分布与并行数据库

实验报告

完成时间：2018 年 1 月 18 日

2017000627	**柴艳峰**	元数据写入文件、select语句查询执行
2017100913	**李昊华**	RPC远程调用、ZooKeeper保存同步元数据文件
2017104173	**汪弘洋**	查询计划生成、优化、可视化
2017100929	**王传雯**	查询语句解析、其他语句执行、分片数据载入

目录

1. 功能实现

表 1 功能列表

功能		SQL 语句关键字
显示数据库已存在表。		SHOW TABLES
创建/删除表。		CREATE TABLE
分片并导入数据。	垂直分片	FRAG -VER
	水平分片	FRAG -HOR
	混合分片	FRAG -MIX
	不分片	FRAG -NONE
查询并返回记录数量和查询时间。		SELECT
删除记录。		DELETE FROM
插入记录		INSERT
更新记录。		UPDATE

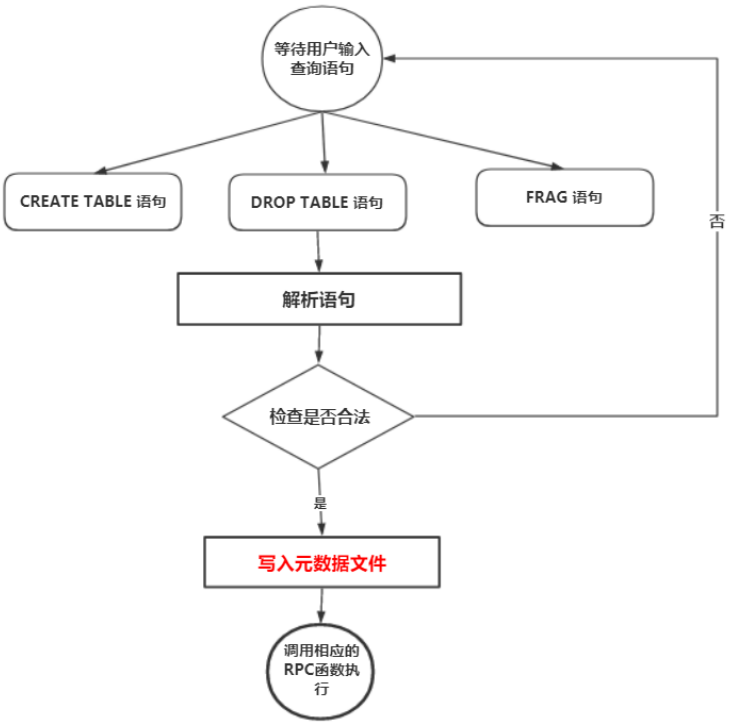
2. 总体流程

不同查询语句的执行流程不太相同，主要分为需要更新的元数据的创建/删除表语句、分片语句；需要根据分片信息分类操作的更新、删除、插入语句；以及需要生成查询计划并优化执行的查询语句。

2.1 不需要生成查询计划的语句。

（1）CREATE TABLE/DROP TABLE/FRAG 语句。

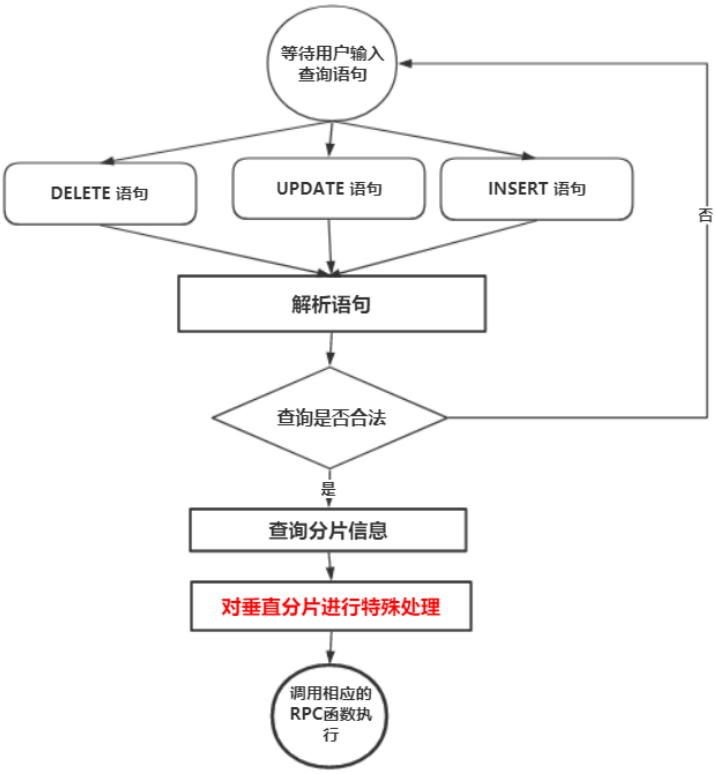
创建表需要向元数据文件中添加表信息，分片语句需要添加分片信息，删除表需要删除表信息、分片信息。具体流程如图



流程图 1 CREATE/DROP TABLE & FRAG 语句

（2）DELETE/UPDATE/INSERT 语句。

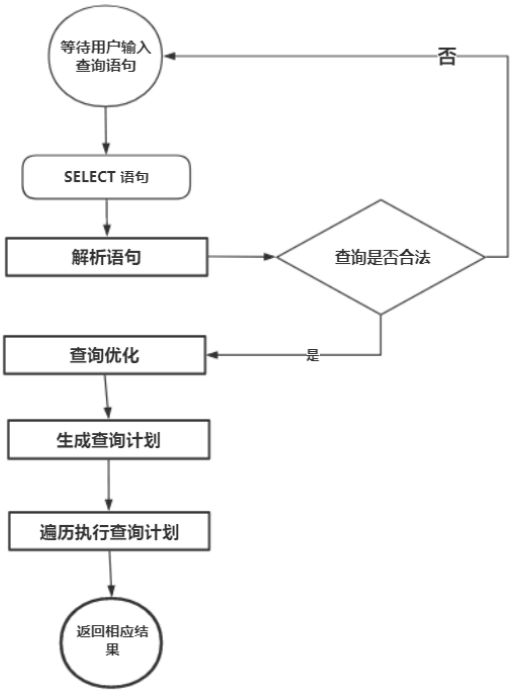
此类语句都需要根据分片信息来执行操作。如果是垂直分片，则需要重构 SQL 语句并去对应站点执行操作。特别地，针对 INSERT 语句，需要根据垂直分片的信息将记录拆分，插入对应站点，具体流程如图



