HSQLDB的存储机制分析

软件62 周展平 2016013253

实验要求

一、分别创建三种表

过程略。

二、对比三种表的打开过程

三种表的不同之处主要在于对于数据的存储方式不同，因此这里主要分析三种表在打开过程中的索引建立过程：

- memory table:

  读取.script文件中记录的语句，执行之后建立起整张表的索引、记录。

- cached table：

  读取.data文件中的索引内容，在内存中建立索引。

- text table：

  读取指定的外存中的存储文件的数据。

三、数据存储

1. 文件与数据存储

- memory table：在.script文件中存储SQL语句，不存储实际数据
- cached table：在.log文件中存储SQL语句，重新打开后会整合到.script文件中；数据存储在.data文件中
- text table：在.log文件中存储SQL语句，重新打开后会整合到.script文件中；数据存储在用户指定的文件中

具体分析在问题思考部分的第五个问题中。

2. 数据存储格式

- .data文件：

  对于索引结点，通过分析NodeAVL.write方法可以看出，对于每一个索引结点，文件中存储了iBalance，iLeft，iRight和iParent四个属性值，这样就相当于将整个AVL树存储了起来。

  对于记录结点来说，我们可以分析RowOutputBase类的几个writeData方法：

```
/**
     *  This method is called to write data for a table row.
     */
    public void writeData(Row row, Type[] types) {
        writeData(types.length, types, row.getData(), null, null);
    }
```

  此方法是将单行记录保存的入口方法，其中调用了另外一个writeData方法：

```
    /**
     *  This method is called directly to write data for a delete statement.
     */
    public void writeData(int l, Type[] types, Object[] data,
                          HashMappedList cols, int[] primaryKeys) {

        boolean hasPK = primaryKeys != null && primaryKeys.length != 0;
        int     limit = hasPK ? primaryKeys.length
                              : l;

        for (int i = 0; i < limit; i++) {
            int    j = hasPK ? primaryKeys[i]
                             : i;
            Object o = data[j];
            Type   t = types[j];

            if (cols != null) {
```

```
                ColumnSchema col = (ColumnSchema) cols.get(j);

                writeFieldPrefix();
                writeString(col.getName().statementName);
            }

            writeData(o, t);
        }
    }
```

