Continuent.org Sequoia 3.0

continuent
Open. Always Available.

Continuent.org Sequoia 3.0 Basic Concepts

Document issue 1.0

Contents

1 About this document

This document provides information about the features of Continuent.org Sequoia 3.0, including a description of the Sequoia architecture and functionality. It also describes the projects at the Continuent.org portal and how to contribute to them.

Use this information to familiarize yourself with the solution prior to Sequoia deployment and installation.

1.1 Related documents

For information on how to install and configure Sequoia, refer to Continuent.org Sequoia 3.0 Installation and Configuration Guide.

Refer to the Continuent.org Sequoia 3.0 Management Guide for:

- detailed instructions on the tasks and procedures involved in managing Continuent.org Sequoia 3.0
- command line syntax and parameter descriptions.

1.2 Typographic conventions

This document uses the following formatting conventions:

Notation	Explanation
Italics	Indicates a reference to another document.
Hyperlink	Indicates a hyperlink to a web page.
Blue italics	Indicates a linked cross-reference to another section in the document.
Bold	Indicates elements in a graphical user interface.
`Courier`	Indicates code, including command line input and/or output.

2 Introduction to Continuent.org Sequoia

Sequoia is an open source database clustering middleware that allows any Java™ application (standalone application, servlet or EJB™ container, etc.) to transparently access a cluster of databases through JDBC™.

Features and benefits of Sequoia

Sequoia can be of use to you if:

1. you have a Java application or a Java-based application server that accesses one or several databases, and
2. the database tier has become the bottleneck of your application or it is a single point of failure, or both.

Note

The Continuent.org projects also provide an interface that allows you to use Sequoia with non-Java client applications. This development is hosted in the Carob project (http://carob.continuent.org).

Sequoia can help you resolve these problems by providing:

- **performance scalability** by adding database nodes and balancing the load among these nodes
- **high availability** of the database tier: Sequoia tolerates database crashes and offers transparent failover using database replication techniques
- **improved performance** with fine grain query caching and transparent connection pooling
- **SQL traffic logging** for performance monitoring and analysis
- support for clusters of heterogeneous database engines.

Sequoia also includes additional features such as monitoring, logging, and SQL request caching.

How Sequoia works

To use Sequoia, you do not have to modify your client application, application servers or database server software. You just have to ensure that all database accesses are performed through Sequoia.

Sequoia implements the concept of Redundant Array of Inexpensive Databases (RAIDb). The database is distributed and replicated among several nodes and Sequoia load balances the queries between these nodes.

Sequoia provides a generic JDBC driver to be used by the clients. This driver forwards the SQL requests to the Sequoia controller that balances them on a cluster of databases (reads are load balanced and writes are broadcasted).

The Sequoia architecture is open, allowing you to plug in your custom requests schedulers, load balancers, connection managers, caching policies, etc.

For more information about the Sequoia software components, see *3 Architecture of Sequoia* on page 7.

Sequoia can be used with any RDBMS (Relational DataBase Management System) providing a JDBC driver, that is to say almost all existing open source and commercial databases. Sequoia allows to build any cluster configuration including mixing database engines from different vendors.

For more information on the possible Sequoia deployment models, that is, the supported client applications and different cluster setups, see *Sequoia deployment models*.

Sequoia licensing

Sequoia is a free, open source project. You can redistribute and/or modify this software under the terms of the Apache v2 license (http://www.apache.org/licenses/LICENSE-2.0.html).

Sequoia is copyrighted by Continuent and the French National Institute For Research In Computer Science And Control (http://www.inria.fr/) (INRIA).

C-JDBC

Sequoia is the continuation of the C-JDBC project (http://c-jdbc.objectweb.org) hosted by the ObjectWeb Consortium (http://www.objectweb.org/), that is, it builds upon the same code base.

The C-JDBC name had to be changed due to Sun Microsystem™'s trademark of JDBC. Therefore, C-JDBC has become Sequoia.

C-JDBC is distributed under an LGPL open source license. While we like the spirit of the license and we think that an open source community can only grow if we contribute modifications back to the community, the LGPL is hard to enforce in practice and it is not always well understood by users who confuse it with the GPL.

Therefore, the contributors and INRIA ,who is the main copyright holder, agreed to re-license the code (Sequoia) under an Apache version 2 license to ease code re-use and facilitate contributions for everybody in the community.

So what will happen with C-JDBC? We are committed to support the technology and we will continue to support the community either through c-jdbc@objectweb.org or sequoia@continuent.org. The C-JDBC LGPL code will remain on ObjectWeb and the

Sequoia APLv2 code will be on Continuent.org. Bugs reported on either side will be backported in best effort mode as we always did.

2.1 Related projects

Hedera

Hedera provides a modular wrapping of controller group communications to give users more choice when choosing a group communication library. By default, Hedera is configured to use the Appia communication library, but use of other libraries such as JGroups is also supported.

