HSQLDB的查询处理分析

软件62 周展平 2016013253

问题思考

1. QuerySpecification类中与select、from、where子句相关的属性

在解析select相关语句的过程中，ParserDQL类调用了XreadQuerySpecification方法进行解析，其中首先调用了XreadSelect方法，如下：

```
QuerySpecification XreadSelect() {
        //创建QuerySpecification对象
        QuerySpecification select = new QuerySpecification(compileContext);

        readThis(Tokens.SELECT); //读取select

        if (token.tokenType == Tokens.TOP || token.tokenType == Tokens.LIMIT) {
            SortAndSlice sortAndSlice = XreadTopOrLimit();

            if (sortAndSlice != null) {
                select.addSortAndSlice(sortAndSlice);
            }
        }

        if (token.tokenType == Tokens.DISTINCT) { //判断是否有distinct限制
            select.setDistinctSelect(); //设置isDistinctSelect = true
            read();
        } else if (token.tokenType == Tokens.ALL) { //判断是否有all限制
            read();
        }
        //分析所有需要查询的colomn
        while (true) {
            Expression e = XreadValueExpression(); //读取关于colomn的表达式

            if (token.tokenType == Tokens.AS) { //判断是否有as
```

```
                read();
                checkIsNonCoreReservedIdentifier(); //检查替换的名称不能重名
            }

            if (isNonCoreReservedIdentifier()) { //检查属性名称不存在
                e.setAlias(HsqlNameManager.getSimpleName(token.tokenString,
                        isDelimitedIdentifier()));
                read();
            }

            select.addSelectColumnExpression(e); //exprColumnList.add(e);
indexLimitVisible++;

            if (token.tokenType == Tokens.FROM) { //出现from则结束
                break;
            }

            if (token.tokenType == Tokens.INTO) { //出现into则结束
                break;
            }

            if (readIfThis(Tokens.COMMA)) { //遇到逗号则继续分析
                continue;
            }
            //出现其他的符号
            if (token.tokenType == Tokens.CLOSEBRACKET
                    || token.tokenType == Tokens.X_ENDPARSE
                    || token.tokenType == Tokens.SEMICOLON) {
                if (database.sqlSyntaxMss || database.sqlSyntaxMys
                        || database.sqlSyntaxPgs) {
                    RangeVariable range =
                        new RangeVariable(database.schemaManager.dualTable,
                                          null, null, null, compileContext);

                    select.addRangeVariable(session, range);

                    return select;
                }
            }

            throw unexpectedToken();
        }

        return select;
    }
```

在此函数中，涉及到了QuerySpecification类的isDistinctSelect属性，代表是否有distinct的限制；exprColumnList属性，储存了查询结果中需要返回的colomn相关表达式；indexLimitVisible属性，代表查询结果的列数。

之后，执行ParserDQL.XreadTableExpression函数，其中XreadFromClause函数负责处理from子句，readWhereGroupHaving负责处理where,group和having子句。XreadFromClause函数调用了XreadTableReference函数，其中QuerySpecification.rangeVariableList属性记录了from子句中的左右RangeVariable。readWhereGroupHaving函数中对于where语句调用XreadBooleanValueExpression函数得到查询条件的表达式并赋值给QuerySpecification.queryCondition属性。

