Clustering implementation for H2 database

Arun Narasani

(GitHub: https://github.com/sfbayhacker/h2database/tree/cs244b-master)

Abstract

This project is an implementation of clustering for H2[1], an open source Java SQL database. The default storage subsystem in H2 is a multi-versioned persistent and log structured key-value store. The project achieves distributed transactions for simple table operations using 2 phase commit protocol (2PC). Replication is achieved by intercepting table level operations and sending messages synchronously to replicas using gRPC as part of the 2PC. And, concurrency is managed using timestamp ordering of operations. The timestamp used for ordering is a cluster level global timestamp assigned by a designated node from the cluster acting as a timestamp server.

1 Introduction

1.1 H2

H2[1] is a light-weight SQL database with support for transactions. H2 supports both embedded mode (in-memory) and server mode (persisted in a file). In server mode, data is persisted in a single file for each database. H2 provides support to encrypt the files.

H2 is written in Java and is extremely popular among Java based frameworks and projects. For example, H2 is widely used as a test dependency in spring-boot projects for testing any SQL database related functionalities. H2 works very well with popular Object Relational Mapping (ORM) frameworks such as Hibernate.

Maven public repository suggests that H2 is used in 5,789 artifacts. It is fair to say that H2 is the de-facto in-memory SQL database used for testing SQL database related functionalities.

H2 comes with a simple clustering feature (hereinafter, referred to as “native” cluster mode or implementation) [3]. The native cluster mode is limited to two nodes and is more akin to active-active high availability mode. In the native cluster mode, replication is achieved at the SQL level. SQL commands are executed independently at both nodes to achieve replication. This approach to clustering comes with certain limitations. One such limitation is related to inconsistent behavior of certain SQL commands/functions supported in H2 such as UUID(), SECURE_RAND(), SESSION_ID(). Because replication is based on SQL commands, same command may result in different values on the two nodes. This limitation means that users will have to forego certain features to run H2 in native cluster mode.

I believe the authors took the simple approach for two reasons. One, the primary use-case for H2 is as an in-memory database for unit tests. And, two, the project is an open source project with only four primary developers. Therefore, the need to support full clustering is probably not the highest of priorities for the authors.

However, there are many use cases that can benefit from using a clustered H2 database. One such example use case is testing behavior of

multiple instances of a micro-service running on docker containers without having to install a physical database server. In a typical micro-services application stack, the micro-services are stateless, and they use a common database server hosted locally or in the cloud. And, making a database server available is not always practical, especially for running tests in the CI/CD pipeline or even in local developer environments.

1.2 Motivation and target use-case

The idea behind this project is to first and foremost learn some concepts related to distributed systems. And, in the process, hopefully, fill the gap in the native cluster implementation of H2 to support additional use cases as discussed.

Given the tight bounds on the project timelines, the project goals were limited to implementation of clustering for databases with simple key-value pair based table structures. Nothing in the implementation assumes key-value pair data structures. However, there is an implicit assumption that there is a key identifying each row. This can be either the key as defined by a primary key definition, or a column guaranteed to have unique value. The current implementation assumes that index level key exists for each row. The assumption is fair and works well with H2, as H2 assigns a primary key (based on a serial number) even when there is no primary key specific for a table.

The implementation addresses the three aspects of replication, atomic commitment, and concurrency control. I chose to use 2PC as a consensus mechanism for replication and as a distributed commit protocol. This should serve the purpose for most testing use-cases while providing reasonable performance. I chose to use timestamp ordering for concurrency control. In section 2, I discuss the overall architecture of how distributed database management is achieved. In section 3, I discuss the implementation details of replication and atomic commitment using 2PC. In section 4, I discuss the implementation details of concurrency control using timestamp ordering.

The primary target use-case for this implementation of clustering feature is test automation for SQL database related functionalities. While performance is not a major focus, reasonable performance is expected when using H2 with clustering enabled. In section 4, I discuss the impact of distributing transactions on transaction throughput by providing performance data, benchmarked against the original implementation of H2.

