

CBASE 功能开发手册

版本信息：CBASE1.2

本手册旨在向CBASE使用人员或者对CBASE感兴趣的数据库开发者，介绍CBASE系统在开源的OceanBase 0.4.2版本基础上新增的功能。

如果想要了解OceanBase的开源版本，请参考OceanBase相关文档。

注：本手册参考OceanBase 0.4.2的相关开源文档。

文档说明

CBASE1.2在OceanBase 0.4.2的基础上新增了如下11个功能模块：

- Decimal
- 二级索引
- Bloomfilter Join
- Semi Join优化
- Sequence
- 批量（插入、删除、更新）
- In子查询
- 主键更新
- 集群管理
- UPS日志同步及优化
- RS/UPS选举

Decimal

1 需求分析

CBASE是基于OceanBase数据库研发的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。为满足业务功能上的需求，在CBASE中增加DECIMAL数据类型，使得数据库能够实现存储，读取decimal类型的数据，并能进行相关的数学运算。

2 模块设计

2.1 表Schema设计

2.1.1 需求

为了使CBASE支持decimal数据类型，首先需要在建表的时候支持包含有decimal数据类型的表的创建，同时需要存储用户在建表时指定的相关列上decimal数据类型的参数，包括有效数字和精度。比如用户定义了某一列为decimal(10,5)。其中precision为10，表示有10位有效数字，scale为5，表示有5位小数。对于CBASE来说，每个server（包括rs，ups，cs，ms）都有自己的ObSchemaManagerV2。ObSchemaManagerV2里存储并管理所有

表的schema，包括每张表有多少列，每一列是什么数据类型，并且保持所有server上ObSchemaManagerV2所管理的表schema是一致的。下面是ObSchemaManagerV2的主要数据结构。

```
class ObSchemaManagerV2{
private:
ObTableSchema table_infos_[OB_MAX_TABLE_NUMBER];
ObColumnSchemaV2 *columns_;
};

class ObColumnSchemaV2
{
private:
bool maintained_; bool is_nullable_; uint64_t table_id_;
uint64_t column_group_id_; uint64_t column_id_;
int64_t size_; //only used when type is char or varchar ColumnType type_;
char name_[OB_MAX_COLUMN_NAME_LENGTH]; ObObj default_value_;
//join info
ObJoinInfo join_info_;
//in mem
ObColumnSchemaV2* column_group_next_;
};
```

每个server运行的时候都会维护自己的ObSchemaManagerV2。这个schema是怎么获得的呢？对于RS来说，第一次启动时从磁盘中读取，每次执行建表语句时对其进行修改。对于其他server来说：1. RS通过心跳通知。2. 建表语句执行的时候，RS会向其他server发送最新版本的schema。

2.1.2 定义

由于CBASE不支持decimal，在实现ObSchemaManagerV2的时候，没有设计数据结构来存储用户定义的精度和有效数字。所以，为了实现decimal，必须对ObSchemaManagerV2进行修改。我们的修改方案是：在ObColumnSchemaV2里面增加了一个数据结构：

```
struct ObDecimalHelper {
uint32_t dec_precision_ :7;
uint32_t dec_scale_ :6;
};
```

相应的，为了使该数据结构能够起作用，需要补上对它进行设置、修改、获取的接口，以及修改ObSchemaManagerV2中schema序列化和反序列化函数。因为ObSchemaManagerV2是各个server在内存中管理各个表schema的结构，无法直接在不同server之间进行传递。注：在修改了ObSchemaManagerV2之后，要对整个集群进行bootstrap（删除集群原有数据）之后才能重启，否则会报错。

2.1.3 设计

建表语句执行的时候，MS会把建表语句进行解析，把用户输入的一些schema信息封装到一个TableSchema数据结构中。这个TableSchema中OB设计了对用户输入的精度和有效数字的存储空间，故无须修改。仅需修改MS封装TableSchema的函数，将用户输入精度和有效数字存到TableSchema里面即可。具体的代码为：

```

Sql/ob_transformer.cpp
gen_physical_create_table()
{
col.data_precision_ = col_def.precision_; col.data_scale_ = col_def.scale_;
}

```

TableSchema封装好之后，MS会把TableSchema发送到RS。RS接收到TableSchema之后，会调用add_new_table_schema()函数根据TableSchema里面的值，对自己的ObSchemaManagerV2进行修改。我们只需要在该函数里，把TableSchema里面的用户输入的精度和有效数字位数存到TableSchema里的ObDecimalHelper即可。具体代码如下：

```

Common/ob_schema.cpp add_new_table_schema()
{
ObColumnSchemaV2 old_tcolumn;
const ColumnSchema &tcolumn = tschema->columns_.at(i); old_tcolumn.set_table_id(tschema->table_id_);
if (tcolumn.data_type_ == ObDecimalType)
{
old_tcolumn.set_precision(static_cast<uint32_t>(tcolumn.data_precision_));
old_tcolumn.set_scale(static_cast<uint32_t>(tcolumn.data_scale_));
}
}

```

由于在ObSchemaManagerV2里面增加了一个数据结构，而ObSchemaManagerV2又是需要在各个server之间进行传输的，所以我们要继续修改ObSchemaManagerV2的序列化与反序列化函数。具体代码如下：

```

Common/ob_schema.cpp
DEFINE_SERIALIZE(ObColumnSchemaV2)
{
...
if(OB_SUCCESS == ret&&ObDecimalType==type_)
{
ret = serialization::encode_i8(buf, buf_len, tmp_pos, ob_decimal_helper_.dec_precision_);
}
if(OB_SUCCESS == ret&&ObDecimalType==type_)
{
ret = serialization::encode_i8(buf, buf_len, tmp_pos, ob_decimal_helper_.dec_scale_);
}
...
}

int ObColumnSchemaV2::deserialize_v4()
{
...
if(ObDecimalType==type_){
int8_t p= 0; int8_t s = 0;
if (OB_SUCCESS == ret) {
ret = serialization::decode_i8(buf, data_len, tmp_pos, &p);
ob_decimal_helper_.dec_precision_ = static_cast<uint8_t>(p) & META_PREC_MASK;;
}
}
}

```

```

if (OB_SUCCESS == ret) {
ret = serialization::decode_i8(buf, data_len, tmp_pos, &s);
ob_decimal_helper_.dec_scale_ = static_cast<uint8_t>(s) & META_SCALE_MASK;
}
}
...
}

```

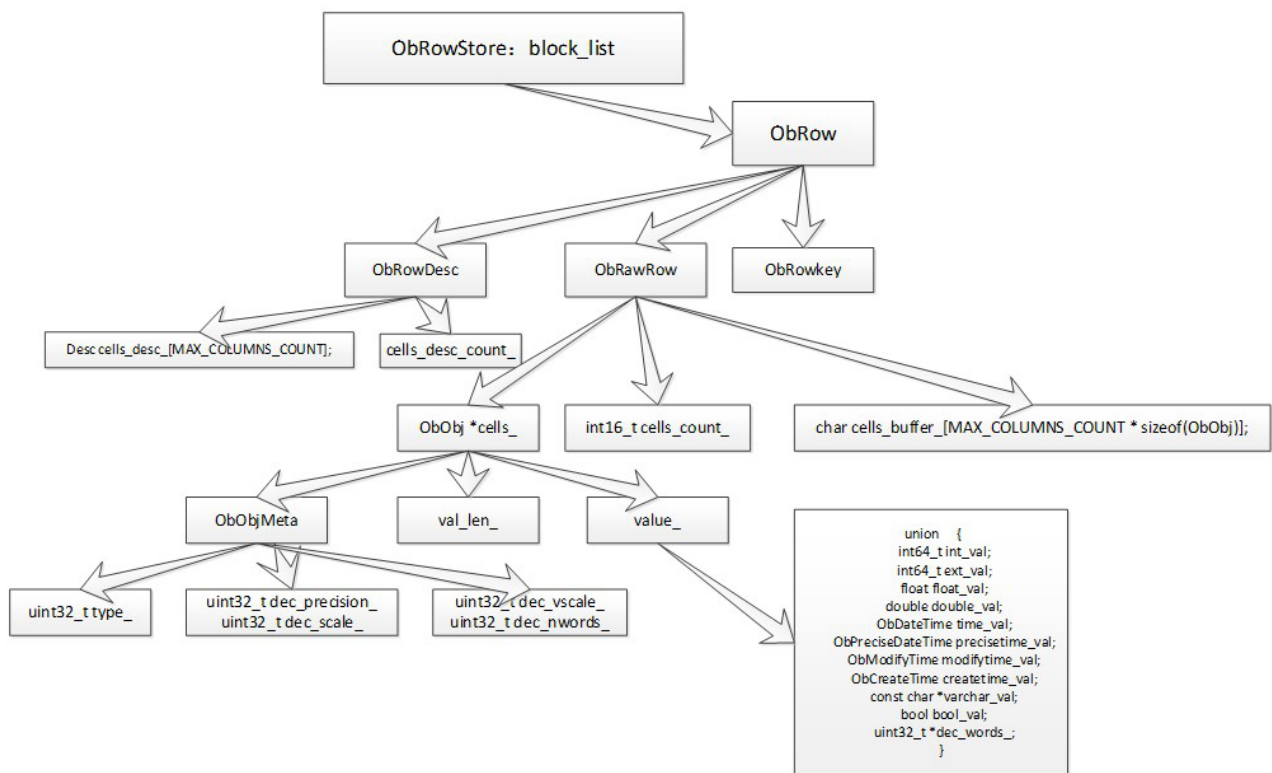
至此，我们完成了对ObSchemanMangerV2的修改。

2.2 ObDecimal基本类设计

在完成了decimal类型的schema的存储以及sql输入decimal数据的识别之后。为了封装整个 decimal的操作（如内存结构，序列化，反序列化，四则运算等），我们增加了一个新的 ObDecimal类。

