C-JDBC Horizontal Scalability
A controller replication user guide

Emmanuel Cecchet, Emic Networks

Version 1.2
March 24, 2004

1. Introduction

This document introduces C-JDBC horizontal scalability and requires the reader to be familiar with the C-JDBC concepts. An introduction on C-JDBC can be found in the white paper that is available from the C-JDBC web site at:
http://c-jdbc.objectweb.org/current/doc/C-JDBC_Flexible_Database_Clustering_Middleware.pdf.
To prevent the C-JDBC controller from being a single point of failure, C-JDBC provides controller replication also called horizontal scalability. A virtual database can be replicated in several controllers that can be added dynamically at runtime. Controllers use the group communications to synchronize updates in a distributed way.
Horizontal scalability is used for both scalability and availability. If a controller has 10 backends with a 200 connection pool for each of them and 1000+ client connections to handle, it means more than 3000 concurrent connections inside the same Java Virtual Machine (JVM) and this does not scale well. In this case, we have to distribute client connections and backends among several controllers.
Figure 1 gives an overview of C-JDBC controller replication. The JDBC URL to use for the C-JDBC driver must contain a comma separated list of controller IP addresses such as: `jdbc:cjdbc://host1:port1,host2:port2/database` where `host` is the machine name (or IP address) where the C-JDBC controller is running, and `port` is the port number the controller is listening for client connections. When a new connection has to be established, a controller is randomly picked from the list. If the currently selected controller fails, another one is automatically picked up from the list.
Controllers use group communication to exchange information and maintain state consistency between each other. Only database updates and commit/rollback commands are sent through the group communication, all other commands are just executed locally to the controller.

