HSQLDB的索引与散列机制分析

文庆福
2011013239 thssvince@163.com
清华大学软件学院11班

2014年 3月 23日

目录

1 实验简介

1.1 实验背景

在数据库中，我们不可避免地要进行一些查询，那么如何快速找到我们想要的记录？一般来说，数据库都会使用索引和散列两种技术来实现。HSQLDB 的索引结构是我们都已经很熟悉的 AVL 树，它提供了三种不同的索引形式：

MEMORY_INDEX，DISK_INDEX，POINTER_INDEX，对应于三种不同类型的数据结点。

索引机制涉及的类主要有：

1) org.hsqldb.TableBase, org.hsqldb.Table

2) org.hsqldb.index.Index, org.hsqldb.index.IndexAVL

3) org.hsqldb.index.NodeAVL 及其子类

4) org.hsqldb.Row , org.hsqldb.RowAVL 及其子类

5) org.hsqldb.persist.PersistentStore, org.hsqldb.persist.RowStoreAVL 及其子类

散列机制设计的类主要有：

1) org.hsqldb.store.HashIndex

2) org.hsqldb.store.BaseHashMap

3) org.hsqldb.lib 包中 BaseHashMap 的实现子类 HashMap、HashSet 等

1.2 实验环境

- 操作系统：Windows 8 企业版
- JDK: OpenJDK 7（64-Bit）
- HSQLDB: 2.3.1
- IDE: Eclipse Standard(Kepler Service Release 1)

2 索引机制的具体实现

HSQLDB 中每一张表中都有一个 IndexList。这里每一张表中建立起了多个索引，建立多索引的目的就是便于查找。HSQLDB 的索引机制主要是由 org.hsqldb.index 包下的所有类来实现的。

Index 接口及其实现类 IndexAVL 提供了一组方法对索引进行操作，在具体实现中，用到的结构是 AVLTree 即平衡二叉查找树。IndexAVL 类会将 AVL 树持久化存储，而 IndexAVLMemory 类则将 AVL 树存储在内存中。

2.1 索引的创建

HSQLDB 中索引最初创建是在 CREATE TABLE 之时。读取 script 文件时会读到建表语句，随后 compileStatement 的时候对建表语句进行分析，间接调用到 ParserDDL 的 compileCreateTabe 方法，该方法中调用 Table 构造方法，初始化一个 Table 实例。

Table 初始化之后，则会调用 compileCreateTableBody 来创建表的具体内容。如果该 CREATE TABLE 语句中有 primary key 等约束关系则会调用 readConstraint 方法读取约束。这里读取的约束会在后面的 executeCompiledStatement 中用到。在 executeCompiled-Statement 中，调用上一步得到的 StatementSchema 对象的 execute 方法执行 CREATE TABLE 语句，若上面得到的约束存在，则调用 ParserDDL 的 addTableConstraintDefini-tions 方法添加临时约束。

```
if (tempConstraints != null) {
    table =
        ParserDDL.addTableConstraintDefinitions(
            session, table, tempConstraints,
            foreignConstraints);
    arguments[1] = foreignConstraints;
}
```

在 addTableConstraintDefinitions 中根据，负责向表中添加约束，不同类型的约束创建不同的索引，如 primary key、unique、foreign key 等。 调用 table 中的 createPrimaryKey 方法:

```
table.createPrimaryKey(indexName, c.core.mainCols, true);
```

createPrimaryKey()方法设置主键后，又调用 createPrimaryIndex() 方法执行。先新建索引对象，然后加入表。

```
public final void createPrimaryIndex(int[] pkcols, Type[] pktypes,
                                      HsqlName name) {
    long id = database.persistentStoreCollection.getNextId();
```

```
        Index newIndex = database.logger.newIndex(name, id, this, pkcols,
            null, null, pktypes, true, pkcols.length > 0, pkcols.length > 0,
            false);

        try {
            addIndex(null, newIndex);
        } catch (HsqlException e) {}
    }
```

addIndex() 方法负责将索引对象按优先级顺序插入 indexList 的适当位置。

```
final void addIndex(Session session, Index index) {

        int i = 0;

        for (; i < indexList.length; i++) {
            Index current = indexList[i];
            int order = index.getIndexOrderValue()
                        - current.getIndexOrderValue();

            if (order < 0) {
                break;
            }
        }

        indexList = (Index[]) ArrayUtil.toAdjustedArray(indexList, index, i,
                1);

        for (i = 0; i < indexList.length; i++) {
            indexList[i].setPosition(i);
        }

        if (store != null) {
            try {
                store.resetAccessorKeys(session, indexList);
            } catch (HsqlException e) {
                indexList = (Index[]) ArrayUtil.toAdjustedArray(indexList,
                        null, index.getPosition(), -1);

                for (i = 0; i < indexList.length; i++) {
                    indexList[i].setPosition(i);
                }

                throw e;
            }
```

```
        }

        setBestRowIdentifiers();
    }
```

