C-JDBC: a Middleware Framework for Database Clustering

Emmanuel Cecchet
INRIA Rhône-Alpes - Sardes team
655, Avenue de l'Europe - 38330 Montbonnot - France
emmanuel.cecchet@inria.fr

Abstract

Clusters of workstations become more and more popular to power data server applications such as large scale Web sites or e-Commerce applications. Successful open-source tools exist for clustering the front tiers of such sites (web servers and application servers). No comparable success has been achieved for scaling the backend databases. An expensive SMP machine is required if the database tier becomes the bottleneck. The few tools that exist for clustering databases are often database-specific and/or proprietary.
Clustered JDBC (C-JDBC) addresses this problem. It is an open-source, flexible and efficient middleware for database clustering. C-JDBC implements the Redundant Array of Inexpensive Databases (RAIDb) concept. It presents a single virtual database to the application through the JDBC interface and does not require any modification to existing applications. Furthermore, C-JDBC works with any database engine that provides a JDBC driver, without modification to the database engine. The C-JDBC framework is open, configurable and extensible to support large and complex database cluster architectures offering various performance, fault tolerance and availability tradeoffs.

1 Introduction

Nowadays, database scalability and high availability can be achieved, but at very high expense. Existing solutions require large SMP machines or clusters with a Storage Area Network (SAN) and high-end RDBMS (Relational DataBase Management Systems). Both hardware and software licensing cost makes those solutions only available to large businesses. Commercial implementations such as Oracle Real Application Clusters [12] that address cluster architectures requires a shared storage system such as a SAN (Storage Area Network). Open-source solutions for database clustering are database-specific. MySQL cluster [13] uses a synchronous replication mechanism with limited support for scalability (up to 4 nodes). Some experiments have been reported using partial replication in Postgres-R [8]. These extensions to existing database engines often require applications to use additional APIs to benefit from the clustering features. Moreover, these different implementations do not interoperate well with each other.
Database replication has been used as a solution to improve availability and performance of distributed or clustered databases [1, 9]. Even if many protocols have been designed to provide data consistency and fault tolerance [4], few of them have found their way into practice [14]. Gray et al. [10] have pointed out the danger

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

of replication and the scalability limit of this approach. However, database replication is a viable approach if an appropriate replication algorithm is used [2, 8, 15]. Most of these recent works only focus on full database replication.

We have introduced Redundant Array of Inexpensive Databases or RAIDb [5] as a classification of different database replication techniques. RAIDb is an analogy to the existing RAID (Redundant Array of Inexpensive Disks) concept, that achieves scalability and high availability of disk subsystems at a low cost. RAID combines multiple inexpensive disk drives into an array of disk drives to obtain performance, capacity and reliability that exceeds that of a single large drive [7]. RAIDb is the counterpart of RAID for databases. RAIDb aims at providing better performance and fault tolerance than a single database, at a low cost, by combining multiple database instances into an array of databases. RAIDb primarily targets low-cost commodity hardware and software such as clusters of workstations and open source databases.

We have implemented Clustered JDBC (C-JDBC), a software implementation of RAIDb. C-JDBC is an open-source Java middleware framework for database clustering on a shared-nothing architecture built with commodity hardware. C-JDBC implements the RAIDb concepts hiding the complexity of the cluster and offering a single database view to the application. The client application does not need to be modified and transparently accesses a database cluster as if it were a centralized database. C-JDBC works with any RDBMS that provides a JDBC driver. The RDBMS does not need any modification either, nor does it need to provide distributed database functionalities. Load distribution, fault tolerance and failure recovery are all handled at the middleware level by C-JDBC. The architecture is flexible, distributable and extensible to support large clusters of heterogeneous databases with various degrees of performance, fault tolerance and availability.

With C-JDBC, we hope to make database clustering available in a low-cost and powerful manner, thereby spurring its use in research and industry. Although C-JDBC has only been available for a year, several installations are already using it to support various database clustering applications [6]. The outline of the rest of this paper is as follows. Section 2 introduces the RAIDb concepts. Section 3 presents the architecture of C-JDBC and the role of its internal components. Section 4 describes how fault tolerance is handled in C-JDBC and section 5 discusses horizontal and vertical scalability. We conclude in section 6.

2 Redundant Array of Inexpensive Databases

One of the goals of RAIDb is to hide the distribution complexity and provide the database clients with the view of a single database like in a centralized architecture. As for RAID, a controller sits in front of the underlying resources. Clients send their requests directly to the RAIDb controller that distributes them among the set of RDBMS backends. The RAIDb controller gives the illusion of a single RDBMS to the clients.

