# DDL

## 表结构缓存

函数：join/metacache.cpp

每次DQL、DML涉及的表都会在proxy通过show create table、show create view获取每个表的表结构信息，然后缓存本地proxy，下次会将当前DB获取的元数据信息与缓存的比对，如果不一致需要更新。

表结构发生变动时，zk节点会将更新的数据同步到计算引擎proxy的路由文件中，使得计算引擎使用的是最新的元数据。

同时，后台也有一个专门的线程负责元数据的更新，每间隔20s更新一次。

## 分区表达式缓存

函数：join/expr\_eval.cpp

在创建表的时候，涉及的分区表达式会在proxy进行缓存，后面如果在插入数据或者查询的时候可以通过分区表达式缓存快速计算具体的分区信息。

## 嵌入式临时表字段裁剪

计算引擎通过配置项enable\_column\_cutoff控制是否加载所有的字段到嵌入式MySQL的临时表，这样可以减少不必要的字段插入，减少计算引擎与数据节点网络负载以及本地存储的空间大小。

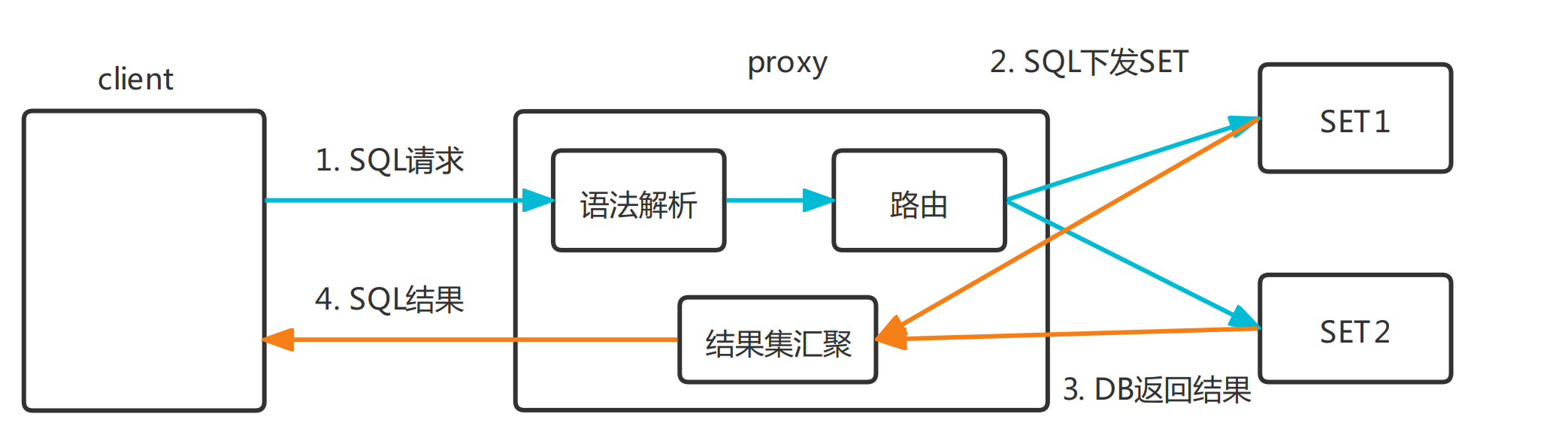
## Online-DDL

# 查询

对于SELECT查询有两种处理模型：一种是流式处理模型，一种是通用处理模型。

流式处理模型主要是处理单表上的查询，而通用处理模型是对流式处理模型的弥补，负责处理分布式的跨节点查询。

## 简单查询

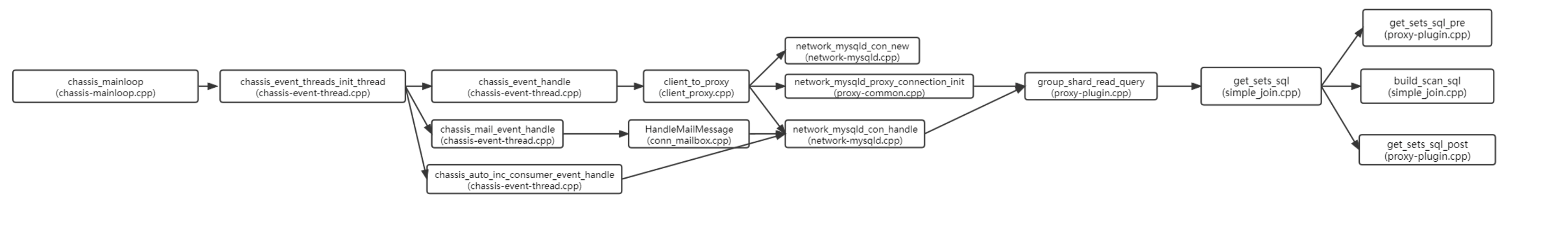


特点：

1. 单表可以下推DB或者多表连接，关联字段为分区键；
2. 同一连接中，采用**流式**处理方式，必须处理完当前SQL，才能继续处理下一条SQL。

简单查询SQL，主要通过get\_sets\_sql设置下发SET的SQL，其中主要通过get\_sets\_sql\_pre，build\_scan\_sql和get\_sets\_sql\_post实现。

