**Replication**

**Topics are:**

* Overview of how it works
* Basic server setup
* Designing more advanced replication configurations
* Managing and optimizing your replicated servers

**Intro**

* Foundation for building large, high-performance applications on top of MySQL, using the so-called “scale-out” architecture
* Replication lets you configure one or more servers as replicas (or slaves) of another server
* Keeps their data synchronized with the master copy
* Not just useful for high-performance applications => also the cornerstone of many strategies for high availability, scalability, disaster recovery, backups, analysis, data warehousing, and many other tasks
* Equally concerned with correctness and reliability when it comes to replication as to performance

**Overview**

* Basic Problem that replication solves is keeping one server’s data synchronized with another’s
* Many replicas can connect to a single master and stay in sync with it and a replica can act as a master
* Masters and replicas can be arranged in many different ways
* MySQL supports two kinds of replication: **statement-based replication (since MySQL 3.23)**and **row-based replication (since MySQL 5.1)**
* Both kinds work by recording changes in the master’s binary log and replaying the log on the replica
* Both are asynchronous => the replica’s copy of the data isn’t guaranteed to be up to date at any given instant
* No guarantees as to how large the latency on the replica might be
* Large queries can make the replica fall seconds, minutes, or even hours behind the master
* MySQL’s replication is mostly backward compatible => newer server can usually be a replica of an older server without trouble
* However, older versions of the server are often unable to serve as replicas of newer versions, because they might not understand new features or the SQL syntax the newer server uses, and there might be differences in the file formats replication uses
* For example, you can’t replicate from a MySQL 5.1 master to a MySQL 4.0 replica
* It’s a good idea to test your replication setup before upgrading from one major or minor version to another
* Upgrades within a minor version, such as from 5.1.51 to 5.1.58, are usually compatible
* Replication generally doesn’t add much overhead on the master
* It requires binary logging to be enabled on the master, which can have significant overhead => need that for proper backups and point-in-time recovery anyway
* Aside from binary logging, each attached replica also adds a little load (mostly network I/O) on the master during normal operation
* If you are replicating a very high-throughput workload (say, 5,000 or more transactions per second) to many replicas, the overhead of waking up all the replica threads to send them the events can add up
* Replication is relatively good for scaling reads, which you can direct to a replica, but it’s not a good way to scale writes unless you design it right
* Attaching many replicas to a master simply causes the writes to be done many times, once on each replica. The entire system is limited to the number of writes the weakest part can perform
* Replication is also wasteful with more than a few replicas, because it essentially duplicates a lot of data needlessly
* For example, a single master with 10 replicas has 11 copies of the same data and duplicates most of the same data in 11 different caches

**Problems Solved by Replication**

* Data distribution
  + MySQL’s replication is generally not bandwidth-intensive, though row-based replication in MySQL 5.1 can consume more bandwidth than statement-based replication
  + It’s useful for maintaining remote data copies, even over intermittent connections, but low replication lag requires a stable, low-latency link
* Load balancing
  + MySQL replication allows you to distribute read queries across multiple servers, making it ideal for read-intensive applications with basic load balancing achievable through simple code changes
  + Load distribution can be managed using methods from hardcoded hostnames and round-robin DNS to advanced solutions like network load balancers and the Linux Virtual Server (LVS) project
* Backups
  + Replication is a valuable technique for helping with backups
  + However, a replica is neither a backup nor a substitute for backups
* High availability and failover
  + Replication can help avoid making MySQL a single point of failure in your application
  + A good failover system involving replication can help reduce downtime significantly
* Testing MySQL upgrades
  + It’s common practice to set up a replica with an upgraded MySQL version and use it to ensure that your queries work as expected, before upgrading every instance

**How Replication Works**

1. The master records changes to its data in its binary log (These records are called *binary log events*)
2. The replica copies the master’s binary log events to its relay log
3. The replica replays the events in the relay log, applying the changes to its own data

