

# Introduction to the MySQL Falcon OLTP Storage Engine

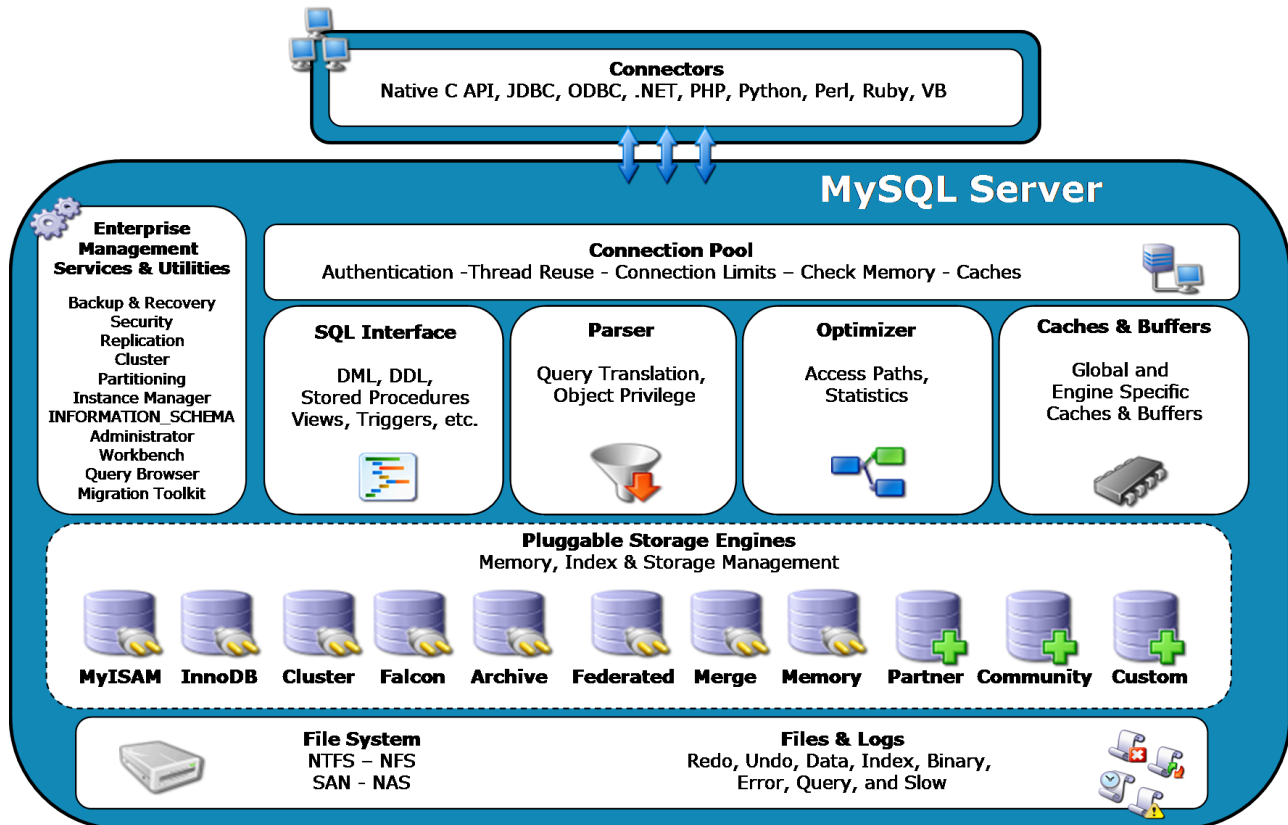
Arjen Lentz (arjen@mysql.com), Support Engineer / Trainer, MySQL AB

September 2006

Original paper by Robin Schumacher, Director of Product Management, MySQL AB

## The MySQL Pluggable Storage Engine Architecture

One key differentiator between MySQL and other database platforms – whether they are proprietary or open source – is the pluggable storage engine architecture of MySQL. This architecture allows a database professional to select a specialised storage engine for a particular application component, while being completely shielded from managing any specific application coding requirements. Graphically depicted, the MySQL pluggable storage engine architecture looks like this:



The pluggable storage engine architecture provides a standard set of management and support services that are common among all underlying storage engines. The storage engines themselves are the components of the database server that actually perform actions on the underlying data that is maintained at the physical server level.

This modular architecture provides performance and manageability benefits for those wishing to specifically target a particular application component's need – such as data warehousing, transaction processing, high availability situations, etc. – while enjoying the advantage of utilizing a set of interfaces and that are independent of any one storage engine.