流程图 2 DELETE,UPDATE & INSERT 语句

2.2 需要生成查询计划的语句。

针对 SELECT 语句，需要调用查询优化和执行的接口，查询计划中包含查询元数据的步骤，在流程图中没有详细展示。



流程图 3 SELECT 语句

3. 模块详细实现

3.1 语句解析模块

语句解析部分使用 flex 识别 SQL 语句中保留的关键字，使用 bison 建立语法规则和对应的语义动作。

3.1.1 词法分析器

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中的词汇模式的程序。当 Lex 接收到文件或文本形式的输入时，它试图将文本与常规表达式进行匹配。 它一次读入一个输入字符，直到找到一个匹配的模式。 如果能够找到一个匹配的模式，Lex 就执行相关的动作（可能包括返回一个标记）。

本数据库定义的保留关键字如图 1 所示，包括 CREATE TABLE, SELECT, DROP, INSERT, FROM, WHERE, ORDER BY 等标准 SQL 语句支持的关键词，并支持每种关键字的大小写格式。此外，还需要处理空格和比较、AND 操作符等。

```
        /* reserved keywords */
CREATE  |
create          return CREATE;
TABLE   |
table           return TABLE;
DROP    |
drop            return DROP;
INDEX   |
index           return INDEX;
SELECT  |
select          return SELECT;
FROM    |
from            return FROM;
WHERE   |
where           return WHERE;
ORDER   |
order           return ORDER;
BY      |
by              return BY;
ASC     |
asc             return ASC;
DESC    |
desc            return DESC;
UNIQUE  |
unique          return UNIQUE;
ALL     |
all             return ALL;
DISTINCT |
distinct        return DISTINCT;
INSERT  |
insert          return INSERT;
INTO    |
into            return INTO;
VALUES  |
values          return VALUES;
DELETE  |
delete          return DELETE;
AND     |
and             return AND;
OR      |
or              return;
INT(EGER)? |
int(eger)?      return INTEGER;
CHAR(ACTER)? |
char(acter)?    return CHARACTER;
varchar |
VARCHAR         return VARCHAR;
float   |
FLOAT           return FLOAT;
EXIT    |
exit    |
QUIT    |
quit            return EXIT;
DATE    |
date            return DATE;
SHOW    |
show            return SHOW;
TABLES  |
tables          return TABLES;
```

图 1 LEX 保留的关键字

```
"="     |
"<>"    |
"<"     |
">"     |
"<="    |
">="                    yylval.strval = strdupxy(yytext);       return COMPARISION;
[-+*/:(),.;]                                                    return yytext[0];

'[^'\n]*'       yylval.strval = strdupxy(yytext);       return STRING;
        /* names */
[a-zA-Z][a-zA-Z0-9_]*           yylval.strval = strdupxy(yytext);       return NAME;
        /* numbers */
[0-9]+          yylval.strval = strdupxy(yytext);       return NUMBER;
```

图 2 LEX 识别的字符串和操作符

3.1.2　语法分析器。

Yacc 全称是 Yet Another Compiler Compiler，是一种语法解析器。Yacc 的 GNU 版叫做 Bison。一个 Yacc 程序用双百分号分为三段。 它们是：声明、语法规则和 C 代码。语法规则识别出的内容可以用”$”符号获取，C 代码可以在识别出自定义的语法规则后作为语义动作调用。

本数据库在 parser.h 头文件中，定义了一些结构体来存储元数据和语法书所需的信息。（对于不需存储信息的 SQL 语句直接调用相关函数来执行。）

3.1.3　结构体定义

(1) CREATE TABLE 语句。

AttrInfo 是记录 Create Table 时表的元数据信息，包括表名 table_name、属性名 attr_name、类型 type 以及占用的字节数 used_size。

```
/*
 * for create, store table meta data info
 * create table_name
 *                                        attr1 type1 size1
 *                                        attr2 type2 size2
 *                                        ...;
 */
struct AttrInfo{
        char* table_name;
        char* attr_name;
        int type;
        int used_size;
};
```

结构体 1 存储表列属性信息的结构

(2) DROP TABLE 语句。

无需存储信息，语义动作是根据识别出的表名调用 droptable()函数。

(3) Select 语句。

SelItem 是记录选择属性的结构体，包括所选择属性属于的表名 table_name，列名 col_name 和对应列的 col_id。

```
/*
 * select sellist from fromlist where condlist orderby orrderlist;
 */
struct SelItem{
        char* table_name;
        char* col_name;
        int col_id;
};
```

结构体 2 SelItem

FromItem 结构体是记录从那些表进行选择的结构体里面记录了涉及到的表名 table_name 和 表对应的编号 tb_id。

```
struct FromItem{
        //Relname
        char* tb_name;
        int tb_id;
//      struct FromList *next;
};
```

结构体 3 FromItem

Join 结构体用于存储识别出来的连接条件，当 bison 识别出形式如*.* compare *.* 的语法时，就调用函数将相关的值存入 Join 结构体。

```
struct Join{
        int op;
        char* tb_name1;
        char* tb_name2;
        char* col_name1;
        char* col_name2;

//      struct Join* next;
};
```

结构体 4 Join

Condition 结构体，存储识别出的其他类型条件。

```
struct Condition{
        /*
         * cond1:tb1.col_name1 op tb2.col_name2
         * cond2:tb1.col_name1 op value;
         */
        int op;
        char* tb_name;
        int tb_id;
        char* col_name;
        char* value;
        int value_type;
//      struct Condition* next;
};
```

结构体 5 Condition

SelectQuery 结构体，存储上述结构体组成的数组以及 distinct 和 all 的标识，用于识别 DISTINCT 和 Select*语句。

```
struct SelectQuery{
        int distinct;
        int all;
        SelItem         SelList[MAX_SELITEM_NUM];
        FromItem        FromList[MAX_FROM_NUM];
        Join            JoinList[MAX_JOIN_NUM];
        Condition       CondList[MAX_COND_NUM];
        Orderby         OrderbyList[MAX_ORDER_BY];
        SelectQuery* next;
};
```

结构体 6 SelectQuery

(4) DELETE 语句

WHERE 语句与 SELECT 中的 WHERE 进行类似的处理。

```
struct DeleteQuery{
        char* tb_name;
        Condition       CondList[MAX_JOIN_NUM];
};
```