* The first part of the process is binary logging on the master
* Just before each transaction that updates data completes on the master, the master records the changes in its binary log
* MySQL writes transactions serially in the binary log, even if the statements in the transactions were interleaved during execution
* After writing the events to the binary log, the master tells the storage engine(s) to commit the transactions
* The next step is for the replica to copy the master’s binary log to its own hard drive, into the so-called *relay log*
* It starts a worker thread, called the *I/O slave thread*, that opens an ordinary client connection to the master, then starts a special *binlog dump* process
* The binlog dump process reads events from the master’s binary log
* If it catches up to the master, it goes to sleep and waits for the master to signal it when there are new events
* The I/O thread writes the events to the replica’s relay log
* The *SQL slave thread* handles the last part of the process
* It reads and replays events from the relay log, thus updating the replica’s data to match the master’s
* As long as this thread keeps up with the I/O thread, the relay log usually stays in the operating system’s cache, so relay logs have very low overhead
* The events the SQL thread executes can optionally go into the replica’s own binary log, which is useful for scenarios we mention later
* Figure 10-1 showed only the two replication threads that run on the replica, but there’s also a thread on the master: like any connection to a MySQL server, the connection that the replica opens to the master starts a thread on the master
* This replication architecture decouples the processes of fetching and replaying events on the replica, which allows them to be asynchronous => the I/O thread can work independently of the SQL thread
* This means updates that might have run in parallel (in different threads) on the master cannot be parallelized on the replica, because they’re executed in a single thread => performance bottleneck for many workloads
* There are some solutions to this, but most users are still subject to the single-threaded constraint

**Setting Up Replication**

* Most basic scenario is a freshly installed master and replica
  + Set up replication accounts on each server
  + Configure the master and replica
  + Instruct the replica to connect to and replicate from the master
* Assuming your servers are called server1 (IP address 192.168.0.1) and server2 (IP address 192.168.0.2)

**Creating Replication Accounts**

* MySQL has a few special privileges that let the replication processes run
* The slave I/O thread, which runs on the replica, makes a TCP/IP connection to the master
* Must create a user account on the master and give it the proper privileges, so the I/O thread can connect as that user and read the master’s binary log
  + **GRANT REPLICATION SLAVE, REPLICATION CLIENT ON \*.\***
* We create this user account on both the master and the replica
  + The account you use to monitor and manage replication will need the REPLICATION CLIENT privilege, and it’s easier to use the same account for both purposes (rather than creating a separate user account for this purpose)
  + If you set up the account on the master and then clone the replica from it, the replica will be set up correctly to act as a master, in case you want the replica and master to switch roles

**Configuring the Master and Replica**

* The next step is to enable a few settings on the master, which we assume is named server1
  + Need to enable binary logging and specify a server ID
* If binary logging wasn’t already specified in the master’s configuration file, you’ll need to restart MySQL
* You must explicitly specify a unique server ID => chose to use 10 instead of 1, because 1 is the default value a server will typically choose when no value is specified
* To verify that the binary log file is created on the master, run SHOW MASTER STATUS
* The replica requires a configuration in its *my.cnf* file like the master, and you’ll also need to restart MySQL on the replica
* Only the server\_id parameter is required on a replica, but we enabled log\_bin too, and we gave the binary log file an explicit name
* Also added two other optional configuration parameters: relay\_log (to specify the location and name of the relay log) and log\_slave\_updates (to make the replica log the replicated events to its own binary log)
* Latter option causes extra work for the replicas, but we have good reasons for adding these optional settings on every replica

**Starting the Replica**

* Next step is to tell the replica how to connect to the master and begin replaying its binary logs
* You should not use the *my.cnf* file for this; instead, use the CHANGE MASTER TO statement
* This statement replaces the corresponding *my.cnf* settings completely
* **CHANGE MASTER TO MASTER\_HOST='server1',**

-> **MASTER\_USER='repl',**-> **MASTER\_PASSWORD='p4ssword',**-> **MASTER\_LOG\_FILE='mysql-bin.000001',**

-> **MASTER\_LOG\_POS=0;**

* The Slave\_IO\_State, Slave\_IO\_Running, and Slave\_SQL\_Running columns show that the replication processes are not running
* Astute readers will also notice that the log position is 4 instead of 0 => 0 isn’t really a log position; it just means “at the start of the log file
* To start replication, run the following command: **START SLAVE;**
* Seconds\_Behind\_Master is no longer NULL => The I/O thread is waiting for an event from the master, which means it has fetched all of the master’s binary logs
* If you make a change on the master, you should see the various file and position settings increment on the replica => should also see the changes in the databases on the replica
* On the replica, you should see two threads. One is the I/O thread, and the other is the SQL thread: **SHOW PROCESSLIST\G**
* These processes will always run under the “system user” user account, but the other column values might vary

