Architecture

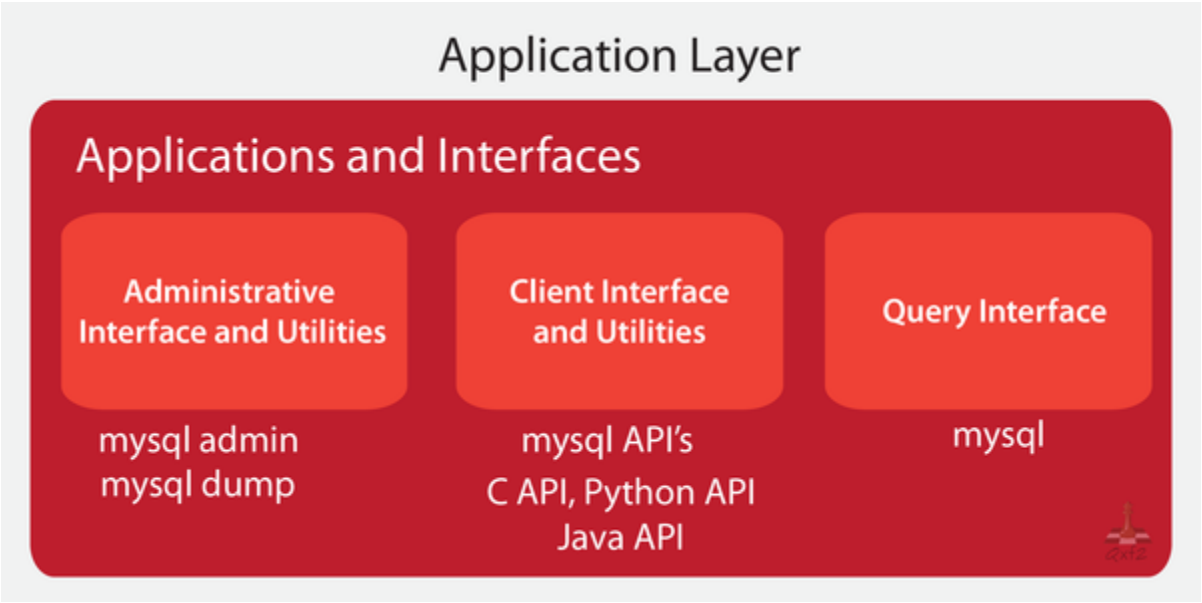
The MySQL architecture is basically a client – server system. MySQL database server is the server and the applications which are connecting to MySQL database server are clients.

Unlike the other databases, MySQL is a very flexible and offers different kinds of storage engines as a plugin for different kinds of needs. Because of this, MySQL architecture and behavior will also change as per the use of storage engines, for example transactional [InnoDB] and non-transactional [MyISAM] engines data storage and SQL execution methods will be different and within the server it will use engine specific components like memory and buffers depending on type storage engine will get used for the SQL operation.

MySQL architecture is broken into three layers basically which can be defined by,

1. Application Layer
2. Logical Layer
3. Physical Layer

MySQL ApplicationLayer:

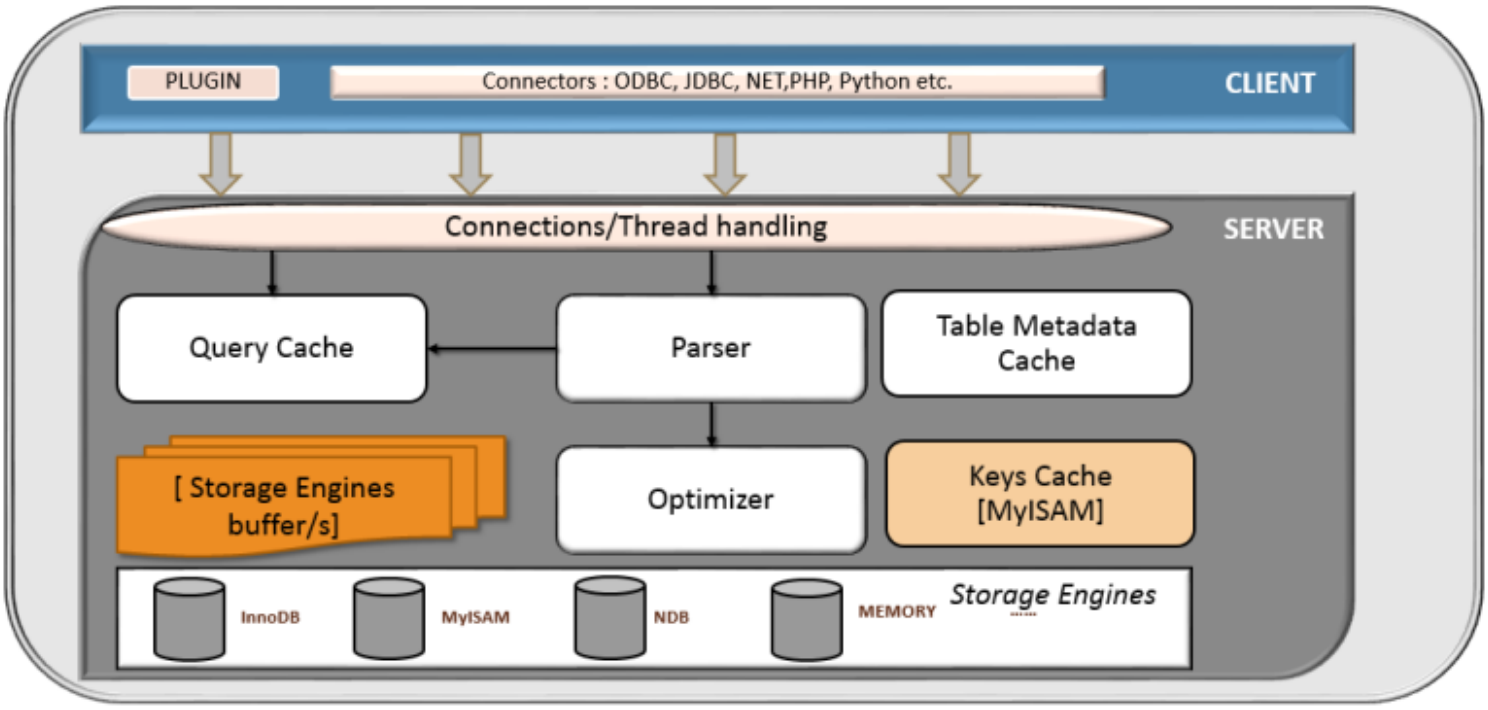


1. Administrators
   Administrators use various administrative interface and utilities like mysqladmin which performs tasks like shutting down the server and creating or dropping databases, mysqldump for backing up the database or copying databases to another server.

2. Clients
   Clients communicate with MySQL through various interfaces and utilities like MySQL API's. The MySQL API sends the query to the server as a series of tokens.

3. Query User
   The query users interact with MySQL RDBMS through a query interface that is *mysql*.

MySQL Logical Architecture:



The Logical Layer takes the data from the Application Layer.

Client :
Utility to connect MySQL server.

Server :
MySQL instance where actual data getting stored and data processing is happening.

mysqld:
MySQL Server daemon program which runs in the background and manages database related incoming and outgoing requests from clients. mysqld is a multi-threaded process which allows connection to multiple sessions listen for all connections and manages MySQL instance.

Invoking "mysqld" will start the MySQL server on your system. Terminating "mysqld" will shutdown the MySQL server.

MySQL memory allocation:
Main MySQL memory is dynamic, examples **innodb_buffer_pool_size** (from 5.7.5), etc.
Working on shared nothing principal which means, every session has unique execution plan and we can share data sets only for the same session.

1. PER Instance :
    a) Allocated once
    b) Shared by all the instance's server process and its threads
2. PER SESSION:
    a) Allocated for each mysql client session
    b) Dynamically allocated and deallocated
    c) Used for handling query result
    d) Buffer size per session

Connection Handling:
When a client connects to server, the client gets its own thread for its connection. All the queries from that client executed within that specified thread. The thread is cached by the server, so they don't need to created and destroyed for each new connection.

Parser:
Check for SQL syntax by checking every character in SQL query and generate *SQL_ID* for each SQL query.
Also, Authentication check (user credentials) will happen at this stage.

Optimizer:
Creates an efficient query execution plan as per the storage engine. It will rewrite a query.
Example: InnoDB has shared buffer so optimizer will get pre-cached data from it. Using table statistics optimizer will generate an execution plan for a SQL query.
Authorization Check (User access privileges) will happen at this stage.

Metadata cache:
Cache for object metadata information and statistics.