Go to http://hedera.continuent.org for more details.

Appia

Appia is a layered communication framework implemented by the University of Lisbon, which provides extended configuration and programming possibilities.

Appia is composed of:

1. a core that is used to compose protocols, and
2. a set of protocols that provide group communication, ordering guarantees, and atomic broadcasting, among other properties.

Go to http://appia.continuent.org for more details.

Carob

Carob is a C++ client library and API that implements the C-JDBC/Sequoia protocol with the controller and an ODBC driver for Sequoia.

Go to http://carob.continuent.org for more details.

2.2 Contributing to Sequoia through the Continuent.org portal

The purpose of the Continuent.org portal is mainly to offer a better support infrastructure to the Sequoia open source community members. Continuent.org aims to provide a flexible framework to start or host new projects related to the Sequoia technology. We can easily host more external contributions or projects related to the technology. Our mission is to build an open source technology of industrial quality.

We warmly welcome all contributions to the Continuent.org projects!

Basically any feature that you need but you do not find implemented in Sequoia can become a contribution topic. Simply send your ideas, documents and developments to the sequoia@continuent.org mailing list.

All Continuent.org projects use GForge and JIRA. To facilitate development, a Sequoia project has been created on the Continuent Forge (https://forge.continuent.org/). This GForge integrates with JIRA (https://forge.continuent.org/jira/browse/SEQUOIA) for issue tracking.

With JIRA you can, for example:

- see the currently open tasks
- see the project roadmap
- watch the progress of issues
- vote on their resolution.

Please use Jira for feature requests and bug reports/fixes: see *2.2.2 Reporting a bug* on page 6.

If you want to receive notifications of the Jira and/or CVS changes, subscribe to the corresponding mailing list(s).

2.2.1 Sequoia mailing lists

There are currently two mailing lists available for Sequoia.

- sequoia@continuent.org is the user mailing list. It is the source to get the latest information about Sequoia, send your feedback and get support from the Sequoia community
- sequoia-commits@continuent.org is a developer mailing list that reports every commit in the Sequoia CVS repository.

Both lists are archived for public review at the Sequoia web site (http://sequoia.continuent.org/).

Feedback is crucial to improve Sequoia: do not hesitate to send us your comments or any other form of input to sequoia@continuent.org.

Note

Please do not send your questions directly to a developer, but to the Sequoia mailing list - thank you!

2.2.2 Reporting a bug

JIRA (https://forge.continuent.org/jira/browse/SEQUOIA) provides support for bug tracking. We strongly encourage you to use the automatic **Report** feature that provides all the details we usually need to figure out what happened. If you cannot use this feature, please include the following information when reporting a bug (when applicable):

- The Sequoia driver and controller version.
- The XML file you used to configure the Sequoia controller.
- JDK vendor and version (example: Sun JDK 1.4.2_08). If you use different JDK for driver and controller, please give as much detail as possible.
- OS vendor and version (examples: Linux 2.6.12 or Windows XP® SP2). If you use different operating systems for clients, controllers and backends, give the appropriate information.
- Database backend version and driver (example: PostgreSQL 8.0.3 Linux with JDBC driver postgresql-8.0-312.jdbc3.jar).
- Detailed error description with possibly the exception stack trace or a logging trace with debugging enabled.

3 Architecture of Sequoia

This section describes the Sequoia software components and their functionality.

The terms used in the following discussion to describe the Sequoia architecture are, as follows:

- **virtual database** refers to the distributed view of a database instance that is loaded inside the different database servers belonging to the cluster
- **node** refers to a physical machine, that can function either as a controller, or a database server, or both
- **database server** is a running instance of a database engine in a virtual database
- **backend** is a Sequoia object that is used to administer an underlying database server. See also *Backend*.
- **controller** is a Sequoia controller instance running in a Java Virtual Machine (JVM)
- **Sequoia connector** connects the client application to a controller. The basic Sequoia installation only includes a JDBC driver for Java clients, but the Carob project provides additional connector types for other client applications.

3.1 Sequoia JDBC driver

The Sequoia JDBC driver is a type 4 JDBC driver, which forwards all database queries to the Sequoia controller.

If you are using Sequoia with a Java client, the client applications connect to the cluster through the Sequoia driver, which replaces the database-specific driver originally used by the client application. Perl clients can also access Sequoia through the DBD::JDBC Perl module and the Sequoia driver.

3.2 Sequoia controller

The Sequoia controller is a Java program that acts as a proxy between the Sequoia connector and the backends. The controller enables the managed databases to be presented to the client application as a single virtual database. The Sequoia controller uses the native database JDBC driver to access the database(s).

3.3 Backend

A backend is a Sequoia object that is used to administer an underlying database server:

- The term *backend* refers to Sequoia's view of a database server instance.
- The backend object can be administered using the cluster management application.
- When a backend is disabled, the underlying database server instance remains operational. A backend is disabled for example for the time of performing a database backup in order to prevent the execution of queries during the backup procedure and to ensure database consistency.