setBestRowIdentifiers() 方法为列设置最佳索引。该方法遍历各索引，将索引对应列 的最佳索引设为该索引。

之后，addTableConstraintDefinitions()遍历每个约束，创建 unique 约束时调用 createAndAddIndexStructure 方法，创建索引。建立索引的过程与主键索引相似：获取对应列，新建索引对象，插入indexList，重 设访问索引和最佳行标识符。

```
Index index =
    table.createAndAddIndexStructure(indexName,
    c.core.mainCols, null, null, true, true, false);

    final Index createIndexStructure(HsqlName name, int[] columns,
                                     boolean[] descending,
                                     boolean[] nullsLast, boolean unique,
                                     boolean constraint, boolean forward) {

        if (primaryKeyCols == null) {
            throw Error.runtimeError(ErrorCode.U_S0500, "createIndex");
        }

        int     s     = columns.length;
        int[]   cols  = new int[s];
        Type[] types = new Type[s];

        for (int j = 0; j < s; j++) {
            cols[j]  = columns[j];
            types[j] = colTypes[cols[j]];
        }

        long id = database.persistentStoreCollection.getNextId();
        Index newIndex = database.logger.newIndex(name, id, this, cols,
            descending, nullsLast, types, false, unique, constraint, forward);

        return newIndex;
    }
```

创建 foreign key 约束时调用 addForeignKey 方法添加约束，在 addForeignKey 方法中同样调用 createAndAddIndexStructure 方法创建索引，与之前类似，不再赘述。

```
Index index = table.createAndAddIndexStructure(refIndexName,
    c.core.refCols, null, null, false, true, isForward);
```

2.2 插入操作

执行数据的插入操作时，层层跟进，是在 RowStoreAVL 类的 indexRow 方法中被调用 IndexAVL 类的 insert 方法来具体实现的。该方法将遍历表中的每个索引，并在每个索引中 插入，以保证数据的一致性。下面就重点分析在一个 IndexAVL 中执行 insert 方法的具体流程。

```
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());

        writeLock.lock();
        store.writeLock();

        try {
            n = getAccessor(store);
            x = n;

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

                return;
            }
        ...
```

首先调用 getAccessor 方法先找到根，如果根为空，则设置当前行的结点为根节点。

```
  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);
            }
        }

        if (compare == 0) {
            Constraint c = null;

            if (isConstraint) {
                c = ((Table) table).getUniqueConstraintForIndex(this);
            }

            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;
        }
    }
```

上面代码段中的 compareRowForInsertOrDelete 方法用于比较两个数据行是否相等，如果查到根据该索引 值比较出相等的结果的两个行，则出错。若不相等则依次向下，查找找到待插入数据的位置。

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

若根结点不空，找到应该插入的位置，并将其父结点记为 x，方向为 isleft，通过 set 方法将当前行对应的结点插入索引 的 AVL 树中，并维护平衡性。

2.3 删除操作

在删除数据时，会调用到 RowStoreAVL 中的 delete 方法，遍历 indexList，从各索引树中删除指定行对应的结点。

```
for (int i = 0; i < indexList.length; i++) {
            indexList[i].delete(session, this, row);
        }
```

最终会调用到 IndexAVL 中德 delete(PersistentStore store, NodeAVL x) 方法，删除的算法与一般的 AVL 树删除操作相似，大致包括：通过修改引用将结点从树中移除，如果需要，寻找合适的结点替代其位置，再进行旋转操作使树保持平衡。

2.4 更新操作

其实这里的索引并不提供所谓的更新操作，因为我们知道，在一棵 AVL 树上，如果改变了某一结点的键值（key），它就不再具有 AVL 树的性质了。因此 AVL 的更新操作应该就是先删除然后再插入的操作。通过测试 update 语句就会发现，程序分别会进行 delete 和 insert 的操作。总体来说更新操作的大致流程如下：

- 删除数据行的索引IndexAVL.delete(PersistentStore, Row)
- 删除索引IndexAVL.delete(PersistentStore, NodeAVL)
  - 删除索引节点
  - 重新平衡
- 删除数据
- 插入所有索引中RowStoreAVL.indexRow()
- 向单个索引插入节点IndexAVL.insert()
  - 如果AVL为空，新建AVL存储器PersistentStore.setAccessor()
  - 沿AVL寻找插入位置,插入新数据IndexAVL.set(),平衡AVL IndexAVL.balance()
- 插入数据