**Initializing a Replica from Another Server**

* The previous setup instructions assumed that you started the master and replica with the default initial data after a fresh installation
* Usually you’ll have a master that has been up and running for some time, and you’ll want to synchronize a freshly installed replica with the master, even though it doesn’t have the master’s data
* Several ways to initialize, or “clone,” a replica from another server
  + Include copying data from the master
  + Cloning a replica from another replica
  + Starting a replica from a recent backup
* You need three things to synchronize a replica with a master:
  + A snapshot of the master’s data at some point in time
  + The master’s current log file, and the byte offset within that log at the exact point in time you took the snapshot (log file coordinates), => find the master’s log file coordinates with the SHOW MASTER STATUS command
  + The master’s binary log files from that time to the present
* **Cold Copy:** Involves shutting down the master to copy its files to the replica, then restarting the master with a new binary log and configuring the replica to start from that log, with the downside of requiring master downtime
* **With a warm copy:** Can copy files while the server is still running
* **Using mysqldump:** For InnoDB tables, use mysqldump --single-transaction --all-databases --master-data=1to dump data from the master, load it into the replica, and sync the replica's coordinates with the master's binary log; for non-transactional tables, use --lock-all-tables for a consistent dump
* **With a Snapshot or Backup:** Initialize the replica by restoring a master snapshot or backup and configuring the replica with the corresponding binary log coordinates using CHANGE MASTER TO, ensuring all binary logs since the backup are retained
* **From another replica:** can clone a replica using snapshot or copy techniques, but the --master-data option with mysqldump won’t work, and you’ll need to use SHOW SLAVE STATUS to get the correct binary log coordinates
  + Biggest disadvantage of cloning one replica from another is that if your replica has become out of sync with the master, you’ll be cloning bad data
* Don’t use LOAD DATA FROM MASTER or LOAD TABLE FROM MASTER! They are obsolete, slow, and very dangerous => also only work with MyISAM

**Recommended Replication Configuration**

* Many possible replication parameters, and most of them have at least some effect on data safety and performance
* Show a recommended, “safe” replication configuration that minimizes the opportunities for problems
* Most important setting for binary logging on the master is sync\_binlog: ***sync\_binlog=1***
* Makes MySQL synchronize the binary log’s contents to disk each time it commits a transaction => don’t lose log events if there’s a crash
* If you disable this option, the server will do a little less work, but binary log entries could be corrupted or missing after a server crash
* On a replica that doesn’t need to act as a master, this option creates unnecessary overhead
* Applies only to the binary log, not to the relay log
* Also recommend specifying a binary log base name explicitly, to create uniform binary log names on all servers and prevent changes in binary log names if the server’s hostname changes
* Specify an argument to the log\_bin option, optionally with an absolute path, but certainly with the base name
* On the replica, we also recommend enabling the following configuration options
* The ***relay\_log*** option prevents hostname-based relay log file names, which avoids the same problems we mentioned earlier that can happen on the master
* The ***skip\_slave\_start*** option will prevent the replica from starting automatically after a crash, which can give you a chance to repair a server if it has problems
* The ***read\_only*** option prevents most users from changing non-temporary tables and the exceptions are the replication SQL thread and threads with the SUPER privilege => avoid giving your normal accounts the SUPER privilege
* A replica can easily break after a crash, because the relay logs and *master.info* file aren’t crash-safe
* If a replica is very far behind its master, the slave I/O thread can write many relay logs
* The replication SQL thread will remove them as soon as it finishes replaying them (you can change this with the ***relay\_log\_purge*** option)
* If it is running far behind, the I/O thread could actually fill up the disk => solution to this problem is the ***relay\_log\_space\_limit*** configuration variable
* If the total size of all the relay logs grows larger than this variable’s size, the I/O thread will stop and wait for the SQL thread to free up some more disk space
* If the replica hasn’t fetched all the relay logs from the master, those logs might be lost forever if the master crashes
* Unless you’re worried about disk space, it’s probably a good idea to let the replica use as much space as it needs for relay logs => reason, why we haven’t included the ***relay\_log\_space\_limit*** setting in our recommended configuration

