solid.

solidDB for MySQL

solidDB

Table of Contents

Introduction 1

Architecture Overview 1

- **MySQL Pluggable Storage Engine Architecture** 1
- **solidDB for MySQL** 2
- **solidDB Storage Engine** 3

Features 4

- **Transaction Durability** 4
  - Strict Durability 4
  - Relaxed Durability 4
  - Which Durability Should You Use? 4
- **Online Backup** 5
  - Starting an On-line Backup 5
  - How the Backup Works 5
  - Restoring from a Backup 5
- **Multiversioned Concurrency Control** 6
  - Multi-Version Concurrency Control 6
  - Pessimistic Concurrency Control 6
  - Optimistic Concurrency Control 7
  - Differences Between Optimistic and Pessimistic Concurrency Control 7
- **solidDB for MySQL Table and Index Structures** 8
  - Main Storage Tree 8
  - Bonsai Tree 8

Performance 9

Conclusion 11

About Solid Information Technology 11

References 12

Introduction/Architecture Overview

Introduction

solidDB for MySQL is a complete relational database management system that combines MySQL Server, solidDB Storage Engine and MyISAM storage engine.

The product that users install, configure and access is not only 100% compatible with MySQL Server, but includes exactly the same code as the regular MySQL Server. As a result, solidDB for MySQL can be accessed with MySQL Drivers and tools that have been developed by MySQL AB and the community.

solidDB Storage Engine provides a fast, scalable, transactional engine to store and manage data, while MySQL Server offers the "upper layers" of the database stack. These include SQL interface, parser, optimizer, connection pool, management services and utilities as well as the database framework with configuration files, directory structure, and command line tools that MySQL users are already familiar with. MyISAM storage engine stores the data dictionary and provides an efficient engine for non-transactional applications.

Architecture Overview

MySQL Pluggable Storage Engine Architecture

Databases are primarily used to share data between multiple concurrent users, and because of this, they need to be sophisticated pieces of software. They are used in many different environments, with different types of requirements and needs. A variety of applications make use of databases, and performance is heavily dependent on how the application is using the database. The workloads in these environments can also be quite different. The "one size fits all" approach used by many databases is very restrictive. It is hard, if not impossible, for software to try to optimize everything simultaneously. If increased database performance and scalability are desired, there must be an understanding of requirements and needs to guide the selection of an appropriate storage engine.

solidDB Storage Engine has been architected to plug into the MySQL Server, using the MySQL Pluggable Storage Engine Architecture, or PSEA. PSEA enables the choice of the most appropriate storage engine for the task at hand, based on particular application workloads. For instance, not all storage engines are transactional. If transactional capabilities are not required, then a storage engine that is optimized for non-transactional operations may be appropriate. On the other hand, many applications do need support for transactions and the requisite storage engine functionality.

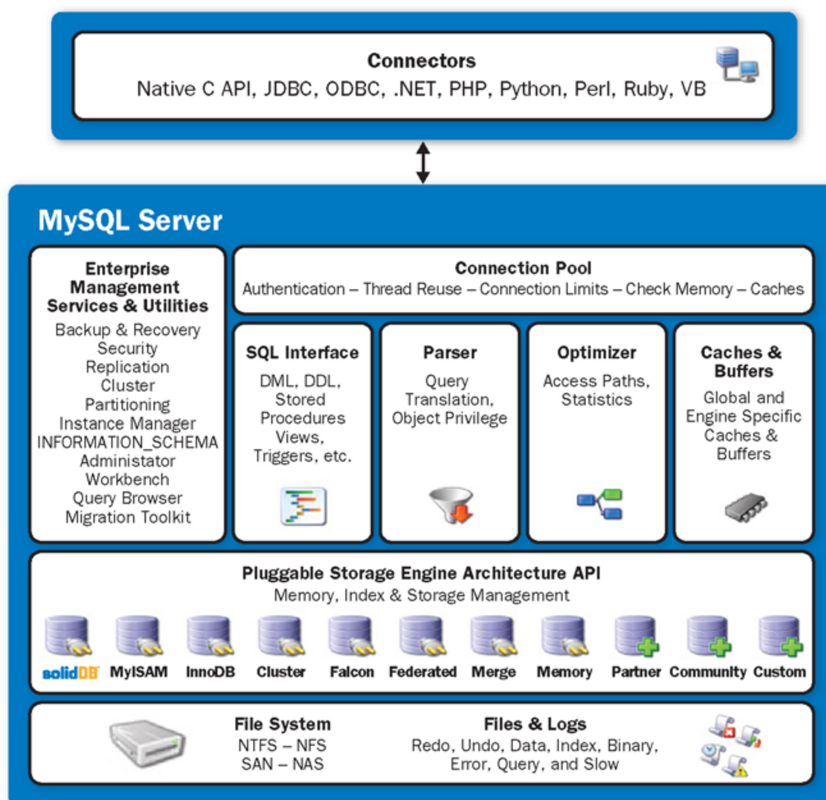

Figure 1. *MySQL Pluggable Storage Engine Architecture*

Architecture Overview

The MySQL PSEA also abstracts details of the storage implementation from the application programmer. Programmers and database administrators can use a single database with a single, consistent SQL interface layer and a single, consistent configuration approach. This greatly simplifies the development task, and related maintenance.