此方法会依次按照每个属性将属性名、以及记录的属性值写入文件，其中写记录调用了第三个writeData函数：

```
    public void writeData(Object o, Type t) {

        if (o == null) {
            writeNull(t);

            return;
        }

        writeFieldType(t);

        switch (t.typeCode) {

            case Types.SQL_ALL_TYPES :
                break;

            case Types.SQL_CHAR :
            case Types.SQL_VARCHAR :
                writeChar((String) o, t);
                break;

            case Types.TINYINT :
            case Types.SQL_SMALLINT :
                writeSmallint((Number) o);
                break;

            case Types.SQL_INTEGER :
                writeInteger((Number) o);
                break;

            case Types.SQL_BIGINT :
                writeBigint((Number) o);
                break;

            case Types.SQL_REAL :
            case Types.SQL_FLOAT :
```

```
            case Types.SQL_DOUBLE :
                writeReal((Double) o);
                break;

            case Types.SQL_NUMERIC :
            case Types.SQL_DECIMAL :
                writeDecimal((BigDecimal) o, t);
                break;

            case Types.SQL_BOOLEAN :
                writeBoolean((Boolean) o);
                break;

            case Types.SQL_DATE :
                writeDate((TimestampData) o, t);
                break;

            case Types.SQL_TIME :
            case Types.SQL_TIME_WITH_TIME_ZONE :
                writeTime((TimeData) o, t);
                break;

            case Types.SQL_TIMESTAMP :
            case Types.SQL_TIMESTAMP_WITH_TIME_ZONE :
                writeTimestamp((TimestampData) o, t);
                break;

            case Types.SQL_INTERVAL_YEAR :
            case Types.SQL_INTERVAL_YEAR_TO_MONTH :
            case Types.SQL_INTERVAL_MONTH :
                writeYearMonthInterval((IntervalMonthData) o, t);
                break;

            case Types.SQL_INTERVAL_DAY :
            case Types.SQL_INTERVAL_DAY_TO_HOUR :
            case Types.SQL_INTERVAL_DAY_TO_MINUTE :
            case Types.SQL_INTERVAL_DAY_TO_SECOND :
            case Types.SQL_INTERVAL_HOUR :
            case Types.SQL_INTERVAL_HOUR_TO_MINUTE :
            case Types.SQL_INTERVAL_HOUR_TO_SECOND :
            case Types.SQL_INTERVAL_MINUTE :
            case Types.SQL_INTERVAL_MINUTE_TO_SECOND :
            case Types.SQL_INTERVAL_SECOND :
                writeDaySecondInterval((IntervalSecondData) o, t);
                break;

            case Types.OTHER :
                writeOther((JavaObjectData) o);
```

```
                break;

            case Types.SQL_BLOB :
                writeBlob((BlobData) o, t);
                break;

            case Types.SQL_CLOB :
                writeClob((ClobData) o, t);
                break;

            case Types.SQL_ARRAY :
                writeArray((Object[]) o, t);
                break;

            case Types.SQL_GUID :
                writeUUID((BinaryData) o);
                break;

            case Types.SQL_BINARY :
            case Types.SQL_VARBINARY :
                writeBinary((BinaryData) o);
                break;

            case Types.SQL_BIT :
            case Types.SQL_BIT_VARYING :
                writeBit((BinaryData) o);
                break;

            default :
                throw Error.runtimeError(ErrorCode.U_S0500,
                                         "RowOutputBase - "
                                         + t.getNameString());
        }
    }
```

此函数根据属性的类型调用不同的写函数将属性的类型与值依次写入文件。至此，单行记录的写入就完成了。

- text table 指定的文件：单行数据对应于用户指定的文件中的一行，每行按照表的属性顺序依次存储了各个属性的值，它们被用户指定的delimiter所分隔

四、数据访问

访问的时候首先通过索引检索对应的Row对象，如果已经在内存中则直接获得，否则需要从文件中读取。cached table与text table会将读取的数据加入缓存中。