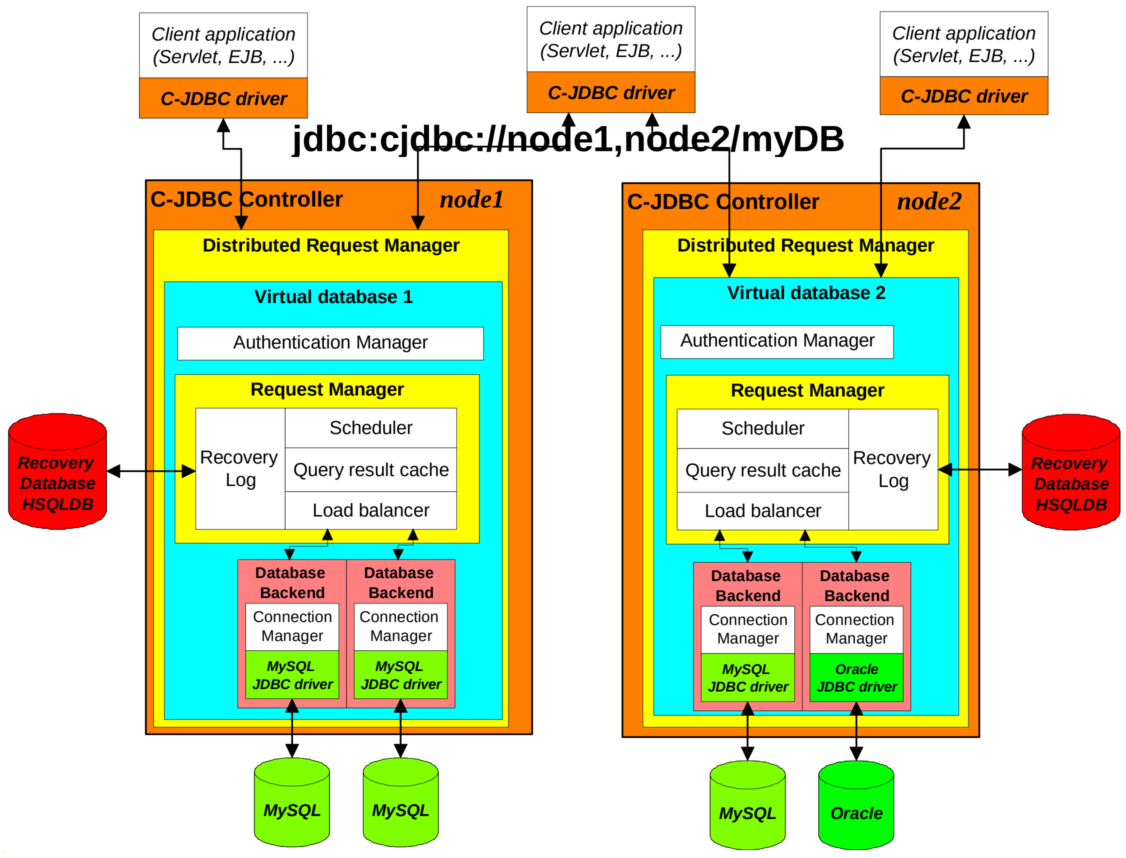

Figure 1. Horizontal scalability overview

2. Basics of controller replication

C-JDBC does not allow databases (same JDBC URL) to be shared between controllers. This means that each controller has its own set of backends and no backend is shared between controllers. As backends are not shared between controllers, there is no chance that backends become inconsistent due to a remote access by another entity.
Figure 2 gives an example of a configuration with 3 controllers. Note that this configuration needs at least 3 backends (3 controllers with 1 backend each) and cannot work with a less backends than controllers.

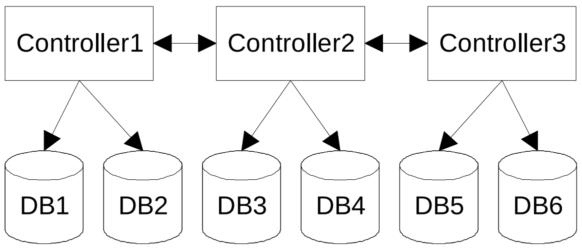

Figure 2. Configuration without backend sharing

2.1.1. Handling write queries

When a write query is received by a controller, it is sent through the group communication that provides a reliable total order delivery. This means that all controllers will receive the queries exactly in the same order and they will all schedule and execute them in the same order.

Basically, the query execution occurs in parallel at all controllers but exactly in the same order. There is no need for extra synchronization messages between controllers, we just ensure that they schedule the queries in the same order thanks to the group communication. Furthermore, it is not necessary to synchronize on remote query completion since each scheduler has a full control on its set of backends. The only exception is in the case of a complete failure of all backends of one controller, we need an extra message to notify controllers if all others failed as well or if there was only one controller to fail (in which case we disable all backends of that failed controller).
Figure 3 gives an example of the messages exchanged between 3 controllers for a write query execution. We just have to send the query to all controllers and wait for all replies. that is (1 multicast + *n* unicast [+ 1 multicast if failure]) messages.

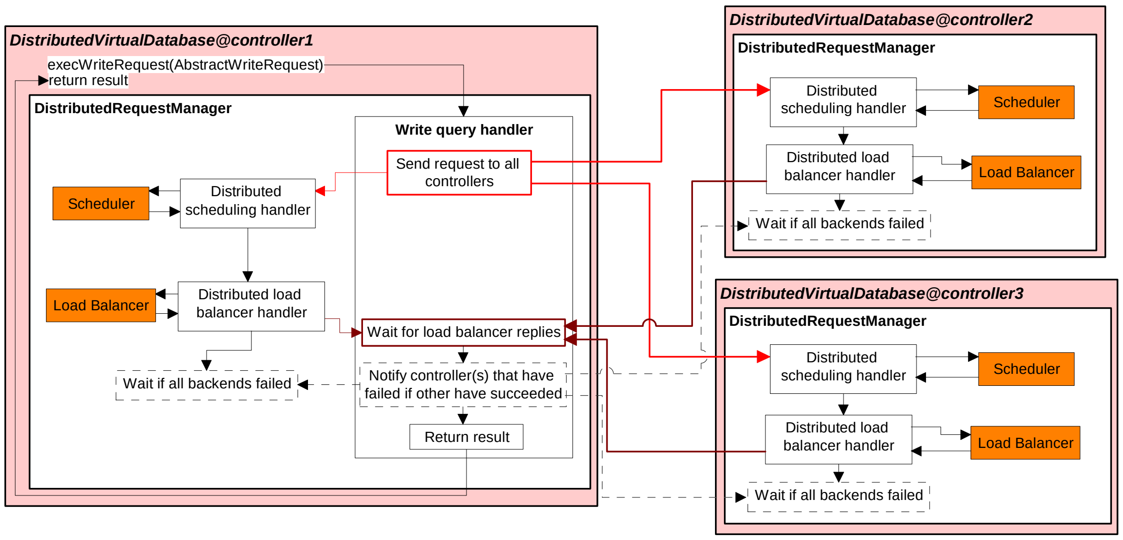
DistributedVirtualDatabase@controller1
execWriteRequest(AbstractWriteRequest)
return result
DistributedRequestManager
Write query handler
Send request to all controllers
Scheduler
Distributed scheduling handler
Load Balancer
Distributed load balancer handler
Wait for load balancer replies
Wait if all backends failed
Notify controller(s) that have failed if other have succeeded
Return result
DistributedVirtualDatabase@controller2
DistributedRequestManager
Distributed scheduling handler
Scheduler
Distributed load balancer handler
Load Balancer
Wait if all backends failed
DistributedVirtualDatabase@controller3
DistributedRequestManager
Distributed scheduling handler
Scheduler
Distributed load balancer handler
Load Balancer
Wait if all backends failed