RAIDb does not impose any modification of the client application or the RDBMS. However, some precautions have to be taken care of, such as the fact that all requests to the databases must be sent through the RAIDb controller. It is not allowed to directly issue requests to a database backend as this might compromise the data synchronization between the backends.

RAIDb defines 3 basic levels providing various degree of replication that offer different performance/fault tolerance tradeoffs.

2.1 RAIDb-0: full partitioning

RAIDb level 0 consists in partitioning the database tables among the nodes. It does not duplicate information and therefore does not provide any fault tolerance guarantees. RAIDb-0 allows large databases to be distributed, which could be a solution if no node has enough storage capacity to store the whole database. Also, each database engine processes a smaller working set and can possibly have better cache usage, since the requests are always hitting a reduced number of tables.

2.2 RAIDb-1: full replication

With RAIDb level 1, databases are fully replicated. RAIDb-1 requires each backend node to have enough storage capacity to hold all database data. RAIDb-1 needs at least 2 database backends, but there is (theoretically) no limit to the number of RDBMS backends. The performance scalability is bounded by the capacity of the RAIDb controller to efficiently broadcast the updates to all backends.
RAIDb-1 provides speedup for read queries because they can be balanced over the backends. Write queries are performed in parallel by all nodes, therefore they usually execute at the same speed as the one of a single node. However, RAIDb-1 provides good fault tolerance, since it can continue to operate with a single backend node.

2.3 RAIDb-2: partial replication

RAIDb level 2 features partial replication which is an intermediate solution between RAIDb-0 and RAIDb-1. Unlike RAIDb-1, RAIDb-2 does not require any single node to host a full copy of the database. This is essential when the full database is too large to be hosted on a node's disks. Each database table must be replicated at least once to survive a single node failure. RAIDb-2 uses at least 3 database backends (2 nodes would be a RAIDb-1 solution).

3 C-JDBC: a RAIDb software implementation

JDBC, often referenced as Java Database Connectivity, is a Java API for accessing virtually any kind of tabular data [19]. C-JDBC (Clustered JDBC) is a Java middleware based on JDBC, that allows building all RAIDb configurations described in the previous section.

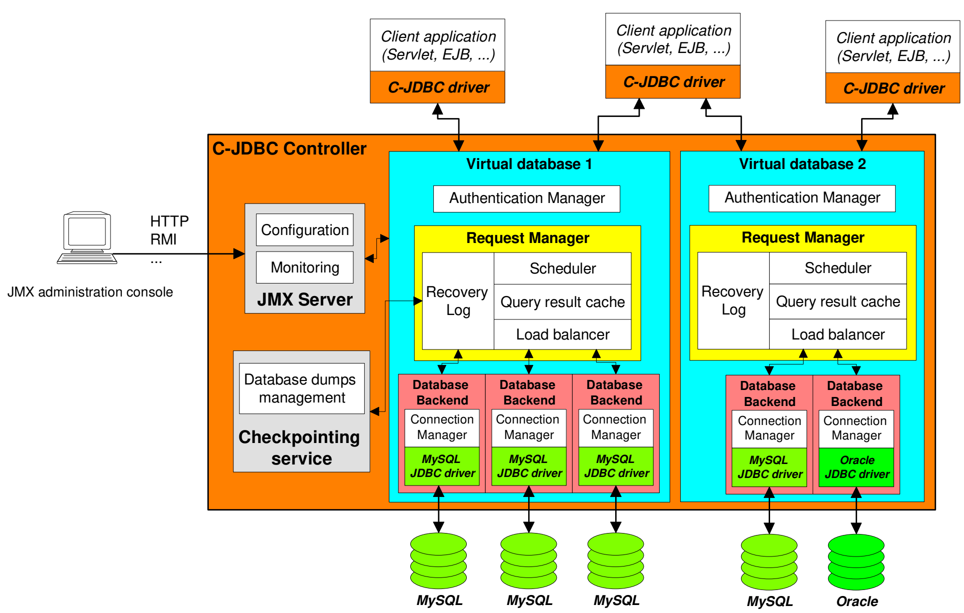

Figure 1: C-JDBC architecture overview.

3.1 Architecture overview

C-JDBC is made of 2 components: a generic JDBC driver to be used by the client application and a controller that handles load balancing and fault tolerance. The database-specific JDBC driver used by the client application is replaced by a generic C-JDBC driver that offers the same interface. The C-JDBC controller is a Java program that acts as a proxy between the C-JDBC driver and the database backends. The distribution is handled by the C-JDBC controller that implements the logic of a RAIDb controller. The degree of replication and the location of the replicas can be specified on a per-table basis. Currently, the tables named in a particular query must all be present on at least one backend. In dynamic content servers, one of the target environments for C-JDBC clusters, this requirement can often be met, since the queries are known ahead of time. Eventually, we plan to add support for distributed execution of a single query.

