18

Case Study: High-Hat Delivers!

To augment his already-stellar decision-making skills, High-Hat Airways' CEO employs a diverse squadron of soothsayers, astrologists, and fortune tellers. For months, these experts have been pushing him to enter the lucrative package shipment marketplace. The CEO has been reluctant; he can't exactly explain why, but he's been waiting for a signal. Finally, the sign arrives in the form of a dream: rows of singing cardboard boxes, each stuffed with cash. The next morning, the CEO summons his executive team to an emergency meeting to deliver the great news: High-Hat is entering the shipping business!

With lightning speed, High-Hat's IT department springs into action. Impending layoffs are delayed. Vacations are canceled; several projects nearing completion are put on hold as the entire team works around the clock to realize the CEO's dream.

This Herculean effort pays off: An entire package tracking application infrastructure is built and deployed worldwide in a matter of weeks. Of course, QA is a bit behind; performance testing isn't even a consideration.

Initially, the new package tracking service is a hit. Wall Street raises earnings estimates, and many executives receive bonuses. The previously delayed IT layoffs now proceed, adding even more to the bottom line.

From a systems perspective, everything appears fine. The new applications have relatively few issues. Results are accurate, and response time is reasonable. During the course of the first month, however, things begin to change. Mysterious reports, detailing sporadic yet horrendous application performance problems, start arriving daily from High-Hat's far-flung empire. Rumors of severe problems leak out to financial analysts, who promptly cut earnings estimates, thereby decimating the stock price. Several executives are demoted, while numerous midlevel managers are moved from their offices into tiny cubicles.

A desperate CIO calls you late one night. High-Hat is very sorry about laying you off, and wants you to return to help overcome these problems. After renegotiating your compensation package, you're ready to go back to work.

Being an astute performance-tuning expert, you know that the first task is to accurately catalog the problems. Only then can you proceed with corrections. Your initial analysis separates the main problems into the following high-level categories.

Problem Queries

After looking into the query situation, you realize that, basically, two types of problem queries exist. The first is encountered when a user tries to look up the status of a shipment. However, it is not consistent: It appears to happen sporadically for most users. The second problem query happens to everyone whenever they attempt to accept a new package for shipment.

Package Status Lookup

Internal employees and external website users have begun complaining that it takes too long to look up the shipping status for a package. What makes this more perplexing is that it doesn't happen all the time. Some queries run very fast, whereas others can take minutes to complete.

The principal tables for tracking package status include the following:

```
CREATE TABLE package_header (
    package_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    dropoff_location_id SMALLINT(3),
    destination_location_id SMALLINT(3),
    sender_first_name VARCHAR(20),
    sender_last_name VARCHAR(30),
...
    recipient_first_name VARCHAR(20),
    recipient_last_name VARCHAR(30),
...
    recipient_fax VARCHAR(30),
...
    INDEX (sender_last_name, sender_first_name),
    INDEX (recipient_last_name, recipient_first_name),
    INDEX (recipient_fax)
) ENGINE = INNODB;


CREATE TABLE package_status (
    package_status_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    package_id INTEGER NOT NULL REFERENCES package_header(package_id),
...
...
    package_location_id SMALLINT(3) NOT NULL,
    activity_timestamp DATETIME NOT NULL,
    comments TEXT,
    INDEX (package_id)
) ENGINE = INNODB;
```

Diagnosis

As an experienced MySQL expert, you know that MySQL offers a number of valuable tools to help spot performance problems. One of them is the slow query log, as discussed in Chapter 2, "Performance Monitoring Options." By simply enabling this log, you can sit back and wait for the troubled queries to make their presence known.

Sure enough, after a few minutes you see some candidates:

```
# Time: 060306 17:26:18
# User@Host: [fpembleton] @ localhost []
# Query_time: 6  Lock_time: 0  Rows_sent: 12  Rows_examined: 573992012
SELECT ph.*, ps.* FROM package_header ph, package_status ps WHERE
ph.package_id = ps.package_id AND ph.recipient_fax like '%431-5979%';
# Time: 060306 17:26:19
# User@Host: [wburroughs] @ localhost []
# Query_time: 9  Lock_time: 0  Rows_sent: 0  Rows_examined: 5739922331
SELECT ph.*, ps.* FROM package_header ph, package_status ps WHERE
ph.package_id = ps.package_id AND ph.recipient_fax like '%785-4551%';
# Time: 060306 17:26:21
# User@Host: [nikkis] @ localhost []
# Query_time: 9  Lock_time: 0  Rows_sent: 0  Rows_examined: 5739922366
SELECT ph.*, ps.* FROM package_header ph, package_status ps WHERE
ph.package_id = ps.package_id AND ph.recipient_fax like '%341-1142%';
```

