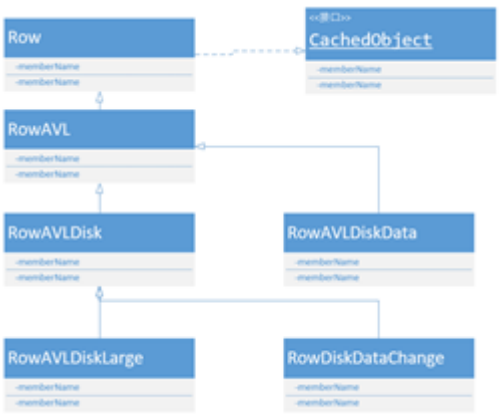
HSQLDB的存储机制分析

软件62 周展平 2016013253

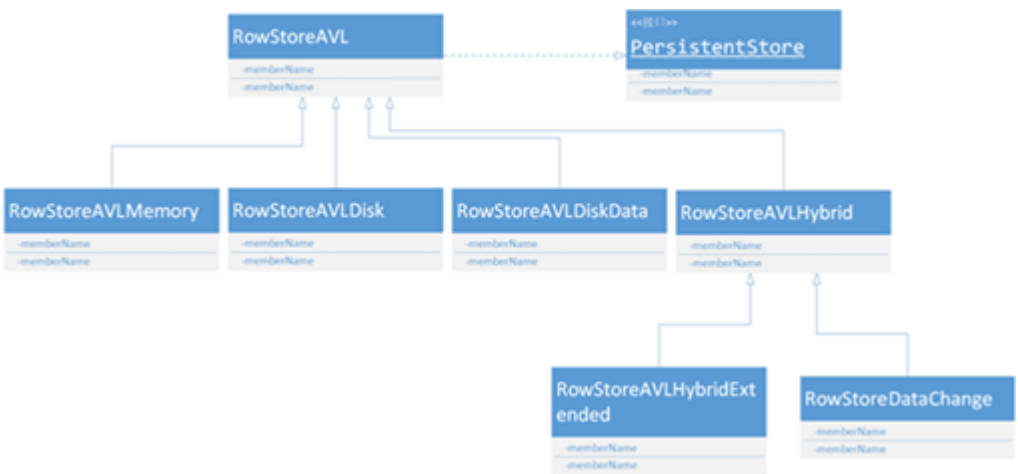
一、三种表类型的表结构与单行数据在HSQLDB中的实现

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preparePut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。