3 索引与记录的对应

这里对 HSQLDB 的索引机制的实现做一个简单的小结。

HSQLDB 中实现的是多索引机制，每一张表对应了多个索引，有一个 IndexList 与之对 应。具体的实现是用 Index 接口，其实现类 IndexAVL 提供对索引进行操作访问和维护的方 法。

表中的记录由表对应的 PersistenceStore 进行管理，Row 是对行的封装类，根据不

同的 表类型有不同的实现。

每一个索引建立一棵索引树，这里的索引树是平衡二叉树。实现上是用的是 NodeAVL 系列的类。 建立索引的目的就是方便进行查找，平衡二叉树查找方便，所以被选在这里作为 理想的索引树实现机制。

每一条记录在每一个索引树中都有一个对应项，在 Row 封装中将这些对影响串成了一 个链表。

相关的数据操作除了会改动数据本身，必要时还会修改索引值。

4 索引的内外存交换机制

IndexAVLMemory 类与 IndexAVL 类的最大区别就是 IndexAVLMemory 中的结点全部保存在内存中，而 IndexAVL 中用到的结点，则必须通过 getLeft、getRight 等方法从外存或缓存中读取。

IndexAVL 类使用 NodeAVLDisk 类作为结点。下面以 NodeAVLDisk.getLeft() 为例:

```
NodeAVL getLeft(PersistentStore store) {

    NodeAVLDisk node = this;
    RowAVLDisk  row  = this.row;

    if (!row.isInMemory()) {
        row  = (RowAVLDisk) store.get(this.row, false);
        node = (NodeAVLDisk) row.getNode(iId);
    }

    if (node.iLeft == NO_POS) {
        return null;
    }

    if (node.nLeft == null || !node.nLeft.isInMemory()) {
        node.nLeft          = findNode(store, node.iLeft);
        node.nLeft.nParent = node;
    }

    return node.nLeft;
}
```

如果当前行不在内存，则调用 RowAVLDisk 类的 get 方法从缓存中读取数据；如果左孩子不存在，则返回 null，否则判断左孩子是否在内存，如果不在，则调用 findNode 方法从缓存中读取，并设置其父结点为当前结点，然后返回。

在读入行数据的过程中建立NodeAVLDisk类型结点，其构造函数从文件中读取索引

信息，但并不设置指向父子结点的引用（如上一段，当需要时才读入），最后将结点加入树中。

```
public NodeAVLDisk(RowAVLDisk r, RowInputInterface in,
        int id) throws IOException {
    row   = r;
    iId   = id;
    iData   = r.getPos();
    iBalance = in.readInt();
    iLeft   = in.readInt();
    if (iLeft <= 0) {
        iLeft = NO_POS;
    }

    iRight = in.readInt();
    if (iRight <= 0) {
        iRight = NO_POS;
    }
    iParent = in.readInt();
    if (iParent <= 0) {
        iParent = NO_POS;
    }
}
```

在清理缓存或关闭数据库的过程中，向文件保存数据时，索引信息一并保存。从主索引结点开始，逐个将节点信息写入文件。

```
private void writeNodes(RowOutputInterface out) throws IOException
{
    out.writeSize(storageSize);
    NodeAVL n = nPrimaryNode;
    while (n != null) {
        n.write(out);
        n = n.nNext;
    }
    hasNodesChanged = false;
}
```

首先写入 iBalance，然后依次是 iLeft、iRight、iParent。如果该行的数据也发生了改变，则索引项写入完成后再写入数据。

```
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);
```

```
}
```

5 散列机制的实现

HSQLDB 中的散列机制用于存储各种信息和映射关系。利用 hash 函数进行散列之后可以实现系统内部所需的数据的快速存取。

5.1 查找操作

BaseHashMap类中的getLookup()方法根据关键字查找其存储位置。

```
protected int getLookup(int key) {
    int lookup = hashIndex.getLookup(key);
    int tempKey;
    for (; lookup >= 0; lookup = hashIndex.linkTable[lookup]) {
        tempKey = intKeyTable[lookup];
        if (key == tempKey) {
            return lookup;
        }
    }
    return lookup;
}
```

该方法首先调用 HashIndex 类的 getLookup 方法，找到给定 key 的第一次出现位置，然后利用 HashIndex 类提供的 linkTable 数组维护的“链”，不断获取下一个数据存储位置，直到该位置存储的数据等于给定关键字为止。

