

## MySQL 5.1 partitions in practice

This article explains how to test the performance of a large database with MySQL 5.1, showing the advantages of using partitions.

The test database uses data published by the US Bureau of Transportation Statistics. Currently, the data consists of ~ 113 million records (7.5 GB data + 5.2 GB index). Getting and loading the data

The data for this exercise comes from the Bureau of Transportation Statistics. The data is provided as CSV files, and they are available from July 1987 to July 2007 (at the time of writing).

The details of getting the data, setting the data structure and loading the data are explained on MySQL Forge. Problem specification

The test affects a database that is bigger than the amount of RAM in the server, and also the indexes are bigger than the RAM.

The server used for this exercise has 4 GB of RAM, and the size of indexes is over 5 GB.

The reasoning behind this specification is that data warehouses have data collections that are far beyond any reasonable amount of RAM that you can possibly install, occupying several terabytes of storage.

In normal database storage, especially for OLTP, the indexes are cached in RAM, to allow for fast retrieval of records. When the data reaches sizes that can't be contained in the available RAM, we need to use a different approach.

One of MySQL 5.1 main features is partitioning, a technique that divides a table into logical portions to speed-up retrievals.

Using MySQL 5.1 partitions looks simple in principle, but there are some tricky points to be aware of while setting the data for maximum performance.

This article will examine the risks and offer some practical advice to achieve the best performance. Partitioning overview

The current implementation of partitioning in MySQL 5.1 is quite simple. You can partition your data by

- range
- list
- hash
- key

Depending on your needs, you may choose different partitioning types. In this article we concentrate on range partitioning, which is perhaps the most interesting for data warehousing.

MySQL partitioning has some constraints that you must be aware of if you want to use this feature effectively.

- \* the partitioning value must be an integer;
- \* if the table has a unique/primary key, the partitioning column must be part of that key.

The first limitation is the one that has the biggest impact on your design decisions. If the column that you need to use for partitioning is not an integer, you need to use a function to transform it. Some additional constraints apply to partitions, as described in the manual, but we are not concerned about it here. Partitioning gotchas Using date columns

What is relevant in this context is the usage of date columns for partitioning. Since the native data type is not supported, we must convert the date into an integer. In addition to the list of allowed functions, we must take into account the fact that only two date functions can trigger the partition pruning. Thus, if we have to deal with a date column, we need to use one of them (YEAR or TO\_DAYS).

When using the YEAR() function, partitioning is easy, readable, and straightforward.

```
CREATE TABLE by_year (
  d DATE
)
PARTITION BY RANGE (YEAR(d))
(
  PARTITION P1 VALUES LESS THAN (2001),
  PARTITION P2 VALUES LESS THAN (2002),
  PARTITION P3 VALUES LESS THAN (2003),
  PARTITION P4 VALUES LESS THAN (MAXVALUE)
)
```

Partitioning by month is trickier. You can't use the MONTH() for two reasons:

- \* you would be limited to 12 partitions, because MONTH does not include the year;
- \* the MONTH function is not optimized for partition pruning, thus performances would be horrible.

Thus, you need to use the other function that is optimized for partition pruning, TO\_DAYS.

```
CREATE TABLE by_month (
  d DATE
)
PARTITION BY RANGE (TO_DAYS(d))
(
  PARTITION P1 VALUES LESS THAN (to_days('2001-02-01')), -- January
  PARTITION P2 VALUES LESS THAN (to_days('2001-03-01')), -- February
  PARTITION P3 VALUES LESS THAN (to_days('2001-04-01')), -- March
  PARTITION P4 VALUES LESS THAN (MAXVALUE)
)
```