Now that you've found what appears to be a problem query, your next step is to run `EXPLAIN` to see what steps the MySQL optimizer is following to obtain results:

```
mysql> EXPLAIN
    -> SELECT ph.*, ps.*
    -> FROM package_header ph, package_status ps
    -> WHERE ph.package_id = ps.package_id
    -> AND ph.recipient_fax like '%431-5979%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: ph
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 521750321
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: ps
         type: ref
possible_keys: package_id
          key: package_id
      key_len: 4
          ref: high_hat.ph.package_id
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

This output provides the answer: MySQL is performing an expensive table scan on `package_header` every time a user searches on recipient fax. Considering the sheer size of the table, it's apparent that this leads to very lengthy queries. It also explains the sporadic nature of the query problem: Most status queries use some other lookup criteria.

When you interview the developer of the query, you learn that this query exists to serve customers, who might not always know the area code for the recipient fax. To make the query more convenient, the developer allowed users to just provide a phone number, and he places a wildcard before and after the number to find all possible matches. He's aghast to learn that this type of query frequently renders existing indexes useless.

Solution

When faced with a large-table query that is not correctly taking advantage of indexes, you have two very different options: Fix the query or add a new index. In this case, it's probably easiest and wisest to just correct the query. The application logic should force the user to enter an area code and fax number. In combination, these two values will be able to employ the index:

```
mysql> EXPLAIN
    -> SELECT ph.*, ps.*
    -> FROM package_header ph, package_status ps
    -> WHERE ph.package_id = ps.package_id
    -> AND ph.recipient_fax like '516-431-5979'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: ph
         type: range
possible_keys: PRIMARY,recipient_fax
          key: recipient_fax
      key_len: 30
          ref: NULL
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: ps
         type: ref
possible_keys: package_id
          key: package_id
      key_len: 4
          ref: high_hat.ph.package_id
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

As you saw earlier during the review of MySQL's optimizer in Chapter 6, "Understanding the MySQL Optimizer," version 5.0 offers better index utilization. In this case, the developer might elect to allow the user to query on several area codes. The new optimizer capabilities mean that MySQL can still take advantage of the index:

```
mysql> EXPLAIN
    -> SELECT ph.*, ps.*
    -> FROM package_header ph, package_status ps
    -> WHERE ph.package_id = ps.package_id
    -> AND ((ph.recipient_fax like '516-431-5979')
    -> OR (ph.recipient_fax like '212-431-5979'))\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: ph
         type: range
possible_keys: PRIMARY,recipient_fax
          key: recipient_fax
      key_len: 30
          ref: NULL
         rows: 2
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: ps
         type: ref
possible_keys: package_id
          key: package_id
      key_len: 4
          ref: high_hat.ph.package_id
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

Shipping Option Lookup

To wring more profit from its shipping service, High-Hat implemented a complex pricing mechanism, with thousands of possible prices based on weight, distance, potential value of the customer, currency, language, and so on. All of this information is stored in a single, vital lookup table:

```
CREATE TABLE shipping_prices (
    price_id INTEGER PRIMARY KEY AUTO_INCREMENT,
    price_code CHAR(17) NOT NULL,
    from_zone SMALLINT(3) NOT NULL,
    to_zone SMALLINT(3) NOT NULL,
    min_weight DECIMAL(6,2) NOT NULL,
    max_weight DECIMAL(6,2) NOT NULL,
    ...
    price_in_usd decimal(5,2) NOT NULL,
    price_in_euro decimal(5,2) NOT NULL,
    price_in_gbp decimal(5,2) NOT NULL,
    ...
    price_in_zambia_kwacha DECIMAL(15,2) NOT NULL,
    price_rules_in_english LONGTEXT NOT NULL,
    price_rules_in_spanish LONGTEXT NOT NULL,
    ...
    price_rules_in_tagalog LONGTEXT NOT NULL,
    price_rules_in_turkish LONGTEXT NOT NULL,
    ...
    INDEX (price_code),
    INDEX (from_zone),
    INDEX (to_zone),
    INDEX (min_weight),
    INDEX (max_weight)
) ENGINE = MYISAM;
```

Users are complaining that it takes too long to look up the potential price to ship a package. In several cases, customers have either hung up on the High-Hat sales representative or even stormed out of the package drop-off centers.

Diagnosis

Given how frequently this data is accessed by users, it seems that it should be resident in memory most of the time. However, this is not what your analysis shows.

The first thing you check is the size of the table and its indexes. You're surprised to see that this table has hundreds of thousands of very large rows, which consumes enormous amounts of space and makes full memory-based caching unlikely.

The next observation that you make is that this is a heavily denormalized table. This means that when a High-Hat representative retrieves the necessary rows to quote a price to a customer in France, each row that she accesses contains vastly larger amounts of information (such as the price in all currencies and shipping rules in all languages), even though this data is irrelevant in her circumstance.

Finally, you examine the query cache to see how many queries and results are being buffered in memory. You're disappointed to see that the query cache hit rate is very low. However, this makes sense: Recall that if two queries differ in any way, they cannot leverage the query cache.

Solution

The underlying problem here is that the database design is horribly inefficient: Had the designers done a better job of normalization, there would be reduced memory requirements for the essential lookup columns; extraneous columns would not even be included in most result sets. Alas, a database redesign is out of the question, so your next course of action is to make the best of a bad situation and help MySQL do a better job of caching information given the dreadful database design.

Often, the least aggravating and time-consuming approach to raising cache performance is to simply plug more memory into your database server. However, in this case, the server has no more storage capacity, so you need to come up with an alternative strategy. The only remaining choice is to focus on MySQL configuration.

You have several choices when deciding how to cache heavily accessed tables containing critical lookup information that is infrequently updated.

- **Switch to a `MEMORY` table**—These fast, RAM-based tables were explored in Chapter 4, "Designing for Speed," overview of MySQL's storage engines. If there was sufficient RAM, you could theoretically load the entire `shipping_prices` table into memory. However, there isn't enough storage, so this option is not workable.
- **Increase utilization of the key cache**—As you saw in Chapter 11, "MyISAM Performance Enhancement," the MyISAM key cache leverages memory to hold index values, thereby reducing costly disk access. However, memory is already a precious commodity on this server, so it's unlikely that you'll be able to extract some additional RAM from your administrators. In addition, this isn't an index problem; instead, the fault lies with the sheer amount of data in each row.
- **Make better use of the query cache**—As you have seen, the query cache buffers frequently used queries and result sets. However, there are several important requirements before a query can extract results from this buffer. One crucial prerequisite is that a new query must exactly match already-cached queries and result sets. If the new query does not match, the query cache will not be consulted to return results.

  In this case, you know from your analysis that there is a high degree of variability among queries and result sets, which means that even the largest query cache won't help.

- **Employ replication**—Recall from earlier in this chapter, the replication discussion that significant performance benefits often accrue by simply spreading the processing load among multiple machines. In this case, placing this fundamental lookup table on its own dedicated machines is very wise. Because there are no other tables with which to contend, it will have the lion's share of memory, so caching hit rates should be somewhat improved.

  The application will then be pointed at this server to perform lookups, which should require minimal code changes. However, given the mammoth amount of data found in this table, it's vital that the replicated server have sufficient memory and processor speed to effectively serve its clients. Last but not least, these large rows have the potential to crowd your network, so it's important that the replicated server be placed on a fast network with ample bandwidth. If not, there's a real risk that you might trade one performance problem for another.

Random Transaction Bottlenecks

You've saved the most difficult-to-pin-down performance obstacle for last. The IT help desk receives high volumes of support calls at sporadic times throughout the day. During these periods of intense activity, users complain of system response problems across the board. These delays affect everything from saving new transaction records to updating existing package status details to running reports. To make matters worse, there doesn't appear to be a correlation with user activity load: Some of the most severe slowdowns happen during off-peak hours.