Query cache:
Shared identical queries from memory. If an identical query from client found in query cache then, the MySQL server retrieves the results from the query cache rather than parsing and

executing that query again. It's a shared cache for sessions, so a result set generated by one client can be sent in response to the same query issued by another client. Query cache based on SQL_ID.SELECT data into view is the best example of pre-cache data using query cache.

key cache:

Cache table indexes. In MySQL keys are indexes (In oracle keys are constraints) if index size is small then it will cache index structure and data leaf. If an index is large then it will only cache index structure. Used by MyISAM storage engine.

Storage Engine:

MySQL component that manages physical data (file management) and locations. Storage engine responsible for SQL statement execution and fetching data from data files. Use as a plugin and can load/unload from running MySQL server. Few of them as following,

1. InnoDB (default 5.5)
    a) Fully Transactional ACID and offers REDO/UNDO for transactions.
    b) Data Storage in tablespaces and multiple data files.
    c) Row Level locking
    d) Logical object structure using InnoDB data and log buffer

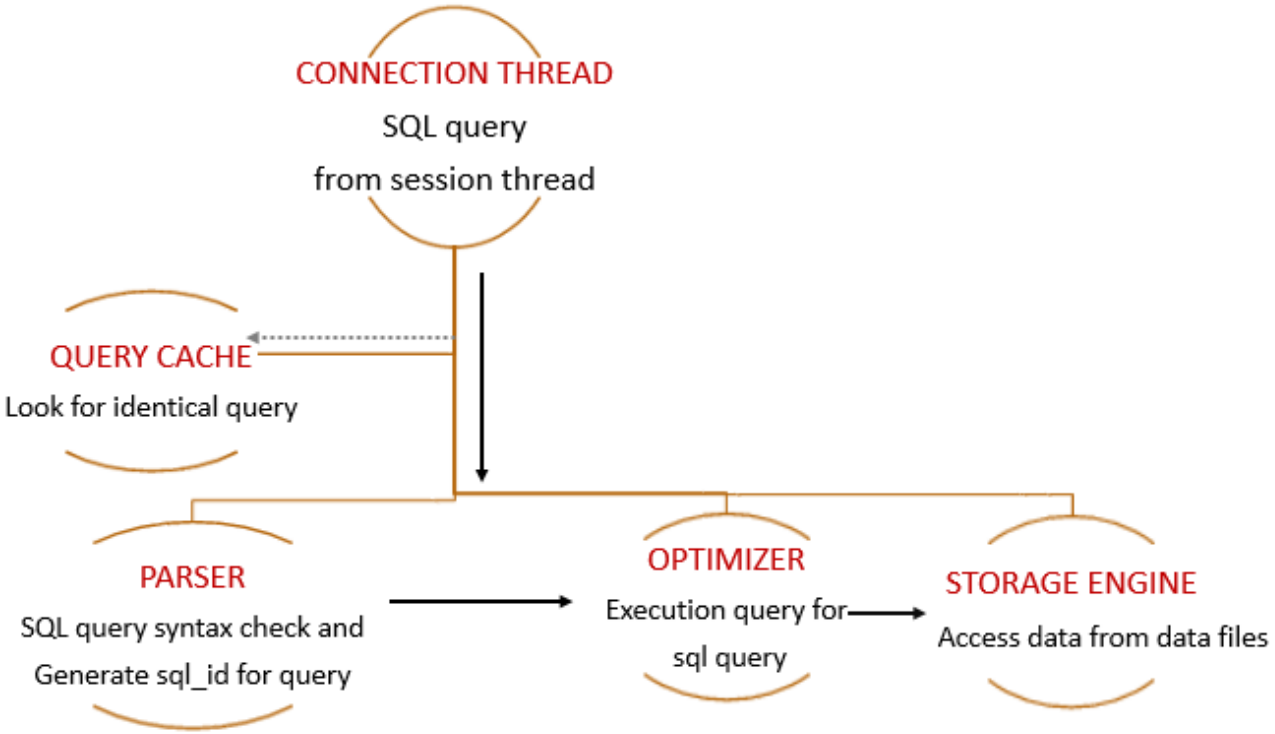
2. MyISAM
    a) Non-transactional with data storage in files
    b) Speed for read and table level locking
    c) MYI for table index and MYD for table read

3. Memory
    a) Non-transactional data stored in memory other than metadata and structure
    b) Table level locking

4. ARCHIVE:
    a) Non-transactional storage engine,
    b) Store large amounts of compressed and unindexed data.
    c) Allow INSERT, REPLACE, and SELECT, but not DELETE or UPDATE sql operations.
    d) Table-level locking.