**Statement-Based Replication**

* MySQL 5.0 and earlier support only statement-based replication (also called logical replication)
* Works by recording the query that changed the data on the master
* When the replica reads the event from the relay log and executes it, it is reexecuting the actual SQL query that the master executed
* Has both benefits and drawbacks
* The most obvious benefit is that it’s fairly simple to implement
* Simply logging and replaying any statement that changes data will, in theory, keep the replica in sync with the master
* Another benefit of statement-based replication is that the binary log events tend to be reasonably compact => doesn’t use a lot of bandwidth
* In practice statement-based replication is not as simple as it might seem, because many changes on the master can depend on factors besides just the query text
* For example, the statements will execute at slightly—or possibly greatly—different times on the master and replica
* As a result, MySQL’s binary log format includes more than just the query text; it also transmits several bits of metadata, such as the current timestamp
* There are some statements that MySQL can’t replicate correctly, such as queries that use the CURRENT\_USER() function
* Stored routines and triggers are also problematic with statement-based replication
* Another issue with statement-based replication is that the modifications must be serializable => requires more locking—sometimes significantly more

**Row-Based Replication**

* MySQL 5.1 added support for row-based replication, which records the actual data changes in the binary log and is more similar to how most other database products implement replication
* The biggest advantages are that MySQL can replicate every statement correctly, and some statements can be replicated much more efficiently
* MySQL can replicate some changes more efficiently using row-based replication, because the replica doesn’t have to replay the queries that changed the rows on the master
* Replaying some queries can be very expensive, see exampel page 461
* This query will scan many rows in the source table but will result in only three rows in the destination table
* Replicating this event as a statement will make the replica repeat all that work just to generate a few rows, but replicating it with row-based replication will be trivially cheap on the replica => in this case, row-based replication is much more efficient
* On the other hand, the following event is much cheaper to replicate with statement- based replication:
  + mysql> **UPDATE enormous\_table SET col1 = 0;**
* Using row-based replication for this query would be very expensive because it changes every row
* Every row would have to be written to the binary log, making the binary log event extremely large
* This would place more load on the master during both logging and replication, and the slower logging might reduce concurrency
* It’s harder to do point-in-time recovery with a binary log that has events in row-based format, but not impossible => a log server can be helpful
* Because neither format is perfect for every situation, MySQL can switch between statement-based and row-based replication dynamically
* By default, it uses statement- based replication, but when it detects an event that cannot be replicated correctly with a statement, it switches to row-based replication => set the ***binlog\_format*** session variable

**Comparison between both**

* In theory, row-based replication is probably better all-around, and in practice it generally works fine for most people
* **Statement-based replication advantages** 
  + Works in more cases when the schema is different on the master and the replica
  + Can be made to work in more cases where the tables have different but compatible data types, different column orders and so on
  + Makes it easier to perform schema changes on a replica and then promote it to master, reducing downtime
  + Statement-based replication generally permits more operational flexibility
  + All changes on the server are taking place through a well-understood mechanism, and it’s easy to inspect and determine what is happening if something isn’t working as expected
* **Statement-based replication disadvantages** 
  + There were tons of bugs affecting replication of stored procedures, triggers, and so on in the 5.0 and 5.1 series of the server
  + There are also lots of problems with temporary tables, mixtures of storage engines, specific SQL constructs, nondeterministic statements, and so on => range from annoying to show-stopping
* **Row-based replication advantages** 
  + There are a lot fewer cases that don’t work in row-based replication => works correctly with all SQL constructs
  + Generally only fails when you’re trying to do something clever such as schema changes on the replica
  + Also creates opportunities for reduced locking, because it doesn’t require such strong serialization to be repeatable
  + Row-based replication works by logging the data that’s changed, so the binary log is a record of what has actually changed on the master => don’t have to look at a statement and guess whether it changed any data
  + In some cases the row-based binary logs record what the data used to be, so they can potentially be more useful for some kinds of data recovery efforts
  + Row-based replication can be less CPU-intensive in many cases, due to the lack of a need to plan and execute queries in the same way that statement-based replication does
  + Can help you find and solve data inconsistencies more quickly in some cases
  + For example, statement-based replication won’t fail if you update a row on the master and it doesn’t exist on the replica, but row-based replication will throw an error and stop
* **Row-based replication disadvantages** 
  + The statement isn’t included in the log event, so it can be tough to figure out what SQL was executed => it’s important in many cases, in addition to knowing the row changes
  + The process of applying row-based changes is pretty much a black box with no visibility into what the server is doing
  + When things don’t work right, it can be tough to troubleshoot
  + If you have multiple replication levels using row-based logging, a statement executed with @@binlog\_format set to STATEMENT will be logged as a statement on the original server
  + However, first-level replicas may convert and relay it as row-based to downstream replicas, causing the statement-based logging to revert to row-based as it propagates
  + Row-based logging can’t handle some things that statement-based logging can, such as schema changes on replicas
  + Replication will sometimes halt in cases where statement-based replication would continue, such as when the replica is missing a row that’s supposed to be changed
  + This could be regarded as a good thing => it is configurable with the slave\_exec\_mode option
  + Many of these disadvantages are being lifted as time passes, but at the time of writing, they are still true in most production deployments

