HSQLDB的并发控制机制分析

软件62 周展平 2016013253

问题思考

1. 寻找当前执行语句等待的session列表的机制

在TransactionManagerCommon类的方法中，setWaitedSessionsTPL函数对于当前语句cs等待的被持有写和读的锁的session进行查找，并保存在Session类的tempSet属性中。函数如下：

```
boolean setWaitedSessionsTPL(Session session, Statement cs) {

        session.tempSet.clear();

        if (cs == null) { //空语句没有冲突
            return true;
        }

        if (session.abortTransaction) { //abort的session无需分析
            return false;
        }

        if (cs.isCatalogLock(txModel)) {
            getTransactionSessions(session);
        }

        HsqlName[] nameList = cs.getTableNamesForWrite(); //cs涉及到的写的表
```

```
        for (int i = 0; i < nameList.length; i++) { //对于每一个表，查找对其加了锁的
session
            HsqlName name = nameList[i];

            if (name.schema == SqlInvariants.SYSTEM_SCHEMA_HSQLNAME) {
                continue;
            }

            Session holder = (Session) tableWriteLocks.get(name); //最多一个session加
了写锁

            if (holder != null && holder != session) {
                session.tempSet.add(holder);
            }
            //查找所有对这张表加了读锁的session
            Iterator it = tableReadLocks.get(name);

            while (it.hasNext()) {
                holder = (Session) it.next();

                if (holder != session) {
                    session.tempSet.add(holder);
                }
            }
        }

        nameList = cs.getTableNamesForRead(); //cs涉及到的读的表

        if (txModel == TransactionManager.MVLOCKS && session.isReadOnly()) {
            if (nameList.length > 0) {
                nameList = catalogNameList;
            }
        }

        for (int i = 0; i < nameList.length; i++) {
            HsqlName name = nameList[i];

            if (name.schema == SqlInvariants.SYSTEM_SCHEMA_HSQLNAME) {
                continue;
            }

            Session holder = (Session) tableWriteLocks.get(name); //最多一个session加
了写锁

            if (holder != null && holder != session) {
                session.tempSet.add(holder);
            }
        }
```

```
        //不需要寻找加了读锁的session，因为读与读不冲突
        if (session.tempSet.isEmpty()) {
            return true;
        }

        if (checkDeadlock(session, session.tempSet)) { //检查是否出现死锁
            return true;
        }

        session.tempSet.clear();

        session.abortTransaction = true; //由于锁的原因当前语句无法执行

        return false;
    }
```

对于当前语句cs涉及到的写入的表，需要查找表的被加上写和读的锁的session，其中至多有一个session加上写锁，而可能有很多session持有读锁。

对于当前语句cs涉及到的读出的表，需要查找表的被加上写的锁，而不需要查找被加上的读的锁，因为读与读之间不发生冲突。

查询被加上写锁的session，首先调用Statement类的getTableNamesForWrite方法查找所有写入的表的名字，然后在tableWriteLocks这个hash表中通过表名查找到对应的session。查询被加上读锁的session，需要在tableReadLocks这个multivalue的hash表中通过表名查找到对应的session链表，然后依次遍历得到所有session。

2.当前执行语句对要访问的表上锁的机制

这里以TransactionManager2PL类为例。在beginAction函数中首先调用setWaitedSessionsTPL方法获取所有的锁冲突的session，之后如果可以执行当前语句，则进行上锁操作：

```
/**
     * add session to the end of queue when a transaction starts
     * (depending on isolation mode)
     */
    public void beginAction(Session session, Statement cs) {

        writeLock.lock();

        try {
            if (hasExpired) {
                session.redoAction = true;

                return;
            }
```

```
            cs = updateCurrentStatement(session, cs);

            if (cs == null) {
                return;
            }

            boolean canProceed = setWaitedSessionsTPL(session, cs); //查看是否出现锁冲
突

            if (canProceed) { //当前语句可以执行
                session.isPreTransaction = true;

                if (session.tempSet.isEmpty()) { //没有锁冲突
                    lockTablesTPL(session, cs); //上锁

                    // we don't set other sessions that would now be waiting for
this one too
                    // next lock release will do it
                } else {
                    setWaitingSessionTPL(session); //等待其他session释放锁
                }
            }
        } finally {
            writeLock.unlock();
        }
    }
```

