

# the *Availability Digest*

## Scaling MySQL with Continuent's uni/cluster

November 2008

A company's data is its life blood in these days of 24x7 operations. Its data must always be available, and it must be protected against loss.

Continuent, Inc., of San Jose, California ([www.continuent.com](http://www.continuent.com)), meets these requirements with *Continuent uni/cluster for MySQL*. Continuent uni/cluster virtualizes two or more MySQL databases to make them appear to the application as a single, highly scalable MySQL database with extremely high reliability. Continuent uni/cluster protects against data loss by synchronously replicating updates across all copies of the database. Furthermore, it improves query performance by load-sharing read queries across the copies. The virtual database can be easily scaled by simply adding additional MySQL database servers.



There is no single point of failure in a uni/cluster. Recovery from a component anywhere in the cluster is transparent to the cluster's clients.

*Continuent uni/cluster for PostgreSQL* provides the same features for PostgreSQL databases.

uni/cluster started out in the MySQL open-source community. It is available today as the Sequoia open-source implementation (<http://community.continuent.com/>). Continuent uni/cluster is the commercially supported version of Sequoia.

### uni/cluster Three-Tiered Architecture

A Continuent uni/cluster is a shared-nothing architecture that virtualizes multiple MySQL database servers to look like a single MySQL database. It comprises three components – client drivers, controllers, and database servers.<sup>1</sup> There are always two controllers in a uni/cluster for redundancy. There may be two or more database servers.

Client systems connect to the virtual MySQL database via proprietary uni/cluster drivers. Each client connects to one of the two uni/cluster controllers via its driver. These connections are completely transparent to the client systems. No modifications to the application software need be made. The application interface provided by the uni/cluster drivers is exactly the same as that used by the applications to access a standard MySQL database.

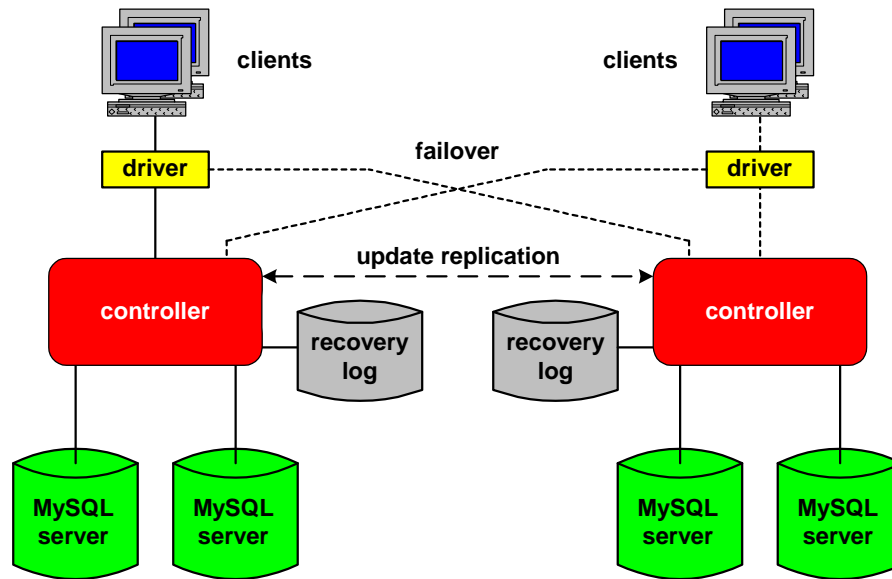
<sup>1</sup> *Continuent uni/cluster for MySQL: High Availability, Scalability and Manageability Services for MySQL – Overview White Paper*, Continuent; 2007.

*Continuent uni/cluster for MySQL – release 2007.1: Basic Concepts*, Continuent; 2008.

*Continuent uni/cluster: High Availability, Scalability and Manageability for Databases – Frequently Asked Questions*, Continuent; February 18, 2008.

Each uni/cluster driver monitors its controller connection. Should that connection fail, or should the controller fail, the driver transparently reconnects to the alternate controller. Connection state and transaction state are preserved. The client is unaware of the failover.