2 Architecture and design

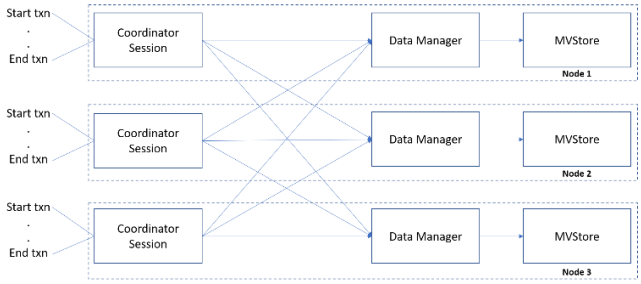

FIG. 1: Distributed Database Architecture

FIG. 1 depicts the distributed database architecture. The distributed database is a collection of homogeneous nodes. Each node is a H2 database server with MVStore[2] (the multi-version store in H2) as the default store for managing data. Each node can act either as a coordinator or a follower for any given transaction.

Users always operate in the context of a session. And there can be only one transaction within a session at any single point in time. And, users may start multiple sessions in parallel at any single node.

One of the primary goals of this project is to support transactions as part of the clustering implementation. While MVStore provides support for transactions in a single node, clustering implementation must take care of atomic commitment across the nodes to support transactions. To achieve atomic commitment across the nodes, two-phase commit or 2PC is used as the consensus mechanism for the nodes to agree on operations being performed on data[5].

In the context of a given transaction, the two-phase commit coordinator interacts with data managers at all nodes including the one at the coordinator to perform operations. The operations are performed at all nodes within the context of the session to which the transaction belongs, where the session is identified by the session id. Followers create a session and cache the local session object for future re-use with the coordinator session id as the key.

Since sessions can be started at the coordinator (which can potentially be different for different transactions) and each coordinator assigns session ids independently, it is possible that session ids can collide. This collision in session ids can create inconsistencies in cache of session objects at the followers. To overcome this problem, the followers identify sessions based on a combination of session id (as defined by H2) and a unique host id assigned to each node in the cluster. Since host id is always unique, the combination of session id and host id is guaranteed to be unique.

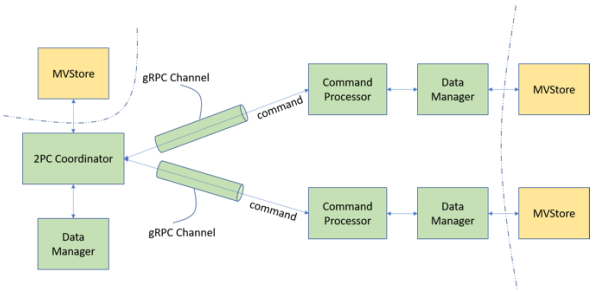

FIG. 2: Component interactions to distribute table level operations

FIG. 2 is a block diagram illustrating the flow of control between the various components.

FIG. 2 illustrates the various components involved in this project. The components highlighted in green are new components added to the existing code base of H2. The distributed database illustrated in FIG. 1 is achieved by way of intercepting row level operations on MVTable (MVTable.java in the source code) and MVPrimaryIndex (MVPrimaryIndex.java in the source code) classes (not shown) that operate on MVStore (MVStore.java in the source code).

2PC Coordinator (TwoPCCoordinator.java in the source code) implements the coordinator logic and is responsible for making calls to followers and receiving messages from followers. 2PC Coordinator makes use for gRPC channel to create gRPC clients to send protocol buffer messages to followers.

Command Processor (CommandProcessor.java in the source code) implements the follower logic to receive messages via gRPC channel and perform necessary operations via the Data Manager. Command Processor with the help of a 2PC Follower (TwoPCFollower.java in the source code) also takes care of sending messages to the coordinator of a transaction. This is needed when a follower recovers after a crash and finds prepare log records for transaction that are yet to be committed.

Data Manager (DataManager.java in source code) implements the data structures for maintaining prewrites, and with the help of a log manager (LogMananger.java in source code) performs necessary operations to persist commit log information and prewrites state to

disk. In addition, Data Manager is responsible for performing necessary concurrency checks for prewrites based on timestamp assigned to transactions.

Data Manager is a new layer added on top of MVStore to facilitate data management in the two-phase commit process and concurrency control using timestamp ordering.

3 Replication and atomic commitment using 2PC

3.1 Replication

Replication is achieved by synchronously distributing operations within a transaction. If the cluster mode is enabled, write operations from the user are intercepted at the node where user is providing requests and corresponding calls are routed to 2PC Coordinator. The 2PC Coordinator is responsible for delegating the calls to Data Managers (local and remote) to perform necessary operations.

Data Managers (local and remote) perform operations in the order received to achieve replication with consistency.