The C-JDBC controller exposes a single database view, called a virtual database, to the C-JDBC driver and thus to the application. A virtual database has a virtual name that matches the database name used in the client application. Virtual login names and passwords match also the ones used by the client. The client therefore perceives there to be a single (virtual) database. Database back-ends can be dynamically added to or removed from a virtual database, transparently to the user applications.

An authentication manager establishes the mapping between the login/password provided by the client application (called virtual login) and the login/password to be used on each database backend (called real login). This allows database backends to have different names and user access rights mapped to the same virtual database setting. All security checks can be performed by the authentication manager. It provides a uniform and centralized resource access control.

A controller can possibly host multiple virtual databases but each virtual database has its own request manager that defines its request scheduling, caching and load balancing policies. Routing of queries to the various backends is done automatically by C-JDBC, using a read-one write-all approach. A number of strategies are available for load balancing and connection pooling, with the possibility of overriding these strategies with a user-defined one. C-JDBC can be configured to support result caching and fault tolerance.

Finally, larger and more highly-available systems can be built by using the horizontal and vertical scalability features of C-JDBC (see section 5). C-JDBC also provides additional services such as monitoring and logging. The controller can be dynamically configured and monitored through JMX (Java Management eXtensions) either programmatically or using an administration console.

3.2 C-JDBC driver

The C-JDBC driver implements the JDBC 2.0 specification and some extensions of the JDBC 3.0 specification. Queries are sent to the controller that issues them on a database backend. The result set is serialized by the controller into a C-JDBC driver `ResultSet` that can be sent through the communication channel. The driver ResultSet contains the logic to browse the results locally to the client machine without requiring further interaction with the controller.

When the ResultSet is too large to be sent at once due to memory constraints, it is fetched by blocks of fixed number of rows. These blocks are transparently sent to the driver when they are required.

The C-JDBC driver can also transparently fail over between multiple C-JDBC controllers, implementing horizontal scalability (see Section 5). The JDBC URL used by the driver is made up of a comma-separated list of 'node/port' items, for instance, `jdbc:cjdbc://node1,node2/db` followed by the database name. When the driver receives this URL, it randomly picks up a node from the list. This allows all client applications to use the same URL and dynamically distribute their requests on the available controllers.

3.3 Request manager

Incoming requests are routed to the request manager associated with the virtual database. The request manager contains the core logic of the virtual database. It is composed of a scheduler, optional caches, a load balancer and a recovery log. Each of these components can be superseded by a user-specified implementation.

Scheduler The *scheduler* is responsible for handling the concurrency control and for ordering the requests according to the desired isolation level. C-JDBC provides both optimistic and pessimistic transaction management. Transaction demarcation operations (begin, commit and rollback) are sent to all backends. Reads are sent to a single backend. Updates are sent to all backends where the affected tables reside. Depending on the RAIDb level, this may be one (RAIDb-0), several (RAIDb-2) or all (RAIDb-1) backends. SQL queries containing macros such as RAND() or NOW() are rewritten on-the-fly with a value computed by the scheduler so that each backend stores exactly the same data.
All operations are synchronous with respect to the client. The request manager waits until it has received responses from all backends involved in the operation before it returns a response to the client. To improve performance, C-JDBC also implements parallel transactions and early response to update, commit, or abort requests. With parallel transactions, operations from different transactions can execute at the same time on different backends. Early response to update, commit or abort allows the controller to return the result to the client application as soon as one, a majority or all backends have executed the operation. Returning the result when the first backend completes the command offers the latency of the fastest backend to the application. When early response to update is enabled, C-JDBC makes sure that the order of operations in a single transaction is respected at all backends. Specifically, if a read follows an update in the same transaction, that read is guaranteed to execute after the update has executed.
If a backend executing an update, a commit or a rollback fails, it is disabled. In particular, C-JDBC does not use a 2-phase commit protocol. Instead, it provides tools to automatically recover failed backends into a virtual database (see Section 4).
At any given time only a single update, commit or abort is in progress on a particular virtual database. Multiple reads from different transactions can be going on at the same time. Updates, commits and aborts are sent to all backends in the same order.