A uni/cluster comprises two controllers backing up each other.<sup>2</sup> A client connects to only one controller. However, should that controller fail, all of its client connections are transparently switched to the other controller to continue operation.



**Continuent uni/cluster Architecture**

A controller receives all SQL queries from its clients and ensures that they are executed across the cluster. Read-only queries are distributed among the database servers to balance the load. Update queries are replicated between the controllers and are synchronously executed simultaneously on all database servers. The controllers coordinate with each other to ensure that all transactions are executed in the same order.

A controller can connect to one or more MySQL database servers. The database servers are any mix of standard MySQL database servers. Should a database server fail, it is removed from the cluster until it is repaired. Upon recovery, its database is synchronized with the cluster's virtual database; and the server is then returned to service.

Each controller maintains a recovery log of all queries and transactions that modified the virtual database. The recovery log is used to recover a failed or downed database node prior to returning it to service. In the event of a controller failure, the Recovery Log on the surviving controller is used to recover all of the database servers on the failed controller after that controller has been returned to service.

## Cost

A major advantage of Continuent uni/clusters is cost. No proprietary database-management systems or proprietary hardware are required. The controllers run under Linux on off-the-shelf hardware.

<sup>2</sup> Later releases may support more than two controllers.

Likewise, commodity hardware is all that is needed for the database servers running MySQL, an open source database.

Communication is via standard TCP/IP over LANs so that only commodity network elements are required.

## uni/cluster Components

### Client Drivers

A client system uses a uni/cluster driver to connect to one of the two controllers. The driver forwards SQL queries to the virtualized database and returns the results of the queries to its client. The driver hides the cluster details from the client. So far as the client is concerned, it is dealing directly with a single MySQL database server.

Each uni/cluster driver monitors its controller connection via periodic pings and automatically switches over to the alternate controller should the connection or the controller fail. The driver maintains not only connection state but also transaction state so that the failover is completely transparent to its client.

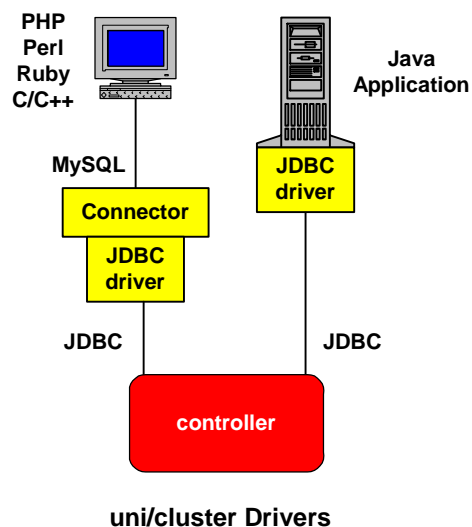
The choice of the controller to which to connect can be based on one of three algorithms:

- *Random* – The controller choice is made randomly when a client initiates a connect request.
- *Round-Robin* – Connections are made to the controllers in a round-robin fashion.
- *Fixed* – The controller to which a client connects is specified at configuration time.

Connections can be persistent or nonpersistent. However, Continuent recommends that nonpersistent connections be used wherever possible for performance reasons. For instance, a database server cannot be taken offline if a client has a persistent connection to the virtualized database.

uni/cluster provides two types of drivers – JDBC drivers and the uni/cluster Connection drivers. If an application already uses a JDBC driver to access the database, it can continue to use the same interface but with a uni/cluster JDBC driver instead. The native JDBC driver must be replaced with a uni/cluster JDBC driver to achieve the benefits of clustering. For instance, as described above, it is the driver that monitors the connection to the controller and that transparently switches over to the alternate controller should its controller or controller connection fail.

Alternatively, if some other application platform is being used to access the MySQL database, a uni/cluster Connector is used instead. The Connector emulates the standard MySQL interface, thus supporting applications such as those written in PHP, Perl, Ruby, and C or C++. The Connector provides conversion between any MySQL interface and the uni/cluster JDBC driver interface, which then connects directly to a uni/cluster controller. Using the Connector, applications use the MySQL cluster exactly as they would a single MySQL database. They require no modification.



