

# HSQldb的事务机制分析

软件62 周展平 2016013253

## HSQldb的事务机制分析

### 问题思考

1. 关闭AutoCommit情况下在commit之前insert语句在内存中的存在形式
2. 关闭AutoCommit情况下commit的执行逻辑
3. 保存点的实现
4. 主键、外键完整性约束的实现
  1. 主键约束
  2. 外键约束
5. insert语句相关的触发器及其触发时间

## 问题思考

### 1. 关闭AutoCommit情况下在commit之前insert语句在内存中的存在形式

在StatementDML的insertSingleRow函数设置断点，执行语句：

```
set autocommit false
insert into account values ('00000002', 'student1', 22)
```

可以发现，插入的记录通过data数组传入该函数，用查看器查看具体信息，如下：

Name	Value
data	(id=1457)
> [0]	"00000002" (id=1460)
> [1]	"student1" (id=1461)
> [2]	BigDecimal (id=1462)
Add new expression	
[00000002, student1, 22.00]	

因次，insert语句要插入的数据是以数组的形式存在的，每一个属性的值对应一个对象。之后，调用了Table类的insertSingleRow函数执行插入操作，首先通过PersistentStore接口的getNewCachedObject方法创建一个Row类及其子类的对象，然后通过Session类的addInsertAction方法将这次insert事物添加到rowActionList中。

## 2. 关闭AutoCommit情况下commit的执行逻辑

Session类的commit方法如下：

```
/**
 * Commits any uncommitted transaction this Session may have open
 *
 * @throws SQLException
 */
public synchronized void commit(boolean chain) {

    //      tempActionHistory.add("commit " + actionTimestamp);
    if (isClosed) {
        return;
    }

    if (sessionContext.depth > 0) {
        return;
    }

    if (isTransaction) {
        if (!database.txManager.commitTransaction(this)) { //commit失败

            // tempActionHistory.add("commit aborts " + actionTimestamp);
            rollbackNoCheck(chain); //进行回滚

            throw Error.error(ErrorCode.X_40001);
        }
    }

    endTransaction(true, chain); //结束本次transaction

    if (database != null && !sessionUser.isSystem()
        && database.Logger.needsCheckpointReset()) {
        database.checkpointRunner.start();
    }
}
```

commit函数首先调用database.txManager.commitTransaction方法尝试进行commit，TransactionManager2PL类的commitTransaction方法如下：

```
public boolean commitTransaction(Session session) {

    if (session.abortTransaction) { //判断是否终止了执行，如果是则直接退出
        return false;
    }
}
```

```

writeLock.lock(); //获取写数据的锁

try {
    int limit = session.rowActionList.size();

    // new actionTimestamp used for commitTimestamp
    session.actionTimestamp = getNextGlobalChangeTimestamp();
    session.transactionEndTimestamp = session.actionTimestamp;

    endTransaction(session);
    //依次获取rowActionList中的action进行commit
    for (int i = 0; i < limit; i++) {
        RowAction action = (RowAction) session.rowActionList.get(i);

        action.commit(session); //对每一个action分别进行commit
    }

    adjustLobUsage(session);
    persistCommit(session); //进行持久化的commit, 写入文件

    session.isTransaction = false;

    endTransactionTPL(session);
} finally {
    writeLock.unlock(); //释放写输入的锁
}

session.tempSet.clear();

return true;
}

```

首先调用writeLock.lock方法获取锁，之后依次对rowActionList中的action调用commit方法进行commit，然后调用persistCommit方法对commit进行持久化，其中主要的函数是PersistentStore接口的commitRow方法，调用Database.Logger类的writeInsertStatement、writeDeleteStatement等方法将执行的操作写入.script和.log文件中。

### 3. 保存点的实现

为了实现保存点，SessionContext类维护了两个属性：savepoints是HashMappedList类型的属性，通过保存点的名称进行hash，保存对应的rowActionList的大小；savepointTimestamps是LongDeque类型的属性，用于存储保存点的时间戳（actionTimestamp）。通过Session类的savepoint函数可以注册一个保存点，如下：

```

/**
 * Registers a transaction SAVEPOINT. A new SAVEPOINT with the

```

```

    * name of an existing one replaces the old SAVEPOINT.
    *
    * @param name name of the savepoint
    * @throws SQLException if there is no current transaction
    */
    public synchronized void savepoint(String name) {

        int index = sessionContext.savepoints.getIndex(name); //查找名称是否已经存在

        if (index != -1) { //如果重名，则删除旧的保存点
            sessionContext.savepoints.remove(name);
            sessionContext.savepointTimestamps.remove(index);
        }
        //添加新的保存点
        sessionContext.savepoints.add(name,
                                       valuePool.getInt(rowActionList.size()));
        sessionContext.savepointTimestamps.addLast(actionTimestamp);
    }

```

该方法会删除同名的旧保存点，并添加新的保存点。

当transaction结束之后需要删除保存点时，调用Session类的releaseSavepoint方法，如下：

```

/**
 * Releases a savepoint
 *
 * @param name name of savepoint
 * @throws SQLException if name does not correspond to a savepoint
 */
    public synchronized void releaseSavepoint(String name) {

        // remove this and all later savepoints
        int index = sessionContext.savepoints.getIndex(name);

        if (index < 0) {
            throw Error.error(ErrorCode.X_3B001, name);
        }
        //依次删除保存点及之后的所有保存点
        while (sessionContext.savepoints.size() > index) {
            sessionContext.savepoints.remove(sessionContext.savepoints.size()
                                             - 1);
            sessionContext.savepointTimestamps.removeLast();
        }
    }
}