Diagnosis

Fortunately, when faced with such a fuzzy, hard-to-define problem, you have a wide variety of tools at your disposal. These range from operating system monitors to network traffic indicators to MySQL utilities. In circumstances in which there doesn't appear to be a consistent problem, it's often best to arrive at a diagnosis by the process of elimination. You can work through a list of possible causes of the transient performance issue:

- **Insufficient hardware**—If your server is underpowered, it's likely that this deficiency is most prominent during periods of peak activity. That isn't the case here. To be certain, it is wise to turn on server load tracking and then correlate that with MySQL response issues.
- **Network congestion**—This is a little harder to rule out, but the performance problems are not always happening during busy hours. Still, a slight possibility exists that some user or process is hogging the network at seemingly random times, which incorrectly gives the appearance of a MySQL problem. Matching a saved trace of network activity with reports of performance problems goes a long way toward completely eliminating this as a possible cause.

- **Poor database design**—Performance problems are the norm, rather than the exception, if the database designers made key strategic errors when laying out the schema. The same holds true for indexes: Generally, an inefficient index strategy is easy to identify and correct.
- **Badly designed queries**—Given the broad constituency that is complaining about these sporadic slowdowns, it seems unlikely that a single protracted query, or even a group of sluggish queries, could be the culprit. The slow query log goes a long ways toward definitively ruling this out.
- **Unplanned user data access**—The widespread availability of user-driven data access tools has brought untold joys into the lives of many IT professionals. Nothing can drag a database server down like poorly constructed, Cartesian product-generating unrestricted queries written by untrained users.

  Aside from examining the username or IP address of the offending client, it's difficult to quickly identify these types of query tools within the slow query log or active user list. However, by asking around, you learn that a number of marketing analysts have been given business intelligence software and unrestricted access to the production database server.

Solution

Now that you've established that decision support users are likely the root cause of these issues, you look at several alternatives at your disposal to reduce the impact of this class of user. Think of the choices as the "Four R" strategy: replication, rollup, and resource restriction. The following list looks at these choices in descending order of convenience.

- **Replication**—This is, by far, the most convenient solution to your problem. Dedicating one or more slave servers is a great way to satisfy these hard-to-please users. There will be some initial setup and testing, but no code or server settings need to be changed, significantly minimizing the workload for developers and administrators.
- **Rollup**—Another way to diminish the negative performance impact of open-ended reports is to aggregate and summarize information into rollup tables. This approach requires no application or configuration changes, but it does necessitate some effort on your part, and might not completely solve the resource contention issues. Moreover, reporting users will be faced with a lag between live data and their view of this information.
- **Resource restriction**—MySQL offers a variety of mechanisms to constrain user resource consumption. These options were discussed in Chapter 10, "General Server Performance and Parameters Tuning," exploration of general engine configuration. They include `max_queries_per_hour`, `max_updates_per_hour`, `max_connections_per_hour`, `max_join_size`, `SQL_BIG_SELECTS`, and `max_tmp_tables`.

This is the least desirable approach. First, it likely requires configuring these settings at either a global or session level. Second, there is a significant possibility that these changes will prove ineffective in your particular environment.

Implementing These Solutions

Now that your first paycheck from High-Hat has cleared and you've decided how to address these problems, what's the safest course of action to implement your solutions? You reviewed this topic in general during Chapter 1, "Setting Up an Optimization Test Environment," exploration of setting up a performance optimization environment.

In the context of the problems identified in this chapter, it's wise to execute changes in the following order:

1. **Package status query correction**—Correcting the application to force entry of an area code when looking up packages by recipient fax number correctly employs the relevant index and eliminates the costly table scans currently plaguing users. This is a low-risk, high-reward alteration.
2. **Roll up tables for business intelligence query users**—These users are wrecking performance at unpredictable intervals. Because you don't have the authority to lock them out of the system, it's a good idea to aggregate data for them in rollup tables. This is lower risk, and requires less work than the next step, replication.
3. **Replicate information for business intelligence query users**—This is the ideal solution for the problem of end-user query writers, but it does require some work on your part (and introduces some minimal risk of "collateral damage") to implement.
4. **Replicate the `shipping_prices` table to a dedicated server**—This change goes a long way toward reducing resource contention on the primary server. Just like its counterpart for business intelligence users, this type of activity comes with its own setup costs and risks. In this case, you must change application logic to point at the right server, which entails work as well as establishes some hazards if you miss a reference in your software.