结构体 7 Delete 查询的结构体

(5) INSERT 语句

记录 Insert 的表格名，解析插入的元组，调用下层提供的 insert 函数。

(6) UPDATE 语句

记录更新的表名，列名，值，类型，条件等信息。

```
struct UpdateQuery{
        char* tb_name;
        char* col_name[256];
        char* col_value[256];
        TYPE type[256];
        int col_count;
        int cond_count;
        Condition       CondList[MAX_COND_NUM];
};
```

结构体 8 UpdateQuery

(7) Frag 语句

需要记录分片类型、站点名称、属性（针对垂直分片）、条件(针对水平分片)。

```
struct FragInfo{
        FRAG_TYPE frag_type;
        char* site_name;
        int cond_count;
        int attr_count;
        Condition CondList[MAX_COND_NUM];
        char* attr_names[MAX_ATTR_NAME_LENGTH];
};
```

结构体 9 FragInfo

3.2 元数据管理模块

3.2.1 元数据写入文件

Metadatamanager 类实现将从解析器获取的结构体信息存入文件。使用 libconfig 库，轻

松将结构体信息写入文件落盘。接口包括保存和删除、分片信息、表信息的函数。

```
Fragment get_fragment_bystr(std::string str){return
void set_fargment_info(Fragment &frg);
void delete_fragmemt_info(std::string str);

void set_siteinfo(SiteInfo &sti);
SiteInfo get_siteinfo_byID(int site_ID){return site
std::string get_IP_by_siteID(int site_ID);

void set_tablemetadata(TableMedata &Tbm);
TableMedata get_tablemetadata(std::string str){retu
void delete_tablemetadata(std::string str);

void setMetadataVer(std::string str);//set the vers
std::string get_metadata_version(){return version;}
```

接口定义 1 元数据写入模块相关函数定义

(1) 表信息按照 3.1.3 中定义的结构体 1 写入文件。

```
table_name_list = [ "publisher", "book", "customer", "orders" ];
publisher :
{
  attr_num = 3;
  ATTR0 :
  {
    attr_name = "id";
    attr_datatype = 1;
    attr_length = 1;
    attr_rulestype = 1;
  };
  ATTR1 :
  {
    attr_name = "name";
    attr_datatype = 2;
    attr_length = 100;
    attr_rulestype = 0;
  };
  ATTR2 :
  {
    attr_name = "nation";
    attr_datatype = 2;
    attr_length = 3;
    attr_rulestype = 0;
  };
};
```

图 3 元数据文件——表信息示例

(2) 站点信息。

```
site_info :
{
  site0 :
  {
    site_ID = 0;
    site_ip = "192.168.8.133";
    site_name = "site0";
    site_port = 0;
    is_control_site = false;
  };
  site1 :
  {
    site_ID = 1;
    site_ip = "192.168.8.134";
    site_name = "site1";
    site_port = 0;
    is_control_site = false;
  };
```

图 4 元数据文件——站点信息

(3) 分片信息。

```
publisher :
{
  db0 :
  {
    isvalid = true;
    con_H1 :
    {
      isvalid = true;
      attr_name = "id";
      attr_condition = "<";
      attr_value = "'104000'";
    };
    con_H2 :
    {
      isvalid = true;
      attr_name = "nation";
      attr_condition = "=";
      attr_value = "'PRC'";
    };
    con_V1 :
    {
      isvalid = false;
      attr_num = 0;
      attr_prikey = "";
      attr_frag_strlist = [ ];
    };
  };
```

```
db1 :
{
  isvalid = true;
  con_H1 :
  {
    isvalid = true;
    attr_name = "id";
    attr_condition = "<";
    attr_value = "'104000'";
  };
  con_H2 :
  {
    isvalid = true;
    attr_name = "nation";
    attr_condition = "=";
    attr_value = "'USA'";
  };
  con_V1 :
  {
    isvalid = false;
    attr_num = 0;
    attr_prikey = "";
    attr_frag_strlist = [ ];
  };
};
```

图 5 元数据文件——分片信息

3.2.2 元数据文件存储及同步

分布式服务框架 zookeeper 负责元数据的存储，保证了不同节点元数据的一致性，确保了节点挂掉重启时可以根据元数据恢复原有状态。

zookeeper的是一个精简的分布式文件系统，它提供了了一些简单的操作和额外抽象的操作。zookeeper一致性的保证需要上层用户自己选择。系统要求元数据的强一致性，否则会造成查询计划的失败。然而不合理的强一致性实现方案会造成庞大的IO开销。我们P2P采用hybrid的方案，客户端每次发送请求时只有leader可以响应其请求、优化查询树并且制定执行计划、协调其他节点执行任务，每个节点都可以成为leader。我们采取的方案是保证协调整个任务执行的leader的元数据实时保证最新以及只有在更新元数据时才进行同步。如图为上层元数据存储的接口，zookeeper节点存储了分布式系统的元数据。主要提供了判断节点是否存在、创建节点、读取节点数据、写节点数据等接口。

```
class ZooKeeper {
public:
  ZooKeeper(const std::string& server_hosts,
            ZooWatcher* global_watcher = nullptr,
            int timeout_ms = 5 * 1000);

  ~ZooKeeper();

  // disable copy
  ZooKeeper(const ZooKeeper&) = delete;
  ZooKeeper& operator=(const ZooKeeper&) = delete;

  bool is_connected();
  bool is_expired();

  //判断节点是否存在
  bool Exists(const std::string& path, bool watch = false, NodeStat* = nullptr);

  //获取节点统计信息
  NodeStat Stat(const std::string& path);

  //创建节点
  std::string Create(const std::string& path,
                     const std::string& value = std::string(),
                     int flag = 0);
  //节点不存在则创建节点
  std::string CreateIfNotExists(const std::string& path,
                                const std::string& value = std::string(),
                                int flag = 0);
  //删除节点
  void Delete(const std::string& path);
  //节点存在则删除
  void DeleteIfExists(const std::string& path);
```