Figure 3. Write query execution overview

No C-JDBC internal components are changed, except the RequestManager that is replaced by a distributed version that takes care of the interface with the group communication. It also means that all optimizations such as 'lazy transaction begin' are still fully functional.
When full replication is not used, it should be possible to optimize some queries so that they are not broadcasted to controllers who don't have any backends with the necessary tables to execute the query. For example, in RAIDb-0, only one controller should be able to execute a given query. This is not implemented yet.

2.1.2. Handling read queries

In the case of RAIDb-0 or RAIDb-2, requests might have to be forwarded to a remote controller if no local backend is able to execute the query. It is possible to check beforehand if the query can execute locally or not (whether by re-performing the job done by the load balancer or having a list of queries having previously successfully executed locally). Else, we can let the query execute and if the load balancer tells that no local backend can execute the query, then we forward it to a remote controller (but we need to parse the query to know which one is the right one to pick).

2.1.3. Handling failures

If one controller fails, other controllers should take over the backends of the failed controller. This can only be achieved if there was no pending updates (write statement or commit operation) on the controller that failed. Client connections must also be re-routed to the other controllers.
Upon controller failure, all open connections will be closed and all open transactions will be automatically rollbacked. If no update was pending, it should be sufficient to replay the open transactions from the recovery log. This feature has not been made available because JGroups does not allow to determine if

there was pending updates at a failing node. Once JGroups has been replaced with a group communication that provides uniform delivery, we will be able to provide this transparent failover.
In the worst case, the backends attached to the controller are disabled and must be fully re-synchronized using another controller.

3. Why backends cannot be shared between controllers?

Figure 4 shows an example of 3 controllers sharing 6 database backends. This configuration is not allowed by C-JDBC and we explain the various issues related to this configuration.

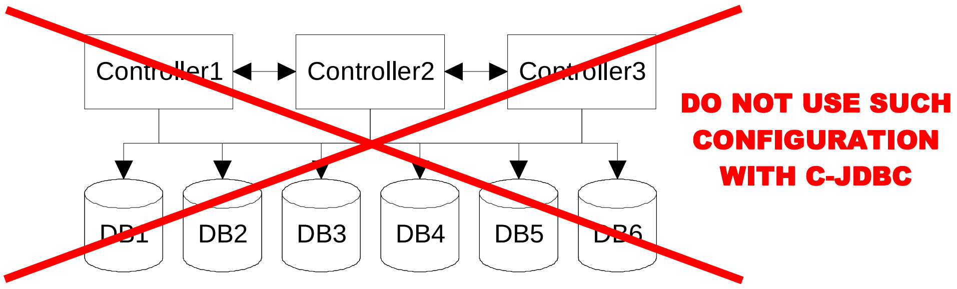

Figure 4. Configuration with full sharing

Every write query must be broadcasted to all controllers so that they can at least invalidate their local cache, so there is no reduction in the usage of the group communication. We explain next the approaches for implementing such strategy and what are the associated issues.

3.1.1. Centralized write approach

The centralized write approach means that a write query is executed by a single controller on all backends. This solution has a major issue illustrated by figure 5.