A client's Connector can be resident either on the client or on the controller. If it is resident on the controller, it must have a copy on both controllers to accommodate the initial connection choice as well as failover.

Therefore, except for the case that a native JDBC driver is replaced with a uni/cluster JDBC driver, there is no change required to be made to any application.

### **Controllers**

The uni/cluster controller manages the execution of all queries. It receives queries from the clients that are connected to it, passes the queries to the virtualized database servers, and returns the responses to the clients. It contains all of the cluster logic required by the uni/cluster.

The uni/cluster controller is also responsible for recovering the database servers connected to it following a database server or a controller failure.

To the clients, the controller looks like a JDBC database. To the MySQL database servers that it drives, it appears as a JDBC client. Thus, it is a transparent insert between the uni/cluster clients and the virtual database made up of the multiple MySQL database servers. It hides the virtualization structure from the clients. So far as the clients are concerned, they are talking directly to a MySQL database.

There are always two controllers configured in a uni/cluster virtual database for fault tolerance. Normally, the two controllers share the load. Some clients connect to one controller, and the other clients connect to the other controller. Should a controller fail, all of the connections to its clients are transferred transparently to the surviving controller by the uni/cluster drivers.

The uni/cluster controller can reside either on a database server or on a dedicated server. In small applications, only one database server may be provided for each controller. In this case, each controller can reside on the node supporting the database.

For larger applications, multiple database servers may be provided for each controller. Each controller must then run in its own dedicated node that connects to the multiple database servers.

It is required that the controllers run under a Linux operating system. Red Hat AS/ES 4 and 5, CentOS 4.4 and later, Fedora 4, SUSE Enterprise Linux 9 and 10, OpenSUSE 10, and Debian Sarge Linux operating systems are supported.

The controller is responsible for distributing queries to the database servers connected to it. For read-only queries, a single database server is selected to execute the query. In this way, the query load is distributed across the database servers connected to the controller. Since total query traffic is distributed between the two controllers via the driver connections, the read query load is distributed across all database servers making up the virtual MySQL database.

A query that modifies the database is sent to all database servers connected to the controller. Such a query is also replicated to the other controller, which sends the query to all of its database servers. Each update query contains a unique sequence number that guarantees that the queries will be executed in the same sequence. Queries that are independent are executed in parallel. However, queries that modify the same tables are executed sequentially. A response is returned to the client when all database servers have successfully completed the update query.

The controller comprises four modules – a Request Scheduler, an optional Request Cache, a Load Balancer, and a Recovery Log.

## Request Scheduler

The Request Scheduler sequences update queries, and it may take advantage of previous results stored in the Request Cache.

An update query may update table data, or it may update the database schema. The Scheduler assigns a unique identifier to each update query. This identifier ensures that all update queries directed to all controllers are executed in the same order to maintain transaction integrity. Independent transactions that do not update the same tables are executed in parallel. Dependent transactions affecting the same table or tables are executed in sequence.

When an update query has been assigned an identifier, it is sent to the companion controller for execution. The reliable group communication protocol that is used for communicating between the controllers ensures that the controllers agree upon the execution order of two update queries received at the same time by each controller.<sup>3</sup>

All queries (read and update) are sent to the Load Balancer for execution. However, they may first be modified for performance purposes based on the contents of the Request Cache.

## Request Cache

Attributes from the results of queries may optionally be cached in the controller. They can be used to modify queries to improve performance. The Request Cache provides three types of caching:

- The *Metadata Cache* caches metadata such as column names and data types that are used to build result sets.
- The *Parsing Cache* stores parsing results such as table names extracted for locking and caching purposes. This cache is especially useful with prepared statements.
- The *Query Result Cache* remembers the result sets of read queries. If a query is executed several times, it only needs to be sent to a database server once.

