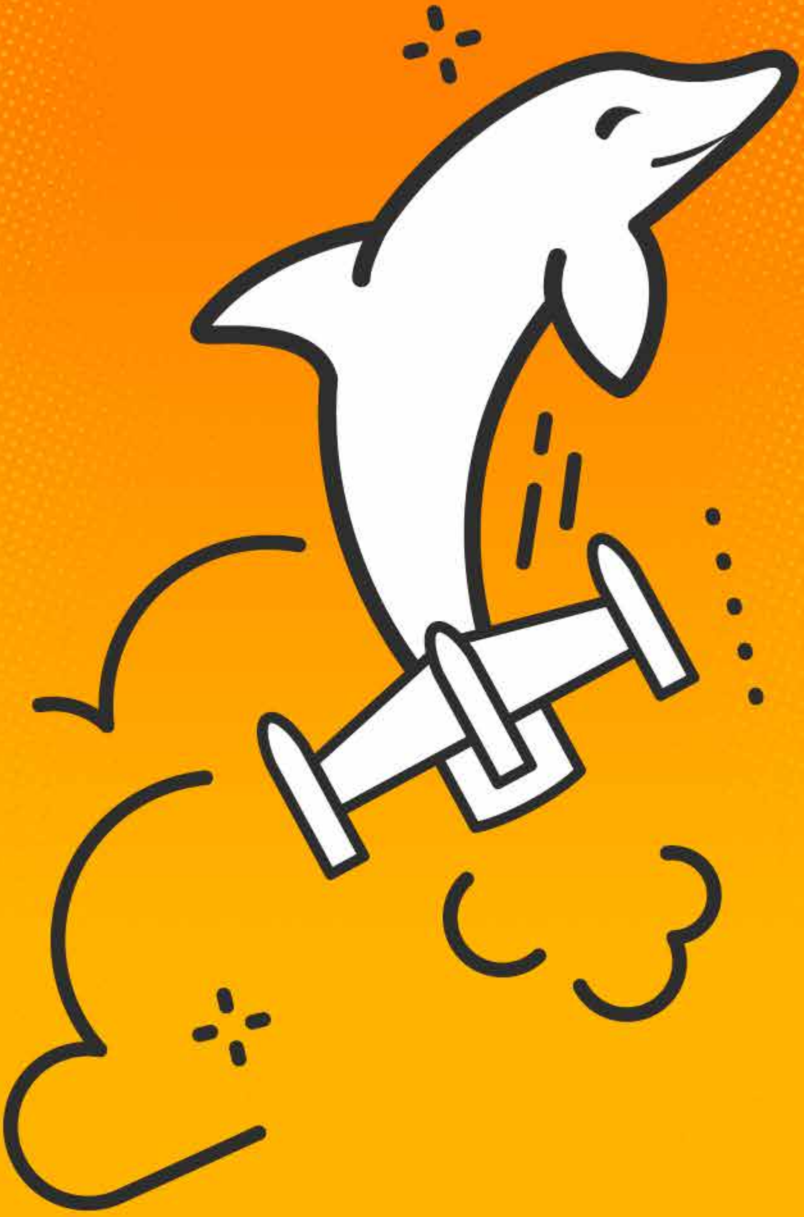
Peter Zaitsev is CEO and co-founder of Percona. A serial entrepreneur, Peter enjoys mixing business leadership with hands-on technical expertise. Previously he was an early employee at MySQL AB, one of the largest open source companies in the world, which was acquired by Sun Microsystems in 2008. Prior to joining MySQL AB, Peter was CTO at SpyLOG, which provided statistical information on website traffic.

Peter is the co-author of the popular book, High Performance MySQL. He has a Master's in Computer Science from Lomonosov Moscow State University and is one of the award-winning leaders of the world MySQL community. Peter contributes regularly to the Percona Performance Blog and speaks frequently at technical and business conferences including the Percona Live series, SouthEast LinuxFest, All Things Open, DataOps LATAM, Silicon Valley Open Doors, HighLoad++ and many more.

Alexander Rubin, Principal Consultant

Alexander joined Percona in 2013. Alexander worked with MySQL since 2000 as a DBA and Application Developer. Before joining Percona he was doing MySQL consulting as a principal consultant for over seven years (started with MySQL AB in 2006, then Sun Microsystems and then Oracle). He helped many customers design large, scalable and highly available MySQL systems and optimize MySQL performance. Alexander also helped customers design big data stores with Apache Hadoop and related technologies.

