H2 database and MVCC - 2nd week

Sungkeun Kim

Report History

Week 1	Understanding MVCC and test program
Week 2	Issue analysis - Isolation level with MVCC - Part 1

1. Overview of weekly report

In this week, I chose one issue registered in the h2 github, learned the domain knowledge of that issue, reproduce it, and finally started to review the issue related codes. The issue that I pick is #174 and #216. These issues claim that h2 database doesn't support transaction isolation level with MVCC mode. By creating test programs to reproduce this issue, I confirmed this is a real issue. Also, to be more correct, I compared H2 with the HSQL database which is known for supporting isolation level with MVCC.

Currently, I am working on the debugging this issue, the next week, I would be continued to debug this issue.

If you want to look at the weekly report real quick, please read section 2, 4, 5, and 6. These sections describes the issue and the result of the reproduction.

2. Issue description

I focused on the issue #174 first. The reporter of this issue claims that in MVCC with **SERIALIZATION** mode, the **"Phantom reads"** phenomenon takes place. This read phenomenon should never occur in the serialization mode. According to the ANSI SQL 92[2], phantom read cannot be happened. However, this happens in H2 database when MVCC mode is enabled.

3. What is the Isolation serialization and Phantom Read?

Isolation serialization

As one of the ACID (Atomicity, Consistency, Isolation, Durability) properties, Isolation determines how transaction integrity is visible to other users and system. Isolation has four levels: Serializable, Repeatable Reads, Read Committed, Read Uncommitted.

In the serializable level with lock-based DBMS, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. When using non-lock based concurrency control, no locks are acquired; however, if the system detects a *write collision* among several concurrent transactions, only one of them is allowed to commit.[1]

Phantom Read

A *phantom read* occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first.[1]

Transaction 1	Transactino 2

SELECT * **FROM** users
WHERE age **BETWEEN** 10 **AND** 30;	

	INSERT INTO users(id,name,age) **VALUES** (3, 'Bob', 27); **COMMIT**;

SELECT * **FROM** users
WHERE age **BETWEEN** 10 **AND** 30;
COMMIT;	

Note that Transaction 1 executed the same query twice. If the highest level of isolation were maintained, the same set of rows should be returned both times, and indeed that is what is mandated to occur in a database operating at the SQL SERIALIZABLE isolation level.[1] The following table shows the concurrency side effects enabled by the different isolation levels.

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

<Table1. The concurrency side effects>[3]

4. Reproduce the issue #174

In order to check if H2 has real problem with MVCC-Serialization mode, I made a test program which simulates the phantom reads phenomenon. The test program could reproduced this issue. If you interested in review the test code, refer my github. Here is the code snippet of the test code.

```
Connection conn1 = DB.connectDB(H2DB.DRIVER, url, USER_ADMIN, "");
conn1.setTransactionIsolation(lockMode);
conn1.setAutoCommit(false);

Connection conn2 = DB.connectDB(H2DB.DRIVER, url, USER_ADMIN, "");
conn2.setTransactionIsolation(lockMode);
conn2.setAutoCommit(false);

// "SELECT id, first, last, age FROM Registration WHERE age BETWEEN 1 AND 30";
int before = select(conn1);

// "INSERT INTO Registration " + "VALUES (104, 'Sungkeun', 'Kim', 2)")
insertRecord(conn2);

conn2.commit();

// "SELECT id, first, last, age FROM Registration WHERE age BETWEEN 1 AND 30";
int after = select(conn1);

Assert.assertEquals(before, after);
```

5. Compare H2 to HSQL

For making sure that my test program is correct, I extended the test program to check every concurrency side effects by the different isolation levels, and compare it to the HSQL which is known for supporting serializable level under MVCC. Following four tables show the result of the test program.

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

< Table 2. Concurrency Side effects from H2 Database with MVCC>

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

< Table 3. Concurrency Side effects from HSQL Database with MVCC>

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

< Table 4. Concurrency Side effects from H2 Database with Non-MVCC>

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

< Table 5. Concurrency Side effects from HSQL Database with Non-MVCC>

The red and blue colored O/X means that the result mismatches with the expected result. The blue “X” just means that my program doesn’t reproduce the side effect and we don’t know whether there is a bug or not. However, look at the red circle. It means that my program causes the side effect(phantom read) that should never occur in the serialization isolation. This is a bug!

6. H2 Documentation on the MVCC and Isolation level

In the H2 document, there is comment on the MVCC and Isolation level. It says that if MVCC is enabled, changing the lock mode (isolation level) has no effect.[4] Based on the document, this issue is not a bug, but unimplemented feature. I want to make it sure and added my comment on the github[5].

