Multi-version Concurrency Control on H2database

Sungkeun Kim
Computer Science and Engineering
Texas A&M University
College Station, Texas 77843
Ksungkeun84@tamu.edu

***Abstract*—This paper illustrates what the multi-version concurrency control system is and how it is implemented in the h2database. Demand in faster concurrent system, the multi-version concurrent control system is replacing two-phase locking method which is commonly used by database management system. The MVCC tries to reduce locking and manages multiple versions of the same row. In order to implement this method, h2database uses the undo log to manage uncommitted changes so that rows can be reconstructed to their previous committed version. By using three simple examples, we expect that readers can understand internal process of the multi-versioning. Although a transaction isolation level is not fully supported under the multi-version concurrency control mode and this mode is not fully tested, the h2database open source project is still actively implementing and stabilizing this feature.**

1. Introduction

In concurrency control theory, there are two ways we can deal with conflicts: (1) The two-phase-locking(2PL) and (2) the multiversion concurrency control(MVCC). 2PL is the simplest way to provide concurrent access to the database with consistent data. 2PL makes all readers wait until the writer finishes its job, which is called a lock. Because this can be very slow, MVCC uses a different method to reduce locking. In this paper we will demonstrate what the multi-version concurrency control is and how it is implemented in the h2database. Also, we will briefly see the limitation of the current MVCC in the h2database.

2. Preliminary

2.1. Concurrency Control

Before we take a look at the internal implementation of MVCC, we need to understand what the MVCC is in a high-level perspective. In the Two-Phase locking(2PL) scheme, every read operation should acquire a shared lock, while a write operation requires taking an exclusive lock.

- A shared lock blocks writers, but allows other readers to acquire the same shared lock
- An exclusive lock blocks both readers and writers accessing the same lock

However, locking causes conflict, and conflict affects scalability. Amdhal's Law or Universal Scalability Law demonstrates how conflict can affect response time. For this reason, database researchers have devised a different concurrency control model which reduces locking to a minimum so that:

- Readers don't block writers
- Writers don't block readers

There is only one scenario that can still generate conflict. When two concurrent transactions try to update the same row, the row is always locked until the transaction that updated this row commits its operation.

In order to achieve the non-locking behavior which is mentioned above, the concurrency control mechanism must operate on a multiversion of the same row, hence this mechanism is called Multi-Version Concurrency Controll(MVCC). For example, when the MVCC database needs to modify an item of data, it will not overwrite the old data with new data, but instead will mark the old data as no longer in use and add the newer version elsewhere. Thus there are multiple versions, but only one is the latest. This allows readers to access the data that was there when they began to read, even if it was modified or deleted by someone else.

2.2. Examples

To understand how MVCC works, we presents three examples: insertion, deletion and update. Each example is illustrated in Figure 1, 2, and 3 respectively.

Inserting a row

1. Both Session1 and Session2 start a new transaction, and Session1 creates a new table called person that has three columns: id, name and age.
2. After Session1 inserts a new row where its id is one, name is Aaron and its age is twenty, it can see this row, but Session2 cannot see it.
3. After Session1 has committed, Session2 can see the row where its id is one.

Deleting a row

1. Both Session1 and Session2 start a new transaction and Session1 executes a query to delete a row where its id is one, but does not commit yet.
2. Although the row is removed from the database, Session2 still can see the row that was deleted by Session1.
3. After Session1 has committed, Session2 no longer sees the deleted row.

The deletion operation does not physically remove a row, it just remembers it as ready for deletion, and the cleaning process will wipe it out when this row is no longer in use by any current running transaction.

Updating a row

1. Both Session1 and Session2 start a new transaction, and Session1 inserts a row into the table and committs its operation.
2. When Session1 updates the row where its id is one, the age is changed from 20 to 30, Session2 can see the previous row which is the latest committed version.
3. Until Session1 commits its transaction, Session2 cannot see the uncommitted version – the age is 20.

While 2PL is pretty much standardized, there is no standard MVCC implementation. Thus, each database management system takes a slightly different approach. In the next part, we will explore the internal implementation of MVCC on the h2databse which is a very active open source project.