From a technical perspective some of the key differentiations in storage engines include:

- **Concurrency/Locking** – some applications have more granular lock requirements (such as row-level locks) than others. Choosing the right locking strategy can reduce overhead and therefore help with overall performance. This area also includes support for capabilities like multi-version concurrency control or “snapshot” read.
- **Transaction Support** – not every application/component needs transactions, and for those that don't, the overhead of such support can be avoided. But for those that do, there are well defined requirements in a number of MySQL's storage engines, like ACID compliance and more.

- **Physical Storage** – this involves everything from the overall page size for tables and indexes as well as the format used for storing data on a physical medium (usually disk). Different storage strategies definitely have an impact on the overall performance of both read and write operations.
- **Index Support** – different application scenarios tend to benefit from different index strategies, and so each storage engine generally has its own indexing methods, although some indexing mechanisms (like B-tree indexes) are common to nearly all engines.
- **Memory Caches** – MySQL offers a couple of storage engines whose data only resides in RAM, which results in very high performance. Others use different variations of memory caches for transactions, transaction logs, table dictionaries, row data, and indexes.
- **Performance Aids** – includes things like multiple I/O threads for parallel operations, thread concurrency, database checkpointing, bulk insert handling, and more.

Each of the pluggable storage engine infrastructure components are designed to offer a selective set of benefits for a particular application. Conversely, avoiding a set of component features helps steer clear of unnecessary overhead. So it stands to reason that understanding a particular application's set of requirements and selecting the proper MySQL storage engine can have a dramatic impact on overall system efficiency and performance. And a DBA can put multiple storage engines in play for the same application, which equates to the DBA having an extremely flexible and high-performance framework at their disposal for satisfying the exact requirements of an application.

Beginning in MySQL 5.1, a storage engine plug-in interface is provided to dynamically and easily install/uninstall storage engines from the MySQL Server. Before a storage engine can be used, the storage engine plugin shared library must be loaded into MySQL using the `INSTALL PLUGIN` statement. For example, if the `EXAMPLE` engine plugin is named `ha_example` and the shared library is named `ha_example.so`, a DBA would load the engine into MySQL with the following statement:

```
INSTALL PLUGIN ha_example SONAME 'ha_example.so';
```

To unplug a storage engine from the MySQL Server, the `UNINSTALL PLUGIN` statement is used:

```
UNINSTALL PLUGIN ha_example;
```

## The Falcon Storage Engine

One of MySQL's new additions is a storage engine code-named "Falcon". Falcon is a next-generation transactional engine especially designed for modern hardware and operating system environments.

### *Jim Starkey & Netrastructure*



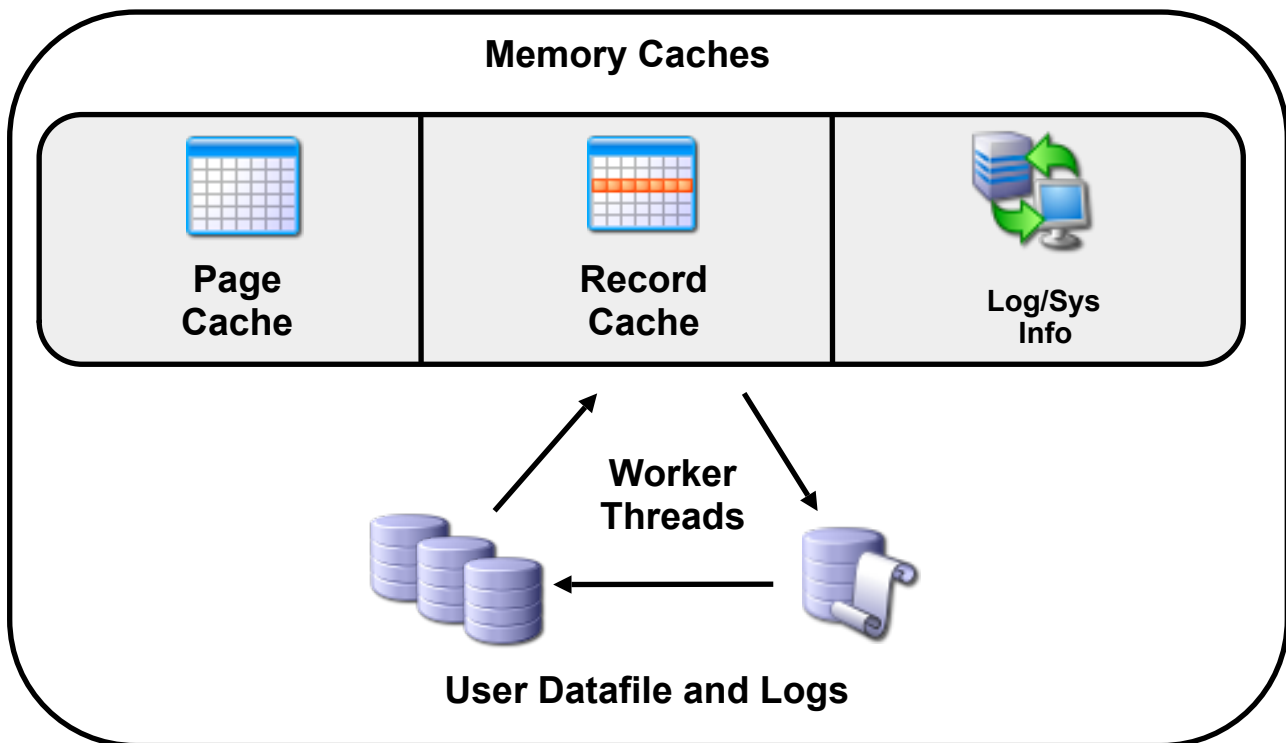
In February 2006, MySQL AB acquired Netrastructure, a company set up by Jim Starkey, one of the more well known architects of modern relational databases. Jim has been responsible for a number of engines and inventions, including Interbase, multi-version concurrency control, and the blob (named after the movie, so Jim insists: the acronym was invented later by some marketing types!)