**Other Systems (master-master, multiple replicas, …)**

* Master-Master
* Replica as master for other replicas

**Partitioned Tables**

* A partitioned table is a single logical table that’s composed of multiple physical subtables
* Partitioning code is just a wrapper around a set of Handler objects that represent the underlying partitions => forwards requests to the storage engine through the Handler objects
* Is a kind of black box that hides the underlying partitions from you at the SQL layer => can see them quite easily by looking at the filesystem, where you’ll see the component tables with a hash-delimited naming convention
* The way MySQL implements partitioning means that indexes are defined per-partition, rather than being created over the entire table
* Different from Oracle, where indexes and tables can be partitioned in more flexible and complex ways
* MySQL decides which partition holds each row of data based on the **PARTITION BY** clause that you define for the table
* Query optimizer can prune partitions when you execute queries => queries don’t examine all partitions, just the ones that hold the data you are looking for
* The primary purpose of partitioning is to act as a coarse form of indexing and data clustering over the table
* Can help to eliminate large parts of the table from being accessed, and to store related rows close together
* Partitioning can be very beneficial, especially in specific scenarios:
  + When the table is much too big to fit in memory, or when you have “hot” rows at the end of a table that has lots of historical data
  + Partitioned data is easier to maintain than nonpartitioned data. For example, it’s easier to discard old data by dropping an entire partition, which you can do quickly. You can also optimize, check, and repair individual partitions
  + Partitioned data can be distributed physically, enabling the server to use multiple hard drives more efficiently
  + You can use partitioning to avoid some bottlenecks in specific workloads, such as per-index mutexes with InnoDB or per-inode locking with the ext3 filesystem
  + If you really need to, you can back up and restore individual partitions, which is very helpful with extremely large datasets
* Partitioning has made the CREATE TABLE and ALTER TABLE commands much more complex
* There are a few limitations that apply to partitioned tables:
  + There’s a limit of 1,024 partitions per table
  + Any primary key or unique index must include all columns in the partitioning expression
  + You can’t use foreign key constraints

**How Partitioning Works**

* Again: Partitioned tables have multiple underlying tables, which are represented by Handler objects
* You can’t access the partitions directly
* Each partition is managed by the storage engine in the normal fashion (all partitions must use the same storage engine)
* Any indexes defined over the table are actually implemented as identical indexes over each underlying partition
* From the storage engine’s point of view, the partitions are just tables
  + Doesn’t really know whether a specific table it’s managing is a standalone table or just part of a bigger partitioned table
* Operations on a partitioned table are implemented with the following logical operations:
* **SELECT *queries***
  + When you query a partitioned table, the partitioning layer opens and locks all of the underlying partitions, the query optimizer determines whether any of the partitions can be ignored (pruned), and then the partitioning layer forwards the handler API calls to the storage engine that manages the partitions
* **INSERT *queries***
  + When you insert a row, the partitioning layer opens and locks all partitions, determines which partition should receive the row, and forwards the row to that partition
* **DELETE *queries***
  + Same as insert and forwards the deletion request to that partition
* **UPDATE *queries***
  + As before and fetches the row, modifies the row and determines which partition should contain the new row, forwards the row with an insertion request to the destination partition, and forwards the deletion request to the source partition
* Some of these operations support pruning => For example, when you delete a row, the server first has to locate it
* Server can prune partitions that can’t contain the row if you specify a WHERE clause that matches the partitioning expression => same for update
* INSERT queries are naturally self-pruned
* Although the partitioning layer opens and locks all partitions, this doesn’t mean that the partitions remain locked and this lock-and-unlock cycle is similar to how queries against ordinary InnoDB tables are executed