The following commands were implemented to achieve replication.

addRow:

In cluster mode, add row operation from user is intercepted to invoke 2PC Coordinator, which subsequently sends the *addRow* command along with session id, host id, transaction id (global timestamp), and the row data. The 2PC Coordinator sends the addRow command to the local DataManager. It also sends a gRPC request to all the followers to perform a *rowOp* with the addRow request. The Command Processor at the followers processes the *rowOp* and sends it to the its DataManager. The local DataManager replicates the addRow request.

updateRow:

In cluster mode, update row operation from user is intercepted to invoke 2PC Coordinator, which subsequently sends the *updateRow* command along with session id, host id, transaction id (global timestamp), the old row data, and the new row data. This is replicated across the followers in a manner similar to the *addRow* request.

deleteRow:

In cluster mode, delete row operation from user is intercepted to invoke 2PC Coordinator, which subsequently sends the *deleteRow* command along with session id, host id, transaction id (global timestamp), and the row key. This is replicated across the followers in a manner similar to the *addRow* request.

readRow:

H2 processes read operation using MVPrimaryIndex as a cache. These read operations are intercepted at the MVPrimaryIndex get request and processed using a *readRow* request on the local DataManager. There is no need to replicate the request to followers as each DataManager has a copy of all the prewrites and can return any updates within that transaction.

3.2 Atomic commitment

Atomic commitment is achieved through the two-phase commit protocol.

In the first phase of two-phase commit, the 2PC Coordinator issues prewrite commands Prewrite (X) (where X is the key in the database index identifying the row) for each of the write requests (create-row, update-row, or delete-row) received from the user. The Data Managers check for conflicts (both read-write and write-write) before accepting the prewrite requests. If

a node is able to allow the prewrite Prewrite (X), the prewrite is recorded in an in-memory data structure and the node responds back with OK vote. If a node cannot allow the prewrite due to conflicts (either read-write or write-write), then the node responds back with ABORT vote. The coordinator proceeds with the transaction if it receives OK vote from all the followers. Otherwise, the coordinator sends ABORT message to all followers, appends abort record to the commit log, and aborts the transaction. In case of failure at any of the nodes before receiving response for prewrite operations, the coordinator waits up to a pre-configured period of time (currently, 200 ms) before aborting the transaction.

When the 2PC Coordinator receives read requests Read (X) (where X is the key in the database index identifying the row), the coordinator need not send the request to followers as the coordinator also maintains the status of prewrites just like any other node participating in the transaction. The coordinator responds by checking its data manager for current value of X.

When the user sends commit request, the 2PC Coordinator sends a PREPARE command to all followers. The followers then persist the prewrite state for the transaction and also the commit log record. At this point, the implementation persists two separate log files – first being log of pending prewrites and second being the commit log. The prewrite log is overwritten every time, and the commit log is an append only log. The commit log is cleaned up only during a recovery scenario. (This can be improved by pruning the log based on a schedule.)

In the second phase of the two-phase commit, the coordinator appends commit entry to its commit log, and sends commit message to all the followers. The data managers make sure that the writes are performed for each of the pending prewrites in the transaction. Successful prewrites during the first phase of the two-phase commit process mean that there will be no conflicts during the second (commit) phase. We will see why this is so when we discuss concurrency in the next section. The coordinator waits for all messages in response to the commit request before committing the transaction and writing transaction end entry to the commit log.

If the coordinator fails after writing commit log entry and before writing transaction end entry, the coordinator will try to recover after restart (manually) by requesting status messages from all followers. Based on the result from the followers, the coordinator performs COMMIT or ABORT locally. This recovery requires all followers to respond with same status for the transaction.

Further, a follower node can fail after prepare phase. After the failed follower node is started (manually), the node will check for any prepared transactions that were not committed from its commit log and asks the coordinator for the result of the transaction(s). Based on the response from coordinator, the node will perform COMMIT or ABORT on its end for the transaction(s). If, the node is unable to contact the coordinator for any reason, the node will keep trying to reach coordinator till it succeeds. This is a limitation of the 2PC process.

Further, in order to avoid reading from disk for transaction status, the log manager keeps a copy of the commit log in-memory.