Each controller hosts a dedicated set of backends: to ensure database consistency, no backends are shared between controllers.

3.4 Controller communication protocol

The controllers use a group communication protocol to exchange information and maintain consistent state information between each other. This controller replication prevents controllers from representing a possible single point of failure.

Only database updates and commit/rollback commands are sent using the group communication protocol. All other commands are executed locally by the controller.

By default, Sequoia controller communication is implemented using the Appia group communication library.

3.5 Virtual database

A virtual database virtualizes a single database but the Sequoia controller virtualizes a Relational Database Management System (RDBMS). In other words, just as a RDBMS can host multiple databases, the controller can host multiple virtual databases.

A virtual database consists of the following components:

- **authentication manager** - authenticates the virtual database username and password during connection establishment and checks that it is correctly mapped to the real database server username and password. For more information, see

Configuring Sequoia usernames and passwords in *Continuent.org Sequoia 3.0 Installation and Configuration Guide*.

- **request manager** - handles the incoming client requests forwarded by the Sequoia connector
- **backup manager** - performs database backup and restore operations and transfers backup files from one controller to another (See *3.5.5 Backup manager* on page 10)
- **backend(s)** - the backend(s) used to administer the underlying database servers(s).

A virtual database and its components are configured in a controller-specific virtual database configuration file. In other words, to configure one virtual database, you need two unique, controller-specific configuration files for that virtual database.

3.5.1 Request manager

The request manager contains the core functionality of the controller. When a client request arrives from the Sequoia connector, it is first routed to the request manager associated with the virtual database.

The request manager consists of the following components, which are described in more detail in the following sections:

- request scheduler
- load balancer
- recovery log.

You can also configure an optional request cache to improve system performance and decrease CPU and memory usage. Refer to the Sequoia DTD for more details.

3.5.2 Request scheduler

The request scheduler schedules the requests and ensures strict query consistency.

Sequoia uses a pass-through scheduling method, where queries are assigned a unique identifier and forwarded as-is to the load balancer. This identifier is used later to ensure that the writes are sent in the same order to all backends.

Each database server ultimately performs the scheduling and the locking. Thus, the locking granularity depends on the database engine.

3.5.3 Load balancer

From the request scheduler, the client request arrives at the load balancer.

Sequoia's load balancing mechanism increases the overall performance of the database cluster. It distributes requests between backends according to a predefined

load balancing method: users can choose a method most suitable for their system. For more information on the load balancing methods, see *5 Load balancing in Sequoia* on page 19.

3.5.4 Recovery log

The recovery log is a transactional log that records all requests and transactions that update the virtual database for database recovery and synchronization purposes.

Note

When configuring the virtual database, note that the recovery log is defined per virtual database and per controller.

The recovery log cannot be shared between controllers: each controller must have its own recovery log.

Make sure you specify different recovery log file names for different virtual databases on the same controller. For example, use the virtual database name as the recovery log file name. See section *Configure the virtual database(s) after Sequoia installation* in *Continuent.org Sequoia 3.0 Installation and Configuration Guide* for detailed instructions.

3.5.5 Backup manager

The backup manager, together with the recovery log, allows the dynamic addition of new backends to the virtual database without first stopping and then restarting the system. Similarly, you can use the backup manager and recovery log to easily re-enable a backend when recovering from a backend failure.

The Sequoia installation includes both a generic and several RDBMS-specific backupers. You can also use your own backuper code. Refer to the *Continuent.org Sequoia 3.0 Installation and Configuration Guide* for detailed instructions on how to configure backupers.

See also *6 Addition and synchronization of cluster nodes* on page 23.

4 Sequoia deployment models

This chapter explains the different configurations, that is, the different Sequoia setups or deployment models that you can use.

- *4.1 Replication models and data distribution* on page 11 describes the different replication models supported by Sequoia
- *4.2 Controller replication and horizontal scalability* on page 17 explains how to improve the fault tolerance and high availability of the cluster by controller redundancy
- *4.3 How the clients connect to Sequoia* on page 18 describes the differences in how the different client applications access the database through the Sequoia middleware.

4.1 Replication models and data distribution

Sequoia uses the concept of RAIDb, which stands for Redundant Array of Inexpensive Databases. This acronym has been used in reference to the RAID (Redundant Array of Inexpensive Disks) concept that achieves scalability and high availability of disk subsystems at a low cost.

RAIDb aims to:

- provide better performance and fault tolerance than a single database by combining multiple inexpensive database instances into an array of databases
- hide the distribution complexity and to provide the database clients with the view of a single database.

As in RAID, also in RAIDb a controller sits in front of the underlying resources. The clients send their requests to the RAIDb controller that balances them among a set of RDBMS backends.