另外，对于聚合函数的处理虽然属于对group by和order by子句的处理函数中，但是由于书写在select子句中，因此也在下面select相关变量中列出。

综合以上分析，QuerySpecification的相关属性有：

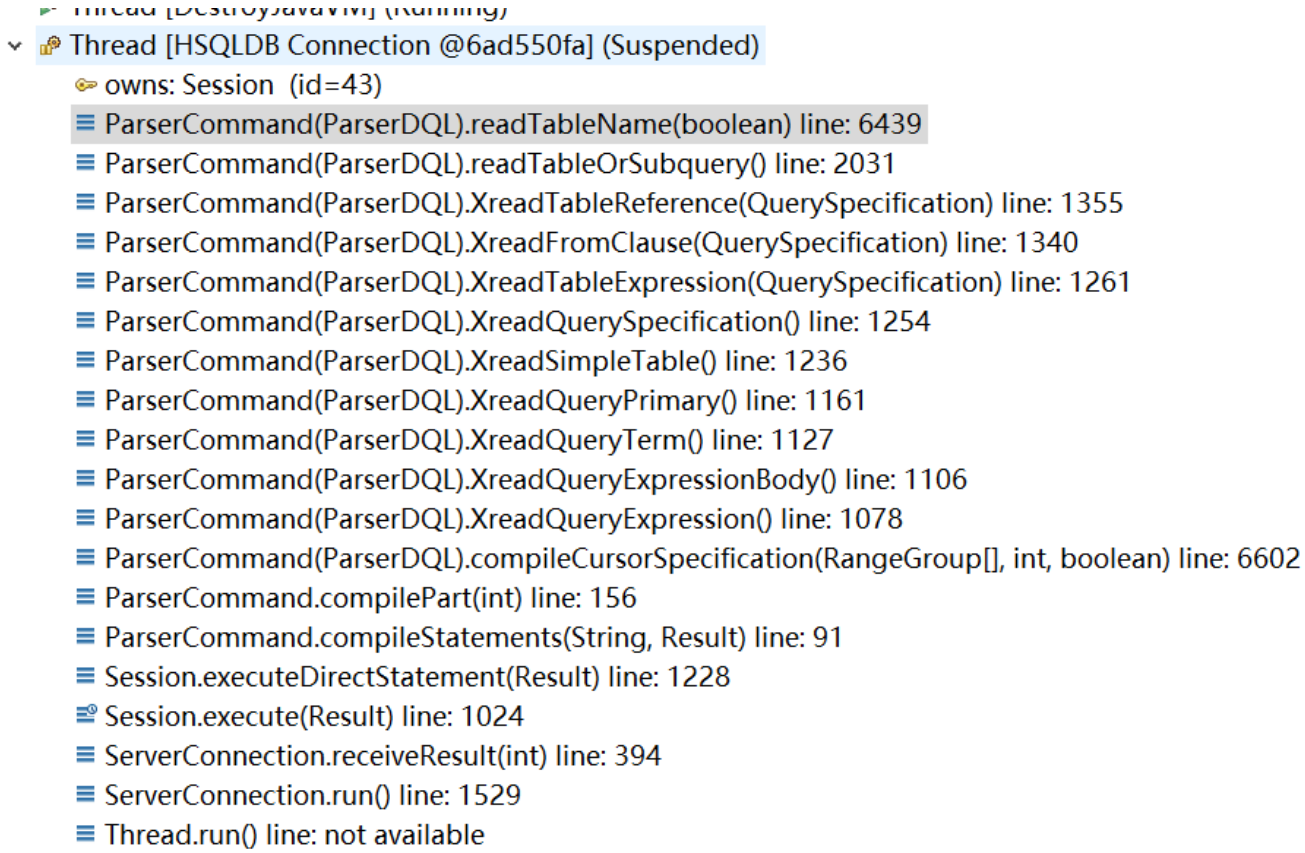
```
//select相关属性
public boolean        isDistinctSelect;     //是否有distinct限制select
public boolean        isAggregated;         //结果是否是聚合函数
public boolean        isGrouped;            //结果是否有group by限制
public boolean        isSimpleDistinct;     //查询的colomns是否是index
Expression[]          exprColumns;          //查询结果中的columns表达式
HsqlArrayList         exprColumnList;       //查询结果中的columns表达式
public int            indexLimitVisible;    //查询结果colomns的数量
Type[]                resultColumnTypes;    //查询结果colomns的类型
Expression            rowExpression;        //与查询结果colomns的类型相关

//from相关属性
RangeVariable[]       rangeVariables;
private HsqlArrayList rangeVariableList;
int                   startInnerRange = -1; //ParseDML用到
int                   endInnerRange   = -1; //ParseDML用到


//where相关属性
Expression            queryCondition;       //一个布尔表达式，即predicate
Expression            checkQueryCondition;
```

2. 如果查询的 table 不存在，其正确性检查在何时进行?