3.3 Example

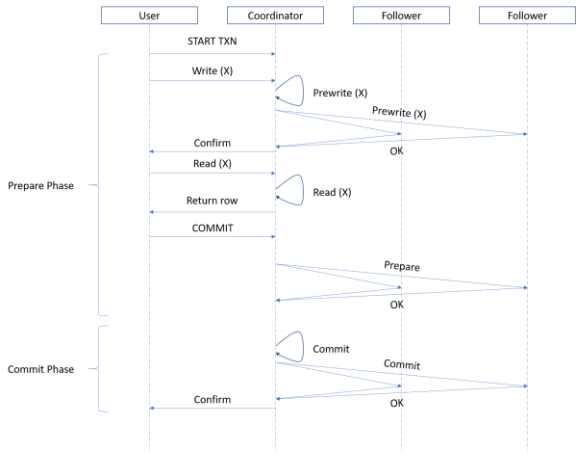

FIG. 3: 2PC for replication and atomic commitment

FIG. 3 illustrates the 2PC process using an example transaction that includes one write operation on X, followed by a read operation on X. In this example, there are two follower nodes. When the user starts transaction, the coordinator creates a transaction locally. When the user requests write on X, the coordinator sends prewrite request to followers. The coordinator sends the session id, the timestamp of the transaction, host id, command, and actual data as part of the request. The followers make concurrency checks (as will be discussed in the next section) before sending OK vote to the coordinator.

Subsequently, in FIG. 2, user requests read on X. The coordinator sends read request to the local data manager only and provides the response.

After the successful, write and read on X, user commits the transaction. At this point, the coordinator sends commit message to the followers. The followers convert pending prewrites (prewrite on X) to write on X. The data mangers issue actual write operation to the H2 MVStore before calling commit on the relevant session.

It is to be noted that the followers may not have an associated user session when the commit request is received by the followers. User session only exists at the coordinator. Therefore, before commit, the followers check if an active session already exists. If not, the followers create a new session for the given combination of session id and host id. The session is re-used for subsequent requests for the same combination of session id and host id. The relevant session created locally is used to perform commit operation at the followers. With the current implementation, the session objects are not being cleaned. The session id is assigned using a sequence number and the number of parallel sessions is limited in H2 to a configurable number. So, the number of session objects can only grow up to the number of maximum sessions allowed. Keeping this in mind, I have not tried to address the problem of increasing number of session objects in cache. An LRU based eviction mechanism can easily be implemented to remove session objects that are not used often.

4 Concurrency using Timestamp Ordering

Data Manager implements concurrency using timestamp ordering. In the current version of the implementation, timestamp ordering is achieved by using a global timestamp for the cluster. The global timestamp is a timestamp issued by a node designated as the timestamp server. For every transaction, the timestamp server issues the transaction id. For transactions started on the timestamp server, the timestamp is issued locally. And, for transactions started on a node that is not a designated timestamp server, the node issues gRPC command to the designated timestamp server for the timestamp.

Since timestamp is issued by a single server, the timestamps are guaranteed to be in order. I acknowledge that the delay added by the gRPC request can cause variances. In the interest of making a version of implementation possible for the project, the variances caused by this delay were ignored. This can be improved by using a timestamp range used in Spanner[6] or in Cockroach DB[7].

The algorithm used for concurrency is derived from [8] and is provided in Table 1 hereunder.

1: Each transaction receives a timestamp when it is initiated at its site of origin;
2: Each read R or write W operation which is required by a transaction has the timestamp of the transaction;
3: Each data item(x) contains the following information:
(i) WTM (x) - the largest timestamp of a write operation on x;
(ii) RTM (x) - the largest timestamp of a read operation on x;
4: Let TS be the timestamp of a prewrite operation PW_i on data item x;
If TS < RTM(x) or TS < WTM (x) then
reject PWi and restart the transaction;
else
put the PWi and its TS into the buffer;
5: Let TS be the timestamp of a read operation Ri on data item x, and TS(PW− min) the lower timestamp of any prewrite in the buffer;
If TS < WTM(x) then
reject Ri and restart the transaction;
else // TS >= WTM (x)
If (no PWi in the buffer)
execute Ri and RTM(x)= max(RTM(x), TS); else
If TS <= TS(PW-min) then
execute Ri and RTM(x) = max(RTM(x), TS);
else // there is one (or more) PW with TS(PW) < TS
Ri is buffered until all transactions which
has TS(PW) < TS commit;
6: Let TS be the timestamp of write operation W_i on data item x. This operation is never rejected; however, it is possibly buffered if there is a prewrite operation PW(x) with TS(PW)<TS. W_i will be executed and eliminated from the buffer when all prewrites with smaller timestamp have been eliminated from the buffer.