```

此方法会删除该保存点及之后的所有保存点。如果需要回滚到某一个保存点，则需要调用Session类的rollbackToSavepoint方法，进而调用TransactionManager接口的rollbackSavepoint方法执行具体的操作。

## 4. 主键、外键完整性约束的实现

### 1. 主键约束

在插入记录时，IndexAVL类的insert方法会根据插入记录的值在AVL树中找到合适的插入位置，主键约束的检查就是在这里进行的，具体函数如下：

```
/**
 * Insert a node into the index
 */
public void insert(Session session, PersistentStore store, Row row) {

    NodeAVL n;
    NodeAVL x;
    boolean isleft      = true;
    int      compare     = -1;
    boolean compareRowId = !isUnique || hasNulls(session, row.getData());

    n = getAccessor(store);
    x = n;

    if (n == null) {
        store.setAccessor(this, ((RowAVL) row).getNode(position));

        return;
    }

    while (true) {
        Row currentRow = n.getRow(store);

        compare = compareRowForInsertOrDelete(session, row, currentRow,
                                              compareRowId, 0);

        // after the first match and check, all compares are with row id
        if (compare == 0 && session != null && !compareRowId
            && session.database.txManager.isMVRows()) {
            if (!isEqualReadable(session, store, n)) {
                compareRowId = true;
                compare = compareRowForInsertOrDelete(session, row,
                                                      currentRow,
                                                      compareRowId,
                                                      colIndex.length);
            }
        }
        //此处插入记录的索引与原有记录相同
        if (compare == 0) {
```

```

        Constraint c = null;

        if (isConstraint) { //违反了主键约束
            c = ((Table) table).getUniqueConstraintForIndex(this);
        }
        //抛出异常
        if (c == null) {
            throw Error.error(ErrorCode.X_23505, name.statementName);
        } else {
            throw c.getException(row.getData());
        }
    }

    isleft = compare < 0;
    x      = n;
    n      = x.child(store, isleft);

    if (n == null) {
        break;
    }
}

x = x.set(store, isleft, ((RowAVL) row).getNode(position));

balance(store, x, isleft);
}

```

其中当compare=0时表明原来存在一个与插入节点的索引值相同的节点，如果isConstraint属性为True，说明违反了主键约束，之后通过getUniqueConstraintForIndex方法获得主键约束并抛出异常。否则没有违反主键约束，记录被正常插入。

## 2. 外键约束

在插入记录时，StatementDML的performIntegrityChecks函数负责检查外键约束是否满足，其中主要调用了Constraint类的checkInsert函数，如下：

```

/**
 * Checks for foreign key or check constraint violation when
 * inserting a row into the child table.
 */
void checkInsert(Session session, Table table, Object[] data,
                 boolean isNew) {

    switch (constType) {

        case SchemaObject.ConstraintTypes.CHECK :

```

```

        if (!isNull) {
            checkCheckConstraint(session, table, data);
        }

        return;

    case SchemaObject.ConstraintTypes.FOREIGN_KEY : //检查外键约束
        PersistentStore store = core.mainTable.getRowStore(session);

        if (ArrayUtil.hasNull(data, core.refCols)) { //如果插入的数据在外键对应的
属性中有null
            if (core.matchType == OpTypes.MATCH_SIMPLE) { //MATCH_SIMPLE情况
下允许

                return;
            }

            if (core.refCols.length == 1) { //外键为单一属性的情况下允许
                return;
            }

            if (ArrayUtil.hasAllNull(data, core.refCols)) { //插入的数据在外键
对应的属性中全部为null
                return;
            }

            // core.matchType == OpTypes.MATCH_FULL
        } else if (core.mainIndex.existsParent(session, store, data,
                                                core.refCols)) {

            return;
        }

        throw getException(data);
    }
}

```

对于检查外键约束的情况，首先通过ArrayUtil.hasNull函数检查在插入记录中是否有外键对应的属性值为null，如果有则继续进行判断，当core.matchType为OpTypes.MATCH\_SIMPLE，或者外键为单一属性，或者插入的数据在外键对应的属性中全部为null时通过检查，否则不通过检查。如果对应的数据全部不是null，则调用IndexAVL类的existsParent函数检查是否存在外键对应的NodeAVL结点，如果不存在则会报错。

## 5. insert语句相关的触发器及其触发时间

通过观察Statement类的insertSingleRow函数，可以发现，一共有3中有关insert操作的触发事件：

- INSERT\_BEFORE\_ROW触发事件，在执行插入动作之前被触发
- INSERT\_AFTER\_ROW触发事件，在执行插入动作之后被触发

- INSERT\_AFTER触发事件，在所有INSERT\_AFTER\_ROW触发事件之后被触发

类似的，可以发现，delete事件和update事件也都分别对应于3种触发事件。以上一共有9种触发事件，定义在Trigger类中：

```
//org.hsqldb.Trigger

// type of trigger
int INSERT_AFTER      = 0;
int DELETE_AFTER      = 1;
int UPDATE_AFTER      = 2;
int INSERT_AFTER_ROW  = 3;
int DELETE_AFTER_ROW  = 4;
int UPDATE_AFTER_ROW  = 5;
int INSERT_BEFORE_ROW = 6;
int DELETE_BEFORE_ROW = 7;
int UPDATE_BEFORE_ROW = 8;
```