At the core of Netrastructure is a new database engine, Jim's latest brainchild, which Jim and his team are now integrating with the MySQL Server as the Falcon storage engine.



### *Falcon Architecture Overview*

The MySQL Falcon architecture is both an advanced and simplified design that makes for a high performing transactional database that requires little maintenance or troubleshooting on the part of database administration staff. The architecture of the Falcon engine is depicted below:



The architecture consists of four basic components:

- **User Datafile** – contains the actual user data stored in a Falcon database.
- **Falcon Log** – contains recently committed transactional information and handles crash recovery activities for a database.
- **Memory Caches** – hold data (including BLOB), index, log, and transaction information for quick access.
- **Worker Threads** – handle all I/O operations such as reading/writing data to the Falcon Log, moving data from datafiles to the memory cache, etc.

Each of these components is covered in more detail in the following sections.

## ***Falcon Storage***

Falcon offers a variety of flexible options and automatic management features that make the engine very easy to setup and maintain. First, unlike some of the other MySQL storage engines that offer fixed page sizes, Falcon allows a DBA to choose the page size that best fits their application. Available page sizes range from 2K to 32K. For the initial integration process, the page size is currently configurable on an instance-wide basis.

### **Storage for User Data**

In the first integration stage, Falcon creates a single datafile for each database in which it stores all table, index, and BLOB data. Falcon creates this datafile automatically when the first Falcon table is created by an end user request. The location of the datafile corresponds to the data directory that is specified in the my.cnf file. The Falcon datafile offers automatic storage extension when needed and basic automatic space reclamation, which makes reorganizing tables and indexes mostly unnecessary. The datafile's maximum size can also be controlled.

Falcon internally uses simplified data types, such as *string* and *number* which are then optimised for speed and minimum storage space. For instance, a small range of numbers around 0 are stored particularly efficiently since they tend to be the most often used. From the MySQL server's perspective though, all the usual data types are available which are automatically mapped to/from Falcon's internal set.

Rows are stored densely on pages, with updates that increase row length being automatically handled by Falcon so DBAs don't need to worry about periodic defragmentation jobs.

The maximum datafile storage size currently stands at over 100TB (around 116TB).

## The Falcon Log

Besides data storage, Falcon uses another storage structure called the Falcon Log for each database to manage write-ahead logging and crash recovery. Two physical files actually make up the Falcon Log. Each block in the log files has a header that includes a unique 64-bit identifier, the length of the block, the creation time of the database to which it belongs, and the block number of the oldest unprocessed block in the log file. Transactions are written to the first Falcon Log file until it reaches a fixed storage limit. Once this limit is reached, it is closed for writing with new transaction data then being applied to the second file, which can automatically extend to accommodate any transaction demand. Log data is flushed from the first Falcon Log file to the database and once this process completes, the first file is re-created and the second Falcon Log file is closed for new transaction writes, with the first file re-opening for incoming transactions (with the second file's data being flushed to the database). This process continues in round-robin fashion.

One distinct aspect of the Falcon Log is there are never any true uncommitted transactions that appear in the log, so it does not serve as an "undo" or rollback mechanism. Simply put, only data that is intended to be found in the database ever makes it to the Falcon Log. This makes for near-instant rollback operations regardless of transaction size.