对于table名称的解析由ParserDQL.readTableName函数完成。函数调用栈如下：



其中XreadFromClause函数在第一个问题中已经涉及到，功能是解析from子句。readTableName函数如下：

```
Table readTableName(boolean orSynonym) {

    checkIsIdentifier(); //检查是否是合法标识符

    lastSynonym = null;

    Table table = database.schemaManager.findTable(session,
                                                   token.tokenString,
token.namePrefix, token.namePrePrefix); //database.schemaManager根据输入的SQL语句中的表名进行查找

    if (table == null) { //未找到
        boolean trySynonym = orSynonym && token.namePrefix == null
            && !isViewDefinition;

        if (trySynonym) { //同义词查找
            ReferenceObject reference = database.schemaManager.findSynonym(
                token.tokenString,
                session.getCurrentSchemaHsqlName().name,
                SchemaObject.TABLE);

            if (reference != null) {
                table = (Table) database.schemaManager.getSchemaObject(
```

```
                    reference.getTarget());
                lastSynonym = reference.getName();
            }
        }

        if (table == null) { //最终确认未找到
            throw Error.error(ErrorCode.X_42501, token.tokenString); //抛出异常
ErrorCode.X_42501
        }
    }

    getRecordedToken().setExpression(table);
    read();

    return table;
}
```

如果最终未找到对应的表，则抛出异常ErrorCode.X_42501，对应的异常信息为：

```
// ErrorCode.java

// HSQLDB database object names
int X_42501 = 5501;                             // user lacks privilege or object not
found
```