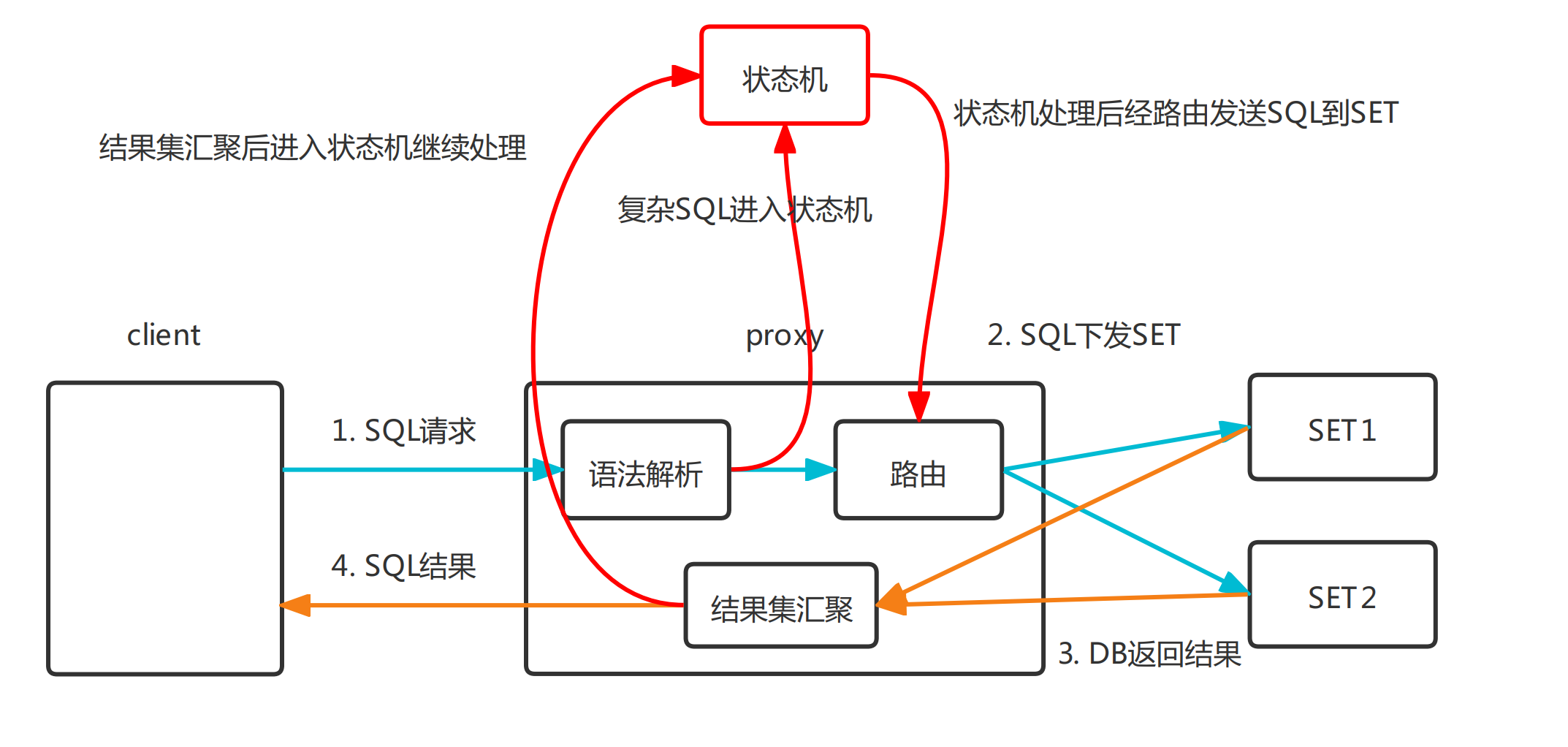
基本流程如下：



## 复杂语句

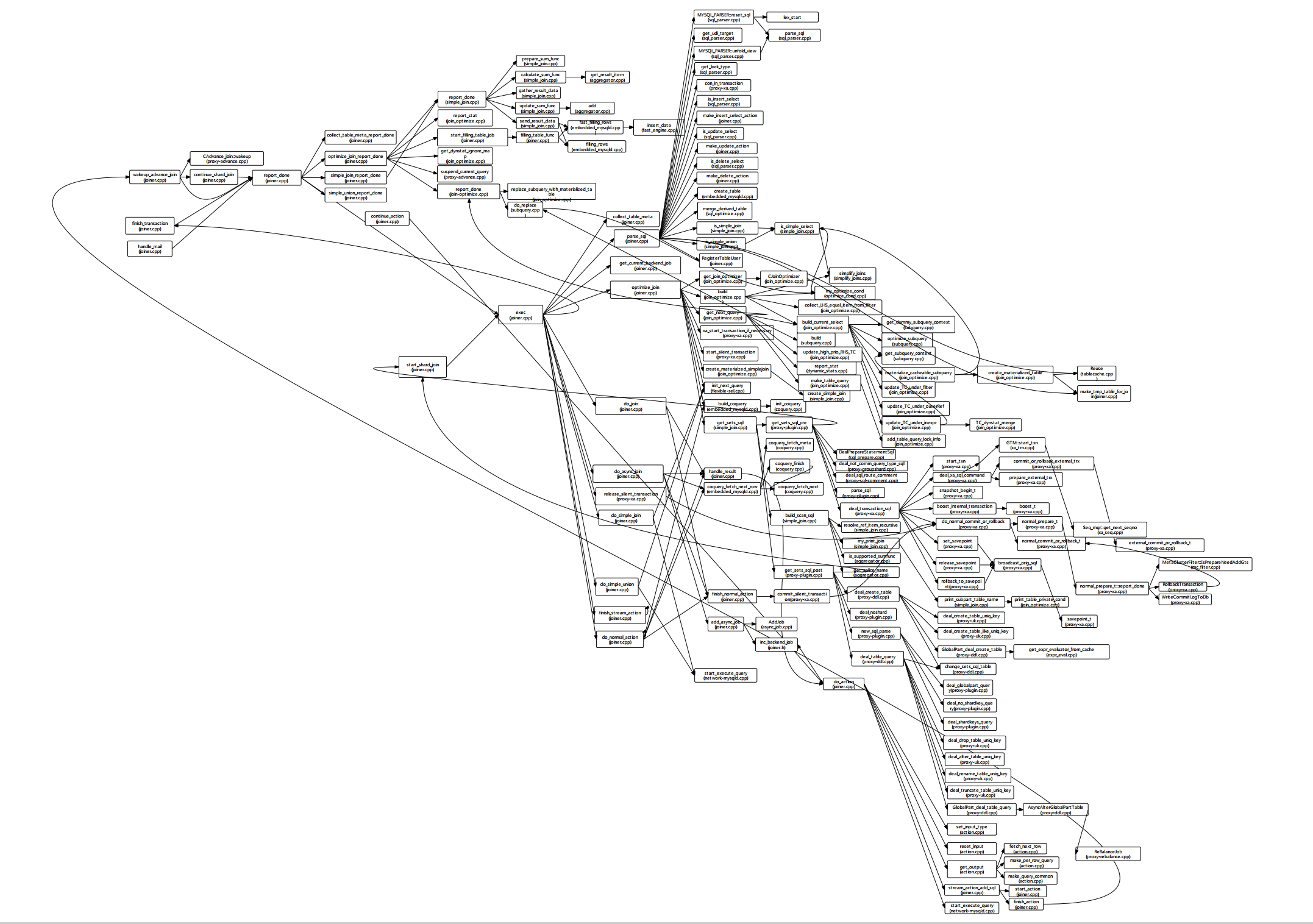
复杂SQL的执行借助状态机实现状态的转换和执行。

基本流程如下：



特点：

1. 无法直接通过SQL下推到DB，需要在proxy进行计算（即需要使用嵌入式）；
2. 对于结果集采用通用处理模式，不需要像流式处理模型那样顺序执行。



从SQL执行顺序，依次优化为：

### 视图展开

joiner\_t::parse\_sql()：

// 将所有的视图进行展开

if (!participate\_view.empty()) {

// 解析SQL

parser->reset\_sql(sql, false, table\_name\_lower\_case, print\_cs\_introducer);

if (parser->is\_error()) {

THROW\_JOIN\_ERROR(parser->get\_error());