5. CSV:
    a) Stores data in flat files using comma-separated values format.
    b) Table structure need be created within MySQL server (.frm)

SQL execution



The MySQL Server (mysqld) executes as a single OS process, with multiple threads executing concurrent activities. MySQL does not have its own thread implementation, but relies on the thread implementation of the underlying OS. When a user connects to the database a user thread is created inside mysqld and this user thread executes user queries, sending results back to the user, until the user disconnects.

When more and more users connect to the database, more and more user threads execute in parallel. As long as all user threads execute as if they are alone we can say that the system (MySQL) scales well. But at some point we reach a limit and adding more user threads will not be useful or efficient.

Here's some basics with regard to MySQL:

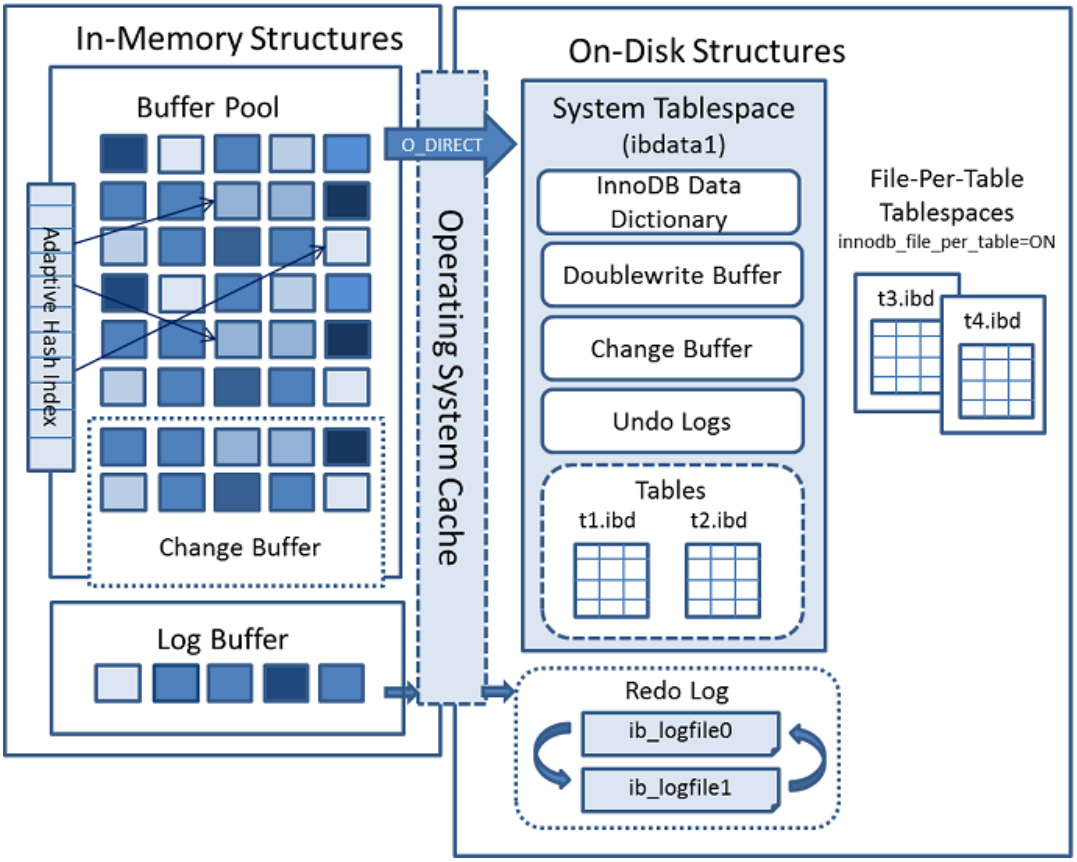
1. MySQL server is a single process application.
2. It is multithreaded.
3. It (usually) acts as a TCP/IP server, accepting connections.
4. Each connection gets a dedicated thread.
5. These threads are sometimes named processes, and sometimes they're referred to as connections.

```
mysql> show processlist;
+----+------+---------------------+------+---------+------+-------+------------------+
| Id | User | Host                | db   | Command | Time | State | Info             |
+----+------+---------------------+------+---------+------+-------+------------------+
| 17 | dba  | localhost           | NULL | Query   |    0 | init  | show processlist |
| 23 | dba2 | localhost           | NULL | Sleep   |  138 |       | NULL             |
| 24 | dba2 | 192.168.1.104:64838 | NULL | Sleep   |    3 |       | NULL             |
| 25 | dba2 | 192.168.1.104:64839 | NULL | Sleep   |    3 |       | NULL             |
+----+------+---------------------+------+---------+------+-------+------------------+

mysql> SHOW GLOBAL STATUS LIKE 'Threads_connected';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Threads_connected | 4     |
+-------------------+-------+
```