5.2 添加操作

BaseHashMap 类将添加和删除操作写在了一起，通过参数区分这两种操作。

```
int     index       = hashIndex.getHashIndex(hash);
int     lookup      = hashIndex.hashTable[index];
int     lastLookup  = -1;
Object  returnValue = null;

for (; lookup >= 0;
        lastLookup = lookup,
        lookup = hashIndex.getNextLookup(lookup)) {
    if (isObjectKey) {
        if (comparator == null) {
```

```
                if (objectKeyTable[lookup].equals(objectKey)) {
                    break;
                }
            } else {
                if (comparator.compare(objectKeyTable[lookup], objectKey)
                        == 0) {
                    break;
                }
            }
        } else if (isIntKey) {
            if (longKey == intKeyTable[lookup]) {
                break;
            }
        } else if (isLongKey) {
            if (longKey == longKeyTable[lookup]) {
                break;
            }
        }
}
```

如果找到（lookup≥0），则表明关键字已存在。若值为引用类型，则使用新值替换 旧值；否则直接返回（因为替换没有意义）。

如果没有找到，则此时lastlookup 的值即为最后一个与待插入关键字具有相同散列值的关键字存储位置。插入数据前首先判断元素数是否已达到阈值，如果是，则扩充映射表。

之后，获取新关键字的插入位置，即第一个空闲位置。修改linkTable，将新关键字加入链接，最后将新关键字和值加入键表和值表中。

```
public int linkNode(int index, int lastLookup) {

    // get the first reclaimed slot
    int lookup = reclaimedNodePointer;

    if (lookup == -1) {
        lookup = newNodePointer++;
    } else {

        // reset the first reclaimed slot
        reclaimedNodePointer = linkTable[lookup];
    }

    // link the node
```

```
    if (lastLookup == -1) {
        hashTable[index] = lookup;
    } else {
        linkTable[lastLookup] = lookup;
    }

    linkTable[lookup] = -1;

    elementCount++;

    modified = true;

    return lookup;
}
```

5.3 删除操作

删除过程首先也是利用散列查找关键字所在位置，如果找到，将关键字和值清除，修改linkTable 将关键字从链接中删除，并将其地址加到可回收地址链表的头部。

```
public void unlinkNode(int index, int lastLookup, int lookup) {

    // unlink the node
    if (lastLookup == -1) {
        hashTable[index] = linkTable[lookup];
    } else {
        linkTable[lastLookup] = linkTable[lookup];
    }

    // add to reclaimed list
    linkTable[lookup]     = reclaimedNodePointer;
    reclaimedNodePointer = lookup;

    elementCount--;
}
```

6 问题总结

6.1 索引的组织结构是怎样的？

一个表可以有多个索引，这通过索引表来维护：每个表项记录该索引的根结点。如果表有 n 个索引，则表中每一行数据会有一个长度为 n 的数组保存该行在 n 个索引中相应的结点。

6.2 增删改记录时索引是怎样变化的？

具体流程可参看 2.2-2.4 节的分析

6.3 索引与记录结点怎样关联在一起关联？

从org.hsqldb.index.NodeAVL继承下来的三种索引结点：

- NodeAVLMemory类（Memory Table的索引节点类型）
- NodeAVLDisk类（Cached Table的索引节点类型）
- NodeAVLMemoryPointer类（Text Table的索引节点类型）

6.4 索引在外存中的如何存储？

在外存中，数据按行保存，首先保存每行对应的各索引的结点信息，包括 iBalance、iLeft、iRight 和 iParent，这记录了相关结点的 id，是一个 32 位整数。保存完索引后，接下来保存的是该行数据。

6.5 索引是怎样实现内外存交换的？

内存 →外存

- 当数据库关闭时，或者需要备份、保存.data文件时
- 将每行数据的索引位置（iBalance、iLeft、iRight、iParent 等）依次以二进制方式与每行数据内容合并
- 将整个数据按照次序存入.data文件

外存 →内存

- 当数据库打开时
- 按照次序，从.data文件中获取每行数据二进制块
- 从数据块头部提取int 类型索引位置数据（iBalance、iLeft、iRight、iParent 等）

6.6 散列的存储机制是怎样的？

散列所需的基本信息保存在 HashIndex 类中，包括一个用数组实现的链表，记录了后继结点下标 (hashTable)，以及散列值对应的起始下标数组(indexTable)。

散列的其他数据保存在 BaseHashMap 类的成员域中，记录键值对等信息。

6.7 散列机制是如何处理冲突的？

散列机制主要涉及到两个表 hashTable 和 linkTable，散列表对应的数据表和 link-Table 有 一一对应关系。hashTable 用于哈希散列后给出指向数据的索引。而 linkTable 则实际管理数 据之间的链接关系。

每一个 hash 值在 hashTable 中有一个索引值指向 linkTable 中的一项，该项表示哈希值 为该值的数据构成的链表的头结点。 这种散列机制处理冲突的方法是链表法，即同一哈希值 的记录链在一个链表上。