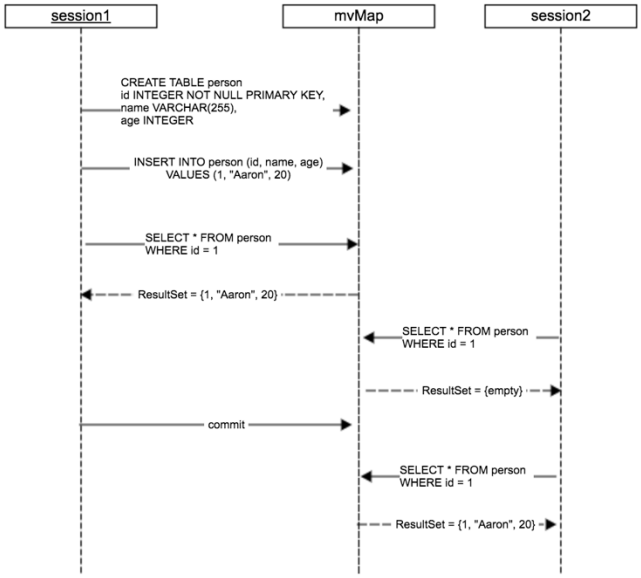

Fig. 1. Inserting a row

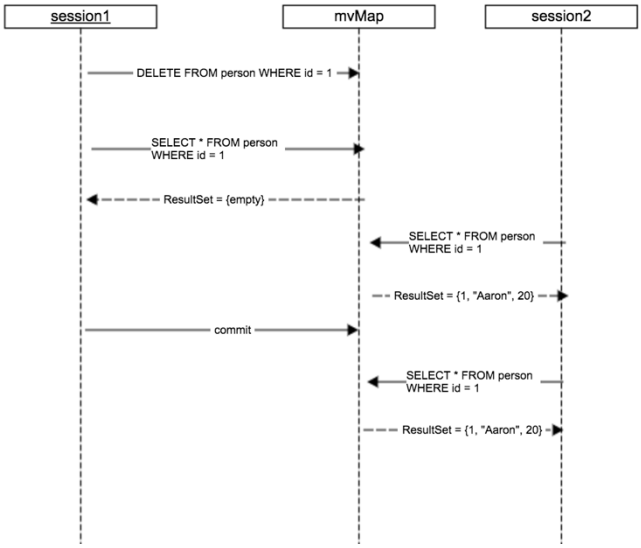

Fig. 2. Deleting a row

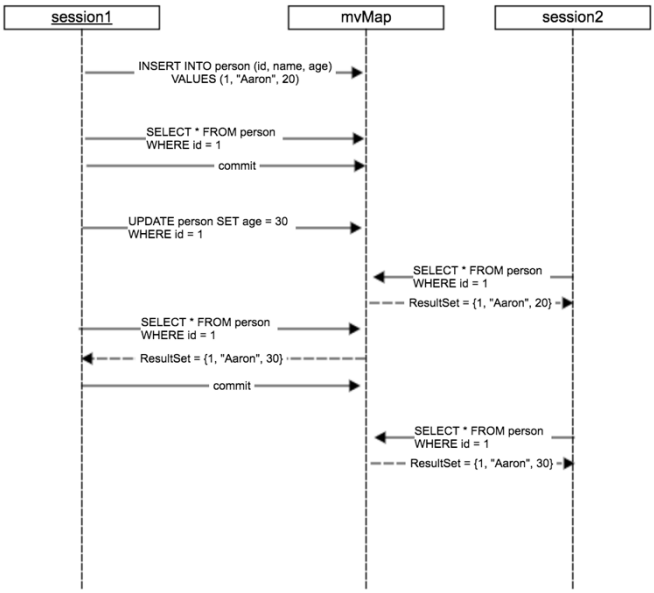

Fig. 3. Updating a row

3. Multi-Versioning in H2database

In the preliminary part, we understood how the multi-version concurrency control system works. Now, we are going to look inside MVCC and understand how it is implemented into the h2database.

Similar to Oracle and MySQL, the h2database uses the undo log to capture uncommitted changes so that rows can be reconstructed to their previously committed version. In other words, the h2database keeps information about old versions of changed rows to support transactional features such as concurrency and rollback. The h2database uses the information in the undo log to perform the undo operations needed in a transaction rollback. It also uses the information to build earlier versions of a row for a consistent read.

Figure 4 shows the internal data structure related to multi-versioning. The h2database has various key-value paired maps to manage tables and related information such as the undo log. Under the mutlti-versioning mode, a *MVTable* class is used for tables, and a *MVMap* class is used for the undo log that stores old versions of changed rows. Also, the *MVMap* is used for special maps that store the latest version of changed rows. Each table has one *MVMap* and this map is stored in another map called *maps* in Figure 4 as a value. Although the *MVMap* has been used for various purposes, let us assume that *the MVMap* is used only for values of the *maps* in Figure 4 for simplicity.