图 6 zookeeper 类及基本函数定义

如下主要为读取与修改元数据的接口，保证了元数据的强一致性。

```
void ZooKeeper::Set(const std::string& path, const std::string& value) {
  Sync(path);
  auto node_stat = Stat(path);
  Println("Set::enter ZooKeeper::Set");
  Println("Set::before zoo_set");
  Println2("Set::path ", path);
  auto zoo_code = zoo_set(zoo_handle_,
                          path.c_str(),
                          value.data(),
                          value.size(),
                          node_stat.version);

  CHECK_ZOOCODE_AND_THROW(zoo_code);
}
```

接口定义 2 Zookeeper 修改元数据文件的接口

```
std::string ZooKeeper::Get(const std::string& path, bool watch) {
  Sync(path);
  std::string value_buffer;
  auto node_stat = Stat(path);

  value_buffer.resize(node_stat.dataLength);
  int buffer_len = value_buffer.size();
  Println2("Get::buffer_len ", buffer_len);
  Println("Get::before zoo_get");
  Println2("Get::path ", path);
  auto zoo_code = zoo_get(zoo_handle_,
                          path.c_str(),
                          watch,
                          const_cast<char*>(value_buffer.data()),
                          &buffer_len,
                          &node_stat);

  CHECK_ZOOCODE_AND_THROW(zoo_code);
  Println2("Get::buffer_len", buffer_len);
  value_buffer.resize(buffer_len);
  return value_buffer;
}
```

接口定义 3 读取元数据的接口

分布式数据库系统的元数据主要分为两类，第一类是不常更新的元数据，比如节点配置信息、表的结构信息、切片信息等。常常更新的信息是用于查询计划优化的统计信息。我们按照一定层次将此两类信息分别存储，减小了IO开销。

3.3 查询树生成及优化

3.3.1 本部分主要工作及输入输出规定。

本部分主要工作是根据 SQL 生成查询树并提供图形界面展示。输入是一个 select 语句和元数据信息。select 语句格式要求：

单词之间空格最多为 1 个，不允许有多余空格；

where 之后的条件部分要求连接条件放在前，选择条件放在后；

条件与条件之间要打上括号；

默认输入的语句是正确的，排错交给解析器处理；

不支持 or 连接条件，原因是考虑合取范式的处理比较复杂，作业要求中没有 or 条件。

总代码量：约 2000 行。

3.3.2 生成带有分片的基本查询树

由于预先指定了 SQL 的语法格式，所以语义解析可以显然可以很容易地做到，首先通过字符串处理，对各部分进行分割。首先生成一颗不带分片的查询树，按如下几步进行操作：

（1）首先根据 SQL 中的表名，把所有需要查询的表插入到树中

（2）根据连接条件，生成一个 join 节点，并连接两个需要做连接的表。（如果一个表已经被 join 节点连接了，再连接另一个 join 节点时，也就是多表连接可能出现的情况，需要用这个表已经连接的 join 节点，也就是祖先，去连接另一个 join 节点）

（3）此时所有表都有一个公共的 join 节点作为祖先，此时可以把所有选择条件放到 join 节点的上方，每个条件作为一个节点插入到树的最上方。

（4）将投影操作选择的属性插入到所有节点的最上方。

（5）此时已经生成了基础的查询树，然后将所有表拆分为分片并用 union 符号连接即可。注意，如果是混合分片的话，需要对相同水平分片条件的分片进行 join 操作，再将 join 操作后的不同水平分片 union 起来。

例子（今后都会使用这样一个很复杂的 SQL 语句，以便有说服力，表如何分片参考 benchmark 中支持水平+垂直部分的分片格式）：

select * from emp,job,sal,asg where (emp.title=sal.title) and (emp.eno=asg.eno) and (job.jno=asg.jno) and (emp.eno<'E0010')

生成的基本查询树如下图：

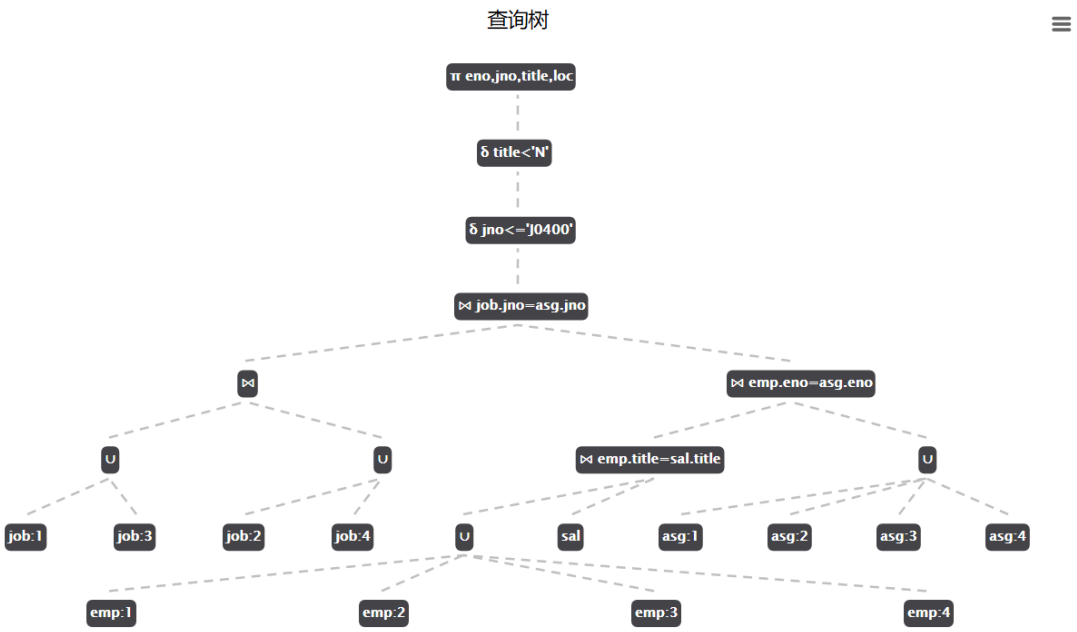


图 8 查询树示例——基础版

3.3.3 查询树的优化

查询树优先度最高的优化有：1.查询条件下推 2.投影条件下推 3.join 下推，还有一些执行上的优化，比如选择数据少的站点进行传输等，由于本部分不包含数据字典的设计，所以这方面的优化归到 SQL 执行部分。

