Using MySQL Table Partitioning with Rails

Ian Kallen

2010-03-15

Goals

- Review a problem I had managing data for a Rails based application
- The solution I found using table partitioning
- Hopefully, it will be applicable to a problem you've encountered
- And this will be helpful for your solution

Problem Statement

- Worked on an application that produced a lot of data from the web
- Detailed views of data older than a week were irrelevant
- Access was optimized with all of the usual suspects (memcached, data denormalized, etc)
- But there was no smooth method of aging data out
- So I turned to partitioning the applicable tables

What is table partitioning?

- Takes per-table InnoDB files a step further
- Segment table data and indexes across multiple files
- Manage the size of a table's working set
- Properly scoped queries can hit a smaller index
- A lot of apps won't benefit from this, assume YAGNI
- We'll get to the use cases

What about merge tables?

- Good question, Padawan
- But, you must turn away from the Dark Side
- Merge tables only work with MyISAM, not InnoDB
- ...may be acceptable for “insert” and “select” only apps
- But if you need updates and deletes
- Anger, misery

What about sharding?

- Orthogonal to table partitioning (use cases coming)
- When write volumes exceed an instance's capacity
- Segment the data across multiple instances
- Which requires moving joins out of the database
- ... relaxing ACID and referential integrity
- Avoid it 'til you need it
- The *data_fabric* rails plugin might ease the pain
- Why are we using a relational database again?

What about...

- Distributing reads with replication to slaves?
- Fragment caching?
- Write-through object caching with cache-money?
- Couchdb? memcachedb? mongodb? redis? riak? voldemort? hbase? cassandra?
- ...there are lots of ways to make your data's life easier but not helpful to the basic data pruning challenge

A common scenario:

- A write-intensive service, lots of inserts and updates
- The app only needs access to the most recent data, each record has ephemeral usefulness
- The data volumes grow, rapid table bloat ensues
- Scaling up hardware isn't a fix, denormalizing to reduce joins isn't enough either
- So, big fat deletes to prune old data are implemented
- Anger, misery

What does this have to do with Rails?

- MySQL 5.1 has pretty good table partitioning support
  (someone else can compare it to PostgreSQL and Oracle's)
- However, it imposes limitations on unique constraints
  (ergo, primary keys, which is why it's only "pretty good")
- If partitioning criteria must use the PK, neatly partitioning by date per the examples won't work
- Rails uses auto_increment PK's (as do lots of frameworks)
- BTW, if validates_uniqueness_of is susceptible to race conditions, serialize writes

Example: tweet tabulation

- A blog CMS used to be, but now Twitter is the new “Hello World”
- Loads of apps built for doing “stuff” with Twitter data
- e.g. consume a Twitter list and tabulate the tweets by
- Something
- So we need models for Twitter statuses and the users that make them

Models

- we have users but let's focus on statuses

```ruby
class CreateStatuses < ActiveRecord::Migration
  def self.up
    create_table :statuses do |t|
      t.integer :user_id, :limit => 8, :null => false
      t.integer :twitter_status_id ,:limit => 8, :null => false
      t.datetime :created_at, :null => false
      t.string :text, :limit => 140
      t.string :source
      t.float :lat
      t.float :lon
      t.boolean :has_geo
    end
    add_index :statuses, [:user_id]
  end

  def self.down
    drop_table :statuses
  end
end
```

```mysql
CREATE TABLE `statuses` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` bigint(20) NOT NULL,
  `twitter_status_id` bigint(20) NOT NULL,
  `created_at` datetime NOT NULL,
  `text` varchar(140) DEFAULT NULL,
  `source` varchar(255) DEFAULT NULL,
  `lat` float DEFAULT NULL,
  `lon` float DEFAULT NULL,
  `has_geo` tinyint(1) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `index_statuses_on_user_id` (`user_id`)
)
```