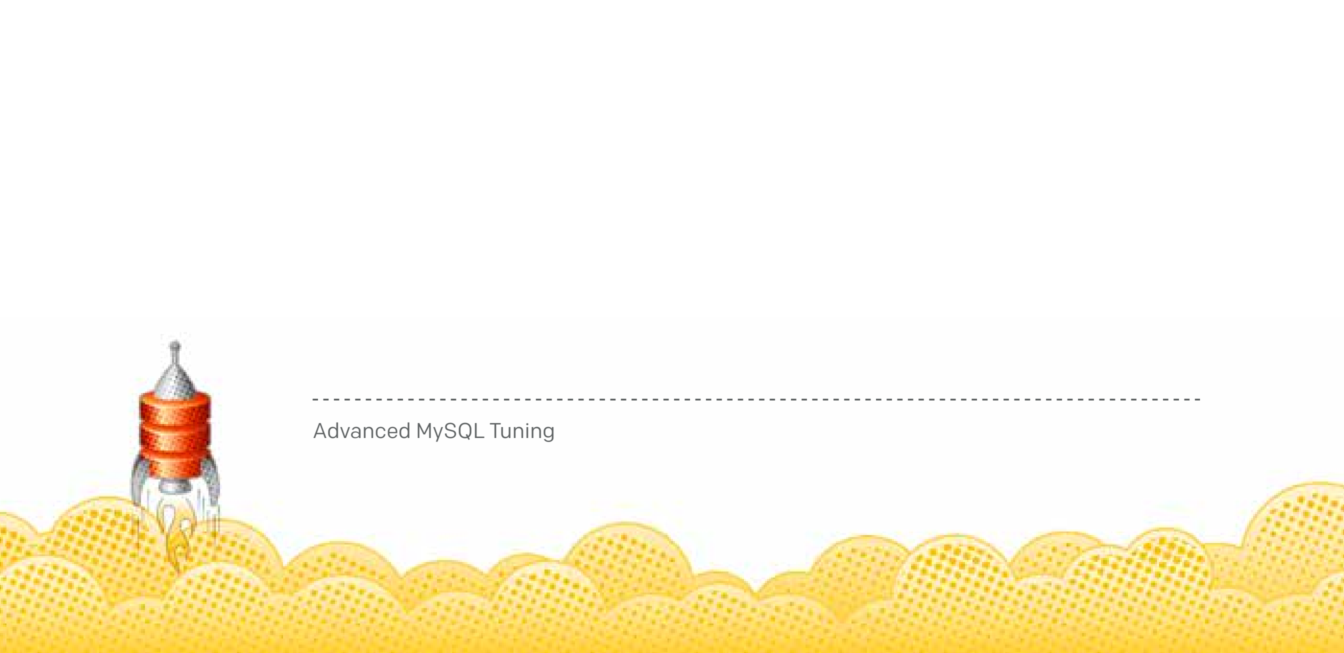
Advanced MySQL Tuning



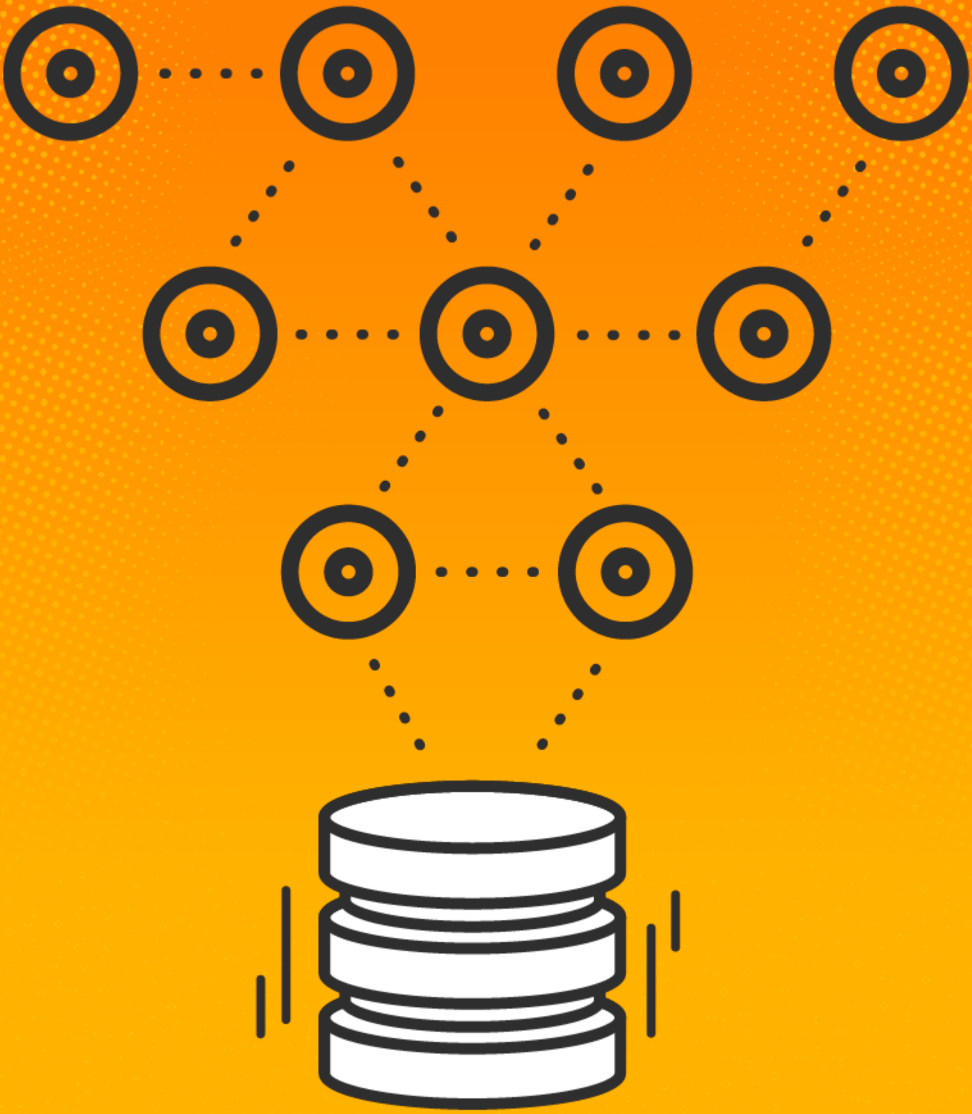
QUERY TUNING

As we can see from the previous chapters, MySQL performance tuning can be critical for business goals and procedures. Query tuning can have a huge impact. In this chapter we will focus on optimizing the typical "slow" queries.

It's important to remember that "slow query" is a relative term. A query can run minutes, but still not affect the normal MySQL functionality or harm the user experience. For example, a backend query that retrieves the latest orders for storage in a data warehousing system can run many minutes and not affect MySQL or end users. At the same time, a really fast query (95th percentile: 1ms) that updates the "user session" table can be really "taxing," as this query can run very frequently with a high level of parallelism.



Indexes



INDEXES

Before starting with query tuning we need to talk first about indexes, and how indexes are implemented in MySQL.

This section will focus on B-tree indexes only (InnoDB and MyISAM only support B-tree indexes). Figure 1 below illustrates a basic B-tree index implementation.

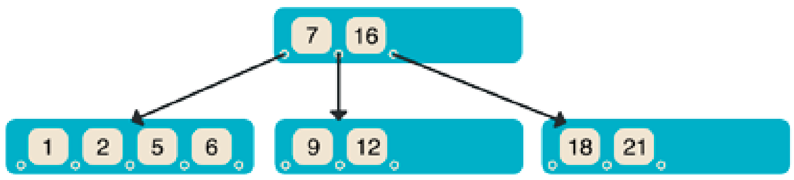


Figure 1: B-Tree Example

B-tree supports both an "equality" (where ID = 12) and a "range" (two examples: where date > "2013-01-01" and date < "2013-07-01", and another example would be id in (6, 12, 18)) search. Figures 2 and 3 illustrate examples of these.

Figure 2 shows an equality search, with a primary or unique key:

```
select * from table where id = 12
```

In this scenario, MySQL is able to scan through the tree and go directly to one leaf and then stop. A primary key search is the fastest index scan operation.

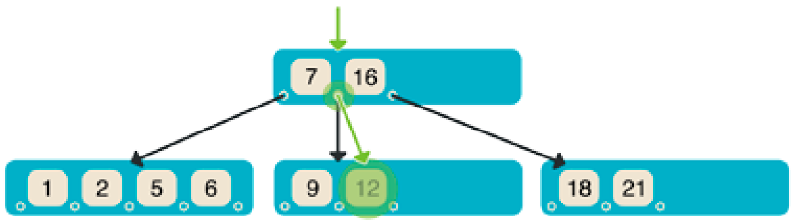


Figure 2: MySQL Primary Key Search

In InnoDB, primary key searches are even faster. InnoDB "clusters" record around the primary key.

```
select * from table where id in (6, 12, 18)
```

In this scenario, MySQL will scan through the tree and visit many leafs/nodes:

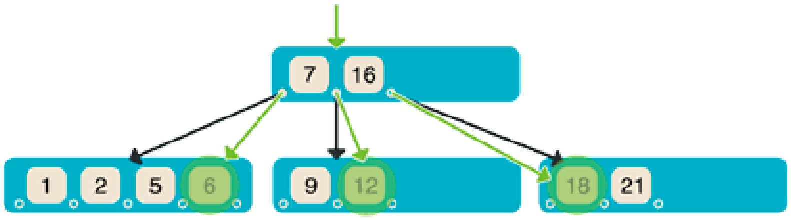


Figure 3: InnoDB Primary Key Search

Table for the Tests

In the next section, I will use a couple of tables for the tests. The first table is the part of the MySQL "world" test database, and can be downloaded from dev.mysql.com.

```
CREATE TABLE City (
    ID int(11) NOT NULL AUTO_INCREMENT,
    Name char(35) NOT NULL DEFAULT ,
    CountryCode char(3) NOT NULL DEFAULT ,
    District char(20) NOT NULL DEFAULT ,
    Population int(11) NOT NULL DEFAULT 0,
    PRIMARY KEY (ID),
    KEY CountryCode (CountryCode)
) Engine=InnoDB;
```

Explain Plan

The main way to "profile" a query with MySQL is to use "explain". The set of operations that the optimizer chooses to perform the most efficient query is called the "query execution plan," also known as the "explain plan." The output below shows an example of the explain plan.

```
mysql> EXPLAIN select * from City where Name =
'London'\G
**************** 1. row ***********
           id: 1
  select_type: SIMPLE
        table: City
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4079
        Extra: Using where
```

As we can see from the explain plan, MySQL does not use any indexes (key: NULL) and will have to perform a full table scan.