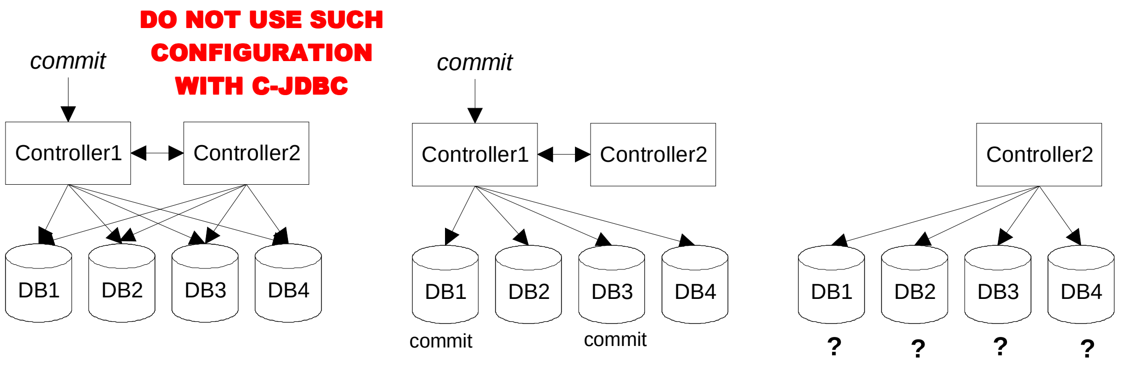

Figure 5. Example of a state loss with shared backends during a commit

Controller 1 & 2 are sharing all backends. Then controller1 receives a commit order (left part of the figure). The commit order is sent through the group communication so that controller2's scheduler can be notified. Controller1 starts to execute the commit operation in parallel on all backends. During the operation controller1 fails but DB1 and DB3 had the time to commit the transaction before the crash (middle part of the figure). However DB2 and DB4 didn't complete the transaction (note that C-JDBC does not use 2 phase commit). When controller2 wants to access the backends, it has no idea of which backends committed the transaction or not. All backends are in an unknown state for controller2. There is no way to recover from such failure.

3.1.2. Distributed write approach

To prevent the problem mentioned in 3.1.1, we have to reduce the set of backends a single controller can write to so that its failure affects a smaller number of backends and let the system being available. In the

distributed write approach, a write request is broadcasted to all controllers and each controller is responsible to write to a subset of backends (all subsets are distinct like in a non-shared configuration).
Backends must be assigned to controllers either in a static or dynamic way. Note that read queries become more complex to schedule in this distributed scenario.
This approach prevents controllers from accessing in read the backends they do not write to, which fallbacks to the C-JDBC configuration where no backends are shared. The problem of sharing backends for read is depicted hereafter.

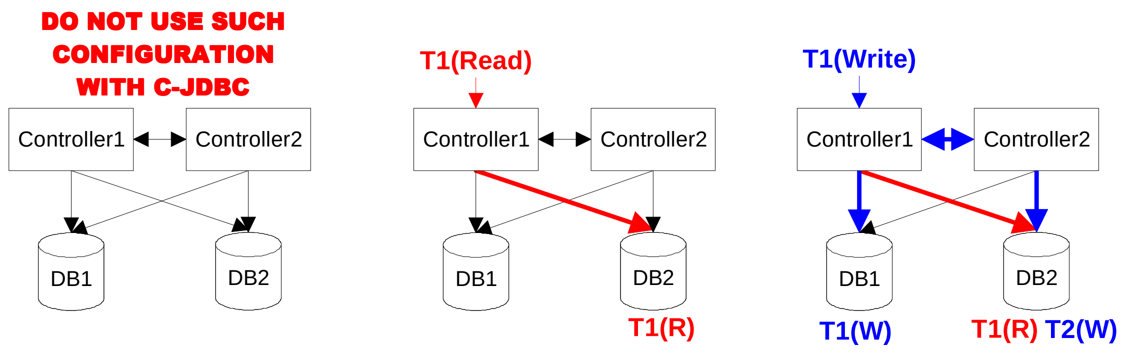

Figure 6. Issues of transactions assigned to connections

Lets consider the example depicted in figure 6 where 2 controllers share 2 backends (left part of the figure), and controller1 is responsible to write to DB1 and controller2 to DB2. Transaction T1 begins with a read that is sent to controller1. The load balancer on controller1 decides to execute this query on DB2 and assigns a specific connection for this transaction (middle part of the figure). Then a write for the same transaction arrives at controller1. Controller1 is only allowed to write to DB1 and controller2 receives the query by a group message to execute it on DB2. The problem is that the connection for transaction1 on DB2 was open by controller1. If controller2 want to execute the query on DB2 it will have to open a new connection which will be considered by the database as a separate (and concurrent) transaction. There is no way for controller2 to access directly the connections opened by controller1.