In Figure 4, there are two tables which are stored in *a tableMaps* as a value and their keys are *table.3* and *table.6* respectively. These two keys of *the MVTable* are stored in the corresponding *MVMaps* as a value and their keys are 7 and 8 respectively. For example, table *person* that is created in Figure 1 is mapped to *table.3* and this table information is stored in a *MVMap* as a value which is mapped to 7. Let this be *MVMap*.7

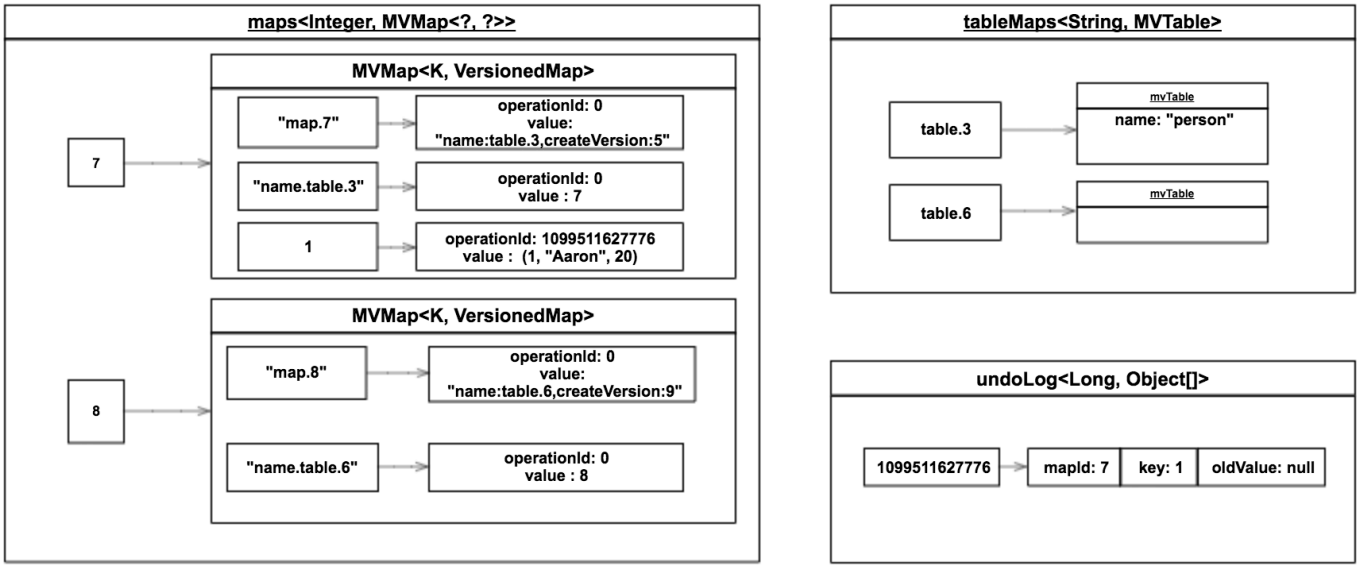

Fig. 4. Internal Datastructure of the h2database with MVCC mode

for short. After the insertion query is executed in Figure 1, the key-value pair is stored in *MVMap.7*: key is one and the value contains the row where its id is one and an operation identifier.

The value of *MVMap* has additional information called an operation identifier for tracing its multiversions of changed rows. It consists of a combination of the transaction id and the log id. Figure 5 shows how the transaction id and the log id are combined. The log id is initialized to zero. It increases by one whenever a new log is added into the undo log and decreases by one whenever a log is removed from the undo log.

```
long getOperationId(int transactionId, long logId)
{
return ((long) transactionId << 40) | logId;
}
```

Fig. 5. Combination the transaction id and the log id to an operation id.

We understood the internal MVCC and how it works so far. From now on, we are going to illustrate how a transaction find its consistent data from *MVMap*. Here, we have a question: what is the consistent data of a specific transaction? The answer is in Figure 6 which shows a simplified algorithm that finds the consistent data. In the multi-versioning scheme on the h2database, the consistent data means that:

- Condition1 - an operation id in the data is zero, which means this data has been committed.
- Condition2 - a transaction id that is mixed in the operation id is the same as the currently operating transaction id.

If current data is not satisfied with the above two conditions, the data is replaced with an another data which is mapped to the operation id and the algorithm Figure 6 keeps checking above conditions. Because the h2database only supports *read committed isolation level* under MVCC scheme, the above two are the only condition of the consistent data.

```
VersionedValue getValue(K key, long maxLog,
                        VersionedValue data) {
    while (true) {
        if (data == null)
            return null;

        // Condition1
        long id = data.operationId;
        if (id == 0)
            return data;

        // Condition2
        int tx = getTransactionId(id);
        if (tx == transaction.transactionId) {
            if (getLogId(id) < maxLog) {
                return data;
            }
        }

        // find the data from the undo log
        Object[] d;
        d = undoLog.get(id);
        data = (VersionedValue) d[2];
    }
}
```

Fig. 6. An algorithm shows how to retrieve the consistent data from the undo log.