So, every new connection gets its own thread. Assuming no thread pool is in use, every new connection makes for the creation of a new thread, and a disconnect causes for that thread's destruction. Hence, there is a 1-1 mapping between connections and active threads.
But then, there is a thread pool, which means there can be threads which are not associated with any connection. So, the number of threads is greater than or equal to the number of connections.

MySQL Physical Architecture:



InnoDB In-Memory Structures (MySQL 5)

1. Buffer Pool
2. Change Buffer
3. Adaptive Hash Index
4. Log Buffer

1. The InnoDB Buffer Pool

On dedicated servers, up to 80% of physical memory is often assigned to the buffer pool.

When a data or index page is read from the ibdata files, it is also cached into the InnoDB Buffer Pool. When cached data changes, it is updated in the Buffer Pool and the page that contains it is flagged as dirty. The page will eventually be written to the ibdata files, but for performance reasons, MySQL will delay it for as long as possible.

As a rule of thumb, you should make the Buffer Pool as large as possible while still leaving enough memory for the MySQL and other processes running in your system. The innodb_buffer_pool_size is one of the most important parameters that can be fine-tuned.

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| innodb_buffer_pool_size | 134217728 |
```

Configuring Buffer Pool Flushing

InnoDB performs certain tasks in the background, including flushing of dirty pages from the buffer pool, a task performed by the master thread. Dirty pages are those that have been modified but are not yet written to the data files on disk. InnoDB aggressively flushes buffer pool pages if the percentage of dirty pages in the buffer pool reaches the innodb_max_dirty_pages_pct threshold.

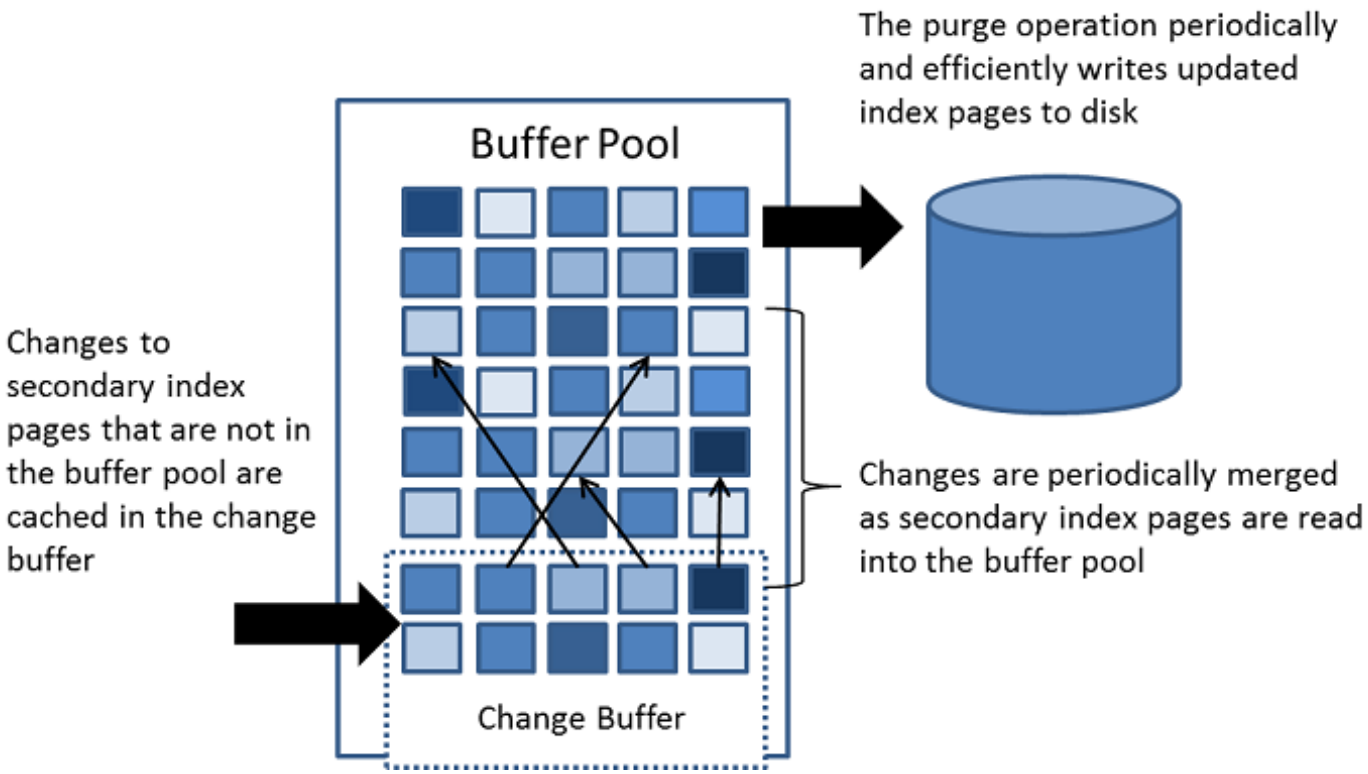
```
mysql> show variables like 'innodb_max_dirty_pages_pct';
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| innodb_max_dirty_pages_pct | 75    |
+-----------------------------+-------+
```

In MySQL 5.5, the default is 75; Means 75% of dirty pages are available in the buffer pool, rest 25% is flushed to the disk.

Setting innodb_max_dirty_pages_pct to zero (0) keeps InnoDB data pages maximally flushed to disk (up to 99.1 %) from the innodb buffer pool.

Without tweaking anything else, this also provides for a faster shutdown of mysqld in the presence of loads of InnoDB data since mysqld flushes the innodb buffer pool on shutdown.

2. Change Buffer



One of the challenges in storage engine design is random I/O during a write operation.

In InnoDB, a table will have one clustered index and zero or more secondary indexes. Each of these indexes is a B-tree. When a record is inserted into a table, the record is first inserted into clustered index and then into each of the secondary indexes. So, the resulting I/O operation will be randomly distributed across the disk. The I/O pattern is similarly random for update and delete operations. To mitigate this problem, the InnoDB storage engine uses a special data structure called the change buffer

1. The change buffer is another B-tree, with the ability to hold the record of any secondary index.

2. There is only one change buffer within InnoDB and it is persisted in the system Tablespace.
3. The size of the change buffer is configured using the innodb_change_buffer_max_size system variable.