4. Recovery Log

As for backends, controllers cannot share a common recovery log. Instead, each controller has to run its own recovery log to reflect the exact state of its set of backends. As depicted on Figure 1, each controller can use a lightweight database such as HSQLDB for the recovery log that should run on the same physical node as the controller to prevent any additional network traffic.
When a controller fails, all backends attached to it are lost as depicted on Figure 7.

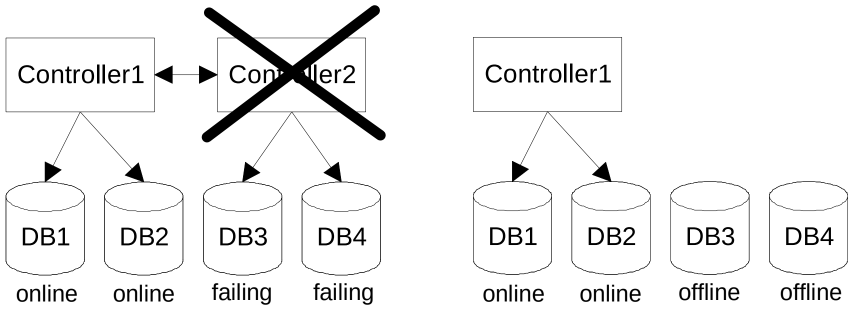

Figure 7. Controller failure lets attached nodes offline

It is then necessary to use an alive controllers (controller1 in our example) to restore a dump and replay the recovery log to resynchronize the failed backends. This is shown by Figure 8. If all controllers have failed, the last failing controller must be restarted and used to recover all backends of the cluster.

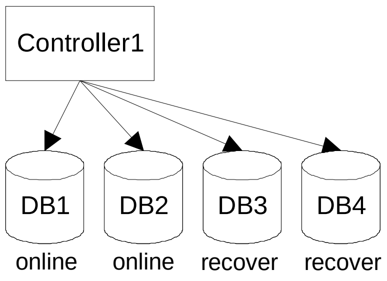

Figure 8. Backends must be recovered from an alive controller

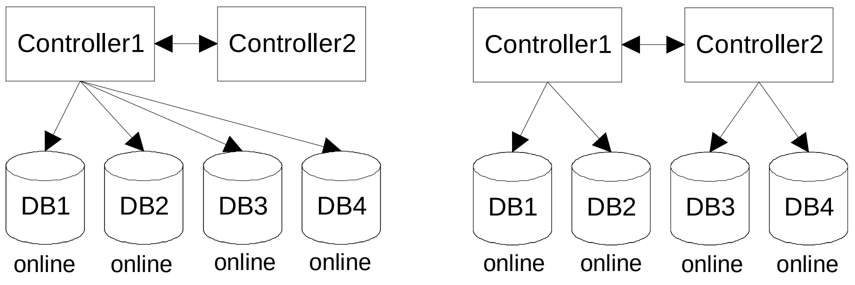

Figure 9. Controller restart and backends transfer

At any time the failed controller can be restarted. Its recovery log is now useless and can be completely cleaned. Once the controller is online, you can use the *transfer* backend command of the C-JDBC console to transfer backends from one controller to another. Note that backends must be fully synchronized before being transferred.
There is currently no command to resynchronize recovery log, therefore a new backup should be made on the restarted controller to have a coherent checkpoint in the recovery log. In Figure 9 example, once DB3 and DB4 have been transferred from controller1 to controller2, the administrator should do a backup of DB3 or DB4 to have a coherent checkpoint in controller2.

5. Cluster partitions

If you use multiple controllers interconnected by long distance links. These links are subject to failures which can result in cluster partitions. Figure 10 shows an example where controller1 and its 2 backends are isolated in partition1 after the failure of the link with controller2 and controller3 that are in partition2. Note that we can have as many partitions as controllers in the system, each controller living in its own partition.