In this case, we can add an index to restrict the number of rows:

```
mysql> alter table City add key (Name);
Query OK, 4079 rows affected (0.02 sec)
Records: 4079  Duplicates: 0  Warnings: 0
```

The new explain shows that MySQL will use an index:

```
mysql> explain select * from City where Name =
'London'\G
*****************1. row**********************
           id: 1
  select_type: SIMPLE
        table: City
         type: ref
possible_keys: Name
          key: Name
      key_len: 35
          ref: const
         rows: 1
        Extra: Using where
```

Index Usages

MySQL will choose only one index per query (and per table if the query joins multiple tables). In some cases, MySQL can also intersect indexes (we won't cover that scenario in this section, however). MySQL uses index statistics to make a decision about the best possible index.

Combined Indexes

Combined indexes are very important for MySQL query optimizations. MySQL can use the leftmost part of any index. For example, if we have this index:

```
Comb(CountryCode, District, Population)
```

Then MySQL can use:

```
CountryCode only part
CountryCode + District
CountryCode + District + Population
```

From the explain plan shown below, we can understand which part(s) get used:

```
mysql> explain select * from City
      where CountryCode = 'USA'\G
***************** 1. row *****************
        table: City
         type: ref
possible_keys: comb
          key: comb
      key_len: 3
          ref: const
         rows: 273
```

Note the key_len part shows "3". This is the number of bytes used from our index. As the CountryCode field is declared as char(3), that means that MySQL will use the first field from our combined index.

Similarly, MySQL can use the two leftmost fields, or all three fields from the index. In this example, the two leftmost fields are used:

```
mysql> explain select * from City
where CountryCode = 'USA' and District =
'California'\G
*******************1. row *****************
        table: City
         type: ref
possible_keys: comb
          key: comb
      key_len: 23
          ref: const,const
         rows: 68
```

So MySQL will use the two first fields from the comb key:

» *CountryCode = 3 chars*

» *District = 20 chars*

» *Total = 23*

Whereas in this explain plan, all three fields are used from the index:

```
mysql> explain select * from City
where CountryCode = 'USA' and District =
'California'
and population > 10000\G
*******************1. row *****************
        table: City
         type: range
possible_keys: comb
          key: comb
      key_len: 27
          ref: NULL
         rows: 6
```

- *CountryCode = 3 chars/bytes*
- *District = 20 chars/bytes*
- *Population = 4 bytes (INT)*
- *Total = 27*

However, if the query does not have the first leftmost part of an index, MySQL can't use it:

```
mysql> explain select * from City where
District = 'California' and population >
10000\G
********************1. row ******************
        table: City
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3868
```

As we do not have the CountryCode in the where clause, MySQL will not be able to use our "comb" index.

If you are too tired to do the math for calculating the actual fields in the index, MySQL 5.6 and 5.7 has a great feature: explain format=JSON:

```
mysql> explain format=JSON select * from
City  where CountryCode = 'USA' and District =
'California'\G
***************1. row ***********************
EXPLAIN: {
  "query_block": {
    "select_id": 1,
    "table": {
       "table_name": "City",
       "access_type": "ref",
       "possible_keys": [
       "comb"
       ],
```

```
            "key": "comb",
             "used_key_parts": [
             "CountryCode",
             "District"
             ],
             "key_length": "23",
             "ref": [
             "const",
             "const"
             ],
             "rows": 84,
        ...
```

With the JSON format, MySQL shows the list of fields inside the index that MySQL will use.

Covered Index

The covered index is an index that covers all fields in the query. For example, for this query:

```
select name from City  where CountryCode =
'USA' and District = 'Alaska' and population >
10000
```

the following index will be a "covered" index:

```
cov1(CountryCode, District, population, name)
```

The above index uses all fields in the query, in this order:

- » *Where part*
- » *Group By/Order (not used now)*
- » *Select part (here: name)*

Here is an example:

```
mysql> explain select name from City  where
CountryCode = 'USA' and District = 'Alaska'
and population > 10000\G
******************* 1. row ****************
        table: City
         type: range
possible_keys: cov1
          key: cov1
      key_len: 27
          ref: NULL
         rows: 1
        Extra: Using where; Using index
```

The extra field here is an informational field inside the explain output that shows additional information (very useful in most cases) about the explain plan and how MySQL will execute the query.

The "Using index" (informational flag) in the extra field of the explain output means that MySQL will use our covered index. That also means that MySQL will use only that index to satisfy the query (and does not have to read the data from the table).

From the previous example, we can see that we need to filter by: CountryCode, District and population. In addition, we will need to retrieve "name". All those fields are already in our "cov1" index, so MySQL can read the index and return the result without touching the data file. That is usually much faster, especially if we have a lots of other fields in our table. This should improve query performance.

Order of the Fields in Index

The order of the fields in the index is very important. The way B-tree works, it is better to have the "equality" comparison field first and the fields with a "range" (more than and less than comparisons) second.

For example, take the following query:

```
select * from City where district =
'California' and population > 30000
```

The best index will be on (district, population), in this particular order.

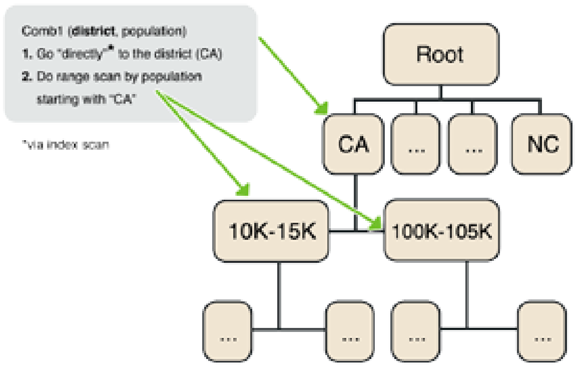

Comb1 (district, population)
1. Go "directly"* to the district (CA)
2. Do range scan by population starting with "CA"
*via index scan
Root
CA
...
...
NC
10K-15K
100K-105K
...
...
...
...

Figure 4: Index for District, Population

In this case, MySQL can go "directly" (via index scan) to the correct district ("CA") and do a range scan by population. All other nodes for the "district" field (other US states in this example) will not be scanned. If we put population first and district second, MySQL will need to perform more work:

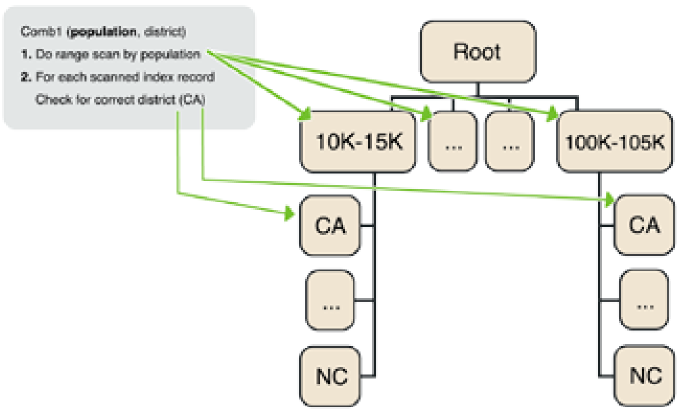
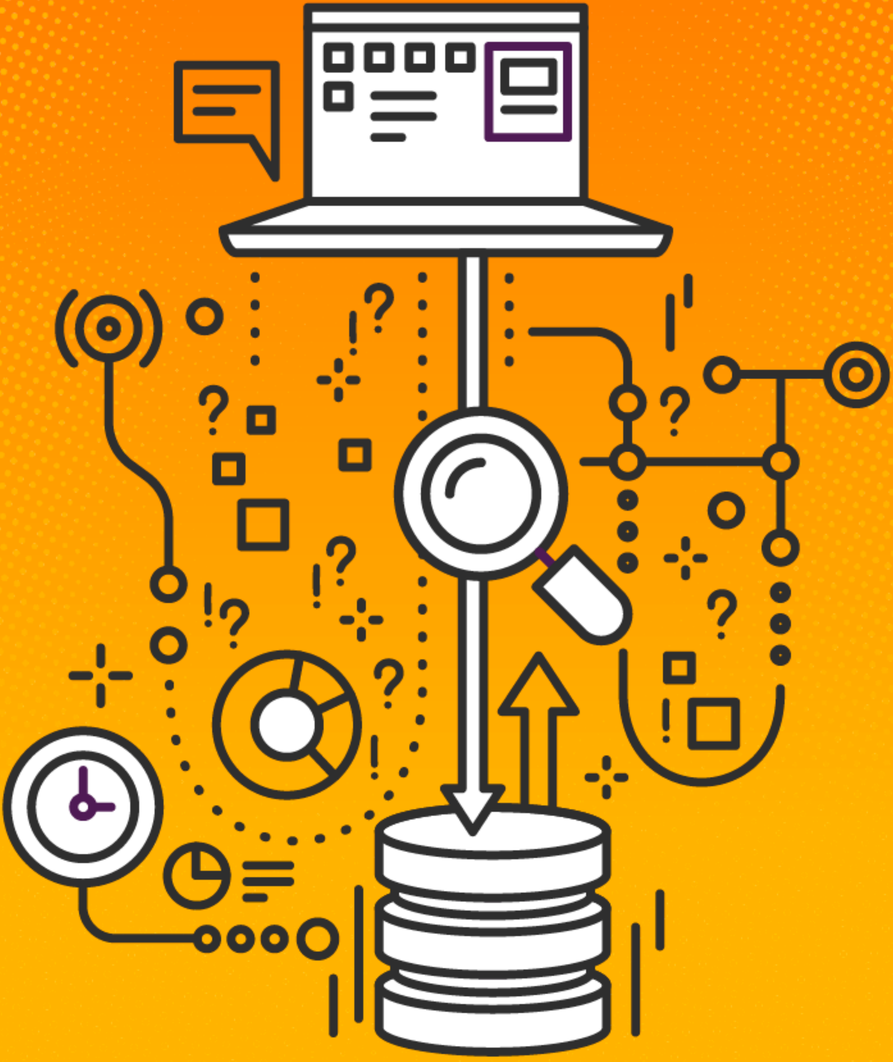

Comb1 (population, district)
1. Do range scan by population
2. For each scanned index record
Check for correct district (CA)
Root
10K-15K
...
...
100K-105K
CA
...
NC
CA
...
NC

Figure 5: Index for Population, District

In this example, MySQL must do a "range" scan for the population, and for each index record, will have to check for the correct district. This is much slower.

Queries



COMPLEX SLOW QUERIES

In this section we will be talking about two major query types:

- *Queries with "group by"*
- *Queries with "order by"*

These queries are usually the slowest ones. We will show how to optimize these queries and decrease the query response time, as well as the application performance in general.

"Group by" Example

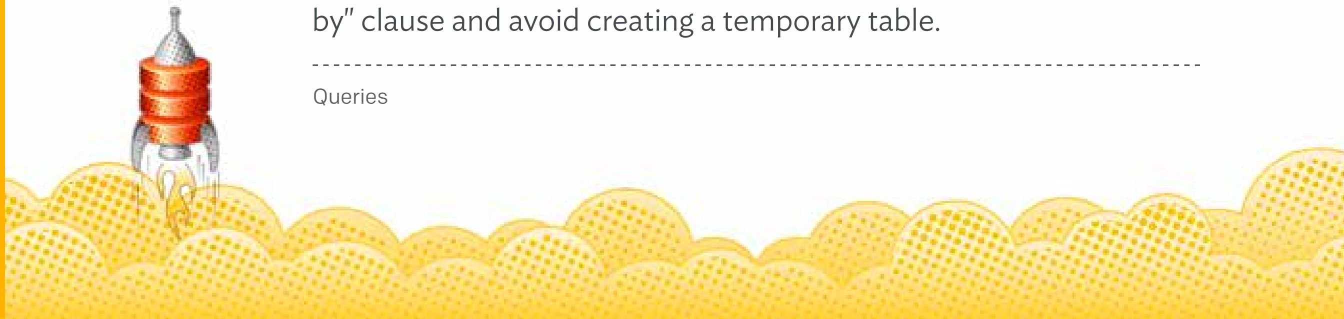
Let's look at this simple example, a query of how many cities there are in each country.

```
mysql> explain select CountryCode, count(*)
from City group by CountryCode\G
           id: 1
  select_type: SIMPLE
        table: City
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4079
        Extra: Using temporary; Using filesort
```

As we can see in the output above, MySQL does not use any indexes (no proper indexes are available), but it also shows "Using temporary; Using filesort". MySQL will need to create a temporary table to satisfy the "Group by" clause if there is no appropriate index.

(You can find more information about the temporary tables here.)

However, MySQL can use a combined index to satisfy the "group by" clause and avoid creating a temporary table.



"Group by" and Covered Indexes

To illustrate the "group by" queries, I will use the following table:

```
CREATE TABLE ontime_2012 (
    YearD int(11) DEFAULT NULL,
    MonthD tinyint(4) DEFAULT NULL,
    DayofMonth tinyint(4) DEFAULT NULL,
    DayOfWeek tinyint(4) DEFAULT NULL,
    Carrier char(2) DEFAULT NULL,
    Origin char(5) DEFAULT NULL,
    DepDelayMinutes int(11) DEFAULT NULL,
    ...
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

The table contains freely available airline performance statistics. The table is six million rows and approximately 2GB in size.

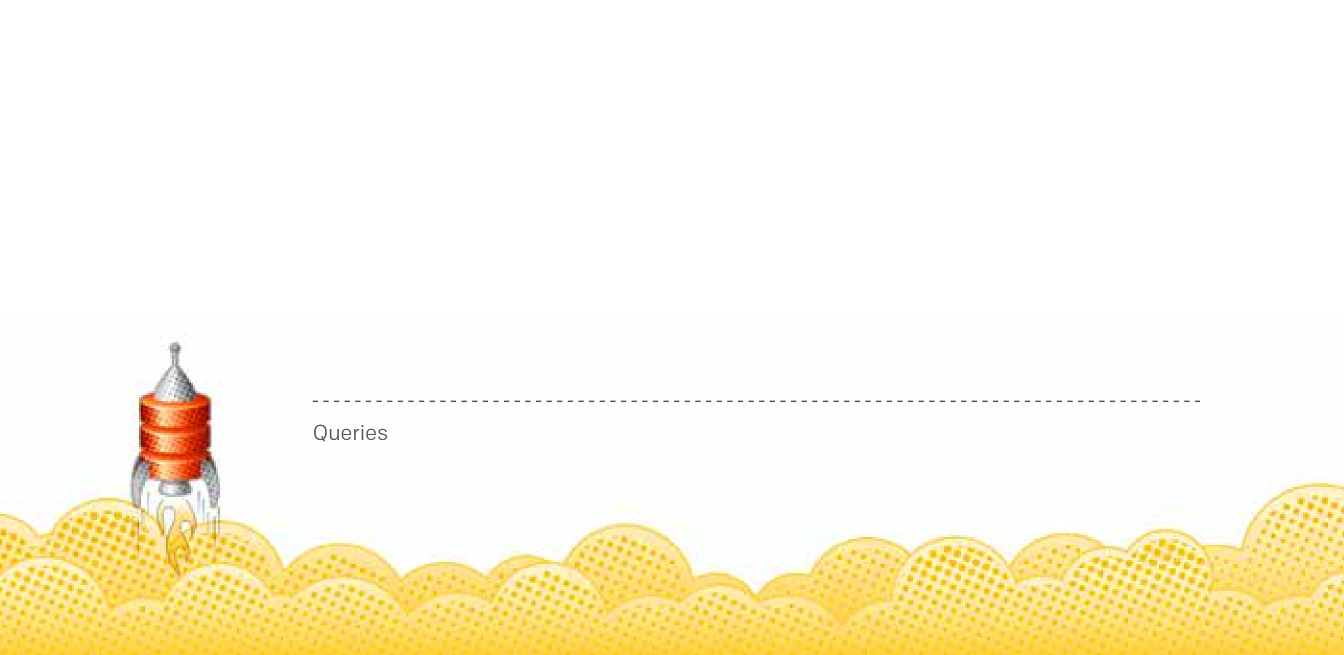
From this data we want to:

» *Find maximum delay for flights on Sunday*

» *Group by airline*

Our example query is:

```
select max(DepDelayMinutes),
carrier, dayofweek
from ontime_2012
where dayofweek = 7
group by Carrier
```



The explain plan is:

```
explain select max(DepDelayMinutes),
carrier, dayofweek
from ontime_2012
where dayofweek = 7
group by Carrier
...
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4833086
        Extra: Using where; Using temporary;
               Using filesort
```

As we can see, MySQL does not use any index and has to scan ~4M rows. In addition, it will have to create a large temporary table.

Creating a "dayofweek" index will only filter out some rows, and MySQL will still need to create a temporary table:

```
mysql> alter table ontime_2012 add key
(dayofweek);
mysql> explain select max(DepDelayMinutes),
Carrier, dayofweek from ontime_2012 where
dayofweek =7
group by Carrier\G
         type: ref
possible_keys: DayOfWeek
          key: DayOfWeek
      key_len: 2
          ref: const
         rows: 817258
        Extra: Using where; Using temporary;
               Using filesort
```

However, we can create a covered index on "dayofweek", "Carrier" and "DepDelayMinutes", in this particular order. In this case, MySQL can use this index and avoid creating a temporary table:

```
mysql> alter table ontime_2012
add key covered(dayofweek, Carrier,
DepDelayMinutes);
explain select max(DepDelayMinutes), Carrier,
dayofweek from ontime_2012 where dayofweek =7
group by Carrier \G
...
possible_keys: DayOfWeek,covered
          key: covered
      key_len: 2
          ref: const
         rows: 905138
        Extra: Using where; Using index
```

As we can see from the explain, MySQL will use our index and will avoid creating a temporary table. This is the fastest possible solution.

Note that MySQL will also be able to use a non-covered index on (dayofweek, Carrier) and avoid creating a temporary table. However, a covered index will be faster as MySQL is able to satisfy the whole query by just reading the index.

"Group by" and a Range Scan

The covered index works well if we have a "const" (where dayofweek=N). However, MySQL will not be able to use an index and avoid a filesort if we have a "range" scan in the "where" clause. Here's an example:

```
mysql> explain select max(DepDelayMinutes),
carrier, dayofweek from ontime_2012
where dayofweek > 5 group by Carrier,
dayofweek\G
...
         type: range
possible_keys: covered
          key: covered
      key_len: 2
          ref: NULL
         rows: 2441781
        Extra: Using where; Using index; Using
               temporary; Using filesort
```

MySQL will still have to create a temporary table. To fix that we can use a simple trick and rewrite the query into two parts with UNION:

```
(select max(DepDelayMinutes), Carrier,
dayofweek
from ontime_2012
where dayofweek = 6
group by Carrier, dayofweek)
union
(select max(DepDelayMinutes), Carrier,
dayofweek
from ontime_2012
where dayofweek = 7
group by Carrier, dayofweek)
```

For each of the two queries in the union, MySQL will be able to use an index instead of creating a temporary table, as shown in the explain plan below:

```
************** 1. row ***********************
        table: ontime_2012
          key: covered
...
        Extra: Using where; Using index
************** 2. row ***********************
        table: ontime_2012
          key: covered
...
        Extra: Using where; Using index
************** 3. row ***********************
           id: NULL
  select_type: UNION RESULT
        table: 
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Using temporary
```

As we can see, MySQL uses our covered index for each of the two queries. It will still have to create a temporary table to merge the results. However, it will probably be a much smaller temporary table as it will only need to store the result sets of two queries.

Please note: IN condition is also an example of range scan (some examples are listed here: http://dev.mysql.com/doc/refman/5.7/en/range-optimization.html), so converting the original query to "where dayofweek in (6,7)" will not help.

```
    mysql> explain
        -> select max(DepDelayMinutes), Carrier,
        ->     dayofweek
        ->     from ontime
        ->     where dayofweek in (6,7)
        ->     group by Carrier, dayofweek\G
    *********** 1. row *************************
               id: 1
      select_type: SIMPLE
            table: ontime
             type: range
    possible_keys: DayOfWeek
              key: DayOfWeek
          key_len: 2
              ref: NULL
             rows: 79882092
            Extra: Using index condition; Using
                   temporary; Using filesort
                   1 row in set (0.00 sec)
```

"Group by" and a Loose Index Scan

A "loose index scan" is another MySQL algorithm that can be used to optimize Group By queries. Loose index scans consider only a fraction of the keys in an index, and therefore are very fast. The following limitations apply:

- *The query is over a single table.*
- *The GROUP BY names only columns that form a leftmost prefix of the index and no other columns.*
- *The only aggregate functions used in the select list (if any) are MIN() and MAX(), same column*

(More information can be found in MySQL documentation on Loose Index Scan.)

For a loose index scan to work, we will need to create an additional index. It should start with columns in the "Group by" clause, and then all fields in the "where" clause (the order of the fields in the index matters). For example, for our query:

```
select max(DepDelayMinutes) as ddm, Carrier,
dayofweek from ontime_2012  where dayofweek =
5  group by Carrier, dayofweek
```

We will need to create this index:

```
KEY loose_index_scan
(Carrier,DayOfWeek,DepDelayMinutes)
```

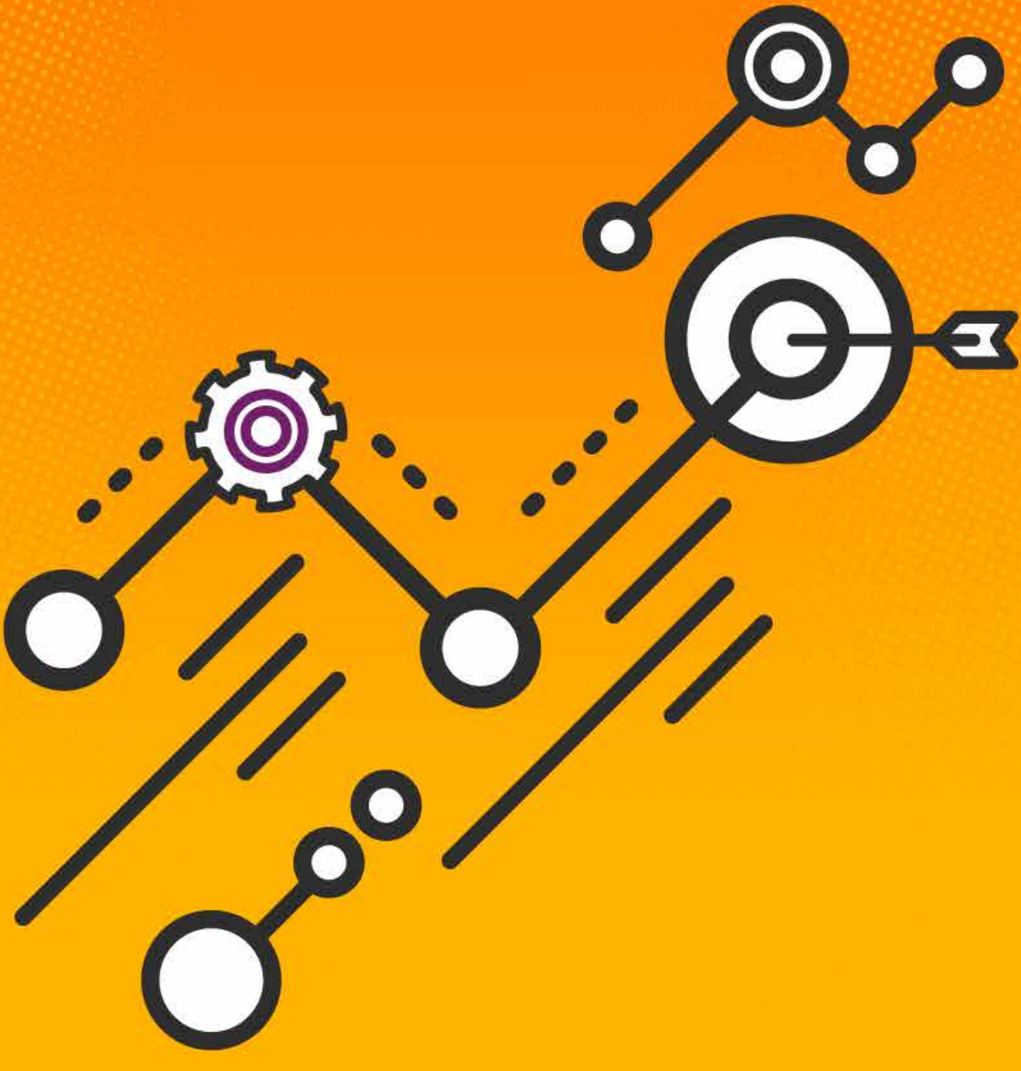
Note that loose_index_scan is a placeholder for the name of the index. It can be any name:

```
mysql> explain select max(DepDelayMinutes)
as ddm, Carrier, dayofweek from ontime_2012
where dayofweek = 5  group by Carrier,
dayofweek
        table: ontime_2012
         type: range
possible_keys: covered
          key: loose_index_scan
      key_len: 5
          ref: NULL
         rows: 201
        Extra: Using where; Using index
               for group-by
```

"Using index for group-by" in the where clause means that MySQL will use the loose index scan. A loose index scan is very fast as it only scans a fraction of the key. It will also work with the range scan:

```
mysql> explain select max(DepDelayMinutes) as
ddm, Carrier, dayofweek from ontime_2012
where dayofweek > 5 group by Carrier,
dayofweek;
table: ontime_2012
         type: range
possible_keys: covered
          key: loose_index_scan
      key_len: 5
          ref: NULL
         rows: 213
        Extra: Using where; Using index
               for group-by;
```

Benchmarks



BENCHMARK

Below is a benchmark that compares query speed with a temporary table, a tight index scan (covered index) and a loose index scan. The table is six million rows and approximately 2GB in size:

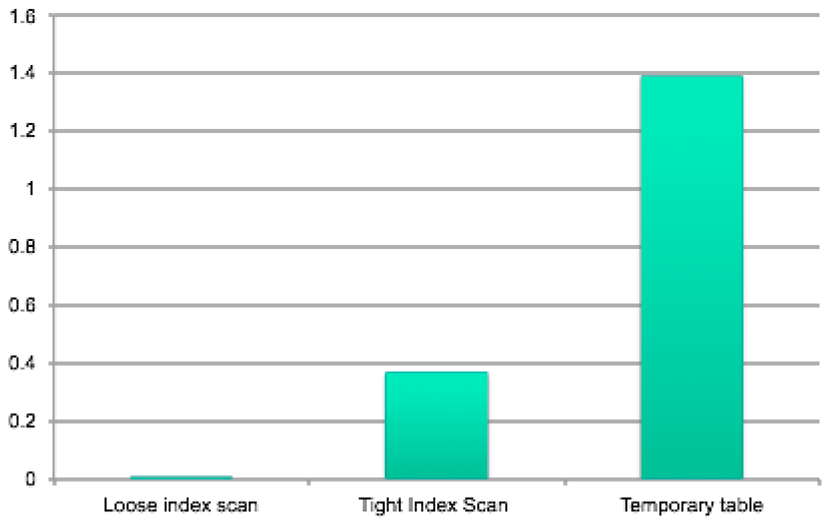


Figure 6: Query Speed (in milliseconds)

As we can see, the loose index scan index shows the best performance. (The smaller the better, the response time (Y-axis) is in seconds.) Unfortunately, loose index scans only work with two aggregate functions, "MIN" and "MAX" for the GROUP BY. "AVG" together with the GROUP BY does not work with a loose index scan. As we can see below, MySQL will use a covered index (not loose_index_scan index) and, because we have a range in the where clause (dayofweek > 5), it will have to create a temporary table.

```
mysql> explain select avg(DepDelayMinutes) as
ddm, Carrier, dayofweek from ontime_2012 where
dayofweek >5  group by Carrier, dayofweek\G
       table: ontime_2012
        type: range
         key: covered
     key_len: 2
         ref: NULL
        rows: 2961617
       Extra: Using where; Using index;
              Using temporary; Using filesort
```

ORDER BY and Filesort

MySQL may have to perform a "filesort" operation when a query uses the "order by" clause.

(You can find more information about "filesort" here: http://dev.mysql.com/doc/refman/5.5/en/order-by-optimization.html.)

The filesort operation below is usually a fairly slow operation (even if it does not involve creating a file on disk), especially if MySQL has to sort a lot of rows:

```
        table: City
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4079
        Extra: Using where; Using filesort
```

To optimize this query, we can use a combined index:

```
mysql> alter table City
add key my_sort2 (CountryCode, population);
mysql> explain select district, name,
population from City where CountryCode = 'USA'
order by population desc limit 10\G
       table: City
        type: ref
         key: my_sort2
     key_len: 3
         ref: const
        rows: 207
       Extra: Using where
```

MySQL was able to use our combined index to avoid sorting: as the index is sorted, MySQL was able to read the index leafs in the correct order.

Sorting and Limit

If we have a "LIMIT" clause in our query, and the limit is relatively small (i.e., LIMIT 10 or LIMIT 100) compared to the amount of rows in the table, MySQL can avoid using a filesort and can use an index instead.

Here's an example:

```
mysql> alter table ontime_2012 add key
(DepDelayMinutes);
```

We can create an index on DepDelayMinutes fields only, and run the explain below (note the query with LIMIT 10):

```
mysql> explain select * from ontime_2012
where dayofweek in (6,7) order by
DepDelayMinutes desc limit 10\G
        type: index
possible_keys: DayOfWeek,covered
         key: DepDelayMinutes
     key_len: 5
         ref: NULL
        rows: 24
       Extra: Using where
```

As we can see, MySQL uses an index on DepDelayMinutes only. Here is how it works. As Index is sorted, MySQL can:

- *Scan the whole table in the order of the index*
- *Filter the results (using the "where" clause condition, but not using the index)*
- *Stop after finding ten rows matching the "where" condition*

This can be very fast if MySQL finds the rows right away. For example, if there are a lot of rows matching our condition (dayofweek in (6,7)), MySQL will find ten rows quickly. However, if there are no rows matching our condition (i.e., empty result set, all rows filtered out), MySQL will have to scan the whole table using an index scan. If we have two indexes, a covered index and an index on the "ORDER BY" field only, MySQL usually is able to figure out the best possible index to use.

Using "ORDER BY" + limit optimization can help optimize your queries.

Calculated Expressions and Indexes

Let us look at another query example:

```
SELECT carrier, count(*)
FROM ontime
WHERE year(FlightDate) = 2013
group by carrier\G
```

The explain plan:

```
mysql> EXPLAIN SELECT carrier, count(*) FROM
ontime
      WHERE year(FlightDate) = 2013 group by
carrier\G
************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: ontime
         type: ALL
possible_keys: NULL
          key: NULL
      Key_len: NULL
          ref: NULL
         rows: 151253427
       Extra: Using where; Using temporary;
Using filesort
Results:
16 rows in set (1 min 49.48 sec)
```

The reason why MySQL is not using an index here is year (FlightDate). "year" is a MySQL function, so MySQL can't just compare the field (FlightDate) to a constant (2013). Instead MySQL will have to "calculate" an expression first (by applying the year function to the field's value, and then comparing it to "2013").

To fix this particular query we can re-write it to use a "range":

```
mysql> SELECT carrier, count(*)
    FROM ontime
    WHERE FlightDate >= '2013-01-01' and
    FlightDate < '2014-01-01'
    group by carrier\G
```

This type of comparison (range of dates) will allow MySQL to use an index:

```
mysql> EXPLAIN SELECT carrier, count(*) FROM
ontime
       WHERE  FlightDate between '2013-01-01'
       and '2014-01-01'
       GROUP BY carrier\G
******************** 1. row *****************
           id: 1
  select_type: SIMPLE
        table: ontime
         type: range
possible_keys: FlightDate
          key: FlightDate
      key_len: 4
          ref: NULL
          rows: 10434762
         Extra: Using index condition; Using
temporary; Using filesort
Results:
16 rows in set (11.98 sec)
```

This is almost ten times faster (12 seconds compared to 1 minute 50 seconds). However, in some cases it can be hard to implement converting the date function to a range. For example:

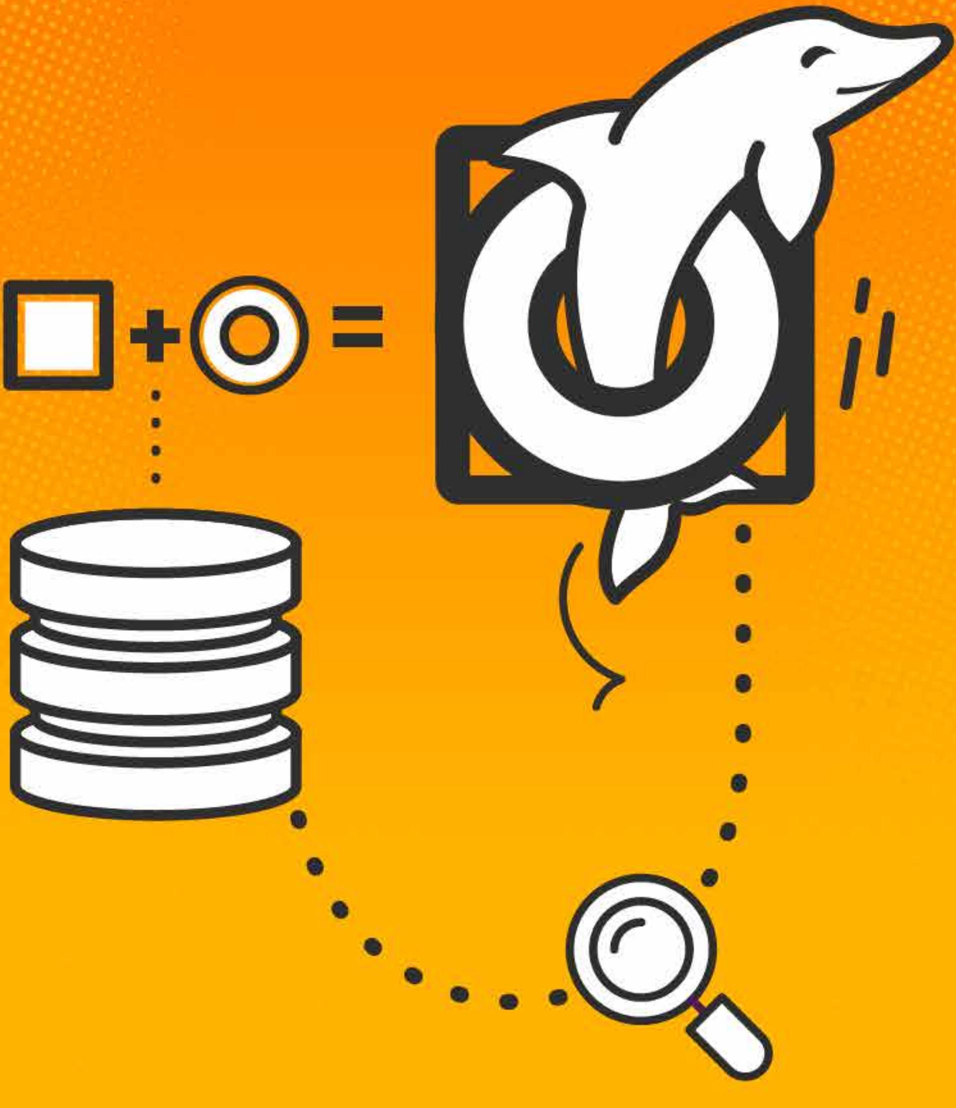
```
      SELECT carrier, count(*) FROM ontime
      WHERE dayofweek(FlightDate) = 7 group by
       carrier
mysql> EXPLAIN SELECT carrier, count(*) FROM
ontime
       WHERE dayofweek(FlightDate) = 7 group
by carrier\G
**************** 1. row *********************
            id: 1
  select_type: SIMPLE
        table: ontime
         type: ALL
possible_keys: NULL
          key: NULL
       key_len: NULL
           ref: NULL
          rows: 151253427
         Extra: Using where; Using temporary;
Using filesort
Results:
32 rows in set (1 min 57.93 sec)
```

Similar to the previous example, MySQL is unable to use an index. But with the dayofweek() function, it is much harder to convert it to a range. The usual way to fix this issue is to "materialize" the field. In other words, store an additional field called "dayofweek":

```
CREATE TABLE ontime (
  id int(11) NOT NULL AUTO_INCREMENT,
  YearD year(4) NOT NULL,
  FlightDate date DEFAULT NULL,
  Carrier char(2) DEFAULT NULL,
  OriginAirportID int(11) DEFAULT NULL,
  OriginCityName varchar(100) DEFAULT NULL,
  OriginState char(2) DEFAULT NULL,
  DestAirportID int(11) DEFAULT NULL,
  DestCityName varchar(100) DEFAULT NULL,
  DestState char(2) DEFAULT NULL, DEFAULT
  DepDelayMinutes int(11) NULL,
  ArrDelayMinutes int(11) DEFAULT NULL,
  Cancelled tinyint(4) DEFAULT NULL,
…
  Flight_dayofweek tinyint NOT NULL,
PRIMARY KEY (id),
KEY Flight_dayofweek (Flight_dayofweek)
  ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

However, we will need to change the application to update this field (or use MySQL triggers which can be slow). In addition, it will require additional storage that will increase the size of the table (not significantly, though).