```
mysql> show variables like 'innodb_change_buffer_max_size';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| innodb_change_buffer_max_size | 25    |
+-------------------------------+-------+
```

3. Adaptive Hash Index

It magically determines when it is worth supplementing InnoDB B-Tree-based indexes with fast hash lookup tables and then builds them automatically without a prompt from the user.

The adaptive hash index feature is enabled by the innodb_adaptive_hash_index variable, or turned off at server startup by --skip-innodb-adaptive-hash-index.

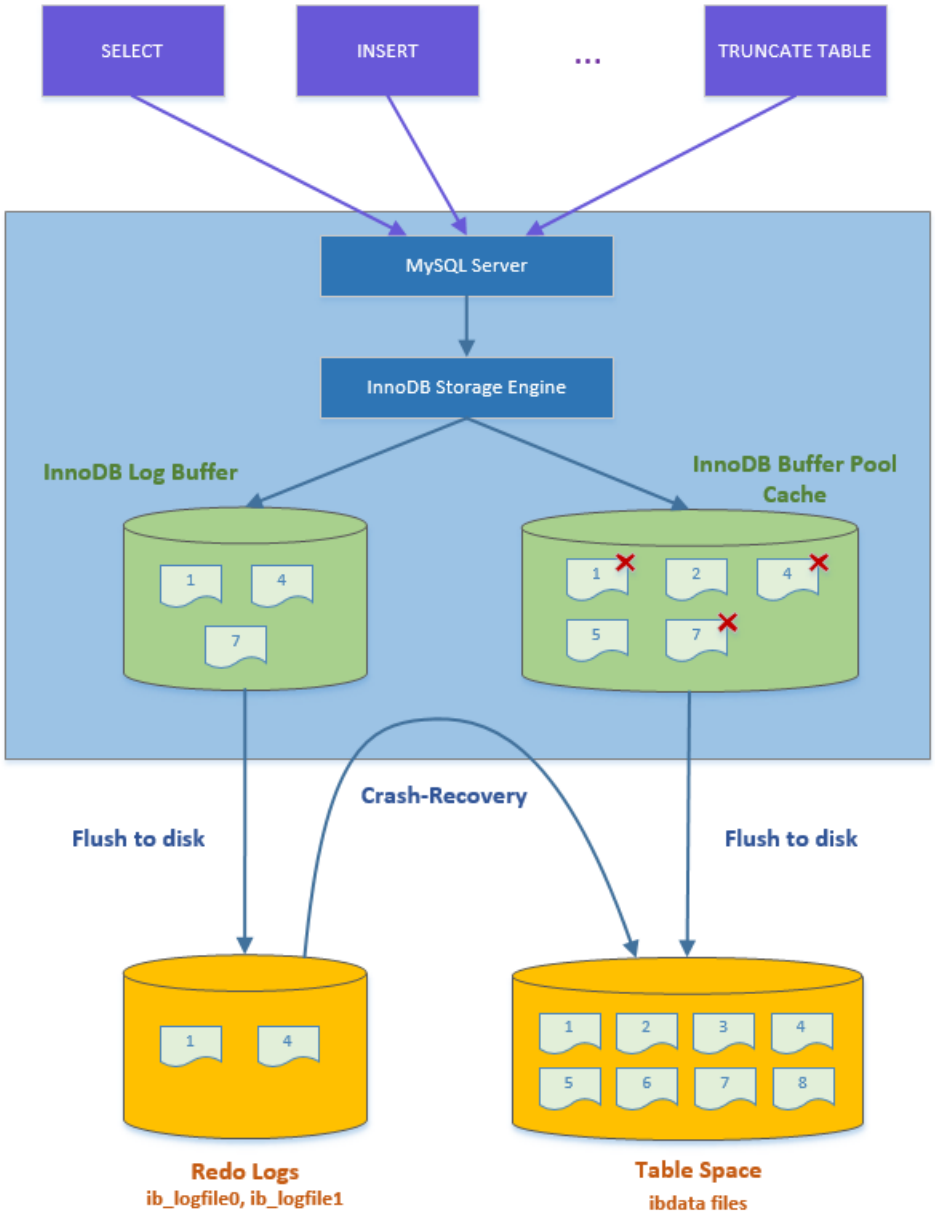
If a table fits almost entirely in main memory, a hash index can speed up queries by enabling direct lookup of any element, turning the index value into a sort of pointer. InnoDB has a mechanism that monitors index searches.

If InnoDB notices that queries could benefit from building a hash index, it does so automatically.

```
mysql> show variables like 'innodb_adaptive_hash_index';
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| innodb_adaptive_hash_index | ON    |
+-----------------------------+-------+
```

4. Log Buffer

The page will also be written to the InnoDB Log Buffer. This log buffer is typically flushed to disk at transaction commit. These are the redo logs, usually named ib_logfile0 and ib_logfile1.
If MySQL crashes, the InnoDB Pool Cache will be lost given it is an in-memory cache. To recover lost data, MySQL will read the InnoDB Logs files and restores all pages that were flagged as dirty. This recovery happens automatically at start up after a crash



Server Crash-Recovery

If MySQL doesn't shutdown cleanly, the InnoDB storage engine will try to recover automatically next time it starts up. It will try to ensure all InnoDB tables are in a consistent state. In a nutshell, MySQL will complete transactions in the redo logs not yet flushed to disk. Similarly, it will roll back transactions not yet committed.