2.2.1 Decimal类型实现：

CBASE 通过ObObj封装了最小数据处理单元，进而实现上层操作对不同数据类型的泛型化操作。但是ObObj未能实现对变长和定长数据的统一泛型化，进而导致在现有OB的基础上增加对decimal类型支持的难度（极大的扩大了代码修改范围）。下面我们首先认识ObObj的结构：



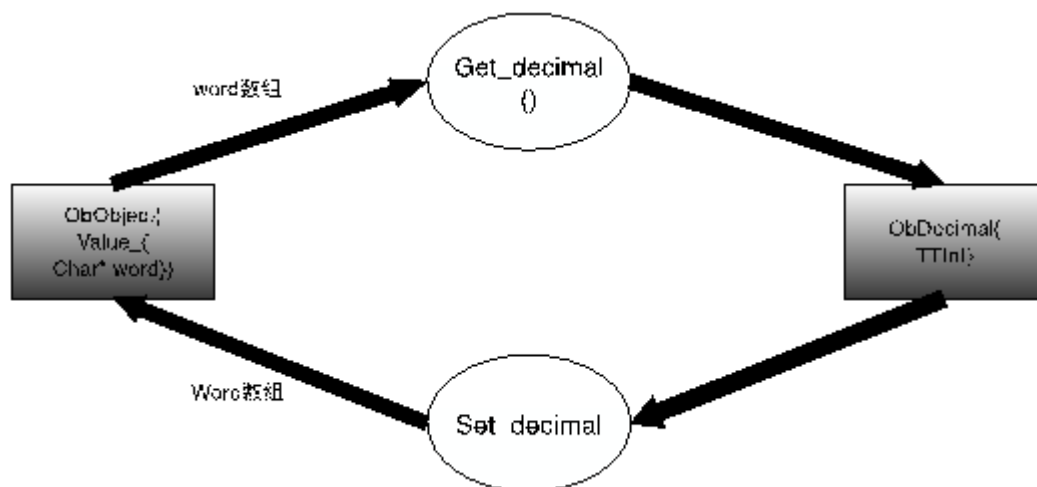
ObObj是CBASE最基本的存储结构。CBASE把用户输入的值封装成ObObj。写到磁盘的时候调用ObObj的序列化。ObObj序列化的时候会根据自己的type，调用不同类型的序列化。同样在读取数据的时候也会根据不同的数据类型进行反序列化与解析。从上图可以看出，以一个联合体union来表示对于不同种数据类型的值。

```

unoin //value实际
{
int64_t int_val;
int64_t ext_val;
float float_val;
double double_val;
ObDateTime time_val;
.....
}value_

```

考虑到Decimal类型的数据的存储空间较大（目前使用N个Byte存储），同时不增加ObObj中value的存储开销（value默认存储大小为8Byte），我们将具体的Decimal使用外部空间来存储（如果将Decimal的数值放在ObObj中将大幅降低修改代码的范围）。为了不增加内存开销，我们的实现手段为，在value_结构体内增加一个char类型的指针*word，这个指针指向一个char类型的数组，这是Decimal的存储结构。在计算方面，我们设计Decimal类型的时候借用了开源的ttmath的Int类，将一个Decimal数转换成大数来处理，运算时也使用这个大数来作运算，TTmath通过这个数组计算转换成一个大数。ObObj中decimal值与ObDecimal中decimal值相互赋值过程如下：



在ObObj类当中，有get_decimal和set_decimal两种方法，get_decimal将value中指向的char数组取出，接着调用ObDecimal类中的from()函数转化成TTInt结构，最后赋值给Decimal类型的TTInt成员。反之，通过set_decimal将一个封装的Decimal类中TTInt的数据取出，然后调用ObDecimal类的to_string()函数转换成ObString，最后赋值给value中的word指针。因为要尽量对得上原先OceanBase的一些接口，所以Decimal类的数据结构设计成与ObNumber类似。

```

Class ObDecimal
{
Private:
uint32_t vscale_;
uint32_t precision_;//存储用户定义的精度 uint32_t scale_; //存储用户定义小数位数
TTInt word[NWORDS]; //用一个ttmath的Int类型ttmath::int来存储实际值
}

```

其中，NWORDS=1，即使用一个TTInt来存储Decimal值。

```
typedef ttmath::Int<2> TTInt;
typedef ttmath::Int<4> TTCInt;
typedef ttmath::Int<8> TTLInt;
```

对于64位机器，在ttmath运算接口当中，ttmath::Int是由一组uint64_t数组来实现大整数，也就是ttmath::Int, size的值可以自定义设置，比如当size=2的时候，用两个uint64_t实现大整数。这时候一个TTInt可以表示的数值范围是 $2^{(64 \times 2)}$ to $2^{(64 \times 2)}$ ，转换成十进制就是38位整数。我们选取的t_size为2，即这个大整数word可以表示的数值范围是38位整数。其中，约定小数部分最多为37位。

2.3 运算设计

2.3.1 四则运算

四则运算，包括加，减，乘，除。计算的时候，直接取出TTInt类型的word进行四则运算，因为总长度是对齐的，所以只需要在最后的结果根据两个数的scale，precision进行结果处理即可。而对于乘法和除法，直接做运算的时候会造成结果溢出，所以用typedef ttmath::Int<8> TTLInt；用于乘除计算。因为这个类型长度四倍于TTInt，所以不会造成溢出，计算时，先将两个乘数转化成TTLInt，再进行乘除计算，结果与MaxDecimalValue/MinDecimalValue比较，做溢出处理。CBASE数据库在存储Decimal时进行数值检测，运算是在存入的数值上进行，因此运算时不需要再进行数值检测，只需要检测是否溢出。在实现运算时，为了保持和CBASE中ObNumber类的运算接口一致，将ObDecimal类运算定义如下：

```
int add(const ObDecimal &other, ObDecimal &res) const;
int sub(const ObDecimal &other, ObDecimal &res) const;
int mul(const ObDecimal &other, ObDecimal &res) const;
int div(const ObDecimal &other, ObDecimal &res) const;
```

其中，other表示的是加数、减数、乘数和除数；res表示的是结果。Add和Sub运算中，将两个Decimal的数值存放在TTInt中，然后调用TTMATH的add和sub进行运算，将运算结果存在res.word[0]中。计算res.word[0]的长度，并与38进行比较来判断是否溢出，最后设置相应的precision和scale。Mul运算中，将两个Decimal的数值存放在TTLInt中，然后调用TTMATH的mul进行运算，计算res.word[0]的长度，并与38进行比较来判断是否溢出，最后设置相应的precision和scale。Div运算中，将两个Decimal的数值存放在TTLInt中并将被除数乘以 10^{37} (num1 = num1 * kMaxScaleFactor;)，然后调用TTMATH的div进行运算，将运算结构存放在res.word[0]中。计算res.word[0]的长度，并与38进行比较来判断是否溢出，最后设置相应的precision和scale。

2.3.2 Decimal逻辑运算

ObDecimal类的逻辑运算的实现类似OCEANBASE数据库中ObNumber类的逻辑运算——重载<、<=、>、>=、==、!=6种逻辑运算符。每种逻辑运算调用ObDecimal类中的sub来实现比较，并返回bool值。

3 总体设计

一个ObDecimal的对象代表一个decimal类型的数值。对于decimal的精度和有效数字位数，我们也同样把他存到ObDecimal类里面。而什么时候把精度和有效数字位数存到ObDecimal里面呢？当然是在插入语句执行的时候。CBASE在执行插入语句的时候，会把用户输入的值转成相应类型的ObObj存起来。如果某一列的类型是decimal，我们会先把用户输入的值转换成一个ObDecimal的对象，然后把该对象存到一个类型为ObDecimalType的ObObj里面。具体的实现函数是int ObExprValues::eval()。在该函数里，我们会获得 ObSchemaManagerV2里面存储的该列的数据类型以及用户输入的精度和有效数字位数，在生成相应的ObDecimal对象的时候，把用户输入的精度和有效数字位数存到该对象中去即可。