Caches C-JDBC provides 3 optional caches. The *parsing cache* stores the results of query parsing so that a query that is executed several times is parsed only once. The *metadata cache* records all ResultSet metadata such as column names and types associated with a query result.
These caches work with query skeletons found in `PreparedStatements` used by application server. A query skeleton is a query where all variable fields are replaced with question marks and filled at runtime with a specific API. An example of a query skeleton is `SELECT * FROM t WHERE x=?`. In this example, a parsing or metadata cache hit will occur for any value of `x`.
The *query result cache* is used to store the `ResultSet` associated with each query. The query result cache reduces the request response time as well as the load on the database backends. By default, the cache provides strong consistency. In other words, C-JDBC invalidates cache entries that may contain stale data as a result of an update query. Cache consistency may be relaxed using user-defined rules. The results of queries that can accept stale data can be kept in the cache for a time specified by a staleness limit, even though subsequent update queries may have rendered the cached entry inconsistent.
We have implemented different cache invalidation granularities ranging from database-wide invalidation to table-based or column-based invalidation. An extra optimization concerns queries that select a unique row based on a primary key. These queries are often issued by application servers using JDO (Java Data Objects) or EJB (Enterprise Java Beans) technologies. These entries are never invalidated on inserts since a newly inserted row will always have a different primary key value and therefore will not affect this kind of cache entries. Moreover, update or delete operations on these entries can be easily performed in the cache.

Load balancer If no cache has been loaded or a cache miss has occurred, the request arrives at the *load balancer*. C-JDBC offers various load balancers for the different RAIDb levels.
RAIDb-1 is easy to handle. It does not require request parsing since every database backend can handle any

query. Database updates, however, need to be sent to all nodes, and performance suffers from the need to broadcast updates when the number of backends increases.
RAIDb-0 or RAIDb-2 load balancers need to know the database schema of each backend to route requests appropriately. The schema information is dynamically gathered. When the backend is enabled, the appropriate methods are called on the JDBC `DatabaseMetaData` information of the backend native driver. Database schemas can also be statically specified by the way of the configuration file. This schema is updated dynamically on each *create* or *drop* SQL statement to reflect each backend schema.
Among the backends that can treat the request (all of them in RAIDb-1), one is selected according to the implemented algorithm. Currently implemented algorithms are round robin, weighted round robin and least pending requests first (the request is sent to the node that has the least pending queries).
In the case of an heterogeneous cluster (database engines from different vendors), it is possible to define rewriting rules based on patterns to accommodate the various SQL dialects. These rules are defined on a per-backend basis and allow queries to be rewritten on-the-fly to execute properly at each backend.

4 Fault tolerance

To achieve fault tolerance, database content must be replicated on several nodes. C-JDBC uses an ETL (Extraction Transforming Loading) tool called Octopus [11] to copy data to/from databases.

Building the initial state The initial database is dumped (data and metadata) on the filesystem in a portable format. We call this the initial checkpoint. Octopus takes care of re-creating tables and indexes using the database specific types and syntax.
Once the system is online, a *recovery log* records all SQL statements that update the database. When a new backend is added to the cluster, the initial checkpoint is restored on the node using Octopus and the recovery log replays all updates that occurred since the initial checkpoint. Once this new backend is synchronized with the others, it is enabled to server client requests.

Checkpointing At any point in time, it is possible to perform a checkpoint of the virtual database, that is to say a dump of the database content. Checkpointing can be manually triggered by the administrator or automated based on temporal rules.
Taking a snapshot from a backend while the system is online requires to disable this backend so that no update occur on it during the backup. The other backends remain enabled to answer the client requests. As the whole database needs to be consistent, trying to backup a backend while leaving it enabled would require to lock all tables in read and thus blocking all writes. This is not possible when dealing with large databases where copying the database content may take hours.
Disabling a specific backend for checkpointing uses a specific data path that does not load the system. At the beginning of the backup, a specific index is inserted in the recovery log (see below). Once the database content has been successfully dumped, the updates that occurred during the backup are replayed from the recovery log on the backend that was used for checkpointing. Once the backend is synchronized with the others, it is enabled again.

Recovery log C-JDBC implements a recovery log that records a log entry for each begin, commit, abort and update statement. A log entry consists of the user identification, the transaction identifier, and the SQL statement. The log can be stored in a flat file, but also in a database using JDBC. A fault-tolerant log can then be created by sending the log updates to a virtual C-JDBC database with fault tolerance enabled.

5 Horizontal and vertical scalability

As mentioned in section 3.2, *horizontal scalability* provides C-JDBC controller replication to prevent the controller from being a single point of failure. To support a large number of database backends, we also provide *vertical scalability* to build a hierarchy of backends. To deal with very large configurations where both high availability and high performance are needed, one can combine horizontal and vertical scalability [6].