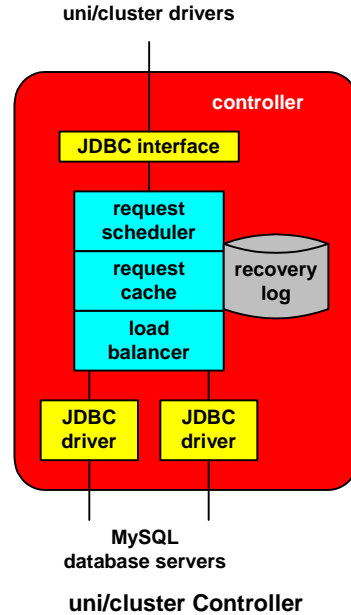
## Load Balancer

The Load Balancer is the execution engine of a uni/cluster. It is responsible for appropriate routing of queries for execution and for the returning of results back to the clients.

### *Load-Balanced Read Queries*

A read query is executed by the controller that has received the query and is sent to only one database server. If the controller has multiple database servers connected to it, it chooses the database server to which the query is to be sent based on one of three algorithms selected at configuration time:

- *Least Server Queue* – The database server with the shortest queue of outstanding query requests is chosen.



<sup>3</sup> For a discussion of distributed event ordering, see Time Synchronization in Distributed Systems – Part 3, Availability Digest, February, 2008.

- *Round-Robin* – The query is sent to the next database server in sequence.
- *Weighted Round-Robin* – Each database server is assigned a weight at configuration time. The weight determines the proportion of queries that the database server will execute. Queries are otherwise distributed in a round-robin fashion.

When the response to a query is received, it is returned to the client. Should a database server be unresponsive to a query, that database server is taken out of service; and the query is resubmitted to another database server. If all of the database servers on a controller fail (for instance, if a controller only has one database server), the controller is taken out of service, its clients reconnect to the other controller, and the query is resubmitted.

#### *Synchronous Replication of Update Queries*

An update query is sent to all of the controller's database servers for execution. It is also sent to the other controller for execution on all of its database servers. The query identifiers described above ensure that all transactions are executed in the same order on each controller. Consequently, an update query is applied synchronously to all database copies in the cluster so that all database copies are kept in synchronism.

When all database servers have executed the update query successfully, the response is returned to the client.

If a database server fails to execute an update query, it is taken out of service. The only exception to this is if all database servers fail to execute the query. In this case, the query is rejected; and all database servers remain in service.

#### Recovery Log

All update queries are written into the controller's recovery log. A pointer to the last database snapshot is also written to the recovery log. The recovery log is used to return a failed database server to service. Server recovery is described in more detail later.

#### **MySQL Database Servers**

Any standard MySQL database server may be used in a uni/cluster. No modifications are required. Each database server is connected to only one controller.

There must be enough database servers connected to each controller to ensure that application query rates can be sustained should a controller fail and remove its database servers from the virtual cluster.

#### **Network**

The cluster network includes the connections between clients and controllers, between controllers and database servers, and between the controller pair. The controller-controller link is used only to exchange update queries and for the commit/rollback commands for those queries.

All communication uses TCP/IP. Except for client connections, WANs are not supported for performance reasons. The communication latency over these channels should be less than a millisecond or two. Therefore, the uni/cluster controllers and the database servers must be collocated and must communicate over LANs.

#### **Failure Recovery**

All uni/cluster components are monitored and are automatically recovered should they fail.

### ***Client-Controller Connection***

The client driver continually monitors its connection to its controller via pings. Should the connection or the controller fail, the driver transparently reconnects to the other controller in the uni/cluster. The client is unaffected by this failover. Connection state and transaction state are preserved by the client driver, and pending requests are resubmitted to the new controller.

### ***Controller***

A controller failure is handled the same way as a client connection failure. Clients switch over to the alternate controller, and pending requests are retried by the client driver.

In this case, the database servers connected to the controller are also down and must be recovered following the controller recovery, as described next. However, in this case, the controller to which they are connected does not have an up-to-date Recovery Log. Therefore, the Recovery Log of the surviving controller is used to recover the database servers on the failed controller.

### ***Database Server***

A database server may be taken out of service intentionally, for instance, to do a snapshot or for other maintenance purposes. It may also be taken out of service by a failure.

If it is removed from service intentionally, a timestamp indicating this action is written into the controller's Recovery Log. When the server is ready to be returned to service, its database is first synchronized with the uni/cluster virtual database by replaying the update queries that have accumulated in the Recovery Log since the out-of-service timestamp.