3.1 词法、语法解析

CBASE在处理insert语句的时候，首先会对sql语句进行词法解析。对于带有小数点的数值，比如3.14，CBASE会在词法解析的时候把它解析成T_DOUBLE类型。然后在生成逻辑计划的时候将该值存成C++中基本的double类型，最后存到ObObj中去。但是如果表的某一列为 decimal类型，在插入该列数据的SQL语句解析过程中，将原本用户输入的包含有小数的数值被转成double类型，将会造成精度上的损失。所以我们的实现是在词法解析里面增加了T_DECIMAL类型，对于带有小数点的数值，全被解析成T_DECIMAL类型，为了保留用户输入数据的全部精度，在SQL解析阶段，我们在T_DECIMAL类型中先存储用户输入的数据。注：科学计数法的数值依然被解析成T_DOUBLE类型。

3.2 生成逻辑计划

MS生成逻辑计划的时候，会将语法树中存的数值转换成中缀表达式，存到逻辑计划树中。

对于T_DECIMAL类型的语法树节点，我们首先把它存到ObObj中，然后存到中缀表达式中。

3.3. 生成物理计划

MS生成物理计划的时候，会将中缀表达式转换成后缀表达式，存到物理操作符中。在 ob_postfix_expression.cpp 中，对于类型为ObDecimalType的中缀表达式，我们把它转换成 后缀表达式的代码如下：

```
case T_DECIMAL: item_type.set_int(CONST_OBJ);
obj.set_decimal_v2(item.dec,item.len); //add xsl ECNU_DECIM
AL
if (OB_SUCCESS != (ret = str_buf_.write_obj(obj, &obj2)))
{
TBSYS_LOG(WARN, "fail to write object to string buffer. err=%d", ret);
}
else if (OB_SUCCESS != (ret = expr_.push_back(item_type)))
{
TBSYS_LOG(WARN, "fail to push item.type. err=%d", ret);
}
else if (OB_SUCCESS != (ret = expr_.push_back(obj2)))
{
TBSYS_LOG(WARN, "fail to push object. err=%d", ret);
}
break;
```

注：因为CBASE的内存分配机制比较特殊，此处尚未解决内存分配问题（在Obj析构的时候释放内存依然存在问题）。主要的原因在于解析后缀表达式的时候，需要将后缀表达式中的每个数据解析成obj并存储到expr_中，然后执行相关的后缀表达式。CBASE中对于变长的类型（如varchar）分配了额外的内存空间，因此需要修正这个OBDecimal的问题，需要采用类似的机制。

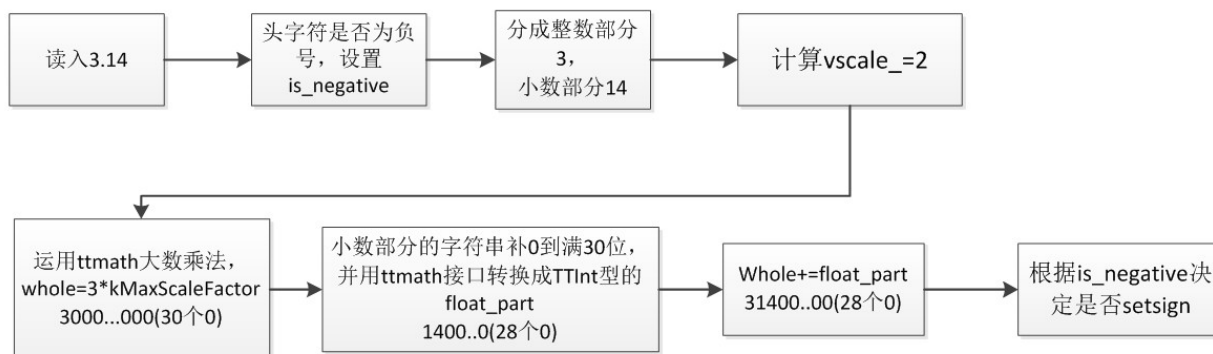
3.4 Decimal类型的转换

对Obj_cast这个函数，需要修改很多地方。下面是具体分析该函数：

```
int obj_cast(const ObObj &orig_cell, const ObObj &expected_type, ObObj &casted_cell, const ObObj
*&res_cell)
```

该函数有四个参数，orig_cell表示原来的数据，即后缀表达式中的数据，expected_type表示表定义时的列的类型，casted_cell表示作类型转换的中间量，res_cell表示转换后的结果。CBASE对该函数的实现是：先判断orig_cell和expected_type的类型是否一致，如果一致，则不作转换，如果不一致，将调用相应的转换函数进行转换，最后的结果作为res_cell返回。由于我们对CBASE的词法解析做了修改，所以必须要在obj_cast函数里面做些相应的修改。对于expected_type为double, float, int 的，我们要写相应的double_decimal, float_decimal, int_decimal函数来供调用。如果expect_type为ObDecimalType，我们会在 Obj_cast函数里根据用户输入的精度和有效数字位数对res_cell进行修改。因为如果用户要插入的数值的小数位数大于用户输入的精度，我们的实现是将用户要插入的数值进行一次截取，将超过精度的小数部分直接截掉，保证存到数据库中的数是截取后的数。

示例：Varchar 与 Decimal之间的相互转化：从char到Decimal之间的类型转化函数为from()与tostring()；其中from()接口为from(const char str, int64_t buf_len)。由char*转化成decimal时，以Decimal(20,7)为例，输入3.14的处理流程如下



其中to_string()接口为to_string(char* buf, const int64_t buf_len)。由decimal转化成char*时，处理流程如下：


```

DEFINE_SERIALIZE(ObObj)
{
...
ObObjType type = get_type(); switch (type)
{
...
case ObDecimalType: ret=serialization::encode_comm_decimal(buf,buf_len,tmp_pos,ob
j_op_flag == ADD,meta_.dec_precision_, meta_.dec_scale_, meta_.dec_vscale_,value_.word,
val_len);
...
}
...
}

```

序列化类型为Decimal的Object的时候，我们将Object的Meta中的内容，与TTInt的内容一同序列化。

参数解释

meta.dec_precision : ObObjMeta中的精度

meta.dec_scale : ObObjMeta中的小数长度

meta.dec_vscale : ObObjMeta中的实际小数长度

value_word : 指向Decimal数据的指针

val_len : Decimal占据char的空间长度个数

因为是用变长存储，在反序列化的时候依次从buf当中读出char。因此，对于每个Decimal 类型的数据会占用掉N byte的空间大小（N = val_len）。

二级索引

1 需求分析

CBASE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。在历史库版本中缺乏对二级索引的支持，使得需要通过手动维护索引表的方式来加快对非主键列信息的查询，产生了额外的工作成本。为了改善这一情况，应相关部门需求，准备开发二级索引功能。

2 适用场景

对非主键列的查询有较高的性能需求。要求非主键列有一定的数据区分能力，例如姓名、日期等属性列，类似性别这种数据基数较小的属性列不适合创建二级索引。

3 功能简述

二级索引功能的主要任务是提高用户SQL查询请求的响应请求，通过对原数据表建立索引并维护索引表来实现更加快速的数据定位。在实际实现当中，每一个被定义的索引实际都维护了一个独立的索引表，当查询sql满足一定条件（查询列全为索引列等）则会将主表的查询转换为对索引表的查询。二级索引机制主要实现SQL功能与数据维护功能，要点列举如下：

SQL功能：

- 创建索引：支持SQL语法，对指定表与列建立索引；
- 删除索引：将特定表上的索引删除；
- 显示索引：将一张表上的索引列出；
- 数据修改：修改索引表中的数据；
- 数据查询：查询索引表中的数据；
- 数据删除：删除索引表中的数据。

索引维护功能：

- 动态索引维护

在维护主表数据的同时，为维护索引表在内存表中同时写入增量更新。考虑到在主表有数据的情况下，新建的索引表无法实现快速复制主表的数据，并拉取磁盘数据，因此，动态数据维护不支持新建索引表在没有进行合并的情况下提供数据查询服务。

- 静态索引维护

索引表需要根据主表写入磁盘的数据构建索引记录，需要实现接口完成收集主表数据的功能。

- 数据校验

为了保证索引表数据与主表数据一致，需要提供数据校验接口。计划使用CRC校验（与原Oceanbase数据库使用的行校验和相似）。每次合并完成（即新索引数据提供服务前）需要按该算法计算其校验和，并且包数据表与索引表之间数据一致。

- 数据采样与索引划分

索引的分片原则决定了索引在集群的分布情况，我们的设计中，索引表分片后的tablet数量接近于主表，过多会造成交互成本提高，过少则影响每日合并。

chunkserver完成局部索引构建后对其tablet进行采样，并将采样信息绘成直方图发送给Rootserver, 由Rootserver接收完全部直方图信息来决策索引的分片。

4 使用方法

- 创建索引

```
create index <索引表名> on <主表名>([<索引列名>,...]) storing([<冗余列名>,...])
```

<索引表名>：允许用户自定义，但部分特殊字符不支持使用。

<主表名>：已经存在的索引表

<索引列名>：使用主表中的任意主键

<冗余列名>：使用主表中的非主键列，（已经被选为索引列的列不支持冗余）

- 删除索引

```
drop index [<索引表名>,...] on <主表名>
```

<索引表名>:已经存在的索引表

<主表名>：索引表的主表

- 显示索引

```
show index on <主表名>
```

<主表名>：索引表的主表

- 数据修改

支持对主表数据更新，SQL语法规则无变化，包括update，replace，insert

- 数据查询

二级索引功能支持两种索引查询计划，既支持自动检测索引列来判断是否使用索引表，也支持用户显示的要求对索引表进行查询。查询语法的规则满足一下两种形式：

1. 自动使用索引表，语法与原select规则相同

若where条件使用的字段为表的全部主键，则这种情况不使用索引，数据库使用全主键查询。

若where条件不是全主键，且也不是主键前缀的字段，但涉及到了索引列，那么数据库自动查询索引表使用索引。如涉及到多个索引表，则使用where条件中最左索引列对应的索引表。