Crash recovery in the Falcon engine is handled by the Falcon Log, with transactions that have not been applied to the user datafile (prior to a system crash) being written to the database upon restart of the system.

The Falcon Log's physical location is defaulted to the data directory of the database, but can be changed by the DBA to be somewhere else on the system, which helps reduce I/O contention at the disk level.

## Falcon Memory Caches

Falcon was designed to perform best on systems with generous amounts of memory. The memory caches utilised by Falcon are similar in some respects with other RDBMSs and MySQL engines; however, the cache structures offer a number of improvements over traditional memory caching strategies. The mechanisms used by Falcon with respect to memory caching include:

- Log/System Cache – log information is kept in memory and flushed to the Falcon Log when transactions commit. Metadata needed by Falcon (system structures, etc.) are also maintained in memory for quick reference.
- Page Cache – the page cache holds index pages and BLOB data read from disk for a particular database, along with various object structures for fast access.
- Record Cache – the record cache is a specialised memory region devoted to holding rows that have been requested by end-user queries for a particular database. Note that this cache differs from traditional data caches in that only specific rows needed by applications reside in the cache as opposed to entire pages (which may contain only subsets of needed information). This technique guarantees that only active data needed to satisfy user requests is in memory, shortens row access time, and reduces cache bloat by not including unrequested information. The record cache also contains in-process transaction information and assists in supporting the multi-version concurrency control (MVCC) mechanisms of the Falcon engine. Because of the support the record cache supplies to transactions, a scavenge process is used to ensure only "hot" data and transactional information resides in the cache. Falcon surveys the demographics of the generational data in the cache, and removes the oldest generations. This process is more complicated than the standard LRU algorithm used by many database systems, but it is more efficient and faster. The scavenge cycle is controlled by a configuration parameter, which can be used by the DBA to determine how often the process is run.

## Falcon Worker Threads

All I/O in the Falcon engine are handled by a pool of worker threads that address various tasks such as managing a database's Falcon Log, reading/writing data from disk and memory, and more. No specific thread(s) are assigned to various tasks, but each are assigned work when it appears in the work queue.

The DBA can control the number of worker threads assigned to manage Falcon needs.

## *Transaction and Object Management*

The following sections cover how the MySQL Falcon engine manages transactions, concurrency, as well as what types of objects are supported within Falcon.

## Transaction and Lock Management in Falcon

The first thing to understand about transactions and Falcon is that the engine takes a multi-generational approach to managing both transactions and concurrency. This means that the engine keeps multiple

iterations/generations of rows available in memory to ensure the highest possible levels of uninterrupted data access.

## **Transaction Handling**

Falcon supports ACID-level transactional operations, which are handled in memory. Once in-process transactions are committed, the operations are flushed to the Falcon Log for asynchronous application to the database files. The only exceptions to this rule are BLOB and index changes, which are immediately applied to the database files upon transaction commit.

Falcon differs from proprietary databases such as Oracle as well as other open source RDBMSs like Firebird in that all transactional operations are kept in memory inside the record cache. Falcon maintains a 4-byte transaction ID in the record header of the rows involved in the transaction, and also transfers this transaction ID to records written to a database's Falcon Log. Each transaction contains a picture/snapshot of the data that existed the moment the transaction was submitted, and therefore, it is very possible to have multiple versions of data being maintained within the Falcon record cache.

As was mentioned, each row involved in a transaction contains a transaction ID of the transaction that created it. Rows with older versions of data include a pointer to the older picture of the data, and deleted rows are represented in memory by a row header with no data and contain a flag that is set to indicate a deleted row.

If the Falcon engine has not been assigned an adequate amount of memory for the record cache, a transaction may page to disk if the operation exhausts the maximum amount of record memory assigned to the database. Note that the Falcon Log is used to manage index and BLOB data that are part of a transaction.

Falcon supports distributed transaction control (via XA), offers auto/non-auto commit (which is accomplished above the storage engine layer in MySQL), and provides savepoints and group commit for intelligent transaction control.

## **Concurrency Control**

As previously stated, Falcon is a multi-generational transaction engine that uses MVCC as its primary concurrency control mechanism. This means that readers never block writers and vice versa, with the end result being exceptional access to data whenever it is needed.

The Falcon engine defaults to the repeatable read isolation mode level with no phantoms being possible. When a transaction attempts to modify or delete a row that another in-process transaction has already obtained, then the previous transaction will wait until the other transaction either commits or rolls back. If the first transaction commits, then the other transaction receives an error and must try again. However, if the first transaction rolls back, then the other transaction will succeed. It's important to note that all transactions will always read and operate on consistent data, which mirrors how many proprietary databases like Oracle operate.