That's already less clear to read than the one partitioned by year. What's worse is that the server won't retain your values, but it will only save the corresponding integers. This is what you get for the above table:

```
show create table by_month\G
***** 1. row *****
      Table: by_month
Create Table: CREATE TABLE `by_month`
  `d` date DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1 /*!50100
PARTITION BY RANGE (TO_DAYS(d))
(
  PARTITION P1 VALUES LESS THAN (730882) ENGINE = MyISAM,
  PARTITION P2 VALUES LESS THAN (730910) ENGINE = MyISAM,
  PARTITION P3 VALUES LESS THAN (730941) ENGINE = MyISAM,
  PARTITION P4 VALUES LESS THAN MAXVALUE ENGINE = MyISAM) */
```

It is advisable to save a copy of the script used to create tables partitioned by month, if you want to have a readable reference of what each partition means. partitioning by function, searching by column

One common mistake that is made when using tables partitioned using a date column is to query by the same function used for partitioning.

For example if your table was created with the clause  
partition by range( YEAR(d) )

and you are told that YEAR and TO\_DAYS are optimized for partition pruning, it seems logical to use a query like  
SELECT count(\*) FROM by\_year  
WHERE YEAR(d) = 2000; # <-- ERROR !!

The partition pruning does not kick, as you can see from EXPLAIN  
explain partitions select count(\*) from by\_year where year(d) = 2001\G  
\*\*\*\*\* 1. row \*\*\*\*\*

```
      id: 1
      select_type: SIMPLE
      table: by_year
      partitions: P1,P2,P3,P4
      type: ALL
      possible_keys: NULL
      key: NULL
      key_len: NULL
      ref: NULL
      rows: 30123
      Extra: Using where
```

The query is doing a full table scan. The meaning of "optimized for partition pruning" is that the search will use partitions when that column is used in the WHERE clause.

The right way of querying is  
 SELECT count(\*) FROM by\_year  
 WHERE d BETWEEN '2001-01-01' and '2001-12-31';

And now the partition pruning is being used:  
 explain partitions select count(\*)  
 from by\_year  
 where d between '2001-01-01' and '2001-12-31'\G  
 \*\*\*\*\* 1. row \*\*\*\*\*  
 id: 1  
 select\_type: SIMPLE  
 table: by\_year  
 partitions: P2  
 type: system  
 possible\_keys: NULL  
 key: NULL  
 key\_len: NULL  
 ref: NULL  
 rows: 30123  
 Extra:

This shows that the query does not cause a full table scan, but it will use only one partition. Using primary keys and indexes

One of the problems I had while testing partitioned tables with a large dataset is that I could not get the performance improvement that I was expecting.

In Robin Schumacher's first article on partitioning the examples compare tables without primary keys. The partitioned table wins the contest hands down.

In my current tests, I started by using a table with a primary key, on the ground that this is what normally would happen. So when I first compared data holding 113 million rows, I used tables with primary keys.

That was a mistake.

A primary key on a table so large that its indexes can't fit in memory is not efficient. To get records from such a table means accessing the disk frequently. Your performance depends completely on the speed of your disk and your processor.

Looking at what others do in data warehousing design, I found out that it's common practice to design large DW sets without using indexes.

In this article we'll see also a performance comparison between partitioned tables with and without primary key. Testing method

In this test, I wanted to compare performance of a large dataset using MyISAM, InnoDB, and Archive storage engines. For each engine, I created one unpartitioned table, with primary key (except for archive) and two partitioned tables, one by month and one by year.

Each topology is tested on a dedicated instance of MySQL server, containing only one database with one table. For each engine, I start the server, run the set of queries and record their results, and then shut down the server.

The server instances were created using MySQL Sandbox. ID storage partitioned records size notes loading time (\*) 1 MyISAM none 113 Mil 13 GB with PK 37 min 2 MyISAM by month 113 Mil 8 GB without PK 19 min 3 MyISAM by year 113 Mil 8 GB without PK 18 min 4 InnoDB none 113 Mil 16 GB with PK 63 min 5 InnoDB by month 113 Mil 10 GB without PK 59 min 6 InnoDB by year 113 Mil 10 GB without PK 57 min 7 Archive none 113 Mil 1.8 GB no keys 20 min 8 Archive by month 113 Mil 1.8 GB no keys 21 min 9 Archive by year 113 Mil 1.8 GB no keys 20 min loading times on a dual-Xeon server.