若where条件不是全主键，但是同时包含了主键前缀的字段与索引列的字段，那么数据默认使用主键前缀进行查询，同时，用户也可以使用hint语法，强制使用某个索引。

2. HINT查询

HINT的优先级高于上述规则，语法类似如下：

```
SELECT c2 /*+ INDEX(test,index_internal_name) */ from test WHERE c1=10;
```

test是数据表名，index_internal_name是索引创建后的内部表名，命名规则为“__3001_idx_idxname”，idxname在创建索引表时由用户指定，3001为table_id。

如果后期支持“alter table tableA rename to tableB”修改表名时，要求改动表名时不能更改数据表的table_id。

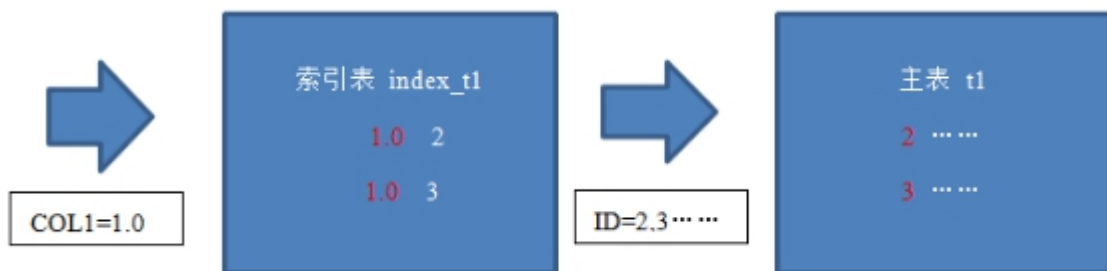
5 概要设计

对于一张主数据表t1，构造的索引，本质上来说是存储在OB的另一张数据表，这个数据表存储了索引列，主键列，和storing列，我们称之为索引表。索引表的设计和用法大致如下：

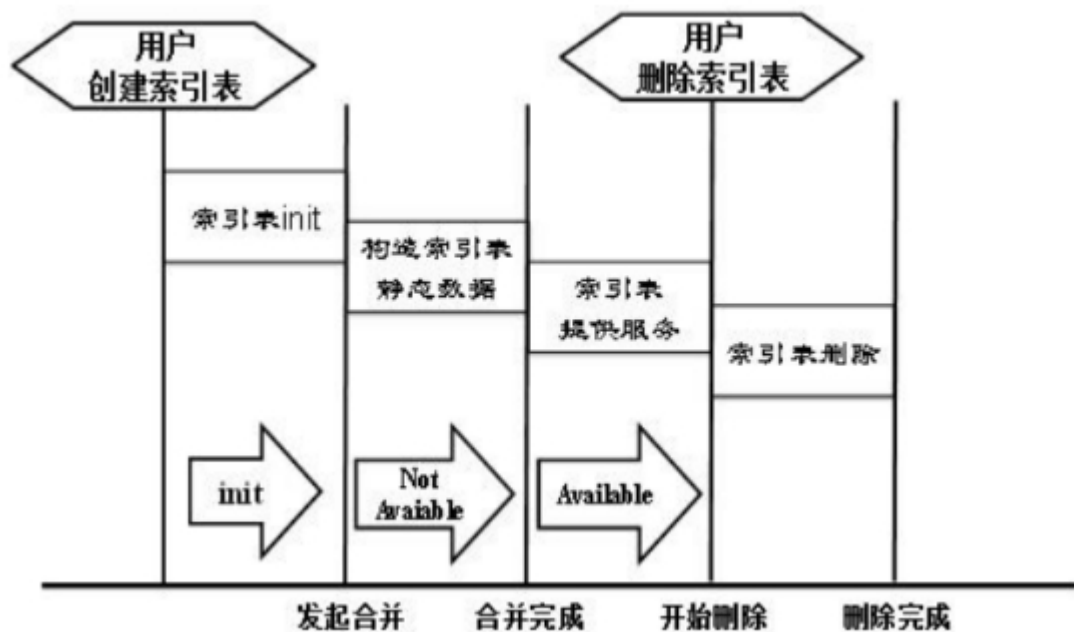
	主表 t1	索引表index_t1
主键	ID	COL1+ID
非主键列	COL1	COL2(STORING字段)
	COL2	

SELECT * FROM t1 WHERE COL1=1.0 ;

1. 查找索引表index_t1表格，找到COL1下的所有ID
2. 回表，根据返回的ID批量查找主表t1的数据
3. 如果查找的是STORING字段，那么不用回表，直接返回索引表的数据

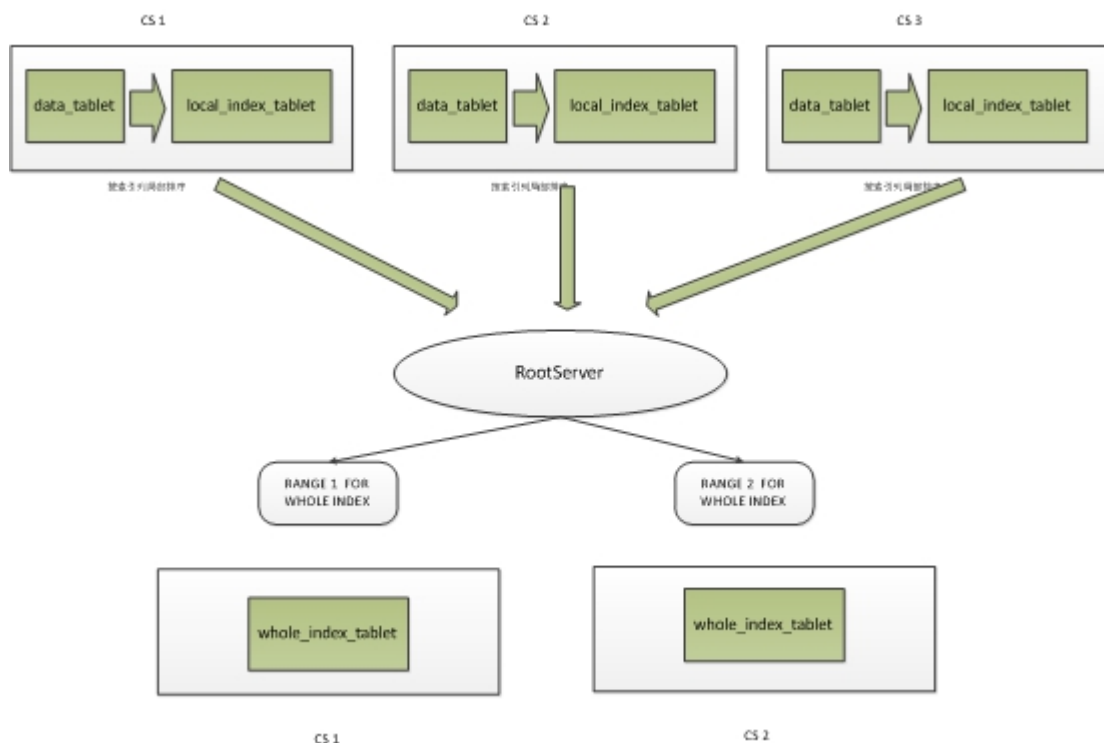


由于索引表也是一张存储的表，所以其也有相应的静态数据与增量数据，相同的，前者存储在CS上，后者存储在UPS的内存表当中。创建索引的时候不但要记录增量数据，也要针对既存数据创建索引。索引将在所有既存数据的索引记录构建完成后变为可用的状态。从索引被创建，直至其可用的过程如下图所示：

