Hash Join功能开发文档

修订历史

版本	修订日期	修订描述	作者	备注
0.3	2017-09-29	Hash Join功能开发文档	茅潇潇	无

1 总体设计

1.1 综述

Cedar是可扩展的关系数据库，实现了巨大数据量上的跨行跨表事务。在线业务中存在大量的多表连接查询，在处理多表连接时，Cedar支持Merge Join、BloomFilter Join、Semi Join运算，特点是Merge Server（MS）分发请求给相应的Chunk Server（CS），CS把过滤后的数据返回给MS，MS排序后做 Merge操作。但是，如果对两张或两张以上的大表进行连接，MS需要对大量的数据进行排序和比较，并且由于上述算子的连接操作都基于排序归并导致需要缓存每张表的数据和中间结果表的数据，对CPU计算资源和内存资源消耗较大。为解决上述问题，Cedar 0.3版本增加了Hash Join的支持。Hash Join首先对前表构建Hash Table，然后流水线处理后表每行数据做连接，一方面对于选择率较高的查询减少了大量的数据对比次数，明显降低了连接计算的时间，另一方面不再需要对后表进行内存中的缓存和排序，减少了时间和空间资源的消耗。生产环境中检验证明Hash Join的应用显著提高了多表连接查询的性能。

1.2 名词解释

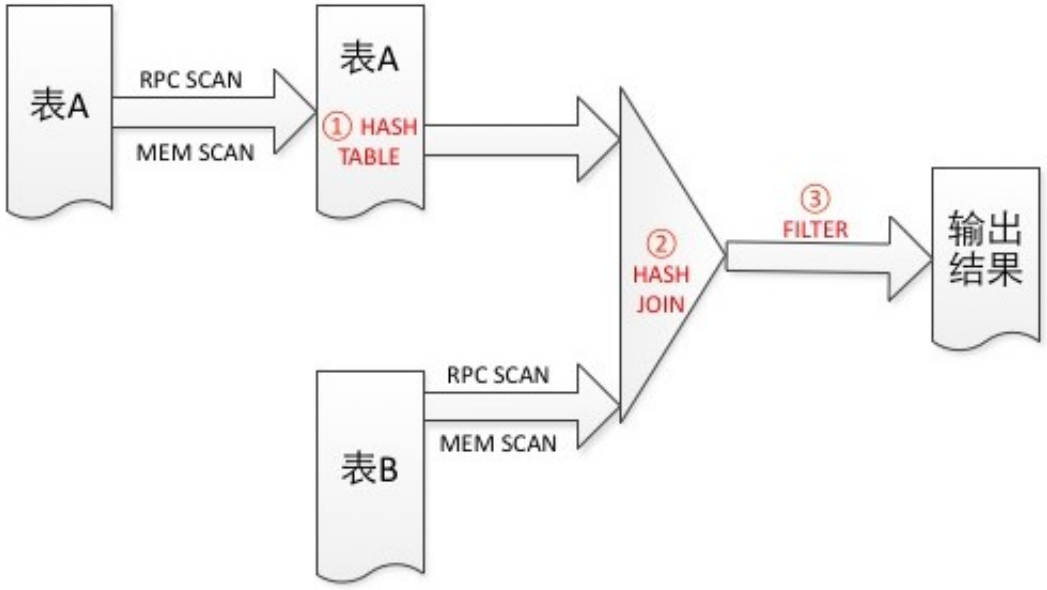
- **CS**：ChunkServer，基线数据服务器，提供分布式数据存储服务，负责存储基线数据。
- **MS**：MergeServer，查询处理服务器，负责接收和解析SQL请求、生成和执行查询计划以及将所有节点的查询结果合并并返回给客户端。
- **哈希函数，哈希表**：给定表M，存在函数f(key)，对任意给定的关键字值key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表M为哈希表，函数f(key)为哈希函数。

1.3 功能

Hash Join使用关系S利用公共属性B上在内存中建立哈希表，然后扫描R表并探测哈希表，找出相匹配的行。假设，节点1上的关系R和节点2上的关系S，在属性R.A=S.B上做连接操作，它的工作流程如下：首先，节点2对关系S的每个元组在属性B上的值使用特定的哈希函数进行计算，并在内存里产生一张哈希表；然后，节点1把关系R中的记录传送给节点2，在节点2上对关系R的每个元组在属性A上的值使用相同的哈希函数进行计算，将计算的结果在哈希表中进行探测；如果探测成功，则一条新的纪录将被创建。

2 模块设计

2.1 基本原理



如图所示，Hash Join处理可以分三个步骤：

1. 生成Hash Table：将参与join的左表生成Hash Table。先将前表所有复合查询条件的数据获取到Merge Server，建立Hash Table。
2. Hash Join处理：根据参与join的后表探测Hash Table，找出Hash Table中与后表匹配的行，进行等值join操作。
3. FILTER处理：按照不等值条件进行过滤，进而输出最终结果。

2.2 解析词法语法子模块设计

在Hint中对指定的Join算法进行解析，并指定两两表之间的Join规则，把用户输入的 SQL 语句解析成语法树。

例如：Select /*+ join(hash_join,merge_join,hash_join,merge_join) */ * from tab1 left join tab2 on tab1.col1=tab2.col2 right join tab3 on tab2.col1=tab3.col2 full outer join tab4 on tab1.col1=tab4.col2 inner join tab5 on tab3.col1=tab5.col2;
/*+ join(hash_join,merge_join,hash_join,merge_join) */括号内需要符合“join_type(,join_type)”规则，按照join顺序，表明两两表之间的Join使用的算法规则。