Crawl statuses like this

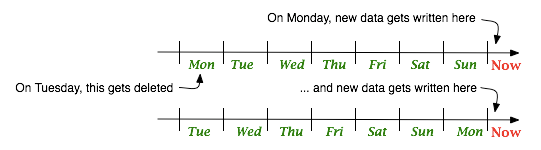
```ruby
class TwitterList
  def refresh_statuses
    get_urls { |url| save_statuses(JSON.parse(open(url).read)) }
  end

  def get_urls
    TWITTER_LIST_CONFIG.entries.each do |user,lists|
      lists.each do |list|
        if block_given?
          yield "http://api.twitter.com/1/#{user}/lists/#{list}/statuses.json"
        end
      end
    end
  end

  def save_statuses(statuses)
    statuses.each { |status|
      tweet =  status.slice('created_at', 'text', 'source')
      tweet['user_id'] = status['user']['id']
      tweet['twitter_status_id'] = status['id']
      if status['geo']
        tweet['lon'], tweet['lat'] = status['geo']['coordinates']
        tweet['has_geo'] = true
      end
      if ! User.find_by_twitter_user_id(:first, tweet['user_id'])
        user = status['user'].slice('name', 'url', 'friends_count', 'followers_count',
          'statuses_count', 'profile_image_url', 'location')
        user['twitter_created_at'] = status['user']['created_at']
        user['twitter_user_id'] = status['user']['id']
        User.create(user)
      end
      Status.create(tweet)
    }
  end
end
```

An example of data with ephemeral usefulness

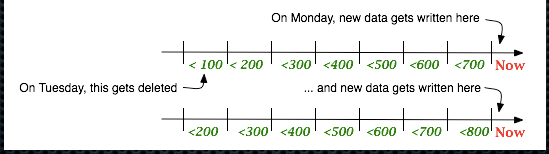
- As statuses usefulness declines with age
- “grow forever” data is not needed
- as data volumes and working set size grows, deletes are increasingly miserable
- which can make you... grumpy
- so that’s our use case

Let's partition by date

- Segment the data into rolling daily partitions like this



- However, the date of the tweet doesn't make sense as a primary key. So partition by the PK



Correlate time & PK ranges

- Both are monotonically ascending
- To ease the pain of determining the daily ID allocation rate, create an index on created_at

```
def daily_id_rate(model)
  date_format = "%Y-%m-%d 00:00:00"
  max_ids = 1.upto(7).map { |d|
    between = "'%s' AND '%s'" % [d.days.ago.strftime(date_format), (d-1).days.ago.strftime(date_format)];
    model.connection.select_value("SELECT max(id) FROM #{model.table_name} WHERE created_at BETWEEN #{between}")
  }.reject { |mi| mi.nil? }.map {|mi| mi.to_i }
  daily_deltas = max_ids.each_cons(2).map { |a,b| a-b }
  approx(daily_deltas.sum/daily_deltas.size)
end

def approx(num)
  zeros = 1.upto(30).reject { |n| 10**n > num }.max
  if zeros > 3
    ((num/(10**(zeros-3))).ceil + 1) * (10**(zeros-3))
  else
    num
  end
end
```

- this is an expensive operation, but not needed frequently

Initial Partitioning

- Let's say you have 1 week of data
- The daily ID allocation rate is 1M
- Create a partition for each day

```
mysql> ALTER TABLE statuses PARTITION BY RANGE (id)
    -> (
    -> PARTITION statuses_0 VALUES LESS THAN (1000000),
    -> PARTITION statuses_1 VALUES LESS THAN (2000000),
    -> PARTITION statuses_2 VALUES LESS THAN (3000000),
    -> PARTITION statuses_3 VALUES LESS THAN (4000000),
    -> PARTITION statuses_4 VALUES LESS THAN (5000000),
    -> PARTITION statuses_5 VALUES LESS THAN (6000000),
    -> PARTITION statuses_6 VALUES LESS THAN (7000000),
    -> PARTITION statuses_maxvalue VALUES LESS THAN MAXVALUE
    -> );
Query OK, 20 rows affected (0.15 sec)
Records: 20  Duplicates: 0  Warnings: 0

mysql>
```