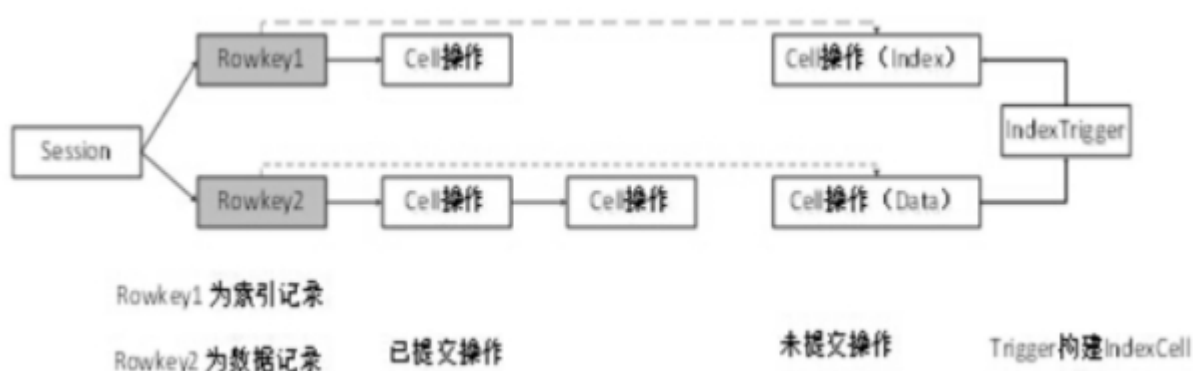


用户创建索引，初始状态为INIT，此状态下是允许维护索引增量数据的，这也是为了构建静态数据时，能够同时得到期间更新的索引。构建静态索引的过程如下：

1. 索引表的数据和数据表一样，包括静态数据和增量数据。静态索引数据的构建在每日合并完成之后进行，建立静态索引数据之后，索引表的状态将变为可用。
2. 首先，等普通的每日合并完成，所有的数据表tablet生成完毕。
3. RS给所有CS发送信息，启动创建索引任务，CS判断自己拥有创建索引的tablet的第一个备份，读取这个tablet的数据，按照索引列进行排序，并写成局部索引的sstable。
4. CS创建完局部索引的tablet之后，向rs汇报局部索引的tablet信息，rs根据这些信息，划分出全局索引数据的tablet分布，并将分布信息写入roottable当中。再次给所有CS发送消息，让其创建全局的索引。x0005
5. S判断自己应该创建一个全局的tablet索引，根据这个range，向其他CS拉取这个range的数据，并写成全局的索引数据的sstable。



当主数据表被更新时，其索引信息也要同时更新，保持数据和索引之间的一致。CEDAR是由Updateserver单独执行写事务，所以索引和数据的更新也是由其单点处理。为了实现同步更新，在MergeServer构建执行计划的时候，将索引相关信息写入一个IndexTrigger加入到执行计划发送给UpdateServer。在修改内存表增量更新的时候，会对索引也进行相应的修改，保证对索引和数据的更新在同一个事务当中，能够同时成功或者回滚。



6 详细设计

6.1 名词解释

- 静态索引：我们将索引以表的形式存储在系统，这些索引也会有基准数据。
- 每日合并：也称为定期合并，主要将UpdateServer中的增量更新分发到ChunkServer中。
- 列校验和：用于校验目的地一组数据项的和。
- CS：ChunkServer，CEDAR的基线数据存储子系统，由多台机器构成，基线数据通常保存2~3副本并且保存在不同的ChunkServer上。
- secondary index：二级索引（或称辅助索引，次索引），用以加快query速度。
- schema：数据库中表的模式。一般指的是表名，约束条件，列名和列属性等等。
- 回表查询(Back to the table)：如果查询涉及到了索引，则第一次先查询索引表，第二次再根据第一次查询的结果来查询原表。
- 不回表查询(Not back to the table)：如果查询涉及到了索引，并且查询到的结果都在索引表中，则只需要查询一次索引表就能获得需要的所有数据。
- hint：用户在sql语句中添加注释，强制使用指定的方式执行sql。使用方式类似于C++语言中/* ... */的注释方式。
- tablet：CEDAR将大表划分为大小约为256MB的子表。
- sstable：每个子表由一个或者多个SSTable组成（一般为一个），每个SSTable由多个块（Block，大小为4KB~64KB之间，可配置）组成，数据在SSTable中按照主键有序存储。

6.2 模块设计

6.2.1 create index 子模块设计

索引是创建在表上的，对数据库表中一列或多列的值进行排序的一种结构。其作用主要在于提高查询的速度，降低数据库系统的性能开销。CEDAR中，新创建的索引需要等待一次每日合并后才生效。

设计原理

- 内部表的改动

新增内部表“_all_secondary_index”。内部表的schema由系统设计人员设计，用户不可改变。为了与原来的“_all_tablet_entry”这张系统表区别开来，在新增内部表中增加了两个新的字：original_table_id和index_status，分别表示索引表对应的原数据表的表id和索引对应的状态。

- 原数据表id和索引表状态

如果这张表是数据表，则在original_table_id这一列上的值为1；索引表的状态index_status有4个值：

0：NOT_AVALIBALE 索引状态不可用 1：AVALIBALE 索引状态可用 2：ERROR 该表不是索引 3：WRITE_ONLY 这张索引表只允许写，不允许读(用于drop index) 4：INIT 这张索引表正在初始化，将于第一次每日合并之后可用

如果这张表是数据表，那么index_status这一列上的值是ERROR。

- schema介绍

CEDAR中三种schema数据结构：1.结构体TableSchema 2.类ObTableSchema 3.schema管理类ObSchemaManagerV2。这三个数据结构是Oceanbase 0.4.2 设计的对于CEDAR中数据表的schema存储及管理的数据结构。

- struct结构体TableSchema是在建表的时候用到的，构建物理计划的过程，就是填充操作ObCreateIndex的TableSchema。新建一张表时，在RootServer中，将TableSchema的信息更新到内部表中。
- ObTableSchema 是schema_manager_v2的一个成员，在schema_manager_v2中有一ObTableSchema数组。创建表/索引后，rootserver会遍历内部表，用内部表的每一行信息，去填充一ObTableSchema。之后，将ObTableSchema加入到schema_manager_v2当中。最后，将schema_manager_v2序列化发送给其他各个Server更新。

索引表的额外的两个字段添加在结构体TableSchema和类ObTableSchema中。通过赋予初始值的方式来区分CEDAR中的数据表和新增的索引表。

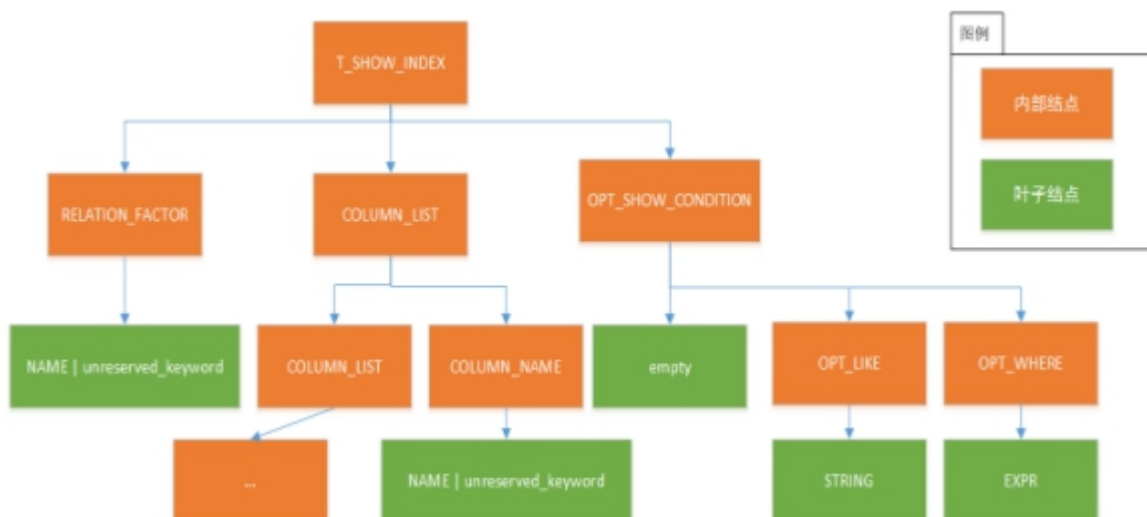
6.2.2 show index 子模块设计

show index模块设计的目的是方便用户查看数据表上建立的索引表的具体条目以及索引表的一些信息。

代码流程

show index的执行流程与show tables的执行流程类似：

- 语法解析，将show index语法解析成语法树，如下图所示；
- 逻辑计划生成，根据语法树生成逻辑计划，将语法树中的信息存储在逻辑计划中，生成中缀表达式等；
- 物理计划生成，由逻辑计划生成物理计划，构造物理操作符，完成中缀转后缀等。



6.2.3 drop index 子模块设计

索引是创建在表上的，对数据库表中一列或多列的值进行排序的一种结构。其作用主要在于提高查询的速度，降低数据库系统的性能开销。当索引过多时，维护开销增大，因此，需要删除不必要的索引。

设计原理

索引表本质上是CEDAR中的一张数据表。但是索引表和数据表存在着一些区别。索引表是依存其原表的，因此在索引表的Schema里面不仅要有普通表的所有信息，还要增加索引表对应的原表信息，索引表的状态信息等等。另外，要保证索引表对应的原表和索引表之间数据的一致性。无论是对原表还是对索引表的操作，都要同步到另一张表上。Drop table 也是如此。因此要分两种情况：1. 删除数据表 2. 删除索引表

删除索引表的流程

为了实现Drop index去删除一张索引表，新增一个继承于ObDropTable的类。这个类专门用来处理Drop index。如下图所示：



单独删除一张索引表，需要新增一个继承于T_DROP_TABLE的物理操作符T_DROP_INDEX。此操作符用于完成单张索引表删除的操作。这样封装的原因在于：索引表大部分信息是跟数据库中的普通表是一样的，比如说：索引表也有自己的tid(表序号)，每个列有自己的cid(列序号)...这些可以使用drop_table来删除。至于索引表特有的息，可以使用drop index来删除。

