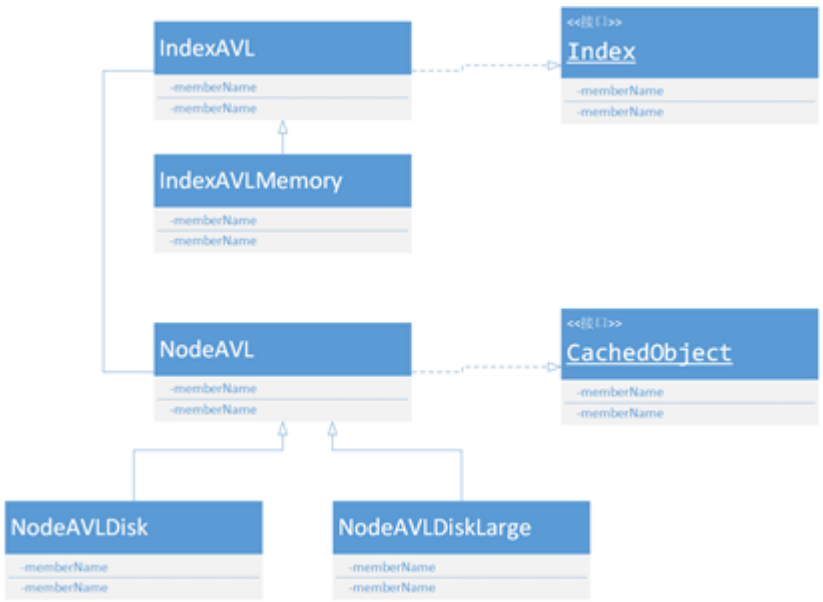
HSQLDB的索引机制分析

软件62 周展平 2016013253

实验要求

一、索引的具体实现机制

相关的包为org.hsqldb.index，索引和AVL树的相关类图如下：



其中，Index接口是索引操作的规约，IndexAVL类对应于索引。cached table与text table使用IndexAVL类的实现索引，而memory table使用IndexAVL类的子类IndexAVLMemory类的实现索引。索引通过AVL树来实现，其中的结点的基类为NodeAVL，代表AVL树中的结点。实例化的时候，选择它的两个子类NodeAVLDisk与NodeAVLDiskLarge之一，后者对应于较大的数据库。每一个NodeAVL类的实例与一个RowAVL类的实例相对应，其中NodeAVLDisk类对应于RowAVLDisk类，而NodeAVLDiskLarge类对应于RowAVLDiskLarge类。

可以通过IndexAVL类的getAccessor方法找到根节点。NodeAVL结点维护了左子树、右子树、父亲结点的信息。通过NodeAVL中的row属性访问结点对应的Row类的实例。nNext属性建立了所有结点在一维上的全序关系。

二、索引的创建时机、增删改记录时索引的变化

在TableBase.createPrimaryIndex函数中设置断点，启动数据库，从堆栈调用中可以看到调用时机：server运行后，在run方法中调用了openDatabases方法调用Database.reopen方法打开这个数据库，之后DatabaseInformation.newDatabaseInformation函数创建了数据库信息对象，在其中经过一系列与数据库信息有关的函数调用，最终调用了Table.createPrimaryKeyConstraint函数与Table.createPrimaryKey函数来创建主键，其中调用了基类TableBase的createPrimaryIndex方法创建主键索引。

增加记录时，Table.insertSingleRow方法将insert任务交给session，session通过TransactionManager调用TransactionManager2PL.addInsertAction方法，最终由RowStoreAVL类的indexRow方法执行具体操作。indexRow方法在表的每一个索引中都要调用IndexAVL.insert方法插入结点。insert方法在本质上就是一个AVL树的插入算法。

删除记录时，调用栈与增加记录时十分类似，区别在于调用的是各个类的删除相关的方法，最终由RowStoreAVL类的delete方法执行具体操作。delete方法在表的每一个索引中都要调用IndexAVL.delete方法删除结点。delete方法在本质上就是一个AVL树的删除算法。