By using the PSEA *(Figure 1)*, all storage engines appear to be the same. They can all be plugged into the MySQL Server in the same way, and the user can interact with the MySQL and its underlying storage engine in the same way. However, despite the common interface and pluggable ability, each storage engine brings a different set of features and performance capabilities. The choice of storage engine must be carefully considered in order to ensure that the data management layer can meet the needs of the applications it supports.

solidDB for MySQL

solidDB for MySQL consists of MySQL Server, MyISAM Storage Engine, solidDB Storage Engine, and a high performance on-line backup utility. solidDB Storage Engine is used for transactional operations and the MyISAM storage engine for non-transactional tasks. Solid offers full support for the entire solidDB for MySQL product as depicted in *Figure 2*.

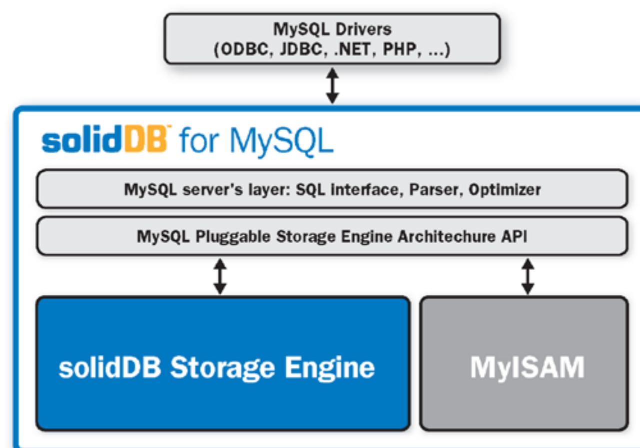

Figure 2. *High level view of solidDB for MySQL*

Solid fully expects that solidDB for MySQL will be used both in scenarios that require full transactional support as well as those that do not. The solidDB for MySQL product contains both solidDB Storage Engine and MyISAM storage engine. There are two reasons why both engines are included:

- For transactional operations, solidDB Storage Engine is recommended because it is designed to scale for transactional operations. MyISAM Storage Engine is written for high performance, non-transactional operations, and Solid recommends MyISAM Storage Engine for those types of workloads.
- The data dictionary for MySQL Server system tables is currently stored using MyISAM Storage Engine.

The appropriate storage engine can be specified on a per table basis, so within a given application both transactional and non-transactional needs can be addressed. If it is unclear how a particular table will be used by an application, solidDB Storage Engine will work well for both transactional and non-transactional operations, even though it has been written to scale especially well for transactional operations. A more detailed view of solidDB for MySQL is shown in *Figure 3*.

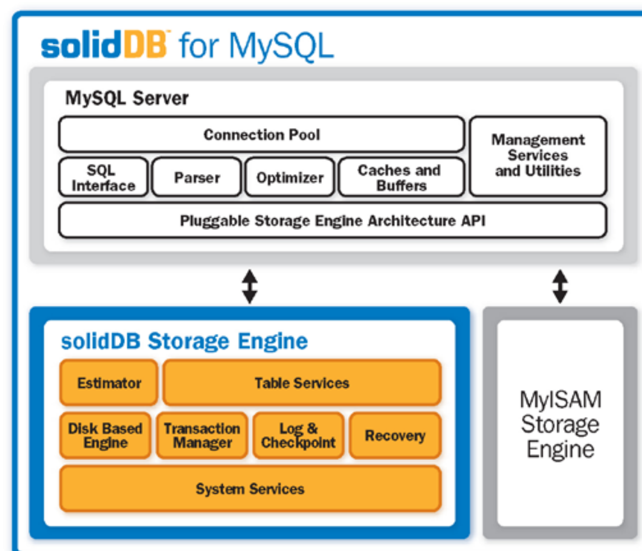

Figure 3. *Detailed View of solidDB for MySQL*

Architecture Overview

solidDB Storage Engine

solidDB Storage Engine does more than just physically read and write data to disk. For example:

- The Cost Estimator performs various types of cost analysis on the incoming query, and generates a lower-level execution plan for how to retrieve data as fast as possible.
- Table Services handle single-table access and all of the standard data types that solidDB Storage Engine supports. Although MySQL Server has a number of caching mechanisms, it is important for the storage engine to have its own caches that can be optimized for the particular storage implementation. The Table Services layer uses both table and index caches.
- The solidDB Disk-based Engine uses Solid's unique Bonsai Tree implementation, and handles the index merging between the Bonsai Tree in memory and the corresponding indexes on disk.
- The Transaction Manager is responsible for maintaining true ACID compliance within the storage engine layer. solidDB supports multi-version concurrency control, and the concurrency conflict checking is performed in the Transaction Manager.
- Logging is very important in a storage engine, as it provides info that the Recovery layer needs. The logging layer also has different durability options.
- Recovery is another important piece of the storage engine. solidDB Storage Engine supports full roll-forward recovery, which is done automatically when the database is restarted after a crash or other severe condition.
- The System Services layer provides an abstraction to the underlying platform where the storage engine is running. This allows solidDB Storage Engine to run easily on many different operating systems and machine architectures.