删除数据表的流程

删除带索引的数据表。开源的OceanBase 0.4.2 版本中还没有实现二级索引。因此，在删除数据表的时候，无需考虑此表是否拥有依赖它的表的存在。因此我们在ObDropTable的类中增加两个私有的成员变量，如下所示：

```
private:
// data members
bool if_exists_;
common::ObStrings tables_;
mergeserver::ObMergerRootRpcProxy* rpc_;
bool has_indexs_; ///< table has index?
common::ObStrings all_indexs_; ///< store all indexs on all tables
```

all_indexs中保存了该table上建立的所有的索引表的名字。在进行删除操作之前，我么首先 会判断all_indexs是否为空。

- 1. 索引表为空，这种情况跟OceanBase 0.4.2 中的流程一样。直接使用 T_DROP_TABLE物理操作符来删除一张表。
- 2. 索引表不为空，这种情况下，会重复调用 6.3.3.1 中的流程:删除一张索引表。直到将 所有的索引表都删除。操作完成之后，这张表就变成了一张没有索引的数据表了。然 后将数据表按照第1种情况删除。

6.2.4 select 子模块设计

索引是创建在表上的、对数据库表中一列或多列的值进行排序的一种结构。其作用 主要在于提高查询的速度，降低数据库系统的性能开销。CEDAR中，新创建的索引 需要等待一次每日合并后才会生效。每日合并完成之后，我们就可以使用我们创建 的索引了。目前只支持根据输出列和过滤条件来判断是否可以使用索引。具体内容 请查看下文中索引使用规则的说明部分。

原select流程（指的是OceanBase 0.4.2 中没有实现二级索引时的select流程）：

- 词法分析，语法分析：把用户输入的sql语句解析成语法树。
- 生成逻辑计划：根据语法树生成中缀表达式和其他信息，存到逻辑计划中。
- 生成物理计划：根据逻辑计划里面存的相应信息，生成物理操作符，并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。
- 物理计划open()：构造一些参数和信息。这些信息将被发送到CS，CS根据这些信息来拉取数据
- 物理计划get_next_row()：向CS发送请求，接受到CS返回的数据，处理后返回给客户端。

现在select流程与原流程最主要的区别：它们有不同的物理计划。

索引使用规则说明

总的来说，优先级是：用户指定hint>不回表>回表>不使用索引。而且这个判断是针对某一张表来说的。判断的依据是两个表达式数组filter_array和project_array，分别存放了sql语句中该表的过滤列和输出列。我们将索引表中是否包含所有的查询信息将查询分为两类：不回表查询和回表查询。详细解释请查看名词解释部分。

索引表使用规则使用一个类完成封装，类名为ObSecondaryIndexService。这个类目前仅用来封装对于查询中的一些优化规则。

判断sql语句能够使用索引表的不回表的索引规则：

1. 遍历filter_array中的每个表达式，如果其中有一个表达式里面的列是原表的第一主键，或者有一个表达式里面包含了2个或2个以上的列，则下面的判断不再执行，直接返回结果：false。
2. 如果上一步的结果是true，则遍历filter_array中的每个表达式，对每个表达式，如果它里面包含的列是原表的某张可用的索引表的第一主键，并且filter_array和project_array里出现的所有列都在这张索引表中，则直接返回true。如果没有表达式符合条件，则返回false。

判断sql语句能够使用索引表的回表的索引规则：

1. 遍历filter_array中的每个表达式，如果其中有一个表达式里面的列是原表的第一主键，或者有一个表达式里面包含了2个或2个以上的列，则下面的判断不再执行，直接返回结果：false。
2. 如果上一步的结果是true，则遍历filter_array中的每个表达式，对每个表达式，如果它里面包含的列是原表的某张可用的索引表的第一主键，则直接返回true。如果没有表达式符合条件，则返回false。

代码流程

关键算法

UPS在接收到MS发过来的物理计划之后，会将insert的物理计划open。Open完之后，会调用apply函数将原表的增量数据存到内存表里面。我们的修改是先调用操作符IndexTrigger的cons_data_row_store()函数，获取原表的数据存到row_store中，再迭代这个row_store将原表的增量数据存到内存表里，之后再调用操作符IndexTriggerIns的函数handle_tringger()，在函数handle_tringger()里将索引表的增量数据存到内存表里。

函数handle_tringger()：

- 1. 获得原表的所有可用索引表
- 2. 对每一张索引表，调用函数handle_one_index_table()处理

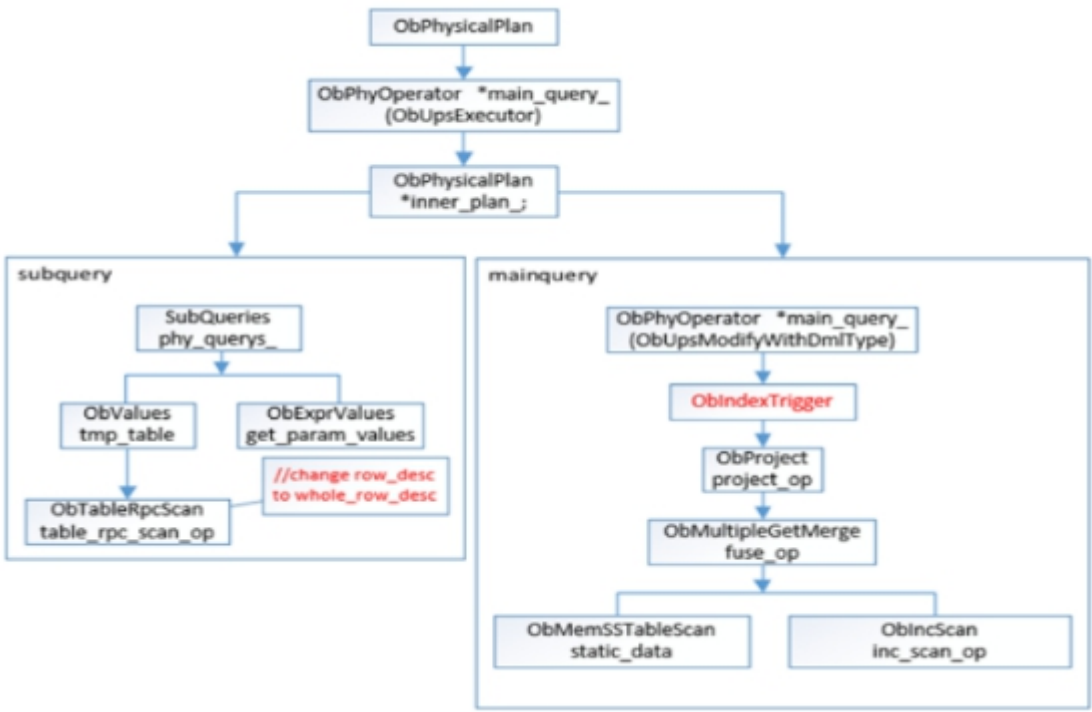
函数handle_one_index_table()：

- 1. 识别当前sql语句的类型
- 2. 如果sql语句为insert语句，通过调用成员变量post_data_row_store_的get_next_row()函数，不断获得sql语句中原表的新增行，根据该新增行以及索引表的行描述信息，构造索引表的新增行
- 3. 调用apply函数将row_store里所有索引表的增量数据存到内存表里面

6.2.6 delete 子模块设计

流程

在原delete的物理计划里面新增了一个物理操作符：ObIndexTrigger。所有对索引表的删除操作都封装在这个操作符里面。该操作符在MS生成，在UPS中open。它在物理计划中的位置是操作符ObUpsModifyWithDmlType的孩子操作符，ObInsertDBSemFilter的父亲操作符。改进后的物理计划如下：



关键算法

我们进行的delete修改点如下：

- 1. 新增物理操作符ObIndexTrigger，对索引表的删除操作封装在这个操作符里面。如果delete的表没有索引表，则走原来的逻辑；只有当delete的表存在索引表时才使用ObIndexTrigger操作符。
- 2. 修改ObTableRpcScan的row_desc参数，原来的row_desc仅仅是主表主键列的row_desc，现在需要将其修改为主表所有所需列的row_desc（包括索引表的主键列）。

UPS在接收到MS发过来的物理计划之后，会将update的物理计划Open。Open完之后，会调用apply函数将原表修改之后的数据更新到内存表中。我们的修改是先调用操作符IndexTrigger的cons_data_row_store()函数，获取原表的需要更新的数据存到row_store中，再迭代这个row_store将原表的数据在内存表中更新，之后再调用操作符IndexTrigger的函数handle_tringger()，在函数handle_tringger()里将索引表的数据在内存表中更新。

函数handle_tringger()：

1. 获得原表的所有可用索引表
2. 对每一张索引表，调用函数handle_one_index_table()处理

函数handle_one_index_table()