以下几节讲述如何进行这些优化，并且根据问题使用了一些很 tricky 的技巧。

(1) 查询条件下推

对于每个表中的条件，考虑是否将其下推：

循环扫描每个分片：

1.如果这个分片没有这个属性，不下推

2.如果有这个属性的情况下。分片条件与查询条件不冲突，则下推；否则，则在这个分片上不会有结果，删除这个分片。

注意：删除分片后要及时清理树，例如下面例子中 job 删除了两个分片，会导致上面的 union 节点只有一个儿子，那么这个 union 节点也要删掉。

select eno,jno,title,loc from emp,job,sal,asg where (emp.title=sal.title) and (emp.eno=asg.eno) and (job.jno=asg.jno) and (jno<='J0400') and (title<'N')

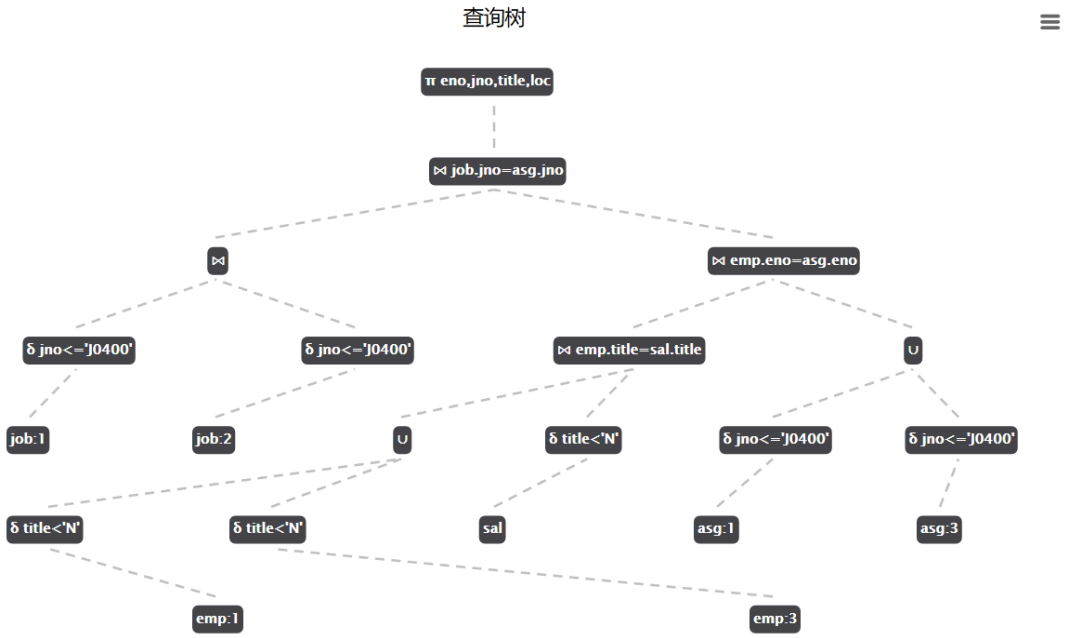


图 97 查询树示例——条件下推

(2) 投影下推

投影下推需要考虑以下几点：这个分片是否有投影属性，并且需要考虑保留将来 join 操作中出现的属性，于是分为以下几步：

(a) 看分片是否具有这个属性，有则要保留

(b) 对于每个分片，观察其上方所有的 join 条件中的属性，join 条件中的属性如果自己拥有则要保留。

(c) 比较 tricky 的地方是，对于相同表分片间的 join，如果一个分片只投影了一个属性，则这个分片肯定是用这个属性做 join 的，并且在这个表上没有选择条件，这种情况可以删掉分片。如果是不同表的分片的话，由于表中的数据可能不同，并不是一个表上

的分片，join 可能并不完全一致，所以还是不能删除的。

考虑过是否将下推后的投影节点，也就是最上方的节点删除的问题，由于连接条件有时会产生不必要的投影属性，所以保留是很必要的，而且并不会过多影响效率。

例子：

select eno,jno,title,loc from emp,job,sal,asg where (emp.title=sal.title) and (emp.eno=asg.eno) and (job.jno=asg.jno) and (jno<='J0400') and (title<'N')

没有使用(c)：

图 80 查询树——投影下推（1）

使用(c)后，（job 分片移动到右侧了），job:1 是没有必要保留的，因为它与 job:2 做 join 结果还是 job:2。

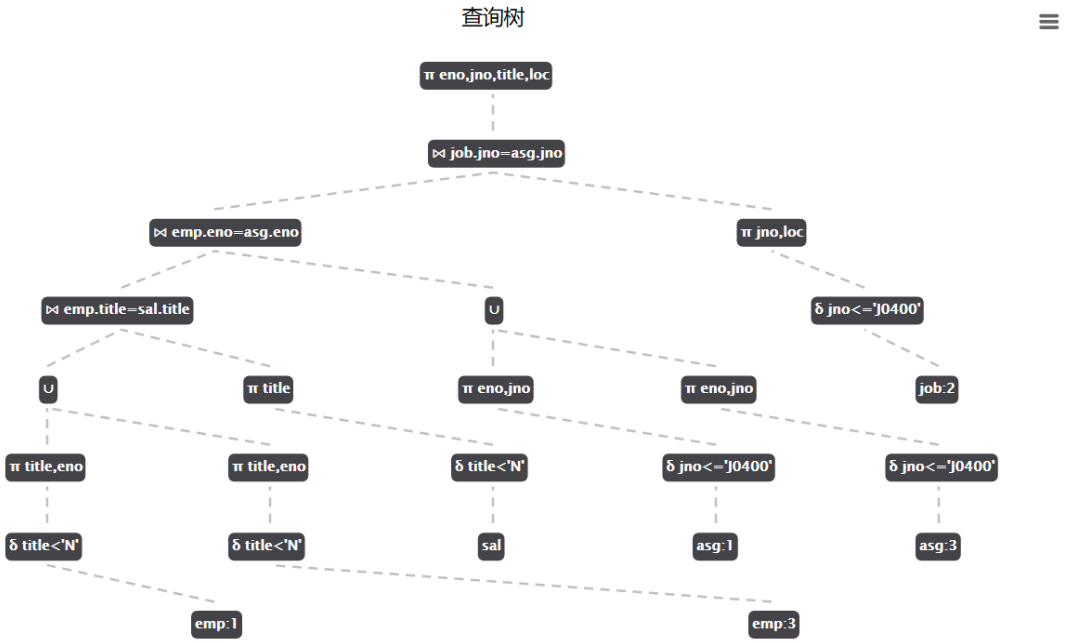


图 91 查询树示例——投影下推（2）

(3) join 下推

这节应该是这部分最难写的部分。实现的主要思想为：后根遍历整棵查询树，也就是先下推下方的 join 节点，对于每个 join 节点，处理时，它下方的 join 节点已经下推过了。