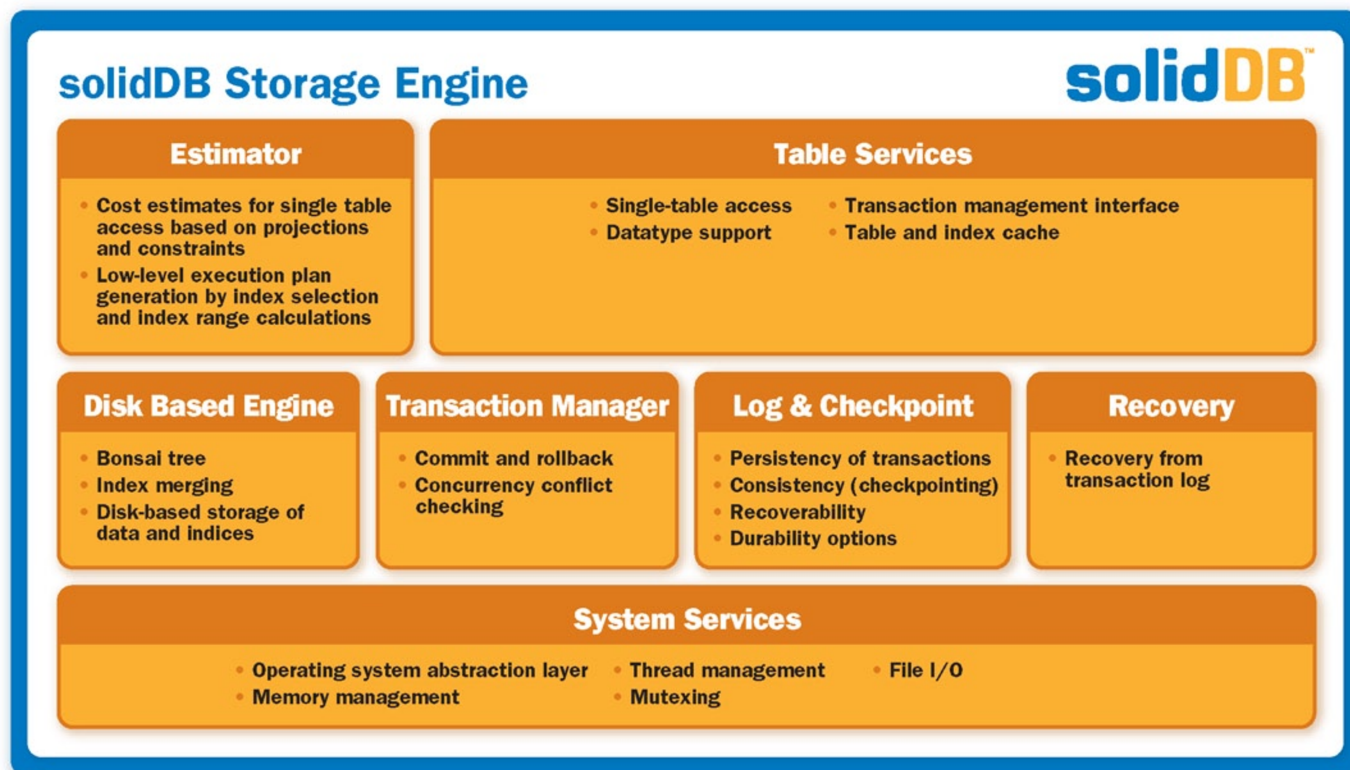

Figure 4. *Detailed View of solidDB Storage Engine*

Features

solidDB for MySQL has a number of additional features and performance improvements that make it uniquely positioned to be a solution for Enterprise-quality workloads.

The following features will be discussed in detail:

- Transaction durability
- On-line backup
- Multi-version concurrency control; including both optimistic and pessimistic concurrency control.
- solidDB Table and Index Structures; including the Main Storage Tree and the Bonsai Tree

Transaction Durability

Normally, when a transaction is committed, the database writes data to two locations: the database file and the transaction log file. However, data is not necessarily written to those two locations at the same time. When a transaction is committed, the database writes the data to the transaction log file as soon as the database commits the transaction. However, the database does not necessarily write data to the database file immediately. The database may wait until it is less busy, or until it has accumulated multiple changes, before writing data to the database file.

If the database shuts down abnormally (due to a power failure, for example) before all data has been written to the database file, the database can recover 100% of committed data by reading a combination of the database file and the transaction log file. The process of reading changes from the log file and updating the database file is called recovery. At the end of the recovery process, the database file is 100% current.

solidDB for MySQL supports both strict and relaxed durability. These durability types will be discussed below.

Strict Durability

The recovery process is automatically executed any time the database restarts after an abnormal shutdown. To have 100% recovery, there must be 100% of the transactions written to the log file. Normally, the database writes data to the log file at the same time that the database commits the data. Committed transactions are stored on disk and will not be lost if the computer is shut down abnormally. This is known as strict durability. The data that has been committed is durable, even if the database is shut down abnormally.