具体的细节见问题思考部分。

五、缓存管理机制

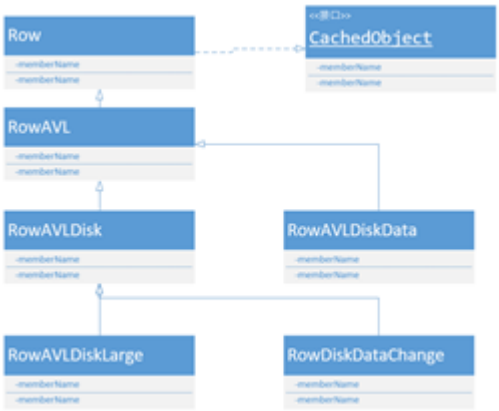
见问题思考部分的第四个问题。

问题思考

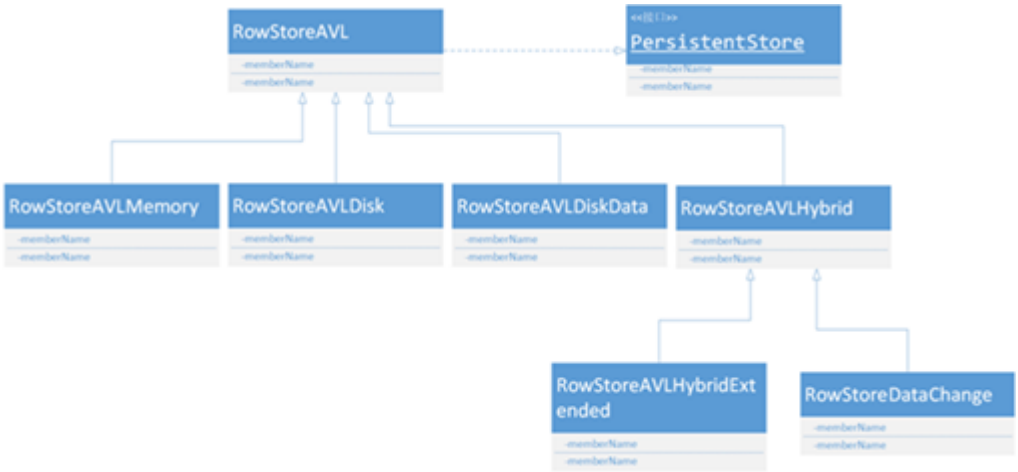
一、三种表类型的表结构与单行数据在HSQLDB中的实现

1.单行数据的实现

- 单行数据相关的类图：



- 持久化存储相关类图：



- 插入记录的过程

  向table中插入一个记录时，需要调用org.hsqldb.Table类的insertSingleRow方法。其中通过org.hsqldb.persist.Logger.newStore方法，根据table的类型创建不同的org.hsqldb.persist.PersistentStore类的子类实例，之后调用org.hsqldb.persist.RowStoreAVL.getNewCachedObject方法创建了一个对应于一个记录(单行数据)的对象，不同的table类型创建不同类的实例。

- memory table

调用的是org.hsqldb.persist.RowStoreAVLMemory类的getNewCachedObject方法，创建的是一个org.hsqldb.RowAVL类的实例。

- cached table

  调用的是org.hsqldb.persist.RowStoreAVLDisk类的getNewCachedObject方法，如果记录很大，会创建一个org.hsqldb.RowAVLDiskLarge类的实例，否则创建一个org.hsqldb.RowAVLDisk类的实例。

- text table

  调用的是org.hsqldb.persist.RowStoreAVLDiskData类的getNewCachedObject方法，创建的是一个org.hsqldb.RowAVLDiskData类的实例。

2. 表结构的实现

memory table与cached table类型的表结构对应的类都是org.hsqldb.Table类，text table则对应于org.hsqldb.Table类的子类org.hsqldb.TextTable类，其中的type属性表示了表的不同类型。在创建新表时，org.hsqldb.TableWorks类负责为Table类的实例增加索引，索引对应于org.hsqldb.index.Index类。之后，org.hsqldb.Database.schemaManager类调用addSchemaObject方法将新添加的索引信息保存。如果创建过程中没有异常，则org.hsqldb.Database.logger类调用writeOtherStatement方法写入日志。

二、内外存数据交换的实现

HSQLDB将文件读写封装了起来，通过org.hsqldb.persist.Logger管理日志文件，通过org.hsqldb.scriptio包中的类管理.script文件的读写，通过org.hsqldb.rowio包中的各个类管理二进制格式与文本格式的文件读写。

org.hsqldb.persist.Cache类对应于缓存，其中对于缓存的替换操作则是通过org.hsqldb.persist.DataFileCache类或org.hsqldb.persist.TextCache类管理的。

cached table通过org.hsqldb.persist.DataFileCache类管理内存与外存的数据交换，其中的get方法负责提供当行数据，如果数据存在于内存中则直接找到并返回，否则需要借助getFromFile方法，调用readObject方法利用org.hsqldb.rowio包中接口RowInputInterface的方法，将一行数据从.data文件读入内存，然后返回。对于写回外存，调用org.hsqldb.persist.DataFileCache类的saveRowNoLock等方法，利用org.hsqldb.rowio包中接口RowOutputInterface的方法写入文件。

text table通过org.hsqldb.persist.TextCache类管理内存与外存的数据交换，这个类继承自org.hsqldb.persist.DataFileCache类，因此在功能上大致相同。