对于一个 join 节点，做如下处理：

a) 因为一个 join 节点肯定有且仅有两个儿子。首先统计这个节点左右儿子分别有多少个分片需要做连接，设分别有 L 和 R 个分片。（这里解释一下，如果 join 节点的儿子是 union 节点，则表示有多个分片需要进行连接，如果不是 union，则代表只有 1 个分片需要连接）。

b) 假如 L 和 R 都等于 1，即是普通的连接，没有下推的必要。若其中有一方大于 1，则将每个左儿子中的分片与右儿子中的分片作 join，并将最后的结果 union 起来。

c) 需要注意的是，如果 SQL 比较特殊，也就是 join 下推后树会特别大的情况，例如那种由于没加限制条件成指数级爆炸的树。为了保证程序的健壮性，会对下推产生的节点数进行准确计算，如果超出了树的容量，则不进行下推。

查询例句：

select eno,jno,title,loc from emp,job,sal,asg where (emp.title=sal.title) and (emp.eno=asg.eno) and (job.jno=asg.jno) and (jno<='J0400') and (title<'N')

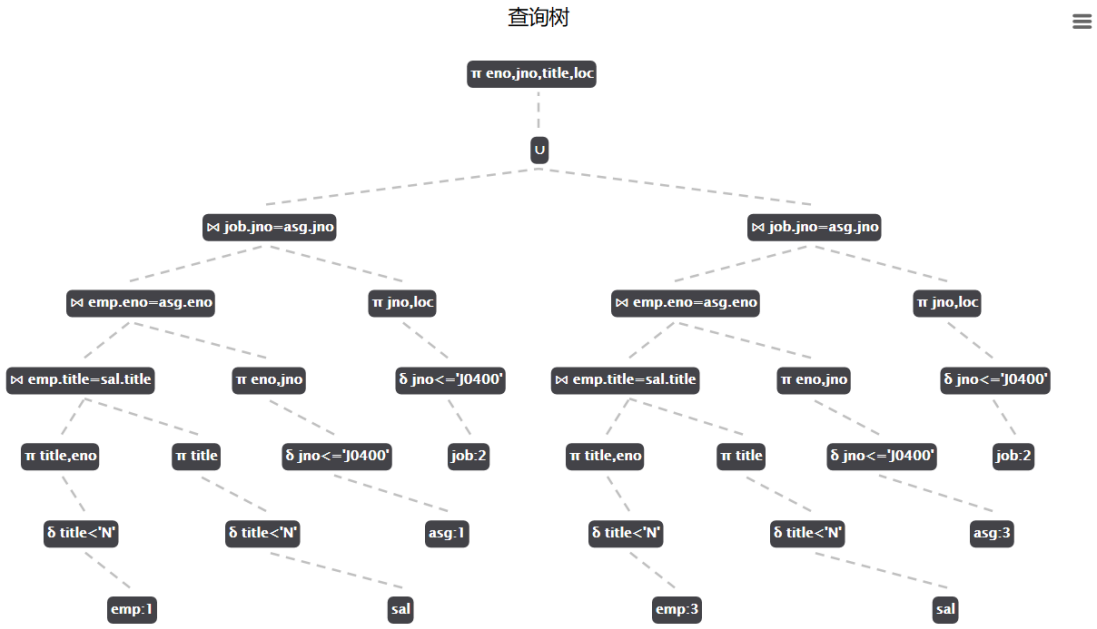


图 102 查询树示例——连接下推

3.3.4 查询树展示

使用了 highchart 的 js 库作为框架，这个框架跟 canvas 类似，需要自己指定坐标。具体实现的方法是：由 c++将生成的树的结构信息输入到 tree.txt 中，用 jsp 读取文件之后，调用 js 绘制到网页上。

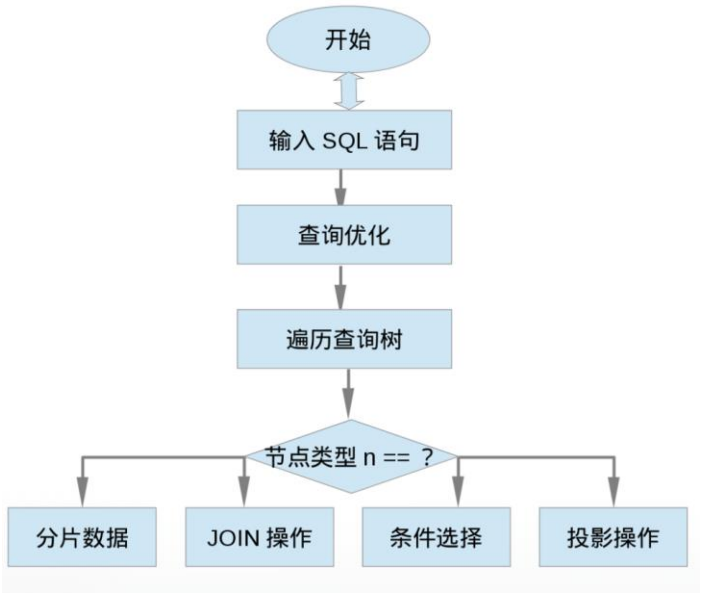
绘图主要是将同一深度的节点画在同一水平线上，并将网页的宽度平均分配给这些节点。

并通过树的指针来进行连线。

3.4 查询执行

3.4.1 总体思想

针对查询优化给出的树结构进行遍历，并生成底层数据库支持的 SQL 语句，调用通讯模块提供的 RPC 函数执行，并回收查询结果进行处理。



流程图 4 查询执行流程图

3.4.2 主要函数定义及功能

(1) void set_treevector_from_querytree(query_tree &qtree);

功能：将生成的优化后的查询树，通过借用 std::vector 容器，完成从根节点到叶节点的遍历，利用 vector 构造栈结构实现从叶节点到根节点的遍历顺序，由于查询树的叶节点为分片信息，执行的第一步即是获取分片数据，同时根据其优化情况，从远端站点尽可能本地选择及投影操作，减少网络传输的数据量。

(2) void exec_SQL_query(std::string SQL);