其中上锁操作的进行是通过lockTablesTPL函数完成的，如下：

```
void lockTablesTPL(Session session, Statement cs) {

        if (cs == null || session.abortTransaction) {
            return;
        }

        HsqlName[] nameList = cs.getTableNamesForWrite();

        for (int i = 0; i < nameList.length; i++) {
            HsqlName name = nameList[i];

            if (name.schema == SqlInvariants.SYSTEM_SCHEMA_HSQLNAME) { //跳过数据库的系
统表
                continue;
            }

            tableWriteLocks.put(name, session); //加上写锁
        }
```

```
        nameList = cs.getTableNamesForRead();

        if (txModel == TransactionManager.MVLOCKS && session.isReadOnly()) {
            if (nameList.length > 0) {
                nameList = catalogNameList;
            }
        }

        for (int i = 0; i < nameList.length; i++) {
            HsqlName name = nameList[i];

            if (name.schema == SqlInvariants.SYSTEM_SCHEMA_HSQLNAME) { //跳过数据库的系
统表
                continue;
            }

            tableReadLocks.put(name, session); //加上读锁
        }
    }
```

实现的机制是，tableWriteLocks和 tableReadLocks是hash表，分别维护了写锁和读锁的从表名到session的映射关系。上锁的操作就是在hash表中通过put方法添加对应关系。

3. 三种锁协议的实现

三种锁协议分别由TransactionManager2PL（two phase lock），TransactionManagerMVCC（multi-version concurrency control），TransactionManagerMV2PL（multi-version two phase lock）这3个类实现，它们都继承了TransactionManagerCommon类。因此，这三个类对外的接口大致相同，区别在于内部的实现。

（1）TransactionManager2PL（Two Phase Locking）

TransactionManager2PL的实现的主要思想是抽象了读锁与写锁两个概念，注意锁是针对整张表的。具体来说，利用Session类的tableReadLocks和tableWriteLocks属性记录持有锁的session，在tempSet属性中存储与该session出现冲突的其他session。另外，Session类的waitingSessions和waitedSessions属性分别记录了等待的与被等待的所有session。在执行过程中，首先判断是否出现锁冲突，如果出现冲突则更新waitingSessions和waitedSessions属性；否则就可以获得相应的锁，更新tableReadLocks和tableWriteLocks属性。

另外，处理死锁问题的是通过TransactionManagerCommon类的checkDeadlock方法进行的，实现的原理就是检查waitingSessions属性中是否有当前session等待的session，如果有则出现了循环依赖，意味着出现了死锁。具体机制见第五个问题。

（2）TransactionManagerMVCC（multi-version concurrency control）

MVCC的核心思想是对于每一个transaction，保存对应的一个版本，并保存其时间戳。在TransactionManagerMVCC类中，committedTransactions属性记录了该transaction的RowAction对象，committedTransactionTimestamps属性则在对应位置记录了该transaction的时间戳，catalogWriteSession属性记录了当前进行对schema进行写操作的session。另外，在基类TransactionManagerCommon类中的liveTransactionTimestamps属性，记录了当前开始进行执行的transaction的时间戳。

另外，Session类的latch属性是CountUpDownLatch类的实例，用来设置session何时开始执行。

当一个transaction开始时，在beginTransaction或beginTransactionResume函数中，其时间戳被加入liveTransactionTimestamps的队尾。在transaction结束时，在endTransaction函数中，其时间戳被从liveTransactionTimestamps队列中移除，同时调用mergeExpiredTransactions函数将所有已经过期的保存版本从committedTransactions和committedTransactionTimestamps中移除。所谓过期，指的是一个保存的transaction的timestamp小于liveTransactionTimestamps中所有的timestamp，这与MVCC的理论完全对应。

（3）TransactionManagerMV2PL（Two Phase Locking with Snapshot Isolation）

TransactionManagerMV2PL类综合了以上两个类的实现。与TransactionManager2PL类似，在执行transaction时需要对整个table加上读锁和写锁；与TransactionManagerMVCC类似，committedTransactions属性和committedTransactionTimestamps属性起到了保存多个版本的作用。

4. 不同隔离级别中的隔离在三种锁协议下的保证

SessionInterface类中定义了4个静态常量，分别标识4种标准的隔离级别：

```
int TX_READ_UNCOMMITTED = 1;
int TX_READ_COMMITTED   = 2;
int TX_REPEATABLE_READ  = 4;
int TX_SERIALIZABLE     = 8;
```

在HSQLDB中，在一些锁协议下多种隔离级别会被合并为一种。

（1）TransactionManager2PL（Two Phase Locking）

在此协议下，READ UNCOMMITTED级别最低，READ COMMITTED是默认的隔离级别，REPEATABLE READ被提升为SERIALIZABLE级别，也就是说REPEATABLE READ的session会按照SERIALIZABLE的隔离级别来执行。

（2）TransactionManagerMVCC（multi-version concurrency control）

在此协议下，READ COMMITTED隔离级别被实现为MVCC策略的READ CONSISTENCY隔离级别，READ UNCOMMITTED隔离级别被提升为READ COMMITTED，而REPEATABLE READ 和SERIALIZABLE则被实现为SNAPSHOT ISOLATION隔离级别。