If durability is strict, data is written to the disk drive at the same time that it is committed to the database. The user is not told that the data has been committed until the data is successfully written to the transaction log on disk. This ensures that the data is recoverable if the database shuts down abnormally. The process is transparent to the user, except that there may be a slight delay before the database is ready to respond to new requests. Strict durability makes it almost impossible to lose data unless the hard disk drive fails.

Relaxed Durability

Historically, the goal of most databases has been to maximize data integrity using strict durability by making sure that data is not lost due to a power failure or other similar problems. This approach is appropriate for many types of data, such as financial data, where it is unacceptable to lose track of even a single transaction.

Some databases have been designed to maximize performance, without regard to data integrity. This is acceptable in situations where the database can operate on the most recent set of data. This is called relaxed durability.

If durability is relaxed, the user may be told that the data has been committed before the data has been written to the transaction log on disk. The database may choose to delay writing the data to disk by waiting until there are several transactions to record. If durability is relaxed, it is possible for the database to lose committed transactions before the data was written to disk, for example, in the event of a crash.

Which Durability Level is Appropriate?

solidDB Storage Engine allows the specification of strict or relaxed durability. If it is acceptable to lose a small amount of recent data, and if performance is crucial, then relaxed durability may be preferable since it increases performance by decreasing the I/O load. Relaxed durability is appropriate when each individual transaction is not mission-critical. For cases where data cannot be lost, like financial transactions, strict durability is most appropriate.

Features

On-line Backup

The solidDB for MySQL product contains an on-line backup function. This online backup function is intended for creating backup copies of solidDB databases and MySQL system tables (i.e., those that reside in the “mysql” database).

Creating a backup is relatively straightforward, involving a single MySQL SET command. For the value of the command, you specify the words “backup” and then the destination directory. For example, on Linux, the user might decide to store a backup in the /tmp/backup/ directory. The destination directory is on the machine where the database is running, not on the client’s machine. The directory needs to exist before performing the backup or an error will result. Since the backup is started by issuing a SQL SET command, it can be issued from any SQL-enabled client, or even from the application. This makes it easy to automate backups without shell access to the database server machine. For example, a short backup script can be written that connects to the database once per day and issues the backup command.

Starting an On-line Backup

The on-line backup can be started by setting value of the SOLIDDB_ADMIN_COMMAND variable:

```
SET GLOBAL SOLIDDB_ADMIN_COMMAND = 'backup <backup_dir>';

SET GLOBAL SOLIDDB_ADMIN_COMMAND = 'backup';
```

The `<backup_dir>` parameter defines the full path to the directory where the backup copy is to be created. It is strongly recommended that the user not choose a directory that resides in the MySQL data directory. That directory is used by solidDB for MySQL for internal information, and using it may cause undesirable effects during backup.

How the Backup Works

When the backup command is executed, the SET command will block (i.e., you will not be able to send any more commands) and the user will need to wait until the backup has completed. When the backup completes, the user will receive a command status from solidDB for MySQL, When run in the MySQL command-line client, the user will be prompted with the mysql> prompt once the backup has completed.

Even though the user is not seeing much, there are a number of actions that are taking place:

1. All DDL statements for solidDB for MySQL tables are blocked. Any attempt to execute such a statement generates an error. The user should not try to make DDL changes during a backup. The backup directory is cleared, and all files and subdirectories are removed from the backup directory.
2. A new MySQL configuration file is created. The MySQL Server maintains many system variables that indicate how the file is configured. Because most of the variables can be changed dynamically by the SET statement, the backup command does not copy the initial option file(s) but creates a new configuration file instead (“my.ini” for Windows and “my.cnf” for other platforms). The configuration file includes value definitions for all those system variables where the current value differs from the default (i.e., the value was set from the command line, option file, or by the SET statement).
3. A checkpoint record is added to the end of log file. Then, the solidDB for MySQL database file and the transaction log file are backed up.
4. All solidDB for MySQL table definition (.frm) and trigger definition (.trg, .trn) files are copied to the backup directory. The table and trigger definition files are backed up using operating system-level copying of all subdirectories of the MySQL data directory to the backup directory. The backup directory may include table and trigger definition files for those tables that are not actually backed up (for example, tables from other storage engines).
5. Finally, all MySQL system tables (residing in the “mysql” database) are backed up and the DDL statements are unblocked.

Restoring from a Backup

Restoration of data from a backup is straightforward. The following steps need to be taken:.

1. Shutdown the MySQL Server
2. Confirm the correct MySQL option (configuration) file to start the database
3. Copy all files and subdirectories from the backup directory to the MySQL data directory
4. Restart the MySQL Server

Features

Concurrency Control

solidDB for MySQL relies on multiple types of concurrency control.

Multi-Version Concurrency Control

Multi-version concurrency control (MVCC) is built in to solidDB Storage Engine. It provides each user with a “snapshot” of the database for that user. MVCC guarantees that each transaction sees a consistent view of the database by reading non-current data for objects modified by concurrent transactions. Changes will not be seen by other users until the transaction is committed.