There are different RAIDb levels or data distribution schemes available, which provide different cost, performance, or fault tolerance tradeoffs. (An evaluation of the different replication techniques in a 6-node C-JDBC cluster with some benchmarking information can be found here: http://c-jdbc.objectweb.org/current/doc/RR-C-JDBC.pdf.)

Refer to the following sections for more information on each RAIDb level.

4.1.1 Full partitioning (RAIDb-0)

RAIDb-0 partitions the database tables among a set of database backend nodes:

- a table itself cannot be partitioned but different tables can be distributed on different backend nodes
- RAIDb-0 requires at least two database backends
- RAIDb-0 can currently only be used with a single controller configuration
- RAIDb-0 provides moderate performance scalability but does not offer fault tolerance.

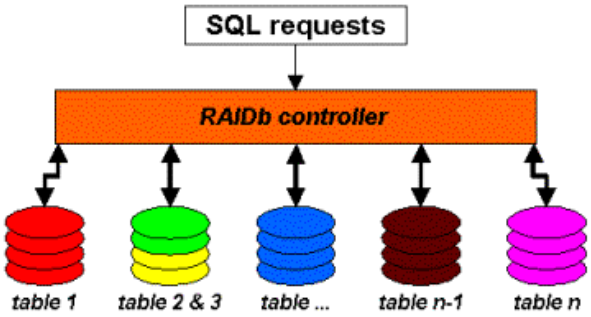

Figure 1. Full table partitioning with RAIDb-0

Note

The current implementation does not support distributed joins. This means that if you have a join between T1 and T2, you have to ensure that both T1 and T2 are on the same machine.

Scalability improvements are limited to the number of tables and the distribution of the workload among the tables:

- RAIDb-0 allows you to distribute a large database among a set of nodes in a situation where no single node has enough storage capacity to store the whole database.
- Furthermore, each database engine processes a smaller working set and can possibly improve the cache usage, since the requests are always targeted to a reduced number of tables.
- RAIDb-0 gives the best storage efficiency since no information is duplicated.

RAIDb-0 requires that the controller knows which tables are available on which node in order to direct the requests to the right node: since no tables are replicated, only one backend can execute a specific request. This knowledge can either be configured

statically in configuration files or built dynamically by fetching the database schema from each database.

4.1.2 Full replication (RAIDb-1)

RAIDb-1 offers a full mirroring or full replication of the database on a set of backends:

- RAIDb-1 requires at least 2 backend nodes, but theoretically there is no limit in the maximum number of backends
- each backend must have enough storage capacity to hold the entire database
- RAIDb-1 allows the use of several controllers in the cluster configuration to achieve high availability for mission critical systems.

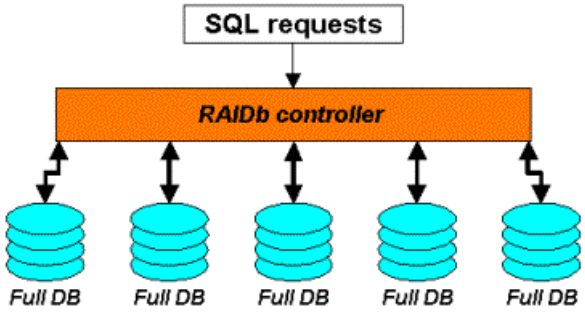

Figure 2. Full replication with RAIDb-1

The performance scalability of a RAIDb-1 setup is limited by the capacity of the controller to efficiently broadcast the updates to all backends. In case of a large number of backend databases, a hierarchical structure like those discussed in *4.1.4 Nested RAIDb Levels (vertical scalability)* on page 15 would give better scalability.

RAIDb-1 provides speedup for read queries because they can be balanced among the backends. On the other hand, there is no speedup on writes (`UPDATE`, `INSERT`, `DELETE` requests) since they have to be broadcasted to all nodes. Write queries are performed in parallel by all backends. Consequently, on a write-only workload, the performance of RAIDb-1 can be lower than that of a single node, whereas a read-only workload will scale linearly with added backeds.

RAIDb-1 includes good fault tolerance, since the system remains operational even with a single enabled backend.

Unlike in RAIDb-0, a RAIDb-1controller does not need to know the database schema, since all nodes are capable of treating any request. However, if the RAIDb-1 controller provides a cache, it will need the database schema to maintain the cache coherence.

4.1.3 Partial replication (RAIDb-2)

RAIDb-2 can be considered an intermediate solution or a tradeoff between RAIDb-0 and RAIDb-1. It provides partial replication to tune the degree of replication of each database table to obtain the best read/write throughput.

RAIDb-2:

- requires at least 3 database backends
- requires that each database table is available on at least two backends to survive the failure of a single node
- does not require any single node to host the full database.