**Types of Partitioning**

* MySQL supports several types of partitioning
* Most common type we’ve seen used is range partitioning, in which each partition is defined to accept a specific range of values for some column or columns, or a function over those columns
* For example, here’s a simple way to place each year’s worth of sales into a separate partition (pls make own example here, should be quite easy)
* Can use many functions in the partitioning clause => main requirement is that it must return a nonconstant, deterministic integer, like **YEAR()**
* In MySQL 5.5 you can use the RANGE COLUMNS partitioning type, so you can partition by date-based columns directly without using a function
* Partitioning by intervals of time is a common way to work with date-based data
* MySQL also supports key, hash, and list partitioning methods, some of which support subpartitions, which we’ve rarely seen used in production
* One use of subpartitions we’ve seen was to work around a per-index mutex inside InnoDB on a table designed similarly to our previous example
* Subpartitioning by hash helped chop the data into smaller pieces and alleviated the problem
* Some Use Cases:
  + You can partition by range using a modulo function to create a round-robin table that retains only a desired amount of data. For example, you can partition date- based data by day modulo 7, or simply by day of week, if you want to retain only the most recent days of data
  + You can partition by an expression such as HASH(id DIV 1000000),which creates a new partition for each million rows inserted => achieves the goal to cluster the recent data together without requiring you to change the primary key

**How to Use Partitioning**

* Imagine that you want to run queries over ranges of data from a really huge table that contains many years’ worth of historical metrics in time-series order
* Want to run reports on the most recent month => return about 100 million rows
* Pretend that you have hardware from 2012 and your table is 10 terabytes
* Much bigger than memory and you have traditional hard drives, not flash
* One thing is sure: you can’t scan the whole table every time you want to query it, because it’s too big
* And you don’t want to use an index because of the maintenance cost and space consumption
* Sometimes this can work around for one or two indexes, but rarely for more
* At very large sizes, B-Tree indexes don’t work
* The cost of maintaining the index (disk space, I/O operations) is also very high
* Only two workable options remain: your query must be a sequential scan over a portion of the table, or the desired portion of the table and index must fit entirely in memory
* Key is to think about partitioning as a crude form of indexing that has very low overhead and gets you in the neighborhood of the data you want
* From there, you can either scan the neighborhood sequentially, or fit the neighborhood in memory and index it
* Partitioning has low overhead because there is no data structure that points to rows and must be updated
* Partitioning doesn’t identify data at the precision of rows, and has no data structure to speak of
* Instead, it has an equation that says which partitions can contain which categories of rows
* Two strategies that work at large scale:
  + ***Scan the data, don’t index it:***Can create tables without indexes and use partitioning as the only mechanism to navigate to the desired kind of rows => if you use a WHERE clause that prunes the query to a small number
  + ***Index the data, and segregate hot data:*** If your data is mostly unused except for a “hot” portion, and you can partition so that the hot data is stored in a single partition that is small enough to fit in memory along with its indexes, you can add indexes and write queries to take advantage of them, just as you would with smaller tables

**What Can Go Wrong**

* The two partitioning strategies are based on two key assumptions
  + Can narrow the search by pruning partitions when you query
  + Partitioning itself is not very costly
* Those assumptions are not always valid
* **NULL*s can defeat pruning***
  + The result of the partitioning function can be NULL
  + Happens even if order\_date is not nullable, because you can store a value that’s not a valid date
  + Any row whose order\_date is either NULL or not a valid date will be stored in the first partition you define (see from example)
  + If you’re querying for the year 2012, you’re checking two partitions instead of one to find the rows, which is definitely undesirable and especially if the first partition is large
  + Workaround with a dedicated partiton: **PARTITION p\_nulls VALUES LESS THAN (0)**
  + Or you can partition by the column itself, instead of a function over the column: **PARTITION BY RANGE COL UMNS(order\_date)**
* ***Mismatched* PARTITION BY *and index***
  + If you define an index that doesn’t match the partitioning clause, queries might not be prunable
  + Suppose you define an index on a and partition by b
  + Each partition will have its own index, and a lookup on this index will open and check each index tree in *every* partition
  + Could be quick if the non-leaf nodes of each index are resident in memory
  + To avoid this problem, you should try to avoid indexing on nonpartitioned columns unless your queries will also include an expression that can help prune out partitions
  + Sounds simple enough to avoid, but it can catch you by surprise
  + For example, suppose a partitioned table ends up being the second table in a join, and the index that’s used for the join isn’t part of the partition clause => each row in the join will access and search every partition in the second table
* ***Selecting partitions can be costly*** 
  + Questions like “Where can I find rows matching this query?” can be costly to answer with range partitioning, because the server scans the list of partition definitions to find the right one
  + Linear search isn’t all that efficient, as it turns out, so the cost grows as the number of partitions grows
  + The queries we’ve observed to suffer the worst from this type of overhead are row- by-row inserts
  + For every row you insert into a table that’s partitioned by range, the server has to scan the list of partitions to select the destination
  + Can alleviate this problem by limiting how many partitions you define
  + A hundred or so works okay for most systems
  + Other partition types, such as key and hash partitions, don’t have the same limitation
* ***Opening and locking partitions can be costly*** 
  + Opening and locking partitions when a query accesses a partitioned table is another type of per-partition overhead.
  + Opening and locking occur before pruning, so this isn’t a prunable overhead
  + This type of overhead is independent of the partitioning type and affects all types of statements
  + Adds an especially noticeable amount of overhead to short operations, such as single-row lookups by primary key
  + Try performing operations in bulk, like multirow inserts
  + Limit the number of partitions you define, improves the situation
* ***Maintenance operations can be costly*** 
  + Some partition maintenance operations are very quick, such as creating or drop- ping partitions
  + Other operations, such as REORGANIZE PARTITION, operate similarly to the way ALTER works: by copying rows around
  + For example, REORGANIZE PARTITION works by creating a new temporary partition, moving rows into it, and deleting the old partition when it’s done
* Other Limitations in the current implementation
  + All partitions have to use the same storage engine.
  + There are some restrictions on the functions and expressions you can use in a partitioning function
  + Some storage engines don’t work with partitioning