To compare the effects of partitions on large and small datasets, I created 9 more instances, each containing slightly less than 2 GB of data, to see if the results were different.

The results were recorded in yet another database instance, for better comparison.

Thus, at any given time during the tests, there were two instances running. One containing the results and the one being tested.

The queries used for this test are of two types

- aggregate queries.

```
SELECT COUNT(*)
FROM table_name
WHERE date_column BETWEEN start_date and end_date
```

- specific record fetch

```
SELECT column_list
FROM table_name
WHERE column1 = x and column2 = y and column3 = z
```

For each query type, I generated queries for different date ranges. For each range, I generated a set of additional queries on adjacent dates. The first query for each range is cold, meaning that such range was hit for the first time. Subsequent queries on the same range are warm, meaning that the range was already at least partially cached.

The list of queries used for the test is on the Forge Results partitioned tables with primary key

Let's start with the wrong approach first.

My first batch of tests used partitioned tables with a composite primary key, the same used in the original table. The total size of the PK was 5.5 GB. Rather than improving performance, PK slowed down the operations.

Queries with partitioned tables, burdened with the index search through a PK that can't fit in RAM (4 GB) performed poorly. This is a lesson to be remembered. Partitions are useful, but they must be used in the right way.

```
+-----+-----+-----+-----+
| status | myisam unpart | myisam month | myisam year |
+-----+-----+-----+-----+
| cold   | 2.6574570285714 | 2.9169642 | 3.0373419714286 |
| warm   | 2.5720722571429 | 3.1249698285714 | 3.1294000571429 |
+-----+-----+-----+-----+
```

The solution was in front of my eyes, but I refused to see it at first. Look at the results for the same set of queries using the ARCHIVE storage engine.

```
+-----+-----+-----+-----+
| status | archive unpart | archive month | archive year |
+-----+-----+-----+-----+
| cold   | 249.849563 | 1.2436211111111 | 12.632532527778 |
| warm   | 235.814442 | 1.0889786388889 | 12.600520777778 |
+-----+-----+-----+-----+
```

The results for the table partitioned by month are better than the ones I got on the corresponding MyISAM table. More comments about this fact later. partitioned tables without primary key

Since the performance on partitioned tables using primary keys was so bad, I decided to jump the gun, and get rid of the primary key.

The key factor is that the primary key for this table is larger than the available key buffer (and larger than the total available RAM in this case). Therefore, any search by key will use the disk.

The new approach was successful. Using partitions only, without primary keys, I got what I wanted. A significant improvement in performance. Tables partitioned by month get a 70 to 90% performance gain.

```
+-----+-----+-----+-----+
| status | myisam unpart | myisam month | myisam year |
+-----+-----+-----+-----+
| cold   | 2.6864490285714 | 0.64206445714286 | 2.6343286285714 |
| warm   | 2.8157905714286 | 0.18774977142857 | 2.2084743714286 |
+-----+-----+-----+-----+
```

To make the difference more visible, I tested with two massive queries that should take advantage of the partition pruning mechanism.

```
# query 1 -- aggregate by year
SELECT year(FlightDate) as y, count(*)
FROM flightstats
WHERE FlightDate BETWEEN "2001-01-01" and "2003-12-31"
GROUP BY y
```

```
# query 2 -- aggregate by month
SELECT date_format(FlightDate,"%Y-%m") as m, count(*)
FROM flightstats
WHERE FlightDate BETWEEN "2001-01-01" and "2003-12-31"
GROUP BY m
```

The results show a performance gain of 30 to 60% for the table partitioned by month, and a gain of 15 to 30% for the table partitioned by year.

```
+-----+-----+-----+-----+
| query_id | m_u   | m_m   | m_y   |
+-----+-----+-----+-----+
| 1 | 97.779958 | 36.296519 | 82.327554 |
| 2 | 69.61055 | 47.644986 | 47.60223 |
+-----+-----+-----+-----+
```

The processor factor

I had some trouble when I moved the test to a different server. The above tests were taken on my home desktop, which uses an Intel Dual Core 2.3 MHz CPU. The new server is much faster, with a dual Xeon 2.66 MHz.