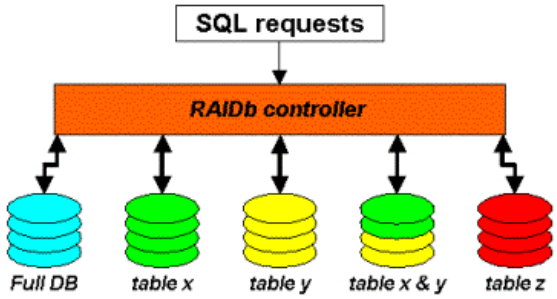

Figure 3. Partial replication with RAIDb-2

Typically, RAIDb-2 is used in a configuration where:

- none or only a few of the nodes host a full copy of the database, and
- a set of nodes host partitions of the database to offload the full databases.

In the example shown in Figure 3. *Partial replication with RAIDb-2* the database contains 3 tables: x, y and z. The first backend contains the full database, whereas the other nodes only host one or two tables. There is a total of 3 copies for table x and y, and 2 copies for table z. Whichever node fails, it is still possible to retrieve the data from the surviving nodes.

RAIDb-2 can be useful with heterogeneous databases. An existing enterprise database using a commercial RDBMS could be too expensive to fully duplicate both in terms of storage and additional licenses costs. With a RAIDb-2 configuration, you can add a number of smaller open-source RDBMS hosting smaller partitions of the database to offload the full database and to offer better fault tolerance. In the example shown in Figure 3. *Partial replication with RAIDb-2*, the first backend node on the left could be the commercial RDBMS and the 4 other nodes the smaller open-source databases.

The fault tolerance of RAIDb-2 is lower than that of RAIDb-1, but it scales better on write-heavy workloads by limiting the broadcasting of updates to a smaller set of nodes.

As RAIDb-0, also RAIDb-2 requires the controller to be aware of the underlying database schemas to be able to route the requests to the appropriate set of nodes.

Warning

There are still some limitations related to the use of RAIDb-2 in the Sequoia code base. Currently, backupers do not support partial dumps: when you back up a backend, you dump all tables on that replica. Similarly, when you restore a backend, all tables are restored. If node1 has tables x, y, and z, you have to dump them all. If you want to restore node2 from that dump, it will receive all tables (x, y, and z).

This becomes an issue during the node recovery process: the recovery log records all queries (for all tables) and tries to replay them all. If a table that is needed to replay a query does not exist on a certain node, then that node cannot be resynchronized. Thus, the recovery code needs to be fixed so that it allows partial replicas to only synchronize those tables that they are hosting.

4.1.4 Nested RAIDb Levels (vertical scalability)

It is also possible to build large scale configurations or meet specific needs by nesting several RAIDb levels.

The next example is a RAIDb-1-0 configuration where a top level RAIDb-1 controller dispatches the requests to three full databases implemented with a RAIDb-0 controller.

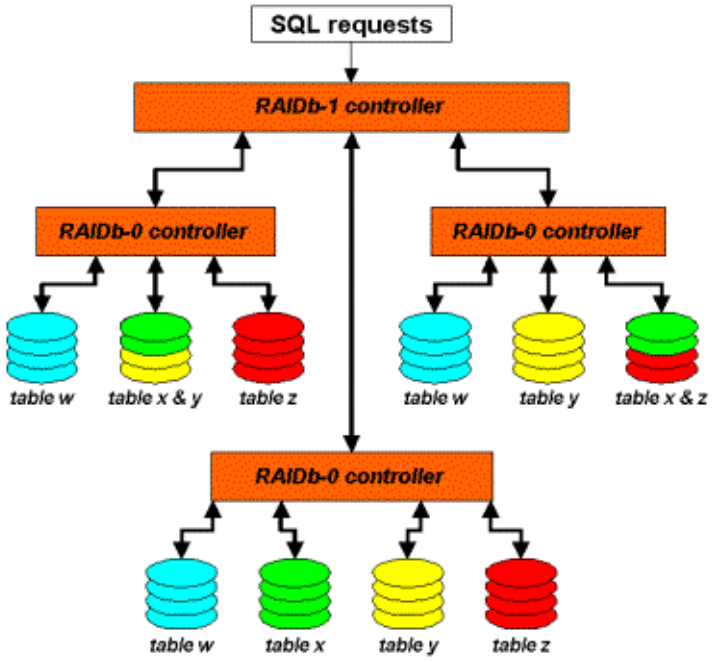

Figure 4. RAIDb-1-0 configuration

This last example shows a RAIDb-0-1 configuration. The top level is a RAIDb-0 controller and fault tolerance is achieved on each partition using a RAIDb-1 controller.