}

……

### FROM优化/derived table

函数：sql\_optimizer.cc/merge\_derived\_table

Derived table（虚表）的优化主要两种思路：

1. 子查询下推：将主查询和子查询下推到DB，减少在proxy层的计算

FROM子查询可以合并的场景需要同时满足：

1. 子查询不存在分组聚合
2. 子查询不存在distinct
3. 子查询不存在limit
4. 使用一个或以上的表
5. 子查询上提：将子查询中的排序字段上提，提高运算效率（消除derived table）

举例：

select count(1) from (select a+b as result from t1) tmp where result<10;

->

select count(1) from t1 where (a+b) < 10;

满足如下条件，将derived table子查询中的排序上提：

1. derived table中不能是视图
2. 上层select不是union
3. 上层select没有group by、聚合函数；
4. 上层select没有distinct；
5. 上层select没有limit；
6. 上层select只包含derived table一个表（不存在join）
7. 表的数量≥1（不能是纯虚表）
8. 不能引用用户变量（值会发生变化）

TODO：对于select 聚合函数 from (select ….)这种场景，如果虚表是广播表或者存在shardkey的条件查询，是存在继续优化的空间的，可以下推。

### JOIN优化

相关函数：joiner\_t::optimize\_join()，joiner\_t::do\_join()

#### 下推

函数：is\_simple\_join--> is\_simple\_select

ON关联条件的等值链，进而判断是否可以下推。

#### 消除外连接/flatten nested join

函数：simplify\_joins .cpp/simplify\_joins

left join -> inner join

**左右连接的消除规则：**

左连接的右表(B)如果出现在过滤条件中“严格函数”中(B.y2=321)（即在left join中可能为NULL的列在表达式中已经排除了，即nullable side不可能为NULL），则左连接退化成内连接；同理，右连接的左表如果出现在过滤条件的“严格函数”中，则右连接也会退化成内连接。同时，ON关联字段添加到WHERE条件查询中。

SELECT \* FROM t1 LEFT JOIN t2 ON t2.a=t1.a WHERE t2.b < 5

-->

SELECT \* FROM t1 INNER JOIN t2 ON t2.a=t1.a WHERE t2.b < 5

-->

SELECT \* FROM t1, t2 ON t2.a=t1.a WHERE t2.b < 5 AND t2.a=t1.a

#### 异步化处理

对于需要在嵌入式创建临时表处理的情况，如果多表Join的结果集比较大，则需要分批次进行数据的插入和查询，这种就是关联操作的异步化。

### WHERE优化

#### 过滤条件下推

gather\_equal\_fields（收集where中等值关系）->check\_relation\_pushdown（等值链合并，根据等值链与shardkey检查是否可以下推）

举例：

Select \* from t1 join t2

on t1.a=t2.b

where t1.b=1 AND t1.c=2

拆分路由后下发DB的SQL：

Select \* from t1 where b=1;

Select \* from t2 where c=2;

#### 条件推导

等价类：

查询：select \* from t1, t2, t3 where t1.a= t2.b AND t2.b=t3.c AND t1.a=10;

=>

select \* from B where t2.b=10;

select \* from C where t3.c=10;

#### 表达式缓存

#### 关联条件推导

#### 异步后台job

### ON优化

#### 谓词化简

函数：my\_optimize\_cond

剔除无用的谓词。

#### 内连接条件下推

函数：

get\_inner\_tbl\_on\_expr

collect\_pushable\_cond

Select \* from t1 join t2

on t1.b=1 AND t2.c=2

where t1.a=t2.b

与前面where条件下推一致。

#### 左连接条件下推

（1）左/右连接的连接条件可以下推给连接的“nullable side”。

（2）过滤条件可以下推给连接“nonnullable side”。

### 子查询

#### 子查询消除

#### 子查询下推

**函数：**SUBQUERY\_CONTEXT::optimize\_subquery

通过等价类和上下文推导的方式判断是否可以下推，可以合并下推的情况：

1. 子查询不能带有group by、聚合函数、having
2. 非single row类型不应该带有聚合函数

举例：

SELECT … FROM t1 WHERE diskey1 IN (SELECT diskey2 FROM t2);（分片键关联）

#### 子查询字段裁剪

函数：remove\_redundant\_subquery\_clauses

a) SELECT \* FROM t1 WHERE t1.a = (<single row subquery>)

b) SELECT a, (<single row subquery) FROM t1

注：移除group by的情况：没有聚合函数和having表达式

#### 子查询上下文构建

函数：

SubqueryOptimizer::get\_subquery\_context

-->SubqueryOptimizer::build\_subquery\_context\_from\_join\_list

-->build\_subquery\_context\_from\_cond

#### 物化子查询

函数：

CJoinOptimizer::build\_current\_select()