- This will take a lot longer than 150 ms on populated table

The MAXVALUE Partition

- Ideally, you never write to it
- You “reorganize” it into new partitions whenever the ones behind it starts getting data written
- Determine when it’s time to reorganize by getting the max(id) and comparing it to the second to last partition behind MAXVALUE
- Use the information_schema database from MySQL

information_schema

- Partitions are defined by the max id they'll contain
- The third to the last partition has the max value to trigger rolling a new one

```
def current_table_partitions(model)
  select_sql = "SELECT partition_ordinal_position AS seq, partition_name AS part_name, " +
    "partition_description AS max_part_id FROM information_schema.partitions WHERE " +
    "table_name='%s' and table_schema='%s' ORDER BY seq" %
    [model.table_name, model.connection.current_database]
  model.connection.select_all(select_sql)
end

def needs_new_partition?(model)
  ctp = current_table_partitions(model)
  model.maximum(:id) > ctp[-3]['max_part_id'].to_i
end
```

Rolling partitions

- When highest “working partition” (statuses_6) is taking data
- Reorganize the last partition (statuses_maxvalue) into statuses_7 and statuses_maxvalue
- Drop the oldest partition (statuses_0)

```
mysql> ALTER TABLE statuses REORGANIZE PARTITION statuses_maxvalue INTO (
    -> PARTITION statuses_7 VALUES LESS THAN (8000000),
    -> PARTITION statuses_maxvalue VALUES LESS THAN MAXVALUE
    -> );
Query OK, 0 rows affected (2.26 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE statuses DROP PARTITION statuses_0;
Query OK, 0 rows affected (1.00 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql>
```

Query Optimization

- EXPLAIN PARTITIONS shows how you can scope which partitions are used in a query

```
mysql> EXPLAIN PARTITIONS SELECT * FROM statuses\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: statuses
   partitions: statuses_2,statuses_3,statuses_4,statuses_5,statuses_6,statuses_7
,statuses_8,statuses_9,statuses_maxvalue
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 9
        Extra:
1 row in set (0.00 sec)
```

```
mysql> EXPLAIN PARTITIONS SELECT * FROM statuses WHERE id BETWEEN 100 AND 109\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: statuses
   partitions: statuses_2
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

Rails Plugin

- A test case

```ruby
def test_roll_partition
  load_statuses(0, 150)
  assert_equal 150, Status.count
  RollingTablePartition.initialize_partitions(Status, 10)
  assert_equal 150, Status.count
  rtps = RollingTablePartition.roll_partition(Status, now=Time.parse("Mon Mar 8 00:00:00 UTC 2010").gmtime)
  # grow the table by 10
  load_statuses(151, 160)
  assert_equal 160, Status.count
  rtps = RollingTablePartition.roll_partition(Status, now=Time.parse("Tue Mar 9 00:00:00 UTC 2010").gmtime)
  assert_equal 'statuses_1', rtps[0].part_name
  assert_equal 9, rtps.size
  # statuses_0 was a fat partition, by droppping it, we've shrunk the table
  assert_equal 71, Status.count
  # grow the table by 10
  load_statuses(161, 170)
  assert_equal 81, Status.count
  rtps = RollingTablePartition.roll_partition(Status, now=Time.parse("Wed Mar 10 00:00:00 UTC 2010").gmtime)
  # statuses_1 was dropped
  assert_equal 'statuses_2', rtps[0].part_name
  assert_equal 9, rtps.size
  assert_equal 71, Status.count
end
```

- http://github.com/spidaman/rtp

Further work

- Create rake tasks for initial partitioning, partition rolling, etc
- Take advantage of hash partitioning to distribute I/O
- Add support for archiving old partitions, not just deleting them
- Make the tests better serve as documentation as specs (RSpec? Cucumber?)
- Integrate table_migrator to make initial partitioning less disruptive

Hope That Helps

- I'm working with an early stage startup with solid entrepreneurs
- We're hiring a few good folks: engineering (python, django, Cassandra... OK, wrong crowd) and ad-operations
- Rawk stars >:0

Thanks

- spidaman@gmail.com
- http://github.com/spidaman/rtp