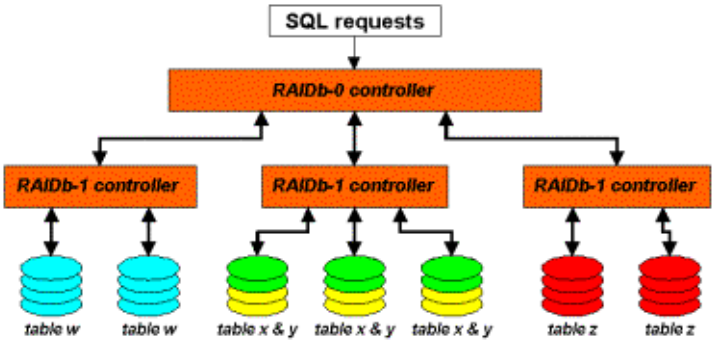

Figure 5. RAIDb-0-1 configuration

4.2 Controller replication and horizontal scalability

In addition to data replication, Sequoia supports controller redundancy. To prevent the Sequoia controller from becoming a single point of failure, you must include two or more controller nodes in your Sequoia configuration.

Sequoia includes two alternative configurations to provide horizontal scalability:

- **collocated controller configuration** - the same machine functions both as a controller and a backend/database server
- **dedicated controller configuration** - the controllers and backends/database servers are installed on dedicated machines.

Note

Each controller must be assigned a dedicated set of backends: backends must not be shared between controllers.

4.2.1 Collocated Controller Configuration for high availability systems

In a collocated controller configuration Sequoia is installed as a configuration of two nodes where both nodes function both as a controller and a backend/database server.

4.2.2 Dedicated Controller Configuration for large production environments

In larger setups the Sequoia controller is installed on a dedicated machine to improve performance.

Adding backends behind the controllers allows Sequoia to serve more requests coming from the client application. The requests will be served by the underlying database servers at their own speed, but as there are several servers behind the controllers, there are also more resources available.

The appropriate number of backends for your system depends on your application type:

- If you have a very read-intensive application, it is better to use a configuration where there are several backends behind each controller.
- On the other hand, if the application is very write-oriented, it is better to use a smaller number of backends behind the controllers.

Note that write requests do not scale in a similar fashion as read requests do:

- a write query is executed on each node at the same time synchronously
- it is returned to the client application when it has been executed on all nodes of the cluster (or on all active cluster nodes, the faulty ones being isolated).

Consequently, for write queries, adding backends behind the controller means that there are several copies of the database being maintained, but it will also add overhead to query handling.

4.3 How the clients connect to Sequoia

You can use Sequoia with the following client applications:

- Java clients
- Perl clients

The way you configure your client application to access the RDBMS database through Sequoia depends on the client application you are using.

- **Java clients** are configured to directly connect to the Sequoia driver
- **Perl clients** can access Sequoia through the DBD::JDBC Perl module and Sequoia driver.

The basic information you must provide when configuring your client application is, however, the same for all the different client applications. The information that the client application requires to connect to Sequoia includes the following:

- the Sequoia URL, which defines the IP addresses of the controller nodes and the name of the virtual database hosted by the controllers
- a username and password combination that is used to authenticate the connection to the virtual database.

Refer to section *Configuring the client application* in *Continuent.org Sequoia 3.0 Installation and Configuration Guide* for detailed instructions and examples on how to configure your client.

Note

The Carob project provides an interface that allows non-Java clients to connect to Sequoia. See http://carob.continuent.org.

5 Load balancing in Sequoia

5.1 Load balancing methods

Load balancing is implemented at two levels, that is, by:

- *Distribution of new client connections between controllers*
- *Distribution of read queries between backends*.

5.1.1 Distribution of new client connections between controllers

When the client application connects to the virtual database, the Sequoia connector connects to a controller using a Sequoia URL. The Sequoia URL contains the IP addresses of the controllers which host the virtual database in question. By default, a Sequoia controller listens to port 25322 for new connections.

If the currently selected controller fails, a new one is automatically picked up from the list defined in the Sequoia URL.

New connections to a controller are established according to a predefined load balancing policy (random, round-robin, ordered). All queries belonging to a given connection go to the same controller but a given controller load balances queries between its underlying backends.

Once the connection is established, a username and password combination (virtual login) for the virtual database is sent with the database name to be checked by the controller's authentication manager.

Related topics

See section *Configuring the client application* in *Continuent.org Sequoia 3.0 Installation and Configuration Guide* for a detailed description of the Sequoia URL, including examples on how to define it.

See section *Balancing client connections to controllers* in *Continuent.org Sequoia 3.0 Installation and Configuration Guide* for detailed instructions on how to configure the load balancing policy that is used to select between the controllers in the Sequoia URL.

5.1.2 Distribution of read queries between backends

The available load balancing methods for balancing read queries between backends are:

- **least pending requests first** - The request is sent to the backend that has the shortest queue of pending requests, that is, the smallest number of pending

requests to be executed. The pending request queue is an accurate approximation of the backend load and thus provides efficient dynamic load balancing.