更改记录时，StatementDML类调用update函数，其中首先执行删除记录的操作，之后执行增加记录的操作。因此，更改记录就是删除和增加记录的串联。

三、索引与记录节点关联

IndexAVL类的getAccessor方法可以获取索引对应的AVL树的根节点，其中主要的函数是RowStoreAVLDisk类的getAccessor方法：

```
// org.hsqldb.persist.RowStoreAVLDisk

public CachedObject getAccessor(Index key) {

        int position = key.getPosition(); // 获取索引在RowStoreAVLDisk中的position

        if (position >= accessorList.length) {
            throw Error.runtimeError(ErrorCode.U_S0500, "RowStoreAVL");
        }
```

```
        NodeAVL node = (NodeAVL) accessorList[position]; // 查找到根节点

        if (node == null) {
            return null;
        }

        RowAVL row = (RowAVL) get(node.getRow(this), false);

        node                              = row.getNode(key.getPosition());
        accessorList[key.getPosition()] = node;

        return node;
    }
```

RowStoreAVLDisk.accessorList数组中存放了多个索引的根节点，按照position访问即可找到。

四、索引在外存中的存储及索引的内外存交换机制

1. 外存中的索引

NodeAVLDisk类的write方法将结点的信息保存到.data文件中，如下：

```
public void write(RowOutputInterface out) {

    out.writeInt(iBalance);
    out.writeInt((iLeft == NO_POS) ? 0
    : iLeft);
    out.writeInt((iRight == NO_POS) ? 0
    : iRight);
    out.writeInt((iParent == NO_POS) ? 0
    : iParent);
}

public void write(RowOutputInterface out, LongLookup lookup) {

    out.writeInt(iBalance);
    out.writeInt(getTranslatePointer(iLeft, lookup));
    out.writeInt(getTranslatePointer(iRight, lookup));
    out.writeInt(getTranslatePointer(iParent, lookup));
}
```

可以看出，外存中的节点存储格式为依次存储iBalance，iLeft，iRight和iParent。

2. 索引的内外存交换机制

打开数据库时，Logger.open方法调用Log.open方法，其中的Log.processScript方法会执行.script文件中所有的操作，包括创建表以及索引的部分，其中Table.setIndexRoots方法初始化了RowStoreAVLDisk.accessorList数组，这个过程中需要创建NodeAVL并且将其与对应的记录RowAVL关联。此时会调用DataFileCache.get方法从缓存中查找，不存在则会调用DataFileCache.getFromFile方法从文件中读取数据。

在访问结点时，最终要借助NodeAVLDisk.findNode函数，如下：

```
private NodeAVLDisk findNode(PersistentStore store) {

        if (row.isInMemory()) { //记录已经在内存中，直接返回结点
            return this;
        }

        RowAVLDisk r = (RowAVLDisk) store.get(row.getPos(), false); //从外存中读取记录

        if (r == null) {
            String tableName = "";

            if (row.getTable().getTableType() == Table.CACHED_TABLE) {
                tableName = ((Table) row.getTable()).getName().name;
            }

            store.getCache().logSevereEvent(tableName + " NodeAVLDisk "
                                            + row.getPos(), null); //添加日志

            return this;
        }

        return (NodeAVLDisk) r.getNode(iId);
    }
```

因此，索引的内外存交换本质上是.script文件的读取与索引中结点对应的记录的内外存交换。

五、缓存的实现机制

见“HSQLDB的存储机制分析”的问题思考部分第四题。

问题思考

一、索引读取的细节

1.索引信息与记录

RowAVLDisk的构造函数如下：

```
public RowAVLDisk(PersistentStore store,
                  RowInputInterface in) throws IOException {

        super(store.getTable(), (Object[]) null);

        position    = in.getFilePosition();
        storageSize = in.getSize();

        int indexcount = store.getAccessorKeys().length;

        nPrimaryNode = new NodeAVLDisk(this, in, 0); //读取第一个结点

        NodeAVL n = nPrimaryNode;

        for (int i = 1; i < indexcount; i++) { //依次读取结点
            n.nNext = new NodeAVLDisk(this, in, i);
            n       = n.nNext;
        }

        rowData = in.readData(table.getColumnTypes()); //读取
    }
```