5.1 C-JDBC horizontal scalability

Horizontal scalability is what is needed to prevent the C-JDBC controller from being a single point of failure. We use the JGroups [3] group communication library to synchronize the request managers of the virtual databases that are distributed over several controllers.
When a virtual database is loaded into a controller, a group name is assigned to the virtual database. This group name is used to communicate with other controllers hosting the same virtual database. At initialization time, the controllers exchange their respective backend configurations. If a controller fails, a remote controller can recover the backends of the failed controller using the information gathered at initialization time.
C-JDBC relies on JGroups' reliable and totally ordered message delivery to synchronize write requests and demarcate transactions. Only the request managers contain the distribution logic and use group communication. All other C-JDBC components (scheduler, cache, and load balancer) remain the same.

5.2 C-JDBC vertical scalability

As a C-JDBC controller gives the view of a single database to the client application, it is possible to implement the backends of a C-JDBC controller with other C-JDBC controllers. In general, an arbitrary tree structure can be created. The C-JDBC controllers at the different levels are interconnected by C-JDBC drivers. The native database drivers connect the leaves of the controller hierarchy to the real database backends.
Vertical scalability may be necessary to scale an installation to a large number of backends. Limitations in current Java Virtual Machines restrict the number of outgoing connections from a C-JDBC driver to a few hundreds. Beyond that, performance drops off considerably. Vertical scalability spreads the number of connections over a number of JVMs, retaining good performance.

6 Conclusion

Clustered JDBC (C-JDBC) is a flexible and extensible middleware framework for database clustering, addressing high availability, heterogeneity and performance scalability. C-JDBC primarily targets J2EE application servers requiring database clusters build with commodity hardware. By using the standard JDBC interface, C-JDBC remains database vendor independent and it is transparent to JDBC applications. Combining both horizontal and vertical scalability provide support for large-scale replicated databases. Several experiments have shown that C-JDBC caches improve performance further even in the case of a single database backend.
There is a growing academic and industrial community that shares its experience and provides support on the `c-jdbc@objectweb.org` mailing list. C-JDBC is an open-source project licensed under LGPL hosted by the ObjectWeb consortium. C-JDBC is available for download from `http://c-jdbc.objectweb.org`.

Acknowledgements

We are grateful for the C-JDBC users community for their feedback and support. We would like to thank all C-JDBC contributors who help us improve the software by testing it in various environments and contribute to the code base with fixes and new features.

References

[1] Christiana Amza, Alan L. Cox, Willy Zwaenepoel - Scaling and availability for dynamic content web sites - *Rice University Technical Report TR02-395*, 2002.

[2] Christiana Amza, Alan L. Cox, Willy Zwaenepoel - Conflict-Aware Scheduling for Dynamic Content Applications - *Proceedings of USITS 2003*, March 2003

[3] Bela Ban - Design and Implementation of a Reliable Group Communication Toolkit for Java - Cornell University, September 1998.

[4] P. A. Bernstein, V. Hadzilacos and N. Goodman - Concurrency Control and Recovery in Database Systems - *Addison-Wesley*, 1987.

[5] Emmanuel Cecchet, Julie Marguerite and Willy Zwaenepoel - RAIDb: Redundant Array of Inexpensive Databases - *INRIA Research Report 4921* - September 2003.

[6] Emmanuel Cecchet, Julie Marguerite and Willy Zwaenepoel - C-JDBC: Flexible Database Clustering Middleware - *USENIX Annual Technical Conference 2004, Freenix track*, June 2004.

[7] P. Chen, E. Lee, G. Gibson, R. Katz and D. Patterson - RAID: High-Performance, Reliable Secondary Storage - *ACM Computing Survey*, volume 26, n2, pp. 145-185, 1994.

[8] Bettina Kemme and Gustavo Alonso - Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication - *Proceedings of the 26th International Conference on Very Large Databases*, September 2000.

[9] Bettina Kemme and Gustavo Alonso - A new approach to developing and implementing eager database replication protocols - *ACM Transactions on Database Systems* 25(3), pp. 333-379, 2000.

[10] Jim Gray, Pat Helland, Patrick O'Neil and Dennis Shasha - The Dangers of Replication and a Solution - *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.

[11] Enhydra Octopus - http://octopus.enhydra.org/.

[12] Oracle - Oracle9i Real Application Clusters - *Oracle white paper*, February 2002.

[13] Mikael Ronstrom and Lars Thalmann - MySQL Cluster Architecture Overview - *MySQL Technical White Paper*, Avril 2004.

[14] D. Stacey - Replication: DB2, Oracle or Sybase - In *Database Programming & Design*, , 7(12), 1994.

[15] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme and G. Alonso - Database replication techniques: a three parameter classification - *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, October 2000.