功能：借助 vector 容器，重新改写 SQL 语句。由于远端传输的分片数据采用本地临时表的方式存储，就需要借助 MySQL 相关工具，在本地创建各表对应的临时表。对于垂直分片的表，进行本地连接操作完成查询。

(3) std::string get_hash_fromstr(std::string str);

功能：由于查询树节点当中包含有特殊字符，有可能 MySQL 在创建本地临时表时无法识别产生错误，故借用 std::hash<string>容器，实现构造特殊表名到 hash 值的映射，也方便临时表的统一管理。

(4) std::string& replace_SQL_tablename_byhash(string& str,const string& old_value,const string& new_value);

功能：原始表名替换为经过 hash 之后的名字，将 SQL 转化为对临时表的查询操作。

3.5 网络通信

leader采用远程过程调用(RPC)的方式来让其他节点执行任务并返回相关结果，RPC隐藏了底层网络的细节，调用者以一种类似于本地调用的方式来得到远程节点函数执行的结果。我们基于rpclib来设计我们的网络通信的接口，设计接口友好为查询计划执行的程序员用

户提供了便利。优化的网络通信的接口测试大数据量的传输速度很快，提升了整个系统的速度。

如图为向上层提供的主要网络通信接口

```
bool RPCExecute(string host_ip, string sql_statement);

string RPCExecuteQuery(string host_ip, string sql_statement);

int RPCExecuteUpdate(string host_ip, string sql_statement);

bool RPCInsertFileToTable(string host_ip, string sql_file, string table_name);
```

接口定义 4 网络通信接口

3.5.1 mysql本地执行

调用mysql C++接口，向上层用户提供了本地sql执行的接口。 如图主要提供了想本地mysql创建、删除与更新表、查询、load数据到表等接口。

```
bool localExecute(string sql_statement){
    cout << "sql_statement " << sql_statement << endl;
    cout << "localExecute(string sql_statement)" << endl;
    MySQL_Driver *driver;
    Connection *con;
    Statement *stmt;
    driver = sql::mysql::get_mysql_driver_instance();
    cout << "   driver = sql::mysql::get_mysql_driver_instance();" << endl;
    con = driver->connect(getMySQLIp(), getUsername(), getPassword());
    con->setSchema("test");
    stmt = con->createStatement();
    cout << "stmt = con->createStatement();" << endl;
    bool ok = false;
    cout << "bool ok = false;" << endl;
    assert(sql_statement.size() > 0);
    ok = stmt->execute(sql_statement);
    cout << "ok " << ok << endl;
    delete con;
    delete stmt;
    cout << "delete stmt;" << endl;
    return ok;
}
```

函数定义 1 mysql 本地执行语句函数

```
int localExecuteUpdate(string sql_statement){
    MySQL_Driver *driver;
    Connection *con;
    Statement *stmt;
    driver = sql::mysql::get_mysql_driver_instance();
    con = driver->connect(getMySQLIp(), getUsername(), getPassword());
    con->setSchema("test");
    stmt = con->createStatement();
    int cnt = false;
    assert(sql_statement.size() > 0);
    cnt = stmt->executeUpdate(sql_statement);
    delete con;
    delete stmt;
    return cnt;
}
```

函数定义 2 本地执行更新函数

```
string localExecuteQuery(string sql_statement){
    MySQL_Driver *driver;
    Connection *con;
    Statement *stmt;
    ResultSet  *res;
    driver = sql::mysql::get_mysql_driver_instance();
    con = driver->connect(getMySQLIp(), getUsername(), getPassword());
    con->setSchema("test");
    stmt = con->createStatement();
    res = stmt->executeQuery(sql_statement);
    int columnCnt = res->getMetaData()->getColumnCount();
    string result;
    while (res->next()){
        for (int i=1;i<=columnCnt;++i){
            result.append(res->getString(i));
            result.append(",");
        }
        result.append("\n");
    }
    return result;
}
```

函数定义 3 本地执行需要返回结果的 SQL 语句函数

```
bool localInsertFileToTable(string sql_file, string table_name){
    string file_name = "tmp";
    ofstream tmp(file_name, std::ios::out);
    tmp << sql_file;
    tmp.close();
    string sql_statement =
    string("LOAD DATA LOCAL INFILE ") +
    "'" + file_name + "'" +
    " INTO TABLE " + table_name +
    " FIELDS TERMINATED BY \t " +
    "LINES TERMINATED BY '\n'";

    MySQL_Driver *driver;
    Connection *con;
    Statement *stmt;
    driver = sql::mysql::get_mysql_driver_instance();
    con = driver->connect(getMySQLIp(), getUsername(), getPassword());
    con->setSchema("test");
    stmt = con->createStatement();
    bool ok = false;
    assert(sql_statement.size() > 0);
    ok = stmt->execute(sql_statement);
    delete con;
    delete stmt;
    return ok;
}
```

函数定义 4 本地执行 Insert 语句函数

4. 个人总结

4.1 汪弘洋

项目伊始，我就预料到工作量可能比较大，所以 benchmark kick off 一出来就开始计划自己的部分。由于当时作业的要求并不具体，我在初始设计时还多考虑了一些情况，如：连接条件中有“OR”的情况，分片条件无限切的情况。之后，老师明确要求参考 benchmark 中的功能要求，最终我就没有考虑上述提到的情况。这使我减少了一些工作，代码实现也顺利了不少。结果，我负责的部分早于 DDL 两周就可以使用了，这为我的组员争取了很多时间。当他们在熬夜工作的时候，我的部分也不至于成为瓶颈。总体来说，我觉得提早制定计划，并按时执行是很明智的决定，也是很良好的习惯。

其次，我很注意自己部分代码的完成质量，模块的输入输出要求定义的很准确。虽然最后和组员对接有些小问题，但是也没有感到太多压力。为了方便调试，查询树的可视化也由我负责。在交付给组员之前，我将 benchmark 里的 SQL 语句都测试了一遍，并且反复确认是没有错误的。为队员的测试提供了便利。

最后，从完成情况来看，我认为我们组完成的进度比另一组稍好一些。总结原因可能是因为本组人少，且每个人分工比较独立，目标也很定的比较实际且容易实现。

当初我是因为想认识认识别的实验室的同学才提出加入他们的，结果看来还是不错的，几个队员都是比较很友善、有趣的人，完成作业的过程比较愉快，总体感觉做大作业

的热情还是比较高的。通过完成这次作业，巩固了之前就比较熟悉的编程语言c++,jsp,js。虽然实现的是一个比较简单的分布式数据库，但是还考虑了非常多细节，从而对分布式数据库实现上的难题有了进一步认识，也为之后实现规模更大的数据库奠定了一些基础。此外，通过这节课认识了几个朋友，也是比较开心的事情。