- **round robin** - Simple round robin load balancing: the first request is sent to the first backend, the second request to the second backend, and so on. This is a continuous loop: after all backends have received a request, the same process starts again from the first backend.
- **weighted round robin** - Same as the round robin method, but enhanced by assigning a weight to each backend. This value determines what proportion of the load a particular backend will receive relative to other backends. For example, a backend that has the weight 2 gets twice as many requests as a backend with the weight 1.

See section *Balancing read queries between backends* in *Continuent.org Sequoia 3.0 Installation and Configuration Guide* for guidelines on configuring a load balancing method suitable for your system.

5.2 Handling of client connection context in Sequoia

The client connections can be divided into the following two types:

- **non-persistent connections** - a non-persistent connection is a connection to a backend that is allocated on-demand for the duration of a query (in `AUTOCOMMIT` mode) or for the duration of a transaction, after which the connection is put back in the backend connection pool.
- **persistent connections** - a client connection that is assigned a dedicated connection to each backend for the duration of the client connection. When a persistent connection is used, information about the connection context and state are preserved by Sequoia.

Both non-persistent and persistent connections are allocated from the Sequoia connection pool: if no connection is available, the controller will wait for the pool to provide an available connection.

Note

By default, the use of persistent connections is disabled in Sequoia. See also *Sequoia URL options* in *Continuent.org Sequoia 3.0 Installation and Configuration Guide*.

5.2.1 Examples of typical usage of persistent connections

Here are some examples of the typical usage of persistent connections:

- setting an environment variable or a connection attribute that is only visible in the scope of a single connection (usually through `SET xxx` commands)
- creating and manipulating a temporary table
- retrieving information about the previously executed request on the connection (in `AUTOCOMMIT` mode), for example `SELECT LAST_INSERT_ID` in MySQL.

5.2.2 Issues related to the use of persistent connections

Use of persistent connections is not recommended with Sequoia for the following reasons.

Degradation of system performance

Opening and closing persistent connections are performed cluster-wide and can thus degrade system performance.

Failures in disabling a backend or shutting down a virtual database

A backend can only be fully disabled when all of its persistent connections have been closed.

Every connection open/close operation is logged in the recovery log. When a checkpoint needs to be added to the recovery log (for example to disable a backend for the duration of a database backup), Sequoia must first wait for all currently opened persistent connections to be closed because the connection context cannot be logged.

Moreover, a virtual database can only be shut down after all of its backends have been disabled. Consequently, if the applications keeps persistent connections opened in the pool indefinitely, the virtual database will fail to shut down cleanly.

How to prevent failures related to persistent connections

The above-mentioned failures can occur if Sequoia is used with:

- standalone applications which do not use an explicit `close ()` to close the connections
- application servers such as JBoss that perform connection pooling on behalf of the application.

Make sure your applications explicitly close their connections. In particular, check the pooling and/or connection re-use schemes.

If you are using Sequoia with an application server such as JBoss, you must configure the connection pool so that the unused connections are renewed and closed regularly (the `idle-timeout` parameter in JBoss).

Note

Use of persistent connections and connection pooling is NOT recommended with Sequoia. Thus, if you do not use persistent connections, you should disable their use. See *Sequoia URL options* in *Continuent.org Sequoia 3.0 Installation and Configuration Guide*.

6 Addition and synchronization of cluster nodes

When a new backend is added to the cluster, its underlying database must be brought to a state that is consistent with the databases residing on the other active cluster nodes.

In Sequoia, when a node is brought into the cluster, entries logged into a recovery log are used to bring the node back into exactly the same state as the other nodes in the cluster. This process does not interfere with the other operations of the cluster and makes it possible to recover from failures without downtime.

Both Sequoia controllers keep a recovery log per virtual database, which logs all requests and transactions that update the virtual database. The recovery log contains:

- snapshots of the database
- a log of queries and transactions that have been applied to the database after the snapshot was made.

6.1 Automatic recovery logging for disabled nodes

When you disable a backend, for example, to perform database maintenance, Sequoia will:

1. automatically insert a checkpoint to the recovery log to store the state of the underlying database server
2. continue recording the incoming client requests.

When you again enable the backend, the recovery log is used to synchronize the underlying database by replaying all client update requests starting from the stored checkpoint.

6.2 Restoring the database from a database dump

If you want to add a new backend to an already active cluster, or you want restore a failed backend, you must take into account that there is no clean checkpoint available in the recovery log that can be used to synchronize the underlying database server. Consequently, the database cannot be synchronized simply by replaying the entries from the recovery log.

In such a case, you must first restore the database from a backup. Restoring the database from a valid backup also attaches a clean checkpoint to the backend that you want to enable.

Note

Backups play an important role in recovering from a hardware or disk failure of a cluster node. Thus, you should back up the database on a regular basis to always have a valid backup available.

6.3 Making database backups

With the backup manager you can backup a database to create a database dump; a record of the table structure and the data of a specific database server. You can use this dump to restore the database of a new backend that you want to add to the cluster.