（3）TransactionManagerMV2PL（Two Phase Locking with Snapshot Isolation）

此协议与2PL协议的隔离级别类似，在two phase lock的基础上增加了SNAPSHOT ISOLATION的相关策略。

下面从源码的角度分析以上各个隔离级别的划分。以下是RowAction类的canRead函数中的片段：

```
switch (session.isolationLevel) {

                case SessionInterface.TX_READ_UNCOMMITTED :
                    threshold = Long.MAX_VALUE;
                    break;

                case SessionInterface.TX_READ_COMMITTED :
                    threshold = session.actionTimestamp;
                    break;

                case SessionInterface.TX_REPEATABLE_READ :
                case SessionInterface.TX_SERIALIZABLE :
                default :
                    threshold = session.transactionTimestamp;
                    break;
            }
```

其中的threshold变量是根据session的隔离级别确定的该session最早能够访问该行数据的时间戳。对于READ UNCOMMITTED协议，将threshold设为Long类型的最大值意味着在此隔离级别下访问不受限制；对于READ COMMITTED隔离级别，threshold就是commit的时间；而REPEATABLE READ被提升为SERIALIZABLE级别，threshold设为transaction结束的时间。

以下是TransactionManagerCommon类的endActionTPL方法的片段：

```
void endActionTPL(Session session) {

        if (session.isolationLevel == SessionInterface.TX_REPEATABLE_READ
                || session.isolationLevel
                   == SessionInterface.TX_SERIALIZABLE) {
            return;
        }

        ...
}
```

当一个action完成之后，如果隔离级别为REPEATABLE READ或SERIALIZABLE，则函数直接返回，不会进行下面的操作。而下面的主要操作如下：

```
writeLock.lock();

try {
    unlockReadTablesTPL(session, readLocks); //释放读锁
```

```
    final int waitingCount = session.waitingSessions.size();

    if (waitingCount == 0) {
        return;
    }

    boolean canUnlock = false;

    // if write lock was used for read lock
    for (int i = 0; i < readLocks.length; i++) {
        if (tableWriteLocks.get(readLocks[i]) != session) {
            canUnlock = true;

            break;
        }
    }

    if (!canUnlock) {
        return;
    }

    canUnlock = false;

    for (int i = 0; i < waitingCount; i++) {
        Session current = (Session) session.waitingSessions.get(i);

        if (current.abortTransaction) {
            canUnlock = true;

            break;
        }

        Statement currentStatement =
            current.sessionContext.currentStatement;

        if (currentStatement == null) {
            canUnlock = true;

            break;
        }

        if (ArrayUtil.containsAny(
            readLocks, currentStatement.getTableNamesForWrite())) {
            canUnlock = true;

            break;
        }
    }
```

```
    if (!canUnlock) {
        return;
    }

    resetLocks(session);  //释放所有锁
    resetLatchesMidTransaction(session); //更新latch
} finally {
    writeLock.unlock();
}
```

可以看出，以上代码的功能就是释放可以被释放的读写锁，同时更新各个session的latch。因此，HSQLDB在REPEATABLE READ或SERIALIZABLE的隔离级别下不会再action结束后释放可以释放的锁，目的是保证不会出现不可重复读和幻读的问题。

5. 死锁检测的实现

在TransactionManagerCommon类中有两个checkDeadlock函数，第一个函数是检查当前session与一个session集合中的每一个session之间是否形成死锁；第二个函数是第一个函数的特例，检查当前session与另一个session是否形成死锁。

```
boolean checkDeadlock(Session session, OrderedHashSet newWaits) { //newWaits是当前session依赖的所有session
        int size = session.waitingSessions.size(); //waitingSessions是依赖于当前session的所有session

        for (int i = 0; i < size; i++) { //检查newWaits集合中的每一个session是否与当前session形成死锁
            Session current = (Session) session.waitingSessions.get(i);

            if (newWaits.contains(current)) { //出现直接循环（2元）
                return false;
            }

            if (!checkDeadlock(current, newWaits)) { //递归地检查依赖图中是否出现环
                return false;
            }
        }

        return true; //没有环，也就不存在死锁
    }

    boolean checkDeadlock(Session session, Session other) { //上一个函数的特例，只检查两个session之间是否有死锁
        int size = session.waitingSessions.size();
```

```
        for (int i = 0; i < size; i++) {
            Session current = (Session) session.waitingSessions.get(i);

            if (current == other) { //2元环
                return false;
            }

            if (!checkDeadlock(current, other)) { //递归检查
                return false;
            }
        }

        return true; //没有环，也就不存在死锁
    }
```