3. 基于代价的对于 JOIN 操作访问 Table 顺序的调整策略

与优化join操作的顺序相关的函数是RangeVariableResolver类的reorderRanges函数：

```
    void reorderRanges(HsqlArrayList starts, HsqlArrayList joins) {

        if (starts.size() == 0) { //没有不需要重新调整顺序
            return;
        }

        int           position = -1;    //寻找到的需要调整位置的table当前的序号
        RangeVariable range    = null;
        double        cost     = 1024;  //当前的最小的检索代价，初始值为1024

        for (int i = 0; i < firstLeftJoinIndex; i++) { //遍历所有rangeVariables
            Table table = rangeVariables[i].rangeTable;

            if (table instanceof TableDerived) { //如果是查询产生的属性涉及到的表，直接跳过
                continue;
            }
```

```
            collectIndexableColumns(rangeVariables[i], starts); //对于单个属性进行判断的表达式，将对应的colomn的索引，根据具体的比较类型放入colIndexSetEqual或colIndexSetOther

            IndexUse[] indexes = table.getIndexForColumns(session,
                colIndexSetEqual, OpTypes.EQUAL, false); //找到可以用于查询属性集colIndexSetEqual的所有索引
            Index index = null;

            for (int j = 0; j < indexes.length; j++) {
                index = indexes[j].index;

                double currentCost = searchCost(session, table, index,
                                                indexes[j].columnCount,
                                                OpTypes.EQUAL); //估计检索代价

                if (currentCost < cost) { //如果发现了更小的检索代价
                    cost     = currentCost; //更新当前最小的检索代价
                    position = i;           //更新得到当前最小的检索代价对应的表
                }
            }

            if (index == null) { //没有可以用于查询目标属性集的索引，说明indexes为null
                Iterator it = colIndexSetOther.keySet().iterator();

                while (it.hasNext()) { //遍历colIndexSetOther
                    int colIndex = it.nextInt();

                    index = table.getIndexForColumn(session, colIndex); //找到用于查询该属性的索引

                    if (index != null) { //存在这样的索引
                        cost = table.getRowStore(session).elementCount() / 2.0; //代价

                        if (colIndexSetOther.get(colIndex, 0) > 1) { //如果属性在条件中出现多次则cost减半
                            cost /= 2;
                        }

                        break;
                    }
                }
            }

            if (index == null) { //仍然没有找到可以用于查询目标属性集的所有索引，放弃估计代价
                continue;
            }
```

```
            if (i == 0) { //firstLeftJoinIndex <= 0
                position = 0;

                break;
            }
        }

        if (position < 0) { //没有找到合适的用于交换的table
            return;
        }

        if (position == 0 && firstLeftJoinIndex == 2) { //无需交换
            return;
        }
        //newRanges是通过将rangeVariables的position位置的元素与第一个元素进行交换得到的
        RangeVariable[] newRanges = new RangeVariable[rangeVariables.length];

        ArrayUtil.copyArray(rangeVariables, newRanges, rangeVariables.length);

        range                = newRanges[position];
        newRanges[position] = newRanges[0];
        newRanges[0]         = range;
        position             = 1;
        //从position开始继续遍历所有table,
        for (; position < firstLeftJoinIndex; position++) {
            boolean found = false;

            for (int i = 0; i < joins.size(); i++) { //找出position后面的、应该换到
position处的range
                Expression e = (Expression) joins.get(i);

                if (e == null) {
                    continue;
                }

                int newPosition = getJoinedRangePosition(e, position,
                    newRanges);

                if (newPosition >= position) { //将newPosition处的range与position处进行
交换
                    range                   = newRanges[position];
                    newRanges[position]    = newRanges[newPosition];
                    newRanges[newPosition] = range;

                    joins.set(i, null);

                    found = true;
```

```
                    break;
                }
            }

            if (found) { //找到了与position交换的rangeVariable，继续向后查找
                continue;
            }

            for (int i = 0; i < starts.size(); i++) { //在starts中继续查找
                Table table = newRanges[i].rangeTable;

                collectIndexableColumns(newRanges[i], starts);

                IndexUse[] indexes = table.getIndexForColumns(session,
                    colIndexSetEqual, OpTypes.EQUAL, false);

                if (indexes.length > 0) {
                    found = true;

                    break;
                }
            }

            if (!found) {
                break;
            }
        }

        if (position != firstLeftJoinIndex) {
            return;
        }
        //将更新后的newRanges复制到rangeVariables
        ArrayUtil.copyArray(newRanges, rangeVariables, rangeVariables.length);
        joins.clear();
        //将tempJoinExpressions的所有元素集中到tempJoinExpressions[firstLeftJoinIndex -
1]中
        for (int i = 0; i < firstLeftJoinIndex; i++) {
            HsqlArrayList tempJoins = tempJoinExpressions[i];

            joins.addAll(tempJoins);
            tempJoins.clear();
        }

        tempJoinExpressions[firstLeftJoinIndex - 1].addAll(joins);
        rangeVarSet.clear();
        //将rangeVariables中的所有range加入rangVarSet中
        for (int i = 0; i < rangeVariables.length; i++) {
            rangeVarSet.add(rangeVariables[i]);
```

```
        }
    }
```

其中调用了searchCost函数来估计检索代价：

```
private double searchCost(Session session, Table table, Index index,
                          int count, int opType) {

        if (table instanceof TableDerived) {
            return 1000;
        } else {
            return table.getRowStore(session).searchCost(session, index,
                                    count, opType);
        }
    }
```

如果table是TableDerived的实例，说明此table是一个查询的结果，此时将代价设置为1000，一方面因为代价难以具体量化，另一方面因为查询结果的数量不会太多，且容易检索，因此设置的代价低于初始设置的阈值1024。其他情况下需要依次调用RowStoreAVL类、IndexAVL类的searchCost函数对代价进行估计。

4. RangeVariableResolver.reorder 函数中starts和joins数组的含义

starts和joins数组包含的元素类型都是Expression，即from子句和where子句中的表达式，starts中的表达式只与单个列（column）有关，例如person.ID='00000001'，此时该表达式的isSingleColumnCondition属性为真；而joins中的表达式涉及到多个列的比较，例如person.ID=personmem.ID，此时该表达时的isColumnEqual属性为真。