3.1. Examples

We have seen the simple examples - insertion, deletion, and update - to understand how the multi-version concurrency control system works. Now, with these examples, we are going to understand its internal operation process.

Inserting a row

Figure 7 shows the status of *MVMap*.7 and the undo log after Session1 finished inserting a new row in Figure 1. Before insertion, there was no key-value pair, but now the new row is stored in the *MVMap*.7 as a value where its operation id is 1099511627776(1/1) – transaction id is one and log id is one. Also, this operation id and the information of the previous version of the new row are stored in the undo log as a key-value pair. In this scenario the previous version is null-value because there was no row where its id is one in the table *person*.

When Session2 tries to select a row where its id is one, it first tries to find the data in the *MVMap*.7. Although there exists the row Session2 asked for, this row was not added by itself but by Session1 – transaction id of Session2 is two and transaction id of the data is one. The data in *MVTable*.7 is not satisfied with the Condition1. Also, the operation id is not zero which means the data is not satisfied with Condition2. Therefore, Session2 tries to look up the latest version which was added by itself or a committed version in the undo log.

Now, Session2 tries to look up a row using operation id which is 1099511627776 in Figure 7. There exists a value mapped to the operation id, and its operation id is zero which means this data has been committed. This is satisfied with Condition1. Therefore, the old value mapped to operation id in the undo log will be returned to Session2 and its actual value is empty.

Deleting a row

Figure 8 shows the status of *MVMap*.7 and the undo log after Session1 deleted the inserted row which is already committed. In the multi-versioning scheme, the row is not physically removed from the *MVMap*.7 immediately when Session1 deletes it. The value mapped to key 7 in the *MVMap*.7 is just replaced to null. Also, a new key-value pair added to the undo log. Its key is the operation id and the value is the deleted row.

When Session2 tries to select the row where its id is one, it looks up the *MVMap*.7 first. Although there exists the data that Session2 asked for but it is not added by itself. This means the data is not satisfied with the Condition2. Therefore, Session2 tries to look up the undo log with operation id stored in the data which is 1099511627776. Because there exists the value which is mapped to the operation id 1099511627776 and its operation id is zero, Session2 can see this value.

Updating a row

Figure 9 shows the status of the *MVMap*.7 and the undo log after Session1 updated the age of the row where the id is one. If the row was updated, the undo log contains the information necessary to rebuild the content of the row before it was updated. When Session2 tries to select the row where the id is one, it looks up the *MVMap*.7 first and then looks up the undo log, if necessary, in the same way that we've mentioned in selecting a row.

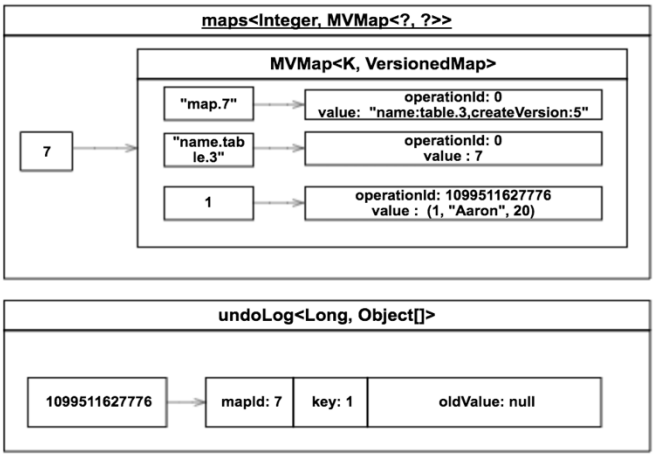

Fig. 7. Datastructures after inserting a row in Figure 1.