因此，读取RowAVLDisk时必然将与其相关的所有NodeAVLDisk读取进了内存。

另外一方面，NodeAVLDisk的构造函数中从一个RowInputInterface的实例in中读取了iBalance, iLeft, iRight和iParent属性，但是并不要求读取对应的RowAVLDisk的数据，在调用getNode方法时才会读取数据。

2. 主索引和次级索引

二、索引与记录结点

见实验要求部分第三题。

三、索引结点与数据结点的保存更新

当记录需要更新时，会一并将索引结点进行更新。其中可能涉及到索引更新的有两种函数调用。

1. RowAVLDisk.write

```
/**
     * Used exclusively by Cache to save the row to disk. New implementation in
     * 1.7.2 writes out only the Node data if the table row data has not
     * changed. This situation accounts for the majority of invocations as for
     * each row deleted or inserted, the Nodes for several other rows will
     * change.
```

```
     */
    public void write(RowOutputInterface out) {

        writeNodes(out);

        if (hasDataChanged) {
            out.writeData(this, table.colTypes);
            out.writeEnd();
        }
    }

    public void write(RowOutputInterface out, LongLookup lookup) {

        out.writeSize(storageSize);

        NodeAVL rownode = nPrimaryNode;

        while (rownode != null) {
            rownode.write(out, lookup);

            rownode = rownode.nNext;
        }

        out.writeData(this, table.colTypes);
        out.writeEnd();
    }
```

事实上，在第一个write函数中，即便记录没有更新，仍然会将索引结点更新。更新索引结点的函数是writeNodes：

```
void writeNodes(RowOutputInterface out) {

        out.writeSize(storageSize);

        NodeAVL n = nPrimaryNode;

        while (n != null) {
            n.write(out);

            n = n.nNext;
        }
    }
```

其中将所有的相关索引结点依次调用了NodeAVL.write方法进行了更新。

2. RowDiskDataChange.write:

```
public void write(RowOutputInterface out) {
```

```
        writeNodes(out);

        if (hasDataChanged) {
            out.writeData(this, table.colTypes);

            if (updateData != null) {
                Type[] targetTypes = targetTable.colTypes;

                out.writeData(targetTypes.length, targetTypes, updateData,
                              null, null);

                RowOutputBinary bout = (RowOutputBinary) out;

                if (updateColMap == null) {
                    bout.writeNull(Type.SQL_ARRAY_ALL_TYPES);
                } else {
                    bout.writeArray(updateColMap);
                }
            }

            out.writeEnd();

            hasDataChanged = false;
        }
    }
```

注意到，其中首先进行了索引节点的更新，之后根据记录是否发生更改决定是否进行记录的更新。

因此，索引节点和数据的更新并不是同步的。

四、缓存机制中的冲突处理

HashIndex类是hash实现的最基本的类，BaseHashMap与Cache都是在其基础上实现的。HashIndex处理冲突采用的是chaining的方法。首先，对于每一个RowAVL对象，将其long类型的position属性longKey转换为int类型hash作为哈希函数，之后在hashTable数组中的第hash个位置存储了另外一个数组linkTable的索引值lookup，而linkTable中每个位置存储了下一个位置。与此相对应的是BaseHashMap中的objectKeyTable，对应的位置就是存储的数据对象。

因此，出现冲突的情况就是当两个对象的position属性转换成相同的int类型的值时相同，此时就会在linkTable中插入一个链表项，同时将数据对象存储到objectKeyTable的对应位置。