三、添加主键约束的实现

添加主键约束需要利用org.hsqldb.TableWorks类的addPrimaryKey方法，如下：

```
void addPrimaryKey(Constraint constraint) {

    checkModifyTable(true);
```

```
    if (table.hasPrimaryKey()) { // 检查是否已经存在主键
        throw Error.error(ErrorCode.X_42532);
    }

    database.schemaManager.checkSchemaObjectNotExists(
        constraint.getName()); // 检查要添加的主键名称是否已经存在

    Table tn = table.moveDefinition(session, table.tableType, null,
                                    constraint, null, -1, 0, emptySet,
                                    emptySet); // 新建一个表并添加主键约束

    moveData(table, tn, -1, 0); // 将数据迁移到新的表中
    // 更新表的信息
    table = tn;

    database.schemaManager.addSchemaObject(constraint);
    setNewTableInSchema(table);
    updateConstraints(table, emptySet);
    database.schemaManager.recompileDependentObjects(table);
}
```

其中，涉及到主键设定的是org.hsqldb.table.moveDefinition方法，该方法基于旧表创建了一个新表，依次添加原有的属性并设置不为主键，之后调用createPrimaryKey方法将新的主键属性添加进去并设置为主键。在createPrimaryKey方法中，最主要的任务是调用setPrimaryKey方法设置主键，同时调用getNewPrimaryIndex和addIndexStructure方法建立主键的索引。

四、缓存的替换机制与容量维护

org.hsqldb.map.BaseHashMap是HSQLDB所有的哈希表的基础类，缓存类org.hsqldb.persist.Cache是它的一个子类。HSQLDB采用了**Least Recently Used (LRU)**的置换算法，在BaseHashMap类中，有一些关于缓存维护规则的属性：

```
protected int       accessMin; //最小的访问序号
protected int       accessCount; //最后一次的访问序号
protected int[]     accessTable; //存储对于每一个位置的访问序号

final float                 loadFactor; //装载因子
final int                   initialCapacity; //初始最大容量
int                         threshold; //阈值，用来动态调整容量
protected int               maxCapacity; //当前最大容量
protected boolean           minimizeOnEmpty; //存储为空时是否最小化
```

DataFileCache通过调用get方法获取单行数据，如果在缓存中则直接调用org.hsqldb.persist.Cache的get方法，否则要调用getFromFile方法，先从文件中读取数据，然后通过put方法加入缓存中。

get方法会将被访问的row的访问序号更新，同时检查是否需要将所有的序号全部重新更新（当最大访问序号大于ACCESS_MAX时）。

put方法首先调用prepare Put方法，如下：

```
boolean preparePut(int storageSize) {

        boolean exceedsCount = size() + reserveCount >= capacity; //是否超过最大数量
        boolean exceedsSize  = storageSize + cacheBytesLength > bytesCapacity; //是否超过最大字节数

        if (exceedsCount || exceedsSize) { //第一次尝试
            cleanUp(false);  //清理缓存

            exceedsCount = size() + reserveCount >= capacity;
            exceedsSize  = storageSize + cacheBytesLength > bytesCapacity;

            if (exceedsCount || exceedsSize) { //第二次尝试
                clearUnchanged(); //清理不需要保留在内存中且未被修改的单行数据
            } else {
                return true;
            }

            exceedsCount = size() + reserveCount >= capacity;
            exceedsSize  = storageSize + cacheBytesLength > bytesCapacity;

            if (exceedsCount || exceedsSize) { //第三次尝试
                cleanUp(true); //最后的办法
            } else {
                return true;
            }

            exceedsCount = size() + reserveCount >= capacity;
            exceedsSize  = storageSize + cacheBytesLength > bytesCapacity;

            if (exceedsCount) { //始终无法满足数据条目数量要求
                dataFileCache.logInfoEvent(
                    "dataFileCache CACHE ROWS limit reached");
            }

            if (exceedsSize) { //始终无法满足字节数要求
                dataFileCache.logInfoEvent(
                    "dataFileCache CACHE SIZE limit reached");
            }

            if (exceedsCount || exceedsSize) {
                return false;
            }
```

```
        }

        return true;
    }
```