**Optimizing Queries**

* Partitioning introduces new ways to optimize queries and corresponding pitfalls
* Again: biggest opportunity is that the optimizer can use the partitioning function to prune partitions
* As you’d expect from a coarse-grained index, pruning lets queries access much less data than they’d otherwise need to (in the best case)
* It’s very important to specify the partitioned key in the WHERE clause, even if it’s otherwise redundant, so the optimizer can prune unneeded partitions
* If you don’t do this, the query execution engine will have to access all partitions in the table, and this can be extremely slow on large tables
* You can use **EXPLAIN PARTITIONS** to see whether the optimizer is pruning partitions
  + mysql> **EXPLAIN PARTITIONS SELECT \* FROM sales \G**
  + result: partitions: p\_2010,p\_2011,p\_2012
  + different for this here: **WHERE day > '2011-01-01'** => only 2 Partitions
* The optimizer is pretty good about pruning; for example, it can convert ranges into lists of discrete values and prune on each item in the list
* The following WHERE clause is theoretically prunable, but MySQL can’t prune it:
  + mysql> **EXPLAIN PARTITIONS SELECT \* FROM sales\_by\_day WHERE YEAR(day) = 2010\G**
* MySQL can prune only on comparisons to the partitioning function’s columns
  + It cannot prune on the result of an expression, even if the expression is the same as the partitioning function
  + Similar to the way that indexed columns must be isolated in the query to make the index usable
  + mysql> **EXPLAIN PARTITIONS SELECT \* FROM sales\_by\_day**-> **WHERE day BETWEEN '2010-01-01' AND '2010-12-31'\G**
  + Because the WHERE clause now refers directly to the partitioning column, not to an ex- pression, the optimizer can prune out other partitions
* The optimizer is smart enough to prune partitions during query processing, too
* For example, if a partitioned table is the second table in a join, and the join condition is the partitioned key, MySQL will search for matching rows only in the relevant partitions
* EXPLAIN won’t show the partition pruning, because it happens at runtime, not at query optimization time

**Merge Tables**

* Partitioning enforces the abstraction rigorously, denying access to the underlying partitions and permitting you to reference only the partitioned table
* Merge tables let you access the underlying tables separately from the merge table
* Like partitioned tables, merge tables are wrappers around underlying MyISAM tables with the same structure
* Partitioning is more integrated with the query optimizer and is the way of the future
* Merge tables are quasi-deprecated and might even be removed someday
* Merge table is really just a container that holds the real tables
* Specify which tables to include with a special UNION syntax to CREATE TABLE
* The underlying tables have exactly the same number and types of columns, and that all indexes that exist on the merge table also exist on the underlying tables
* MERGE tables (also known as UNION tables) were removed in MySQL 8.0
* However, due to modern alternatives and their limitations, MySQL deprecated and later removed MERGE storage support
* In contrast, a partitioned table’s partitions are hidden by the MySQL server and are accessible only through the partitioned table