It avoids conflicts between reads and writes of the same object by maintaining several versions of the object.

Unlike other database products, solidDB for MySQL supports the additional optimistic concurrency control capabilities to support non-blocking reads and writes.

Pessimistic Concurrency Control

Pessimistic concurrency control is also known as locking. Locking allows multiple users to safely share a database as long as all users are updating different data. While some storage engines only support table-level locks, solidDB Storage Engine supports row-level locks. Row-level locks improve the performance in typical multiple-user scenarios because updates to different rows in the same table can be done simultaneously.

When pessimistic concurrency control is used, as soon as any piece of the row is updated it is locked making is impossible for two users to update a row at the same time. The disadvantage of pessimistic concurrency control is that the system must also check whether the requested row(s) are already locked by another user (or connection) requiring overhead for every operation. This overhead occurs whether or not two or more users are actually trying to access the same row. This overhead is small, but accumulates because every row that is updated requires a lock.

Pessimistic concurrency control is called “pessimistic” because the system assumes the worst – it assumes that two users will want to update the same row at the same time, and then prevents that possibility by locking the row, no matter how likely or unlikely conflicts actually are.

Here is an example of pessimistic concurrency control operation:

> **Note:** To activate pessimistic concurrency control, you must start mysqld with the “–soliddb_pessimistic=true” configuration option.
>
> On Transaction 1, enter the statements below:

```
create table t2(a int not null) engine=soliddb;
insert into t2 values (1)(2);
commit;
set autocommit = 0;
update t2 set a = 5 where a = 1;
```

> On Transaction 2, enter the statements below:

```
set autocommit = 0;
update t2 set a = 7 where a = 1; /* This query waits for
Transaction 1 to release locks on table t2 */
```

> On Transaction 1, enter the statements below:

```
commit;
```

> Transaction 2 can now continue execution.

Features

Optimistic Concurrency Control

Optimistic concurrency control is an alternative to pessimistic concurrency control. Optimistic concurrency control assumes that contention for a given set of data is rare. Instead of locking every row when used, the database looks for indications that two users are trying to update the same row at the same time. If this is the case, then one user's updates are discarded and the user is informed.

When using optimistic concurrency control, each time that the database tries to update a row the row is versioned and the version number is stored for reference purposes. When it is time to write the updated data back to the disk drive, the database compares the original version number with the version number that the disk drive contains. If the version numbers are identical, then there has not been data contention and the row can be written to the database. However, if the row versions do not match then action needs to be taken to ensure data integrity. Here is an example of optimistic concurrency control operation:

On Transaction 1, enter the statements below:

```
create table t1(a int not null) engine=soliddb;
insert into t1 values (1)(2);
commit;
set autocommit = 0;
update t1 set a = 5 where a = 1;
```

On Transaction 2, enter the statements below:

```
set autocommit = 0;
update t1 set a = 7 where a = 1;
```

On Transaction 2, you can see:

```
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

Differences Between Optimistic and Pessimistic Concurrency Control

When using optimistic concurrency control, the conflict isn't revealed until just before the updated data is written to the database. When using pessimistic concurrency control, the conflict is revealed as soon as data read is attempted. By analogy, pessimistic locking is like having a guard at the bank door that checks account numbers upon entry into the bank. If someone else (e.g., a spouse) is already in the bank accessing the account, then nobody else can enter until the other person finishes her transaction and leaves. On the other hand, optimistic locking allows someone to walk into the bank at any time and conduct business. However, the risk exists that when leaving the bank the guard will prevent exit because there was a transaction conflict that requires the account holder to get back in line and execute the transaction again.

In addition to blocking updates, optimistic and pessimistic concurrency control differ in another important way. Pessimistic locking blocks a user from reading even a locked row. Optimistic concurrency doesn't check for conflicts until the point when updated data is being written to disk.

solidDB Storage Engine contains a sophisticated implementation of optimistic concurrency control. Rather than giving each user whatever version of data that is on the disk at the moment it is read, solidDB Storage Engine can temporarily store multiple versions of each data row. Each user's transaction sees the database as it was at the time that the transaction started. This way, the data that each user sees is consistent throughout the transaction, and users are able to concurrently access the database. Data is always available to users because locking is not used, and access is improved since deadlocks no longer apply. However, users run the risk that their changes will be thrown out if those changes conflict with another user's changes.

Even if a table uses pessimistic locking, and even if a row within that table has an exclusive lock, another user may execute read operations on the locked row under specific conditions. If the reader explicitly sets a transaction to be a read-only transaction, then versioning can be used rather than locking. This only occurs if the user explicitly declares the transaction as read-only by issuing the command:

```
SET TRANSACTION READ ONLY;
```

So, User1 might put an exclusive lock on a row and update it. When the row is updated, its version number changes. User2, who is using a read-only transaction, can read the previous version of the row even though the row has an exclusive lock on it.

Features

Pessimistic locking offers an option that optimistic locking does not. We said earlier that pessimistic locks fail "immediately"—that is, if you try to get an exclusive lock on a row and another user already has a lock (shared or exclusive) on that row, you will be told that you can't get a lock. solidDB Storage Engine provides the option of either failing immediately or waiting a specified number of seconds before failing. For example, if a wait period of 30 seconds is implemented, a user may initially be unsuccessful in achieving the lock but the database will continue trying to get the lock until either it is successful or until the 30 seconds has elapsed. In many cases, especially when transactions tend to be very short, there is benefit to setting a brief wait that allows activity to continue that otherwise would have been blocked by locking.

This solidDB Storage Engine wait mechanism applies only to pessimistic locking and not to optimistic concurrency control. There is no such thing as "waiting for an optimistic lock." If data has been changed since the time that it was read, no amount of waiting will prevent a conflict that has already occurred. In fact, since optimistic concurrency methods do not place locks, there is literally no "optimistic lock" to wait on.

Neither pessimistic nor optimistic concurrency control is right or wrong. When properly implemented, both approaches ensure that data is properly updated. In most scenarios, optimistic concurrency control is more efficient and offers higher performance, but in others pessimistic locking is more appropriate. In situations where there are a lot of updates and a relatively high chance of users trying to update data at the same time, pessimistic locking is recommended. If the chances of a conflict are very low (e.g., many rows and relatively few users or very few updates and mostly "read" operations), then optimistic concurrency control is usually the best choice. The decision will also be affected by how many rows each user updates at a time.

solidDB Storage Engine Table and Index Structures

In solidDB Storage Engine, the main data structure used to store data, including tables and indexes, is a variation of a standard B-tree. The database uses two of these structures:

- The main storage tree holds permanent data
- The Bonsai Tree stores new data temporarily, until they are ready to be moved to the main storage tree

Main Storage Tree

The main storage tree contains all of the data in the database, including tables and indexes. Internally, the database stores all data in "indexes"—there are no separate tables. Each index contains either complete primary keys (i.e., all the data in a row) or secondary keys (i.e., what the SQL language refers to as "indexes", where just the column values that are part of the SQL index). There is no separate storage method for data rows, except for Binary Large Objects (BLOB) and other long column values.

All of the indexes are stored in a single tree, which is the main storage tree. Within that tree, indexes are separated from each other by a system-defined index identification inserted in front of every key value. This mechanism divides the index tree into several logical index subtrees, where the key values of one index are clustered close to each other.

Bonsai Tree

The Bonsai Tree is a small active "index" (i.e., data storage tree) that efficiently stores new data (i.e., deletes, inserts, updates) in central memory, while maintaining multi-version information. Multiple versions of a row can co-exist in the Bonsai Tree. Both the old and new data are used for concurrency control and for ensuring consistent read levels for all transactions without any locking overhead.

With the Bonsai Tree, the effort needed for concurrency control is significantly reduced. When a transaction is started, it is given a sequential Transaction Start Number (TSN). The TSN is used as the "read level" of the transaction; all key values inserted later into the database from other sessions are not visible to searches within the current transaction. This offers consistent index read levels that appear as if the read operation was performed atomically at the time the transaction was started. This guarantees read operations are presented with a consistent view of the data without the need for locks, which have higher overhead.