A backup operation is perfomed, as follows:

1. If the backend was in enabled state, it will be disabled for the time of the backup. The underlying database server remains up and running.
2. While the database backup is in progress, the incoming client requests are recorded in the recovery log.
3. If the backend whose database is being backed up was initially in enabled state, it will automatically synchronize the database content after the backup operation using the recovery log entries and then enable itself.

Warning

If you perform a backup when there is only one enabled backend available, note that the backup operation will also bring the remaining backend to disabled state. Thus, the whole cluster will stop serving requests for the time of the backup. Back up the database on a regular basis to avoid a situation where you need to stop the whole cluster.

7 Automatic failure handling

Sequoia offers fully transparent failover handling. The handling of the different failure scenarios is exlained in more detail in the following sections.

7.1 Handling of controller connection failures

If a connection to a controller instance fails, the Sequoia connector transparently reconnects the client connections to another controller according to a predefined policy. See *5.1.1 Distribution of new client connections between controllers* on page 19.

If a failure occurs inside a transaction, the transactional context is automatically restored upon reconnection.

7.2 Handling of controller failures

If one of the controllers fails, the backends hosted by the failed controller are disabled. Existing client connections are transparently reconnected to another controller according to a predefined policy.

Sequoia also automatically generates a trace file named `sequoia.report` to the `log` directory when a controller cannot recover from an error.

Possible pending queries are handled, as follows:

- read queries are transparently retried (even within a transaction)
- if there were write queries pending when the controller failed, a notification message is sent to the client application telling that the write might or might not have been executed. The application should be configured to rollback the transaction in such a case.

7.3 Handling of backend failures

If a backend fails, it is automatically removed from the load balancer. The handling of the failed operation depends on the query type:

- If a backend fails during a read operation, the operation is retried on another backend. If the retried operation succeeds, the first backend is disabled. If the query fails on all backends, the query is regarded faulty.
- If a backend fails during a write operation, the failure is ignored if the write operation was successful on another backend.

Failures can be handled without any noticeable failover time.

8 Logging in Sequoia

Sequoia provides logging services based on the Log4j logging framework.

The logging system allows you to select, for example:

- which software components are being logged
- which log files are to be created
- different logging levels that determine how detailed and fine-grained the logging information will be
- different log appenders that determine how the log messages are output. By default, Sequoia log entries are both stored into a log file and displayed on the standard output. You can also select other appenders, such as sending the logs to a remote log server or an e-mail address.

When you install Sequoia, a logger configuration file, called `log4j.properties` is automatically created in the `config` directory of the installation directory. You can control the logging configuration at runtime by modifying the logger configuration file. See section *Change the logging configuration* in *Continuent.org Sequoia 3.0 Management Guide* for details.

8.1 Log files

The log file(s) containing the log messages are stored to the `log` directory under the Sequoia installation directory.

By default, Sequoia creates the following two log files to the `log` directory:

- `cluster.log`
- `full_cluster.log`.

By default, only the messages pertinent to the operation of the cluster are both shown in the standard output and and written to the `cluster.log` file. The cluster's internal log messages are not shown on the standard output, they are only written to the `full_cluster.log` file.

You can also create the following two log files by editing the `log4j.properties` configuration file:

- `request.log` - used to log requests received by the controller
- `distributed_request.log` - used to log the execution of distributed requests at each controller.

Sequoia also generates a trace file named `sequoia.report` to the `log` directory when:

- the controller receives a shut down command, or
- the controller cannot recover from an error.

The settings of the `sequoia.report` file are included in the controller configuration file.

Using cluster.log and full_cluster.log

By default, events with the severity level `INFO` are shown in the standard output and recorded to `cluster.log` (see also *Logging levels*.)

The log messages can be categorized into the following types:

- log messages related to performing cluster administration through the cluster management application (CLC)
  - a start message indicating that the cluster is now performing the requested cluster administration task
  - a completion message indicating the successful completion of a cluster administration task
  - an error message reporting the failure of a cluster administration task
- log messages indicating a cluster state change
- log messages related to serious runtime errors that cause an unexpected and unrecoverable change in cluster status
- log messages related to user authentication errors.

If you notice a failure in the operation of the cluster, first check the contents of the `cluster.log` file. If the reason of the failure could not be deduced based on cluster.log, open the `full_cluster.log`, which includes all log messages related to the cluster operation.

8.2 Logging levels

You can specify the level of detail included in the logs by selecting a logging level for each logged component. If the logging level is set to `OFF`, logging is disabled for the given Sequoia component.

There are five logging levels available:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`.

The logging level determines the severity level of the logged events. For example, if the logging level is set to `ERROR`, only events with the severity level `ERROR` or `FATAL` are logged.

See section *Logging management* in *Continuent.org Sequoia 3.0 Management Guide* for detailed instructions.