4.2 李昊华

总体来讲，我在这门课学习了分布式系统的知识、提升了coding能力、锻炼了团队合作的能力。

在大作业中，我主要负责网络通信模块的实现、元数据存储与一致性、mysql本地化执行的设计等部分。网络传输部分对其他模块的依赖度相对较低，因此学生在课程中后期就集中精力写完了网络通信的模块。在网络通信模块的过程中积累了网络的知识。网络通信模块主要是底层网络协议需要扣细节、比较琐碎，需要为上层的用户封装底层的细节，向上提供易用的接口。

系统要求不同节点的元数据保持强一致的特性，否则会导致查询计划执行失败，zookeeper本身并不保证存储数据的强一致性，需要上层用户选择一致性的类型。比如保证强一致性最笨的方法就是每次读取或者写入元数据之后都同步一遍，这种方案的系统开销是非常大的，为了保证数据的强一致性，我们采取的方案是保证协调整个任务执行的leader的元数据实时保证最新以及只有在更新元数据时才进行同步。学生向上层提供的接口保证了元数据的强一致性和低开销。

Mysql本地执行的接口部分不做赘述，在小组中学生的任务完成的比较早。后期承担了部分衔接项目不同模块的debug工作，以及部分运维工作。组员都很靠谱很给力，合作很愉快。传雯组长认真负责，柴师兄代码给力，弘洋同学远在深圳也时刻关注项目的进度，一直写代码做贡献。考核前晚上一起去明法楼通宵到第二天早上的经历令人印象深刻。另外一组同学给人留下认真、聪慧、友好的形象，和他们的技术交流彼此颇有收获。选择上这门课收益很大，对个人偏向分布式存储的专业方向也很有帮助。

4.3 柴艳峰

经过一个多月的团队合作，克服重重困难，比较圆满地完成了分布式数据库项目作业，结束验收之后，感想颇多，特此记录。

首先，团队合作首先要克服的问题就是1+1<2的问题，即时团队中每个人的个人能力都很突出，但在团队项目中往往会出现整体效果难以尽如人意的现象。我们小组共有4人，每个人的编码能力、对项目的理解能力都有不同。这就面临着1+1+1+1<<4的问题，那么最重要的沟通问题就必须解决。而沟通，就需要Leader从中协调和把握各个队员之间的连接，在这一点上，选择传雯作为队长是正确的，发挥了Leader应有的作用，在各个关键节点上选择了最优的策略，对整个项目的推进至关重要。

其次，在项目连接过程中，各个模块的对接与调试是一个相当繁琐和麻烦的过程，在这过程当中更需要每个组员的无私奉献，面对未知的bug和停滞不前的进度，组员之间并没有选择互相埋怨，而是选择互相鼓励，为彼此积极探索解决思路。可以说，每个组员之间都是平等而友好的。在此次分布式数据库当中，弘洋的部分相对比较独立，由于前期GDD定义规范较晚，他个人利用模拟数据进行查询优化部分的推进，较早地完成了查询优化部分，为我们后期争取了较多的调试时间，为顺利通过验收打下了良好的基础。在网络通讯

部分，昊华积极学习新知识，利用RPC通讯较好地解决了各个站点间的数据传输，并在后期调试阶段搭建实验环境付出了很多的心血，这些都是为项目顺利验收做好了坚实的保障。

最后，我个人在与队员相处的过程当中，也再一次锻炼了自己的代码能力和学习新知识的能力，很多编程的经验必须经过实战才能真正领悟其中真谛。

在与队员的合作过程当中，真正体会到了一起通宵战bug的快乐，可以说为我刚刚开始的学习生活开了一个好头。感谢范老师在本学期的悉心授课，感谢团队成员的热心帮助，虽然在过程当中走了很多弯路，但是这段经历却是让我一生难忘的回忆。

4.4 王传雯

通过完成这个“简陋版”分布式数据库，我掌握使用bison,lex解析工具、libconfig以及编写makefile等程序员必备技能，了解了如何使用C++连接数据库，学习了分布式数据库执行查询的基本流程，在最后的验收中也收获了合作的成就感和满足感。

首先，先对我负责的代码部分行简单总结。我负责除select语句之外的所有语句的处理。如报告中所述，不同的语句需要考虑不同的执行情况，设计到元数据保存、检查，根据分片进行插入、删除等。其中难度最大的是实现分片语句解析，数据生成，下发。在项目后期，我主要的工作都落在使用RPC执行语句和调用元数据管理模块存储元数据上。这两个都是在解析器的部分进行调用，我感觉压力还是很大的，好在柴艳峰和李昊华给予我很多帮助。

其次，我从团队合作中真切感受到了代码模块连接的难度。本组最后完成验收得益于两点，一个是老师及时跟进进度并时常打预防针，解决了部分拖延症的问题【我认为老师可以把验收时间再提前一点时间，可能帮助更大】。另一个是小组成员都很靠谱：汪弘洋负责的查询计划生成、优化部分高度独立，且完成很早，为我们之后的对接奠定了基础；李昊华负责的网络通信模块（Zookeeper和RPC调用）质量比较高，几乎没出什么bug，在对接的时候，直接静态编译成链接库了；柴艳峰负责的查询执行部分需要与汪弘洋连接，难度大，开始的时间较晚，从而压力也大，但最终也保质保量的完成了。更重要的是，三个队友脾气都比较温和，对我提出的各种要求和疑问都能耐心解答。

最后，老师的教课方式也充分体现出小班教学的优势。选这门课的很大部分理由是因为人少。当时选课时还有另外一门分布式的课程，也属于本专业的专业必修，但是人太多，以本人的性格肯定会“水”过去。于是，我果断选择了范举老师的这门课，果然这门课真是一点也不“水”。范老师认真负责，对每个组给予了充分关注，为了这门课我贡献了本学期四次熬夜中的两次，欣赏了人大独具特色的冬日星空。

另外，搭配数据库系统实现和操作系统这两门课，会进一步加深对数据库和分布式的认识。

总结一下，选这门课，值。