7. Debugging preparation

So far, I learned what is the Isolation and its side effects. Also I tested and reproduced the issue #174. After than, I started to review the source codes.

7.1. Starting point of H2 database server

This is the first time to review h2 database, I need to know where is the starting point of the h2 database server. Therefore, I looked up the manifest file in the jar file. org.h2.tools.Console is the starting point of the h2 database server.

```
Manifest-Version: 1.0
Implementation-Title: H2 Database Engine
Build-Jdk: 1.8
Created-By: 1.8.0_101-b13 (Oracle Corporation)
Main-Class: org.h2.tools.Console // ← this is the main class
Bundle-Activator: org.h2.util.DbDriverActivator
```

<Code 1. META-INF/MANIFEST.MF>

7.2. Enable debugging logs

To enable debugging system on the h2 database, change the DEBUG level to DEBUG.

```
/**
 * The default level for file trace messages.
 */
public static final int DEFAULT_TRACE_LEVEL_FILE = OFF;  // ERROR, INFO, DEBUG, ADAPTER
```

Then, debugging logs are printed. Here is an interesting feature that prints the client program code. This would be helpful for me to navigate the source code.

Client code	`Connection conn2 = DB.connectDB(H2DB.DRIVER, url, USER_ADMIN, "");`
Log	`/**/Connection conn2 = DriverManager.getConnection("jdbc:h2:tcp://localhost/~/test;MVCC=TRUE", "SA", "");`

Server code	`/**`
	` * Write trace information as a method call in the form`
	` * objectName.methodName(param) where the parameter is formatted as a long`
	` * value.`
	` *`
	` * @param methodName the method nam`
	` * @param param one single long parameter`
	` */`
	
	`protected void debugCodeCall(String methodName, long param) {`
	`    if (trace.isDebugEnabled()) {`
	`        trace.debugCode(getTraceObjectName() + "." +`
	`                methodName + "(" + param + ");");`
	`    }`
	`}`

```
04-06 10:47:46 jdbc[3]:
/*SQL l:43 #:1 t:1*/INSERT INTO SESSIONS VALUES(?, ?, ?, NOW()) {1: 3, 2:
'jdbc:h2:tcp://localhost/~/test;MVCC=TRUE', 3: 'SA'};
04-06 10:47:46 lock: 3 shared read lock unlock SESSIONS
04-06 10:47:46 jdbc[3]:
/*SQL #:2 t:2*/SELECT id, first, last, age FROM Registration WHERE age BETWEEN 1 AND 30;
04-06 10:47:46 jdbc[4]:
/*SQL #:2 t:1*/SELECT id, first, last, age FROM Registration WHERE age BETWEEN 1 AND 30;
04-06 10:47:46 lock: 3 shared read lock requesting for REGISTRATION
04-06 10:47:46 lock: 3 shared read lock ok REGISTRATION
04-06 10:47:46 jdbc[3]:
/*SQL #:1 t:6*/INSERT INTO Registration VALUES (3, 'Sungkeun', 'Kim', 2);
04-06 10:47:46 jdbc[3]:
/*SQL #:3 t:1*/SELECT id, first, last, age FROM Registration WHERE age BETWEEN 1 AND 30;
04-06 10:47:46 jdbc[4]:
/*SQL #:2*/SELECT id, first, last, age FROM Registration WHERE age BETWEEN 1 AND 30;
```

8. Debugging the issue

In order to debug this issue, I divided it into two categories.

8.1 Does isolation level saved to its value well?

Yes, it does.

```
04-06 23:08:18 jdbc:
/**/conn1.setTransactionIsolation(8); // 8: serializable mode
```

```
public void setTransactionIsolation(int level) throws SQLException {
    try {
        debugCodeCall("setTransactionIsolation", level);
```

8.2 When the write operation is occurred by conn2 connection, what happen to the conn1 connection?

I am working on it. It takes some time to understand internal logic and find related codes. This would be the next week's major tasks.

References

[1] Isolation (database systems)

https://en.wikipedia.org/wiki/Isolation_(database_systems)

[2] A Critique of ANSI SQL Isolation Levels

http://www.cs.umb.edu/~poneil/iso.pdf

[3] Isolation Levels in the Database Engine
https://technet.microsoft.com/en-us/library/ms189122(v=SQL.105).aspx

[4] H2 - Advanced

http://www.h2database.com/html/advanced.html#mvcc

[5] H2 Github-issue

https://github.com/h2database/h2database/issues/174

[6] Github of Test programs

https://github.com/danguria/H2Exercise/tree/serializable_isolation