Calculated Fields in MySQL 5.7

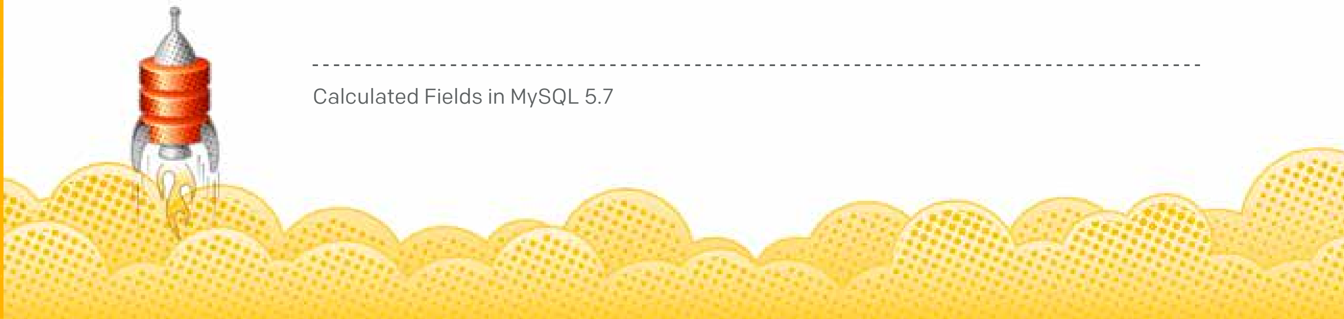


CALCULATED FIELDS IN MYSQL 5.7

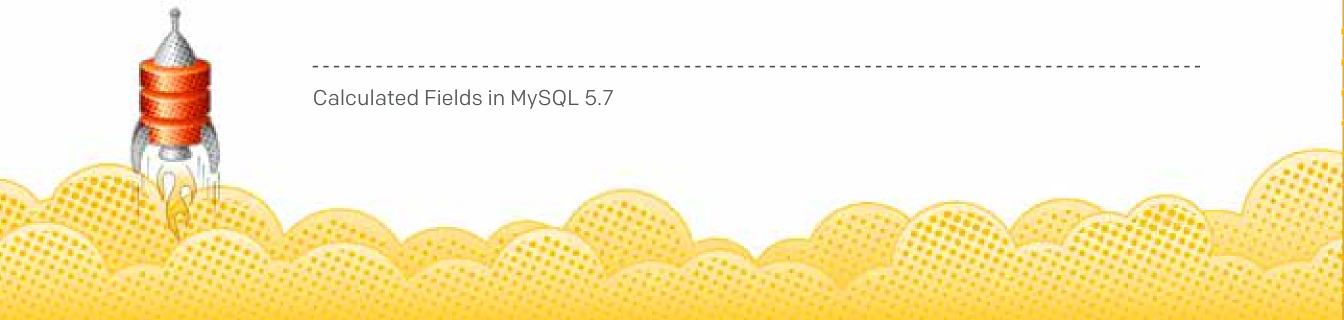
Starting with MySQL 5.7, we can use a new feature called "Virtual" or "Generated" columns:

```
CREATE TABLE ontime_virtual (
  id int(11) NOT NULL AUTO_INCREMENT,
  YearD year(4) NOT NULL,
  FlightDate date DEFAULT NULL,
  Carrier char(2) DEFAULT NULL,
  OriginAirportID int(11) DEFAULT NULL,
  OriginCityName varchar(100) DEFAULT NULL,
  OriginState char(2) DEFAULT NULL,
  DestAirportID int(11) DEFAULT NULL,
  DestCityName varchar(100) DEFAULT NULL,
  DestState char(2) DEFAULT NULL,
  DepDelayMinutes int(11) DEFAULT NULL,
  ArrDelayMinutes int(11) DEFAULT NULL,
  Cancelled tinyint(4) DEFAULT NULL,
...
  Flight_dayofweek tinyint(4)
  GENERATED ALWAYS AS (dayofweek(FlightDate
  VIRTUAL,
PRIMARY KEY (id),
KEY Flight_dayofweek (Flight_dayofweek)
  ) ENGINE=InnoDB;
```

Here the Flight_dayofweek is a virtual column, so it will not be stored in the table. We can index it, however, and the index will be built and stored. MySQL can use the index:



```
mysql> EXPLAIN SELECT carrier, count(*) FROM
ontime_virtual  WHERE Flight_dayofweek = 7
group by carrier\G
**************** 1. row *********************
           id: 1
  select_type: SIMPLE
        table: ontime_virtual
   partitions: NULL
         type: ref
possible_keys: Flight_dayofweek
          key: Flight_dayofweek
      key_len: 2
          ref: const
         rows: 165409
     filtered: 100.00
        Extra: Using where; Using temporary;
               Using filesort
```



Conclusion

CONCLUSION

We have discussed different indexing strategies to optimize your slow queries.

- *Covered indexes are a great MySQL feature and, in most cases, can increase MySQL performance significantly.*
- *Some queries may also be optimized with a separate index, which will enable the loose index scan algorithm.*
- *"Order by" optimizations can be done with a covered index and with the "order by+limit" index technique, as described above.*

Now you can look again at your MySQL slow query log, and optimize the slower queries.

How Percona Can Help

How Percona Can Help

Potential performance killers are easy to miss when you are busy with daily database administration. Once these problems are discovered and corrected, you will see noticeable improvements in database performance and resilience.

Percona Consulting can help you maximize the performance of your database deployment with our MySQL performance audit, tuning and optimization services. The first step is usually a Performance Audit. As part of a Performance Audit, we will methodically and analytically review your servers and provide a detailed report of their current health, as well as detail potential areas for improvement. Our analysis encompasses the full stack and provides you with detailed metrics and recommendations that go beyond the performance of your software to enable true performance optimization. Most audits lead to substantial performance gains.

If you are ready to proactively improve the performance of your system, we can help with approaches such as offloading workload-intensive operations to Memcached. If your user base is growing rapidly and you need optimal performance on a large scale, we can help you evaluate solutions. If performance problems lie outside of MySQL or NoSQL, such as in your web server, we can usually diagnose and report on that as well.

Percona Support can provide developers and members of your operation team the 24x7x365 resources they need to both build high-performance applications and fix potential performance issues. Percona Support is a highly responsive, effective, affordable option to ensure the continuous performance of your deployment.

Our user-friendly Support team is accessible 24x7x365 online or by phone to ensure that your databases are running optimally. We can help you increase your uptime, be more productive, reduce your support budget, and implement fixes for performance issues faster.

If you want to put your time and focus on your business, but still have peace of mind knowing that your database will be fast, available, resilient and secure, Percona Database Managed Services may be ideal for you. With Percona Managed Services, your application performance can be proactively managed by our team of experts so that problems are identified and resolved before they impact your business. When you work with Percona's Managed Service team you can leverage our deep operational knowledge of Percona Server for MySQL, Percona Server for MongoDB, MongoDB, MariaDB®, OpenStack Trove, Google Cloud SQL and Amazon® RDS to ensure your databases performing at the highest levels.

Our experts are available to help, if you need additional manpower or expertise to improve and ensure the performance of your system. To discuss your performance optimization needs, please call us at +1-888-316-9775 (USA), +44 (203) 6086727 (Europe), visit http://learn.percona.com/contact-me or have us contact you.

ABOUT PERCONA

Percona is the only company that delivers enterprise-class software, support, consulting, and managed services solutions for MySQL, MariaDB and MongoDB across traditional and cloud-based platforms that maximize application performance while streamlining database efficiencies. Our global 24x7x365 consulting team has worked with over 3,000 clients worldwide, including the largest companies on the Internet, who use MySQL, Percona Server for MySQL, Percona Server for MongoDB, Amazon RDS for MySQL, Google Cloud SQL, MariaDB and MongoDB.

Percona consultants have decades of experience solving complex database and data performance issues and design challenges. Because we are both broadly and deeply experienced, we can help build complete solutions. Our consultants work both remotely and on site. We can also provide full-time or part-time interim staff to cover employee absences or provide extra help on big projects.

Percona was founded in August 2006 by Peter Zaitsev and Vadim Tkachenko, and now employs a global network of experts with a staff of over 140 people. Our customer list is large and diverse and we have one of the highest renewal rates in the business. Our expertise is visible in our widely read Percona Database Performance blog and our book High Performance MySQL.

Visit Percona at www.percona.com

PERCONA

Percona Corporate Headquarters

8081 Arco Corporate Drive, Suite 330
Raleigh, NC 27617, USA

usa +1-888-316-9775
eur +44 (203) 6086727

www.percona.com
info@percona.com