The MySQL Falcon engine uses a sophisticated graphing model for deadlock detection that uses a lock table to resolve lock dependencies and deadlock issues. This form of deadlock resolution is superior to the standard timeout and like models that other databases systems use.

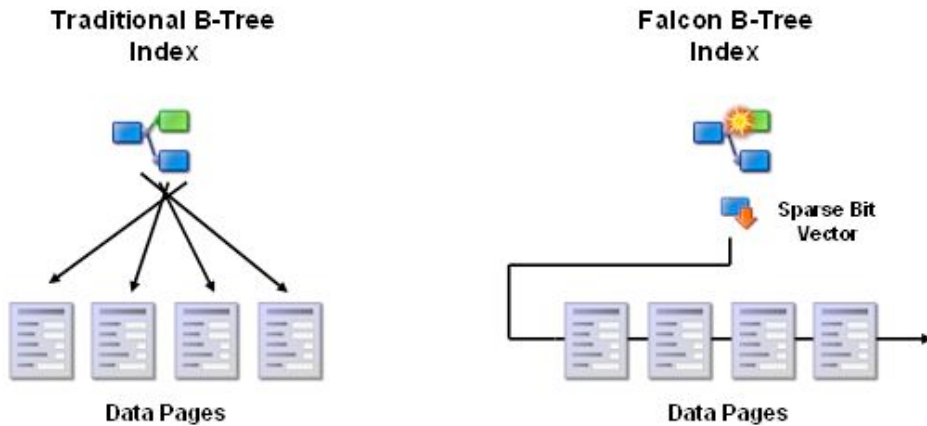
## ***Table and Index Management in Falcon***

In terms of storage, both tables and indexes are stored within the user datafile that is created by the Falcon engine when the first Falcon table is created in a MySQL instance. Falcon supports standard heap tables along with all datatypes available within MySQL. Tables can support up to 32,000 columns and house up to four billion rows.

With respect to indexes, Falcon uses an advanced form of B-tree indexing that provides a number of the benefits of clustered indexes without the drawbacks that come from the use of such structures. Traditional database index implementations traverse indexes by bouncing between index pages and database pages, which can oftentimes lead to inefficient or costly disk access.

Clustered indexes (or index-organised tables) are structured so that the physical ordering of records corresponds to the index order, with the actual leaf pages being the data pages. While some applications benefit from this organization, the actual physical implementation of clustered indexes can lead to space management problems, such as page splitting and more.

Falcon B-tree indexes work different in that the index is scanned first, with bits being set in a sparse bit vector to indicate selected records. Records and data pages are then processed in bit order, which is also physical order by disk.



Falcon's indexing schema results in a number of benefits. First, all indexes behave like well-tuned clustered indexes. Second, indexes aren't subject to two phase locking. Finally, Boolean "and" and "or" operations can be performed on intermediate bitmaps at virtually no cost, eliminating the need for the optimiser to chose between alternative indexes.

Internally, the Falcon engine already has an advanced full text indexing system which can operate across multiple tables. This feature is however not part of the initial integration process (so that the engine becomes available for use as soon as possible), but instead will be worked in in the next stage.

One final note on Falcon indexes is that they are online in nature: adding/dropping indexes do not block table access for any DML operations.

## Comparison of Falcon Engine Features

The following table compares and contrasts the Falcon engine with other popular MySQL storage engines in use.

Feature	Falcon	InnoDB	NDB Cluster	MyISAM
Crash Recovery	☑	☑	☑	
Online Parameter Support	☑			☑
Foreign Key Support	☑	☑		
Tablespaces	*	☑	☑	
Automatic File Extension	☑	☑	☑	☑
Data Caching	☑	☑	☑	
Index Caching	☑	☑	☑	☑
ACID Transactions	☑	☑	☑	

Distributed Transaction Support	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Implicit Savepoints	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
MVCC (Full/Real) Support	<input checked="" type="checkbox"/>	*		
Row-level Locks	*	<input checked="" type="checkbox"/>		
Deadlock Detection	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
GIS support	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
B-Tree Indexes	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Clustered Indexes	*	<input checked="" type="checkbox"/>		
Full Text Indexes	V2			<input checked="" type="checkbox"/>

## Timeline

Jim's team is working hard to make the Falcon storage engine available for users within 2006. It is then upto you, our 10 million users, to put this engine to the test and catch those bugs that can only be caught in real world environments. We encourage everybody to download the early versions and try them on their development and testing systems. It should then become production quality within 2007.