Old versions of rows (and the newer versions of those same rows) are maintained in the Bonsai Tree for as long as there are transactions that need to see those old versions. After the completion of all transactions that reference the old versions, the "old" versions of the data are discarded from the Bonsai tree, and new committed data are moved from the Bonsai Tree to the main storage tree. The presorted key values are merged as a background operation concurrently with normal database operations. This offers significant I/O optimization and load balancing. During the merge, the deleted key values are physically removed.

Performance

Solid has conducted benchmarks of the solidDB for MySQL product using a benchmark called Telecom One, or TM1. The TM1 benchmark simulates a typical Home Location Register database, which is used by many mobile carriers. The Home Location Register, or HLR, is a central database that contains details of each mobile phone subscriber that is authorized to use the network. Although the benchmark is simulating an HLR, the benchmark is applicable to more than just the telecom space. The workloads that are generated by TM1 are similar to the workloads of many Enterprise applications.

The TM1 benchmark generates a flooding load on a database by performing seven pre-defined transactions that insert, update, delete and query the data in the database. The reads and writes are done using an approximately 80% to 20% read/write ratio. The TM1 results show Maximum Qualified Throughput (MQTh) of the target database system, and the response time distributions per transaction types for all seven types of transactions. The Maximum Qualified Throughput is the number of successful transactions per second. *Figure 5* shows the throughput of the solidDB for MySQL after running the TM1 benchmark, as we vary the number of call subscribers. A subscriber is effectively a row in the database.

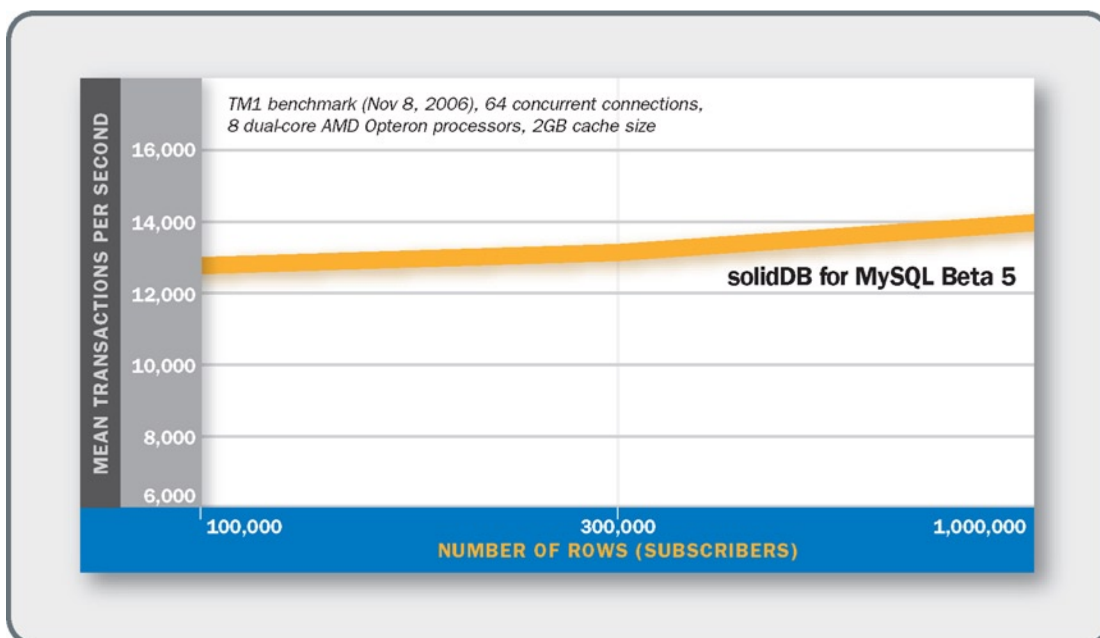

Figure 5. *solidDB for MySQL Throughput*

The benchmark test were run using an AMD Opteron machine with 8 dual-core processors, and the cache size for the database has been configured to be 2 GB. The number of concurrent client connections for this benchmark run was 64. The left side of the graph is the Maximum Qualified Throughput, which is effectively the number of transactions per second.

It is interesting to note that even though the number of subscribers was increased ten-fold, there was no corresponding degradation in the number of transactions processed. The solidDB for MySQL database was easily able to handle this workload, and performed slightly better at a million rows than it did with 100 thousand. A partial explanation for this is the 16 core server, and how solidDB Storage Engine made good use of multiple threads running on these cores. It also made good use of the cache size.

Performance

Figure 6 shows what happens if the number of CPUs are varied instead of the number of rows in the database.

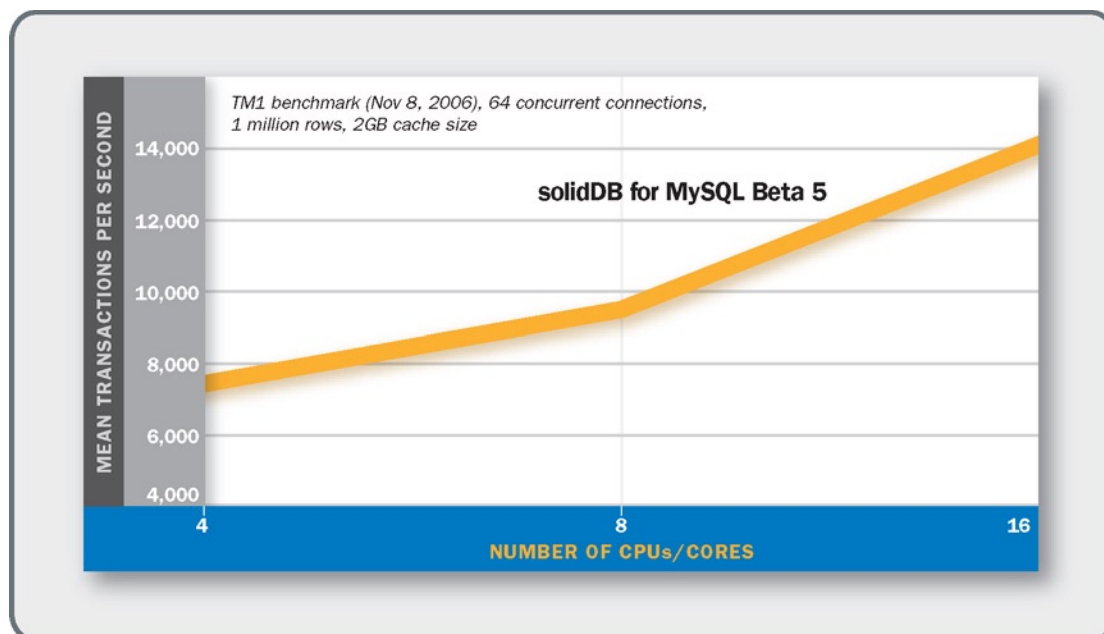

Figure 6. *solidDB for MySQL Scales with Increasing Number of Processors/Cores*

In this test, the same machine and settings are being used (i.e., an AMD Opteron with 64 concurrent client connections and 2 GB cache size). The high-end of subscribers, 1 million, was also used. The bottom of the graph represents the number of cores, from 4 to 8 to 16. The number of cores could be enabled or disabled during this benchmark so that other machine-specific factors could be held constant. The left side of the graph represents the number of transactions per second.

As the number of cores are varied, the number of transactions per second actually increased. This illustrates that solidDB Storage Engine is making very efficient use of the number of cores. Other storage engines that are less efficient typically have more of a flat line in this type of benchmark.

Additional performance information can be found in the TM1 Benchmarking Results whitepaper located at www.soliDB.com/TM1Results website.

Conclusion/About Solid Information Technology

Conclusion