对于缓存的清理分为3个级别，先尝试影响最小的清理方法，如果仍然不能满足要求则继续进行影响更大的清理方法。其中发挥作用的是cleanUp函数与clearUnchange函数。

cleanUp函数如下：

```
private void cleanUp(boolean all) {

        updateAccessCounts();

        int savecount    = 0;
        int removeCount  = size() / 2;
        int accessTarget = all ? accessCount + 1
                               : getAccessCountCeiling(removeCount,
                                   removeCount / 8);
        int accessMid = all ? accessCount + 1
                            : (accessMin + accessTarget) / 2;

        objectIterator.reset();

        for (; objectIterator.hasNext(); ) {
            CachedObject row = (CachedObject) objectIterator.next();
            int          currentAccessCount = objectIterator.getAccessCount();
            boolean oldRow = currentAccessCount < accessTarget
                             && !row.isKeepInMemory(); // 缓存较旧，应该被替换
            boolean newRow = row.isNew()
                             && row.getStorageSize()
                                >= DataFileCache.initIOBufferSize;
            boolean saveRow = row.hasChanged() && (oldRow || newRow); //缓存需要保存更
新

            objectIterator.setAccessCount(accessTarget);

            synchronized (row) {
                if (saveRow) {
                    rowTable[savecount++] = row; // 将需要保存更新的数据保存到rowTable中
                }

                if (oldRow) { // 移除过期缓存
                    row.setInMemory(false);
                    objectIterator.remove();

                    cacheBytesLength -= row.getStorageSize();
                }
```

```
            }

            if (savecount == rowTable.length) {
                saveRows(savecount); // rowTable已满，需要批量保存更新

                savecount = 0;
            }
        }

        saveRows(savecount);
        setAccessCountFloor(accessTarget);

        accessCount++;
    }
```

根据参数all的真假情况，首先设置了accessTarget，也即衡量新旧缓存的标准值，小于这个值的访问序号被认为是**过期**的缓存。之后对于缓存中的数据依次遍历，oldRow标识了单行数据是否已经过期（accessCount < accessTarget），saveRow标识了数据是否需要保存更新。之后，如果需要替换缓存数据且需要保存更新，则将其保存到rowTable数组中，当数组装满后调用saveRows函数进行批量的全部更新。

clearUnchanged函数如下：

```
void clearUnchanged() {

        objectIterator.reset();

        for (; objectIterator.hasNext(); ) {
            CachedObject row = (CachedObject) objectIterator.next();

            synchronized (row) {
                if (!row.isKeepInMemory() && !row.hasChanged()) {
                    row.setInMemory(false);
                    objectIterator.remove();

                    cacheBytesLength -= row.getStorageSize();
                }
            }
        }
    }
```

该函数的替换策略是，将所有不需要保存在缓存中且没有被更改过的数据进行清除。

五、cached table与text table增删改数据时外存文件的变化

- cached table

进行增删改数据时，.log文件保存了执行的SQL语句，.data文件保存了必要的数据。结束数据库server进程并重新启动server进程，.log文件中的记录被清空。

在org.hsqldb.persist.Logger类中出现了dataFileExtension属性：

```
public static final String dataFileExtension       = ".data";
```

此后，DataFileCache.dataFileName和org.hsqldb.persist.Log类都用到了这个属性值。DataFileCache类负责维护.data文件。

- text table

  首先创建一个text table并指定文件：

  ```
  create text table person (id char(8), name varchar(8), age int, undergraduate
  boolean,
  birth_year numeric(4,0), birthdate date, primary key (id));

  set table public.person source "person;fs=|"
  ```

  发现.log文件中记录了以上两条SQL语句。之后向表中添加一条数据：

  ```
  insert into person values ('00000001', 'student1', 22, true, 1995, '1995-01-01');
  ```

  发现在data/person文件中出现了我们添加的数据：

  ```
  00000001|student1|22|true|1995|1995-01-01
  ```

  最后，执行删除操作：

  ```
  delete from person
  ```

  data/person文件中的数据被清空。

  结束数据库server进程并重新启动server进程，.log文件中的记录被清空。