- 若hint中的join类型个数多于后面数据表的两两连接数，则忽略hint中多余的join类型；若hint中的join类型个数少于后面数据表的两两连接数，则后面未指定的连接都执行merge join。
- 对于select * from tab1,tab2 where tab1.col1=tab2.col2; 此类Join查询需要DBA改为Inner Join后再指定Hint，完成Bloomfilter Join；否则将执行原Merge Join操作。

修改内容：

1. 分别向词法定义文件sql_parser.l和语法定义文件sql_parser.y中添加Hash Join的词法规则和语法规则，使解析器能够识别Hash Join的语法。

2.3 生成逻辑计划子模块设计

根据语法树生成中缀表达式和Hint信息，判断hint中指定的join信息是否符合规则，存到逻辑计划中。

- 在生成逻辑计划时，判断Hint信息是否与SQL中Join运算一一对应，若不对应则按照原Merge Join查询。

修改内容：

1. 对语法树生成中缀表达式和Hint信息，增加hint是否有指定join信息的分支case T_JOIN_TYPE_LIST。
2. 增加方法oceanbase::sql::generate_join_hint()对Hint中的Join信息进行判断是否符合规则，并将Hint信息中每一个join类型有序地存到一个动态数组中，供生成物理计划时使用。

2.4 生成物理计划子模块设计

根据逻辑计划里面存的相应信息，然后按照指定的Join算法生成相应的物理操作符，包括Merge Join运算符，BloomFilter Join运算符，Semi Join运算符和Hash Join运算符，并按照Join先后顺序确定操作符之间的父子关系。从逻辑计划中取出中缀表达式，转成后缀表达式存到相应的物理操作符里面。

- 按照SQL中的Join顺序生成物理查询计划

```
int ObTransformer::gen_phy_joins(
    ObLogicalPlan *logical_plan,
    ObPhysicalPlan *physical_plan,
    ErrStat& err_stat,
    ObSelectStmt *select_stmt,
    ObJoin::JoinType join_type,
    oceanbase::common::ObList<ObPhyOperator*>& phy_table_list,
    oceanbase::common::ObList<ObBitSet<> >& bitset_list,
    oceanbase::common::ObList<ObSqlRawExpr*>& remainder_cnd_list,
    oceanbase::common::ObList<ObSqlRawExpr*>& none_columnlize_alia
s,
    int32_t hint_idx)
|--->//原Join逻辑不变
|--->//while (ret == OB_SUCCESS && phy_table_list.size() > 1)
|------>判断phy_table_list.size()是否大于hint_idx
|------>并对Hint中Join信息join_array_.size()和指定inner(left,right,ful
l) join判断
        如果没有Hint信息或不是inner(left,right,full) join则执行原来的Mer
ge Join
|------>根据不同Join类型建立物理操作符
        CREATE_PHY_OPERRATOR(hash_join_single_op, ObHashJoinSingle,
physical_plan, err_stat);
        CREATE_PHY_OPERRATOR(join_op, ObMergeJoin, physical_plan, e
rr_stat);
|------>提取ObQueryHint中的信息，判断Join使用的算法
|--->//最后释放join_array_中的join列
```

修改内容：

1. 修改SQL中的Join物理查询计划sql::ObTransformer ::gen_phy_joins()，增加判断Join类型的语句，识别Hash Join类型，建立Hash Join的物理操作符。

2.5 Hash Join物理操作符子模块设计

Hash Join主要实现三个函数：Open()、Close()、Get_next_row()。

- 添加ObHashJoinSingle::open()方法

```
int ObHashJoinSingle::open(){
|--->left_op_->open();打开前表,构造参数和信息
|--->hash_table_.create(HASH_BUCKET_NUM);
|--->while(left_op_->get_next_row(row))
|--------->//迭代前表每一行数据，建立hash table
|--------->curr_row->get_cell(c1.tid,c1.cid,temp)
|--------->hash_key = temp->murmurhash64A(hash_key);
|--------->hash_table_.set_multiple(hash_key, pair)
|--->right_op_->open();打开后表,构造参数和信息
}
```

- 添加ObHashJoinSingle::get_next_row()方法

```
根据join_type选择具体的join算法（left、right、full out、inner join）
int ObHashJoinSingle::inner_hash_get_next_row(const common::ObRow
*&row){
|--->const ObRow *right_row = NULL;
|--->while(OB_SUCCESS == ret)
|--------->//每次迭代后表的get_next_row()取一行数据
|--------->ret = right_op->get_next_row(right_row);
|--------->//从相应的哈希桶中取出一行前表数据
|--------->get_next_equijoin_left_row(left_row, *right_row, bucke
t_hash_key, pair)
|--------->curr_row_is_qualified ();判断非等值连接
|--------->output
|--------->row = &curr_row_;
}
```

- 添加ObHashJoinSingle:: close ()方法

```
int ObHashJoinSingle::close(){
|--->//释放空间，重置运算符中的数据
|--->//row_desc_.reset();
|--->//ret = ObJoin::close();
}
```

3 模块接口

3.1 对外接口

- 对两个或多个数据表使用hint, 指定join类型(hash join，semi join，bloomfilter join或

merge join)的连接。语句如下：
select /*+ join(hash_join,merge_join,…)*/ <查询内容> from <表名> inner (left, right, full outer) join <表名> on <join条件>；
/*+ join (hash_join, merge_join, hash_join, merge_join) */括号内需要符合“join _ type (, join_ type)”规则，按照join顺序，表明两两表之间的join使用的算法规则。

- 若hint中的join类型个数多于后面数据表的两两连接个数，则忽略hint中多余的join类型；若hint中的join类型个数少于后面数据表的两两连接个数，则后面未指定的连接都默认执行merge join。

4 使用限制条件和注意事项

- hash join不支持在不同类型的连接列上做等值连接。
- 多表连接时，根据连接顺序对每两张表使用以上规则，以达到优化效果。