1. 识别当前sql语句的类型
2. 如果sql语句为update语句，通过调用成员变量pre_data_row_store_的get_next_row()函数，不断获得sql语句中原表更新之前的数据，根据该行数据以及索引表的行描述信息，构造索引表需要删除的数据
3. 通过调用成员变量post_data_row_store_的get_next_row()函数，不断获得sql语句中原表更新之后的数据，根据该行数据以及索引表的行描述信息，构造索引表需要插入的数据
4. 调用apply函数将row_store里所有索引表需要删除的数据从内存表里删除，需要插入的数据在内存表里更新

6.2.8 replace 子模块设计

流程

执行REPLACE时，需要考虑是否有索引。如果REPLACE数据表无二级索引，那么生成物理计划时不需要向CS请求静态数据；否则，生成物理计划需要添加获取静态数据和MemTable数据。

- No Index Physical Plan：与未实现二级索引版本的REPLACE的物理计划基本上一样，仅仅把ObUpsModify物理操作符修改为ObUpsModifyWithDmlType物理操作符。
- Modify Index Physical Plan：这部分物理计划结合了INSERT和UPDATE物理计划的生成过程。数据表的修改过程按照原来的流程即可。修改索引表时，首先根据插入值来识别哪些索引需要修改。如果需要修改，那么增加操作符ObIndexTrigger，主要负责索引的更新。索引的更新遵循先删除后插入的规则，即先找到索引表中受到影响的那一行数据，将其删除，再将更新后的新行，插入到索引表中，完成更新。实现二级索引的REPLACE物理计划，如下图所示。

关键算法

在MS生成物理计划阶段，如果判断索引表需要修改，则需要构造向CS获取静态数据和向UPS获取MemTable数据的物理计划，同时还需拿到所有索引的Schema，同时需要在物理计划里需要添加操作符ObIndexTrigger。

UPS在接收到MS发过来的物理计划之后，会将物理计划Open。Open完之后，会调用apply函数将原表修改之后的数据更新到内存表中。我们的修改是先调用操作符IndexTrigger的cons_data_row_store()函数，获取原表的需要更新的数据存到row_store中，并判断当前所获取的记录是否为空行，如果不为空行，说明需要构建索引表删除行；再迭代这个row_store将原表的数据在内存表中更新，之后再调用操作符IndexTrigger的函数handle_tringger()，在函数handle_tringger()里将索引表的数据在内存表中更新。

函数handle_tringger():

1. 获得原表的所有可用索引表
2. 对每一张索引表，调用函数handle_one_index_table()处理

函数handle_one_index_table():

1. 识别当前sql语句的类型
2. 如果sql语句为replace语句，通过调用成员变量pre_data_row_store_的get_next_row()函数，不断获得sql语句中原表更新之前的数据；如果需要构造索引表删除行，则根据该行数据以及索引表的行描述信息，构造索

引表需要删除的数据

3. 通过调用成员变量`post_data_row_store`的`get_next_row()`函数，不断获得sql语句中原表更新之后的数据，根据该行数据以及索引表的行描述信息，构造索引表需要插入的数据
4. 调用`apply`函数将`row_store`里所有索引表需要删除的数据从内存表里删除，需要插入的数据在内存表里更新

6.2.9 静态数据构建子模块设计

当我们在有数据的表上建立索引的时候，必须考虑的一个问题是：将数据表上的数据复制到索引表上。

因为从本质上来说，索引表是存在CEDAR中的另一张表，所以，一张索引表也包含了静态数据和动态（增量）数据两个部分，相同的，前者存储在CS上，后者存储在UPS的内存表当中。为了使一张索引表可用，就必须完整地构建索引表的静态数据，在静态数据和增量数据合并的阶段，我们进行索引表的静态数据构建，也就是说，直到第一次数据合并完成之前，索引表都是不可用的。

静态数据构造过程可分为以下几个阶段：

1. 静态数据构建的准备阶段。为了使得索引表能够获得最新的数据，我们在构建索引表之前需要进行一次每日合并。因此，我们将静态数据构建放在每日合并之后。
2. 局部索引构建阶段。当普通的每日合并完成之后，便开始索引的静态数据部分的构建。当CS接收到了RS的创建索引的信号之后，获得一个数据表的tablet，对这个tablet按索引列进行排序，并写到一个局部的sstable。完成以上步骤之后，CS向RS汇报局部索引sstable的信息。
3. 全局索引构建阶段。RS接收到了CS发过来的采样信息之后，对RANGE进行划分，尽量使得每个CS上的tablet的数量相等，以达到负载均衡。RS将切分后的RANGE信息发送给CS。CS根据这个RANGE信息互相之间拉取数据。这个阶段完成后，每个CS便完成了全局的索引构建。
4. 索引表静态数据构造完成阶段。复制全局索引备份，检查列校验和，修改索引表状态为可用。创建索引静态数据期间，索引的状态为`WRITE_ONLY`，也就是只允许写，在此期间更新有索引的数据表的时候，是允许写入到ups里的。

静态数据构建过程中主要涉及到如下几个模块：

1. 多线程处理模块。CS在接收到`OB_CS_CREATE_INDEX_SIGNAL`包之后，开始创建静态索引的过程。因为一张表在一个CS上可能不仅仅只有一个tablet，有些tablet可能会很大，为了提高创建索引的速度，使用多线程来实现创建静态数据的索引。
2. 信息交互模块。CS在构建静态数据的过程当中，涉及到多次和RS的交互。这其中最主要的两个部分是：CS采样信息统计模块和RS的RANGE信息切分模块。
3. 列校验和模块。作为一张独立的表，索引表虽然要求与原表数据保持一致，但是无法否认的是，并没有绝对的方法来保证索引表的完全一致（特别是在分布式环境下不同节点的数据存在出现不一致的可能）。因此，为了确保对外提供的索引表必然是正确可用的，需要使用列校验和来进行数据一致性的确认。

6.3 模块接口

可以在客户端下执行如下命令，指定创建一张索引的时间上限，如果超过时间限制索引仍未完成，则跳过当前索引，创建下一张。

```
alter system set monitor_create_index_timeout='3600s' server_type=rootserver;
```

时间以秒作单位，默认限制是1800s。

7 使用限制条件和注意事项

索引只优化索引列的等值查询，范围扫描会在之后的版本优化。

Bloomfilter Join

一、概述

1.1 目的

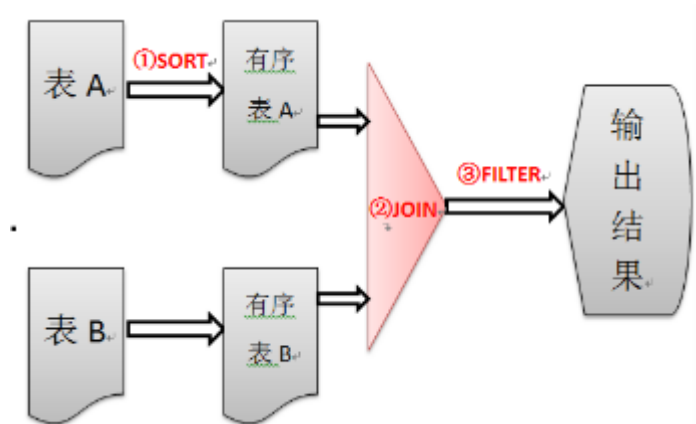
OceanBase是可扩展的关系数据库，实现了巨大数据量上的跨行跨表事务。业务中存在大量的多表连接查询，在处理多表连接时，目前Ocean Base进行Merge Join运算，在Merge Server（MS）分发请求给相应的Chunk Server（CS），CS把所有数据返回给MS，MS进行排序后做Merge操作。若对大表进行查询，CS将传输大量不会产生连接关系的无效数据到MS，浪费了大量的网络传输时间和无效数据的排序时间等，导致查询缓慢。

若两张表连接查询，一张小表，一张大表，小表对于大表的区分性较好，则使用Hash Join的查询效率会优于Merge Join；若两张表数据量都很大，目前实现的Merge Join需要将两表数据存入内存，导致内存不足，使用Nested Loop Join可解决此问题。但是，在0.4.2.8版本中缺乏对Hash Join、Nested Loop Join的支持，为提高查询效率，在OceanBase_BankComm版本中增加Bloomfilter Join操作，作为Hash Join、Nested Loop Join在OceanBase中的实现。

本文档的编写目的，是为了介绍对Bloomfilter Join功能的实现。以便后续开发人员在学习Join操作或者完善Join功能的时候能够用到。

二、原理分析

2.1 Merge Join基本原理图



如图所示，JOIN处理可以分三个步骤：

① SORT处理：将参与JOIN的表按照等值条件的列进行排序。多个等值条件时，按照多个条件的列依次排序，即先按第一个列排序，然后在第一列相等的子集中按照第二列排序。两个表需按照相同的列处理顺序排序。