If it can't, it will shut down the server with an appropriate error message.

Upon restart of the MySQL server, the Tablespace discovery process starts. The tablespace discovery process is carried out by InnoDB to find out the tables for which "REDO" logs have to be applied.

1. Redo present after the last checkpoint Aplied ---> Tablespace
2. log buffer Flushed → Tablespace
3. Uncommited changes rollback via Undo → Tablespace

Crash recovery operation works regardless of the innodb_fush_log_at_trx_commit setting.

It is important that the **innodb_flush_log_at_trx_commit** is set to the default value 1 to maintain full ACID compliance. This property determines how frequently the log buffer is flushed to disk. With value 1 it is guaranteed that after a transaction commit log files are flushed to disk.

a) With value **0** (log buffer → logfile,Every 1 sec), the contents of the log buffer are written to logfile approximately once 1sec.
A value of 0 is the fastest choice but less secure. Any mysqld process crash can erase the last second of transactions.

b) With value **1** (log buffer → logfile(Commit)), the contents of the log buffer are written out to the log file at each transaction commit.
A value of 1 is the safest choice because in the event of a crash you lose at most one statement or transaction from the binary log. However, it is also the slowest choice.

c) With value **2** (log buffer → logfile(Commit),Every 1 sec), the contents of the log buffer are written out to the log file at each transaction commit, or approximately 1sec.
A value of 2 is faster and more secure than 0. Only an operating system crash or a power outage can erase the last second of transactions.