-->CJoinOptimizer::materialize\_cacheable\_subquery

不能优化的非关联子查询

#### ANY/ALL

**函数：**SUBQUERY\_CONTEXT::optimize\_subquery

**优化（基于规则）：**

any/all->min/max

### GROUP BY优化

#### 下推/消除加载操作

函数：simple\_join\_t::build\_scan\_sql

下推规则：

1. 如果存在group by分组字段，则将分组字段加入到排序字段中；
2. 如果group by分组字段是select list的子集，那么不需要在临时表去重；
3. 所有聚合函数都有distinct，则下推distinct；
4. 所有聚合函数都没有distinct，则下推group by和聚合函数。

### 聚合函数

函数：simple\_join\_t::build\_scan\_sql、is\_supported\_sumfunc

**下推规则：**

1. 聚合函数带变量（类似sum(@a:=@a+1)）不能下推（is\_supported\_sumfunc）；
2. 不支持的聚合函数不能下推（build\_scan\_sql）。

#### 下推

TDSQL对于shardkey聚合函数或者其他函数相关计算优化下推还是不足。

#### AVG优化

**函数：**agreator.cpp/Aggregator\_quick::get\_splice\_name

**优化：**

avg=sum/count

### HAVING优化

函数：simple\_join\_t::build\_scan\_sql

### SELECT优化

#### 列裁剪

cutoff\_unused\_columns\_from\_tables创建临时表，查询的时候可以直接使用select \*

### DISTINCT优化

#### 下推/消除加载操作

函数：simple\_join\_t::build\_scan\_sql

1. 如果所有的聚合函数都带有distinct，则将distinct下推；
2. 如果没有group by分组字段，且没有聚合函数，则将distinct下推；
3. 如果没有group by分组字段，但存在聚合函数，结果集单行的情况，则忽略distinct。

### ORDER BY优化

#### 下推/消除加载操作

函数：

simple\_join\_t::build\_scan\_sql

下推规则：

1. 非单行结果集，没有使用临时表去重，且order by排序列是group by分组列或distinct去重列的子集，则下推排序操作；
2. 非单行结果集，没有使用临时表去重，没有group by，没有distinct，没有聚合函数，则下推order by排序列；
3. 其余情况，利用临时表对结果集进行排序。

### LIMIT优化

#### 下推

函数：simple\_join\_t::build\_scan\_sql

下推规则：

1. 不使用临时表，且不存在聚合函数（或者聚合函数可下推），且不存在having

### UNION优化

#### 下推

函数：is\_simple\_union

说明：parse\_sql调用，即在执行语法解析的时候判断union是否可以进行下推。

**下推规则：**

1. 包含不能下推的用户自定义变量不能下推；
2. 子查询不能下推，union不能下推；
3. 子查询下推方式是否一致，不一致不能下推；
4. 子查询group by分组列不包含partition by分区字段，不能下推；
5. 子查询不存在group by分组时，如果存在聚合函数或者having，则不能下推；
6. 子查询包含order by，不能下推；
7. 子查询包含limit，不能下推；
8. 如果包含distinct，或者为union语句，则结果字段必须要求包含分区字段。

#### 收包数调整

默认proxy从DB收取SQL结果集的条数是4000，可以根据具体数据量调整，减少网络开销。

### 全局唯一索引

### 锁优化

采用乐观锁和悲观锁并存的方式控制并发，可以灵活控制。如果是乐观锁则采用lock in share mode，否则采用排它锁for update。

此外，还实现了无锁的并发控制，即分布式MVCC，采用全局的gts时间戳控制并发。

### 执行器/嵌入式临时表

这里执行器主要是控制多set情况下的异步执行，比如UNION ALL的并发，异步查询结果。

注：这里与TiDB和ShardingSphere的DistSQL模块功能类似。

#### 异步查询

#### 嵌入式数据库

**函数：**embedded\_mysqld.cpp

采用嵌入式临时表（MyISAM存储引擎）暂存结果集，并且会定期清理，proxy的嵌入式处理时采用协程异步化处理，提高效率。

## Hint

# DML

## 下推

对于DML操作需要判断参与表的分片规则，根据表级别分片规则判断是否可以合并下推。

## 拆分

如果是走嵌入式的场景，比如INSERT SELECT语句涉及多条记录的时候，会根据每一行的结果计算setname，然后根据setname的情况，将多个共同下发到一个set的INSERT合并，减少不必要的网络IO。

**函数：**action.cpp/ Action\_base::get\_output

将update/insert/delete拆分为select->insert/update/delete操作。