Repeating the above tests, I got this surprise:

```
+-----+-----+-----+-----+
| status | myisam unpart | myisam month | myisam year |
+-----+-----+-----+-----+
| cold | 0.051063428571429 | 0.6577062 | 1.6663527428571 |
| warm | 0.063645485714286 | 0.1093724 | 1.2369152285714 |
+-----+-----+-----+-----+
```

The original table, with primary key, is faster than the partitioned ones. The times for partitioned tables are the same I got on the slower server, but the performance of the original table has improved, making partitions unnecessary.

What to do? Since this server seems to take advantage of indexes so well, I added an index on the partitioning column to the partitioned tables.

```
# original table
create table flightstats (
  AirlineID int not null,
  UniqueCarrier char(3) not null,
  Carrier char(3) not null,
  FlightDate date not null,
  FlightNum char(5) not null,
  TailNum char(8) not null,
  ArrDelay double not null,
  ArrTime datetime not null,
  DepDelay double not null,
  DepTime datetime not null,
  Origin char(3) not null,
  Dest char(3) not null,
  Distance int not null,
  Cancelled char(1) default 'n',
  primary key (FlightDate, AirlineID, Carrier, UniqueCarrier, FlightNum, Origin, DepTime, Dest)
)
```

```
# partitioned tables
create table flightstats (
  AirlineID int not null,
  UniqueCarrier char(3) not null,
  Carrier char(3) not null,
```

```

FlightDate date not null,
FlightNum char(5) not null,
TailNum char(8) not null,
ArrDelay double not null,
ArrTime datetime not null,
DepDelay double not null,
DepTime datetime not null,
Origin char(3) not null,
Dest char(3) not null,
Distance int not null,
Cancelled char(1) default 'n',
KEY (FlightDate)
)
PARTITION BY RANGE ...

```

The results were then much more satisfactory, with a 35% performance gain.

```

+-----+-----+-----+-----+
| status | myisam unp   | myisam month | myisam year |
+-----+-----+-----+-----+
| cold   | 0.075289714285714 | 0.025491685714286 | 0.072398542857143 |
| warm  | 0.064401257142857 | 0.031563085714286 | 0.056638085714286 |
+-----+-----+-----+-----+

```

Lessons learned

Testing partitions has been a tiring experience. Gaining performance improvements is not a straightforward operation. What I assumed was a painless change turned out to be a long trial-and-error process. There is no silver bullet

Applying a partition change to a table is no guarantee for performance improvement. The gain depends on several factors:

- the column used for partitioning;
- the function used for partitioning, where the native column type is not an integer;
- the server speed;
- the amount of RAM.

Nothing should be taken for granted. Run benchmarks before applying changes to a production system

Depending on the usage of your database, you may get a huge performance gain, or nothing at all. You can also get a performance decrease, if you are not careful.

Consider this: a table partitioned by date with month intervals can get you dreamy performance gains if your queries always include a date range. It can be a full table scan if your queries don't include a date range. Archive tables can be an excellent compromise

Archive tables achieve a huge performance gain when partitioned. Again, it depends on your usage. Without partitioning, any query to an Archive table is a full table scan. If you have historical data that does not change and you need to perform statistical queries by range, the Archive engine is an excellent choice. It uses 10 to 20% of the original storage, and it can perform better than the original MyISAM or InnoDB table for aggregate queries.

Again, it is all a matter of benchmarks. A well tuned partitioned MyISAM table performs far better than a corresponding Archive table, but it needs ten times more storage. Summing up

Partitioning is the key for performance gain with large databases. The definition of large depends on the hardware at your disposal.

Applying partitions blindly is no guarantee of achieving performance improvements, but with the aid of some preliminary benchmarks it can become your perfect solution.