```
mysql> show variables like 'innodb_flush_log_at_trx_commit';
+--------------------------------+-------+
| Variable_name                  | Value |
+--------------------------------+-------+
| innodb_flush_log_at_trx_commit | 1     |
+--------------------------------+-------+
```

Doublewrite Buffer (MySQL 5.6) – (Cache for 100 pages incase disk is unavailable)

The doublewrite buffer is a storage area located in the system tablespace where InnoDB writes pages that are flushed from the InnoDB buffer pool, before the pages are written to their proper positions in the data file.
Only after flushing and writing pages to the doublewrite buffer, does InnoDB write pages to their proper positions.
If there is an operating system, storage subsystem, or **mysqld** process crash in the middle of a page write, InnoDB can later find a good copy of the page from the doublewrite buffer during crash recovery.

Innodb uses this buffer to avoid data corruption that happens in case of any partial page writes, this partial page write may happen in case of disk get completely full, may be due to power outages, crash or a bug.
This buffer is a reserved area capable of storing 100 pages (16KB*100).It is located inside ibdata1 and it is more like a backup of the recently written 100 pages. When innodb flushes its pages from buffer to the disk, it first writes to this buffer and then to the disk. Every data is written twice with double write buffer. In case of a partial page writes to the disk the Innodb could recover the pages from the double write buffer, this happens vice versa, whenever the Innodb discovers a page corruption, its checks this buffer or clean page. The performance over head by enabling this buffer is negligible as the writes are sequential. The variable which controls this buffer is innodb_doublewrite make it 0 or 1 to disable and enable accordingly.

The doublewrite buffer is enabled by default in most cases. To disable the doublewrite buffer, set innodb_doublewrite to 0.

```
mysql> show variables like 'innodb_doublewrite';
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| innodb_doublewrite | ON    |
+--------------------+-------+
```

Redo Log (Inno db only)

The redo log is a disk-based data structure used during crash recovery to correct data written by incomplete transactions. During normal operations, the redo log encodes requests to change table data that result from SQL statements or low-level API calls. Modifications that did not finish updating the data files before an unexpected shutdown are replayed automatically during initialization, and before the connections are accepted.

By default, the redo log is physically represented on disk by two files named ib_logfile0 and ib_logfile1.

MySQL writes to the redo log files in a circular fashion. Data in the redo log is encoded in terms of records affected; this data is collectively referred to as redo. The passage of data through the redo log is represented by an ever-increasing LSN value.

```
mysql> show variables like 'innodb_log_file_size';
+----------------------+----------+
| Variable_name        | Value    |
+----------------------+----------+
| innodb_log_file_size | 50331648 |
+----------------------+----------+


-- To increase the number of log files
mysql> show variables like 'innodb_log_files_in_group';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| innodb_log_files_in_group | 2     |
+---------------------------+-------+
```

As of MySQL 5.6.8, the innodb_fast_shutdown setting is no longer relevant when changing the number or the size of InnoDB log files. Additionally, you are no longer required remove old log files, although you may still want to copy the old log files to a safe place, as a backup.

To change the number or size of InnoDB log files, perform the following steps:

1. Stop the MySQL server and make sure that it shuts down without errors.
2. Edit my.cnf to change the log file configuration. To change the log file size, configure innodb_log_file_size. To increase the number of log files, configure innodb_log_files_in_group.
3. Start the MySQL server again.

If InnoDB detects that the innodb_log_file_size differs from the redo log file size, it will write a log checkpoint, close and remove the old log files, create new log files at the requested size, and open the new log files.