solidDB for MySQL is a turbo-charged version of MySQL database that leverages solidDB Storage Engine to support heavy transaction volumes and large numbers of concurrent users. It leverages Solid's proven transactional storage engine, which is used in products from companies including HP, Siemens, Nortel, Cisco, and NEC. Leveraging this heritage, solidDB for MySQL is designed to support large number of concurrent client connections and multiprocessor/multi-core hardware configurations. In these same configurations, solidDB for MySQL also provides very fast response times. This has been demonstrated in the TM1 Public Benchmark available at www.solidDB.com/TM1

About Solid Information Technology

Solid Information Technology is the leading provider of fast, always-on and affordable database solutions for both OEM/ Embedded customers and Enterprise end-users.

Originally founded in 1992 in Helsinki, Finland, Solid had the early vision to provide innovative data management solutions that allow fast, failure-proof access to data. The resulting product line, solidDB, first shipped in 1994. Since then, Solid has continued to serve the most demanding needs of real-time applications in deployments that include next generation networks, trading rooms, multimedia network printers, medical devices, fleet management systems and Point of Sale (POS) solutions. Today, Solid has hundreds of customers worldwide in a variety of industries. There are more than 3,000,000 deployments of Solid's database technology worldwide in telecommunications networks, enterprise applications, and embedded software and systems. Market leaders such as Alcatel, Cisco, EMC2, HP, Nokia and Siemens rely on Solid for their mission-critical applications.

Firmly established as both a visionary and a technology leader for mission-critical database solutions, Solid extended its product line to serve the growing demand for open source, mission critical databases. In partnership with MySQL, Solid contributed its core transactional technology to the open source community. Solid believes this will further the adoption of open source databases among Enterprises seeking fast and scalable database solutions. As a result, Solid introduced solidDB for MySQL in 2006.

While retaining a strong commitment to the OEM and Embedded database market with its fast, always-on and affordable proprietary offering, Solid is now well equipped to serve the broad Enterprise market with its open source database offering. Solid will continue to bring additional functionalities to the open source market under the well known MySQL Server interfaces to satisfy the most demanding needs for mission-critical open source databases.

Worldwide demand for database technology continues to grow, with the fastest growth in high-availability and open source solutions. Solid is well-poised to meet the diverse needs of this evolving market by offering a proven portfolio of open source and proprietary database solutions: **solidDB**

Solid has worldwide headquarters in Cupertino, California, and regional offices in North America, Europe and Asia.

References

1. Solid Information Technology Web Site
   http://www.solidtech.com
2. solidDB for MySQL Product Web Site
   http://www.soliddb.com/solidDBforMySQL/
3. solidDB for MySQL Open Source Community Site
   http://dev.soliddb.com/en/index.php
4. MySQL 5.0 Reference Manual, Chapter 14, “Storage Engines and Table Types”
   http://dev.mysql.com/doc/refman/5.0/en/storage-engines.html
5. MySQL Pluggable Storage Engine Architecture Web Site
   http://solutions.mysql.com/engines.html

www.solidDB.com | **email: info@solidDB.com**

Corporate Headquarters Solid Information Technology 20400 Stevens Creek Blvd., Suite 200 Cupertino, CA 95014 Ph +1 408 454 4700

Asia Pacific Solid K.K. 43rd Floor The Landmark Tower Yokohama 2-2-1 Minatomirai, Nishi-ku Yokohama 220-8143 Japan Ph +81 45 224 2525

EMEA Solid EMEA Headquarters Itälahdenkatu 22 B FIN-00210 Helsinki Finland Ph +358 424 8888 1

Germany Solid Germany Stefan-George-Ring 19 D-81929 München Ph +49 89 218959 10

China Solid Information Technology China Rm 1214, Tower A, International Chamber of Commerce Building Fuhua 1 Road
FuTian, ShenZhen, PRC. 518048 Ph +86(0)755 8293 1369

© 2006 Solid Information Technology. All rights reserved.
Solid Information Technology, the Solid Logo, solidDB, EmbeddedEngine, BoostEngine, CarrierGrade Option, and SmartFlow Option are either registered trademarks or trademarks of Solid Information Technology in the United States and/or other countries. All other products, services, companies, and publications are trademarks of their respective owners. Specifications are subject to change without notice. L-WP-MySQL 12/15/06a