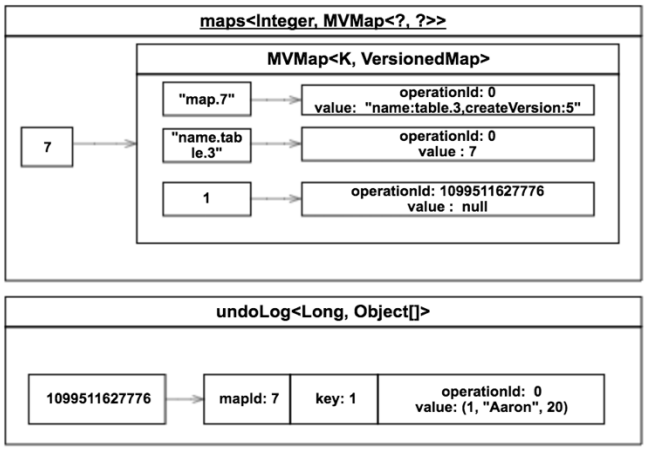

Fig. 8. Datastructures after deleting a row in Figure 2

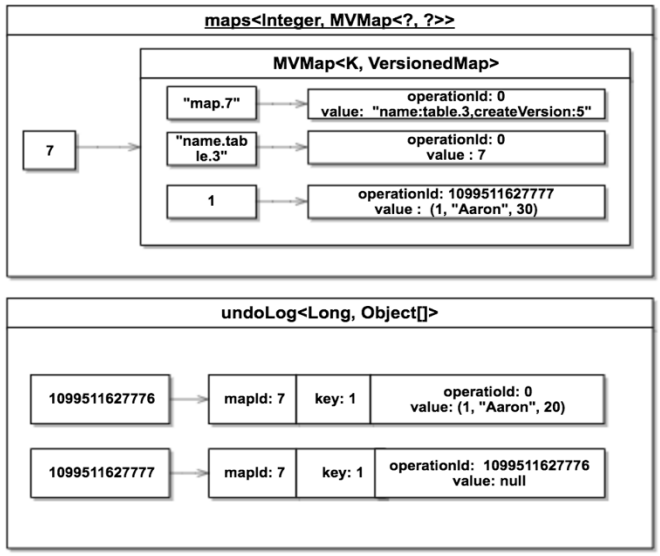

Fig. 9. Datastructures after update the row in Figure 3.

4. The current progress of MVCC

4.1. Unsupported Lock mode

Currently, the h2database fully supports the transaction isolation level only for the non-MVCC scheme, while the *read committed isolation level* is used for MVCC scheme[1]. It means that changing isolation level has no effect on MVCC.

Transaction Isolation Level

Transaction isolation levels are a measure of the extent to which transaction isolation succeeds. In other words, transaction isolation levels are defined by the presence or absence of the following phenomena:

- Dirty Reads - occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.
- Non-repeatable Reads – occurs when a transaction reads the same row twice but gets different data each time.
- Phantom Reads – occurs when a transaction executes two identical queries and the collection of rows returned by the second query is different from the first.

The four transaction isolation levels as defined by SQL-92 are defined in terms of the phenomena above. In the following Figure 10, an "O" sign indicates each phenomenon that can occur. That is, the three phenomena could be occured in Read uncommitted level, but the only dirty read phenomenon could be occurred in Read committed level. None of the phenomena could not be occurred in Serializable level.

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
Read uncommitted	O	O	O
Read committed	X	O	O
Serializable	X	X	X

Fig. 10. Standard concurrency side effects

As mentioned earlier in this part, changing the isolation level has no effect if the h2database uses the MVCC scheme. Figure 11 shows that every isolation level acts like the read uncommitted isolation.

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
Read uncommitted	X	O	O
Read committed	X	O	O
Serializable	X	O	O

Fig. 11. Concurrency side effects from the H2database with MVCC

5. Conclusion

We have presented the concurrency control systems such as 2PL and the MVCC system. Also we have illustrated what the MVCC is and how it work both the high-level and low-level perspective. Finally, we have seen the currency progress of the MVCC in the h2database. The MVCC feature is not fully tested yet and many bugs are reported to the open source project of the h2database but this project is actively implementing and stablizing this feature. This paper would be a foundation of our research related to the concurrency control and we will keep improving this paper with new technical knowledge. We will also contribute our knowledge to the h2database by resolving bugs related to the concurrency.

REFERENCES

[1] H2 database engine. Retrieved May 11, 2017, from http://www.h2database.com

[2] H2 database git hub. Retrieved May 11, 2017, from https://github.com/h2database/h2database

[3] InnoDb multi-versionning. Retrieved May 11, 2017, from https://dev.mysql.com/doc/refman/5.7/en/innodb-multi-versioning.html

[4] Vlad Mihalcea, "How does mvcc works" Retrieved May 11, 2017, from https://vladmihalcea.com/2017/03/01/how-does-mvcc-multi-version-concurrency-control-work/

[5] Microsoft online documents, "Transaction Isolation Level". Retrieved May 11, 2017, from https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels

[6] Wikipedia, "Multiversioni concurrent control". Retrieved May 11, 2017, from https://en.wikipedia.org/wiki/Multiversion_concurrency_control