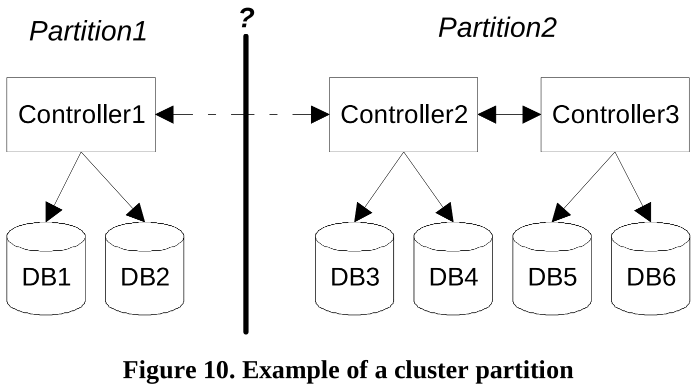

Figure 10. Example of a cluster partition

5.1. *Partition issues*

If the partitions contain disjoint data (nothing shared) then there is no issue except that clients connected to controller1 might receive exceptions that it is not possible to execute the request since no backend has the available data. Vice-versa, clients of controller 2 and 3, will receive exceptions if they try to query data contained in DB1 or DB2.
With RAIDb-1 (full replication) or RAIDb-2 (partial replication) configurations, we have the problem that data might become inconsistent between the 2 partitions. If both partitions evolve on their side, how do we reconcile both partitions when the link comes back up? The answer is simple, we don't.

A possible solution is to force a controller to be read-only if it becomes alone (it has lost connection with its peers). This has to be done by a user program listening to the JMX notification that reports the possible cluster partition (see 5.2). The program has then to take the appropriate actions such as turning backends to read-only mode (through JMX again).

5.2. *Failure versus Partition*

The group communication layer detects failures and manages group membership. However, how controller 2 & 3 will make the difference between a simple link failure (resulting in a partition) and a real failure of controller1? In both cases, we will detect that controller1 is out of the group, but we have no way to distinguish a partition from a failure, therefore the user should assume the worst, that is a partition.

6. Checklist for setting up controller replication

Controllers use the JGroups group communication middleware to synchronize updates in a distributed way. The JGroups stack configuration is found in `config/jgroups.xml` and should not be altered unless you specifically know what you are doing. Keep in mind that total order reliable multicast is needed to ensure proper synchonization of the controllers. More information about JGroups can be found on the JGroups web site. Note that JGroups requires proper network settings, here are a few guidelines:

- a default route must be defined (check with `/sbin/route` under Linux) for the network adapter which is bound by JGroups (usually eth0). If such route does not exist, either the group communication initialization will block or controllers will not be able to see each other even on the local host. If you don't have any default entry in your routing table you can use a command like '`/sbin/route add default eth0`' to define this default route.
- issues have been reported with DHCP that can either block (under Windows) or just fail to properly set a default route and leads to the issue reported above. We strongly discourage the use of DHCP, you should use fixed IP addresses instead.
- name resolution should be properly set so that the IP address/machine name matching works both ways. Often improper `/etc/hosts` or DNS configuration leads to group communication initialization problems.

In order for a virtual database to be replicated, you must define a `Distribution` element in the virtual database configuration file (see Section 9.2.1, "Distribution" in the C-JDBC user documentation). There are several constraints for different controllers to replicate a virtual database:

- give the list of all controllers that you plan to use for replication of your virtual database in the C-JDBC driver URL. Even if all controllers are not online at all times, the driver will automatically detect the alive controllers: `jdbc:cjdbc://node1,node2,node3,node4/myDB`
- the virtual database must have the same name and use the same groupName (in the Distribution element).
- each controller must have its own set of backends and no backends should be shared between controllers (C-JDBC checks the database URLs, having different backend names is not sufficient).
- each controller must have its own recovery log since recovery logs cannot be shared. It is possible for a controller not to have a recovery log but this controller will have no recovery capabilities.
- the authentication managers must support the same logins.
- schedulers and load balancers must implement the same RAIDb configuration.
- database schemas (if defined) must be compatible according to the RAIDb level you are using.

Note

As backends cannot be shared between controllers, it is not possible to use a SingleDB load balancer with controller replication. If each controller only has a single database backend attached to it, then you must use a RAIDb-1 configuration since in fact you have 2 replicated backends in the cluster.

7. How to contribute?

If you have comments, ideas, use cases, suggestions, links, articles, code, patches, or any other contribution that can be sent by email, please share it with us on the mailing list c-jdbc@objectweb.org. If you are interested in contributing to the implementation or testing of C-JDBC, contact us on the mailing list as well.