A database server is deemed to have failed if it is unresponsive to queries or if it cannot apply an update query that was successfully applied by at least one other database server. If a database server fails, before it can be restored to service, the last snapshot must be loaded to its database. All update queries that occurred after the snapshot are replayed to bring its database into synchronism. It can then be returned to service.

### ***Network***

Client-controller network failures have been discussed above.

Should a controller-database server connection fail, the database is deemed to have failed and is revived, as previously described.

Should the controller-controller connection fail, split-brain operation must be avoided. This occurs if the two controllers continue to execute update queries without being able to inform the other. This condition is detected by the group communication protocol used between the controllers. In this case, one of the controllers is taken down. The controller to be taken down is specified at configuration time.

### ***Snapshots***

A point-in-time snapshot of the database is required to revive a failed database server or to add a new server. Upon recovery or installation, the server's database is first loaded with the snapshot; and then all subsequent update queries are replayed from a Recovery Log.

To provide a consistent snapshot of the dynamically changing virtual database without taking down the cluster, one database server in the cluster must be taken out of service. Its database

contents are copied to a snapshot file, and a pointer to that file is placed in the Recovery Logs of both controllers. The database server is then returned to service by reviving its database from the Recovery Log.<sup>4</sup>

## Adding Capacity

Adding capacity to the uni/cluster is simply a matter of providing an additional database server and connecting it to one of the controllers. Its database is then synchronized with the cluster database by loading the last snapshot and by replaying all subsequent update queries. It can then be put into service. It will participate in further read queries via the uni/cluster's load balancing mechanism.

## Management

The Continuent uni/cluster provides utilities for configuring the uni/cluster, monitoring its operation, and issuing error alerts. These utilities include:

- *clusteradmin* to debug and administer the cluster.
- *clustersql* to issue regular SQL statements against the cluster.

In addition, the uni/cluster provides the facilities for creating backup snapshots of the database, for removing database servers or a controller for maintenance, and for recovering downed or failed database servers and controllers. Clients can be added to the cluster without impact.

SNMP is not yet supported by uni/cluster, though it may be in the future.

## Master/Slave

Continuent also supports a Master/Slave configuration in which changes made to the master database are replicated asynchronously to one or more slave databases using log-based replication. The content of the slave databases are slightly behind the master database due to the replication latency of the replication mechanism. Should a master fail, one of the slaves is promoted to be the new master.

The Master/Slave configuration currently supports either MySQL or Oracle databases in either role.

Both uni/cluster and Master/Slave provide scale-out functionality that allows the cluster to scale gracefully as application transaction rates grow. The various components for uni/cluster and Master/Slave are organized into a Scale-Out Stack called Tungsten. Tungsten enhances reuse of the various Continuent components in scalable clusters.

## uni/cluster's Open-Source Roots

uni/cluster started as the open-source project C-JDBC under the Object Web umbrella. As the core developers of the project joined Continuent, the project transformed into Sequoia, which is the currently-maintained open-source version. Sequoia is the reference middleware

---

<sup>4</sup> Note that a uni/cluster with only one database server per controller will have two single points of failure during the snapshot interval – the remaining controller and the remaining database server. This situation can be avoided by having a minimum of three database servers, two on one controller and one on the other. The redundant database server on the one controller can be taken offline for the snapshot, still leaving a redundant system with two controllers, each with a database server.

implementation of the GORDA specification. GORDA is a research project supported by the European Community and aims at database-replication standardization.

Continuent uni/cluster for MySQL and Continuent uni/cluster for PostgreSQL are the commercialized versions of Sequoia. They are supported fully by Continuent.

There is no need to license MySQL or PostgreSQL for use in the uni/cluster. However, for critical applications, it is recommended that commercially supported versions of MySQL and PostgreSQL be used.

## **Continuent**

Continuent is a privately-funded company with a strong Nordic heritage. Its corporate office is in San Jose, California. It has sales offices in Espoo, Finland, and in Hong Kong.

Continuent products are used by such companies as Capgemini, Telstra, Sanoma, Thomson, La-Z-Boy, Alcatel-Lucent, CNET Networks, and NOAA.