Table 1: Timestamp ordering with 2PC [8]

The implementation logic for the algorithm described above can be found in the Data Manager (DataManager.java in the source code).

5 Performance evaluation

The performance data is provided here in the following in Table 2. The data is included for H2 as a single node, H2 native cluster (two nodes), H2 cluster single node, H2 cluster with 1 follower, and H2 cluster with 3 followers.

Single Node

Insert 100 rows: 20 ms; Commit: 6 ms
Select 100 rows: 10 ms
Update 100 rows: 16 ms; Commit: 4 ms

Two Node HA (native cluster mode)

Insert 100 rows: 23 ms; Commit: 10 ms
Select 100 rows: 8 ms
Update 100 rows: 30 ms; Commit: 5 ms

H2 Cluster (Coordinator only)

Insert 100 rows: 100 ms; Commit: 38 ms
Select 100 rows: 12 ms

Update 100 rows: 16 ms; Commit: 4 ms
<u>H2 Cluster (1 follower)</u>
Insert 100 rows: 374 ms; Commit: 44 ms *Select 100 rows: 11 ms* *Update 100 rows: 669 ms; Commit: 78 ms*
<u>H2 Cluster (3 followers)</u>
Insert 100 rows: 452 ms; Commit: 56 ms *Select 100 rows: 13 ms* *Update 100 rows: 745 ms; Commit: 55 ms*

Table 2: Performance data

The performance data shows that there is significant degradation in performance. The latencies introduced are primarily due to the gRPC delays and the additional log writes.

In the current state of implementation, each data operation command (*addRow*, *updateRow*, *deleteRow*) is being sent to followers separately. And the delays accumulate with increasing number of statements.

Further, performance on update statements is twice worse because update is implemented as two operations in H2 – an add operation followed by a delete operation. And, therefore, there are 2X messages being sent for update operation.

I believe this can be improved significantly by accumulating all write operations, and sending all the operations along with the PREPARE command.

6 Future work

As noted throughout the paper, many improvements and optimizations are possible within the current implementation. The notable ones are: combining data and commit log into a single log file, using timestamp range for timestamp ordering instead of a global timestamp, and replicating all write operations in a transaction as part of PREPARE command.

Further, the project currently provides an additional layer of concurrency to make things work in cluster mode. This can be improved by integrating with H2's MVCC (Multi-Version Concurrency Control) implementation. Specifically, MVCC can be integrated with global timestamp so that MVCC snapshots are synchronized across nodes. This way it is possible to work with H2's native concurrency mechanism (with some updates) instead of introducing another layer of concurrency control on top of the database. I believe this can provide additional performance benefits.

7 Conclusion

In conclusion, this project provides a simple implementation of clustering for H2. The implementation allows using H2 in a real cluster mode across many nodes without any major configuration requirements. The implementation can be easily adapted to embedded mode of H2 as well. The implementation can make use of many performance enhancements noted in the description. However, I believe the current implementation is a good starting point, and the work can be enhanced easily to make it useful for others.

8 Acknowledgements

I would like to thank David Mazieres and Jim Posen for making the course possible during the difficult times of COVID-19. Their ideas and guidance for this project have been very valuable. Also, the open-ended nature of the course made it possible for me to work on

something relevant to my interests and at the same time learn concepts taught during the course. Thank you!

REFERENCES

1. H2 Database: https://www.h2database.com/html/main.html
2. H2 Database MVStore reference: https://www.h2database.com/html/mvstore.html
3. H2 Database Native Clustering reference: https://www.h2database.com/html/advanced.html#clustering
4. Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. ACM Comput. Surv. 13, 2 (June 1981), 185–221. DOI:https://doi.org/10.1145/356842.356846
5. Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc., USA.
6. James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. ACM Trans. Comput. Syst. 31, 3, Article 8 (August 2013), 22 pages. DOI:https://doi.org/10.1145/2491245
7. Cockroach DB: https://www.cockroachlabs.com/blog/living-without-atomic-clocks/
8. Luiz Alexandre Hiane da Silva Maciel. A TIMESTAMP-BASED TWO PHASE COMMIT PROTOCOL FOR WEB SERVICES USING REST ARCHITECTURAL STYLE