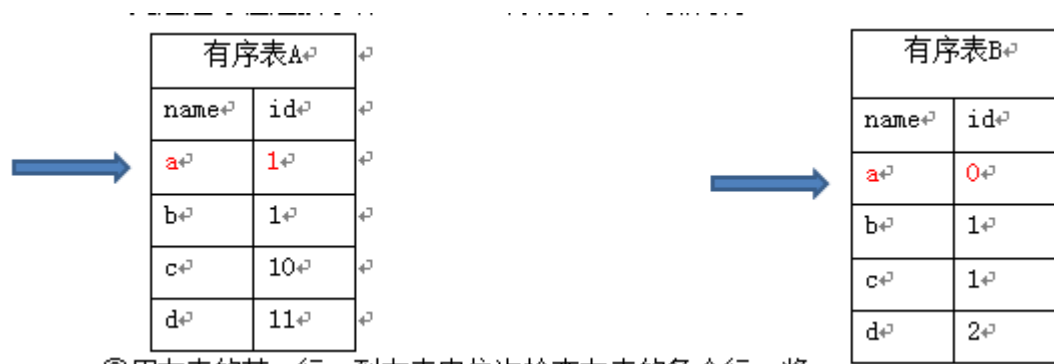
② JOIN处理：将有序表进行等值JOIN操作。由于表是有序的，在数据量庞大时，可以显著提升排序的效率，但是，这也是OB中限制至少有一个等值条件的原因。

③ FILTER处理：按照不等值条件进行过滤，进而输出最终结果。

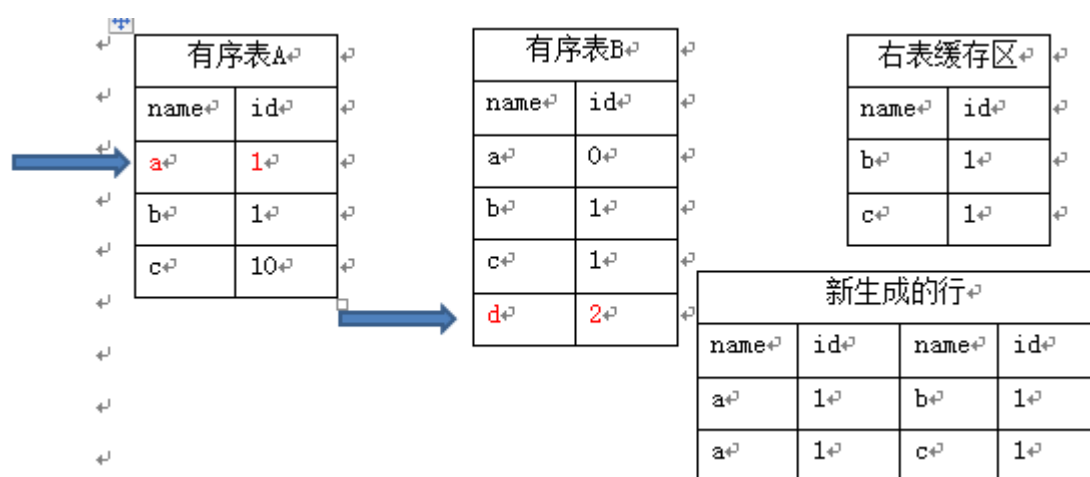
2.2 Merge Join算法

假设join操作: `select * from A left join B on A.id=B.id;`

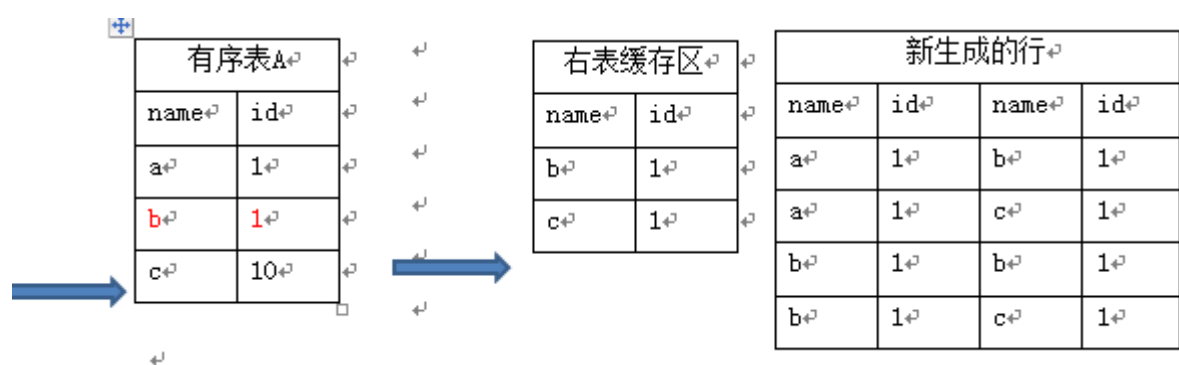
先通过等值连接条件A.id=B.id分别将A,B表排好序。



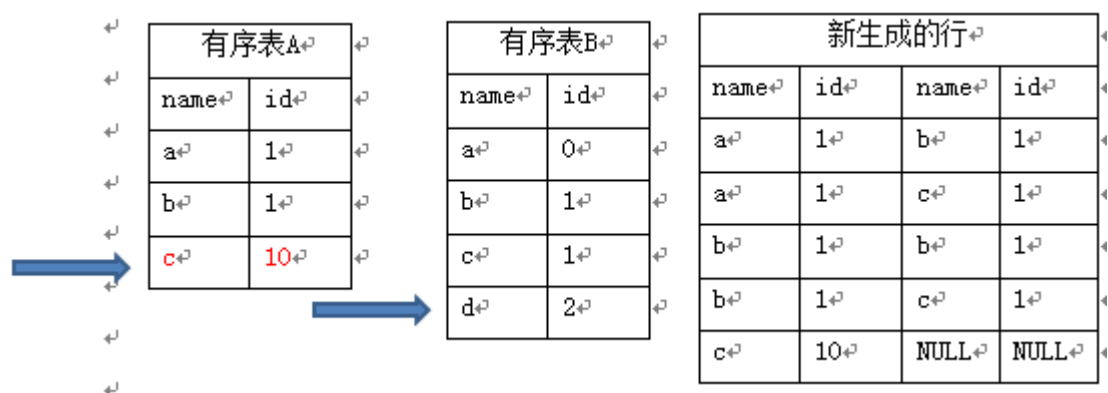
用左表的某一行，到右表中依次检查右表的各个行。将满足连接条件的左表的行和右表的行join成新行。同时将右表中满足等值连接条件的行保存到一个右表缓存区中



取左表的下一行，并判断和左表的上一行是不是满足等值连接条件。若满足，则和右表缓存区中的行进行join；否则清空右表缓存区，同时和右表中剩下的行进行join



对于左表的某一行，如果右表中没有一行和其满足等值连接条件。对于内连接，则抛弃这一行；对于外连接，则将相应位置NULL之后join成新生成的行



对于新生成的行，用不等值条件进行过滤。对于内连接，抛弃不满足不等值连接条件的行；对于左连接，将不满足不等值连接条件的行中来自右表的数据全部置为NULL之后输出

来自右表的数据置NULL后输出			
name	id	name	id
a	1	NULL	NULL

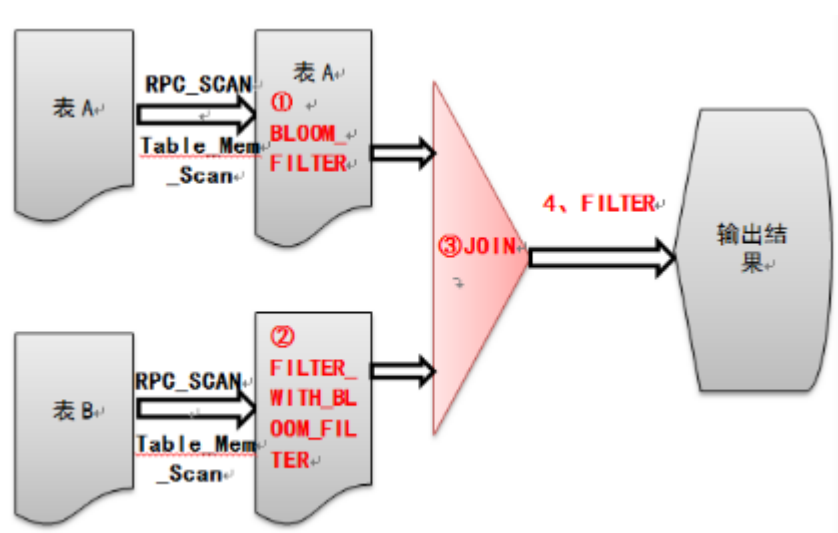
对于右连接，将不满足不等值连接条件的行中来自左表的数据全部置为NULL之后输出；

来自左表的数据置NULL后输出			
name	id	name	id
NULL	NULL	b	1

两表join的前提是两表的数据必须是有序的，不然算法会出错。所以Ocean Base限制join操作至少要有有一个等值连接条件。

两表join的前提是两表的数据必须是有序的，不然算法会出错。所以Ocean Base限制join操作至少要有有一个等值连接条件。

2.3 Bloomfilter Join基本原理图



如图所示，JOIN处理可以分四个步骤：


1. 生成Bloom Filter：将参与JOIN的前表生成Bloom Filter。先将前表所有复合查询条件的数据获取到Merge Server，并生成Bloom Filter。

2. Bloom Filter过滤：将参与JOIN的前表生成的Bloom Filter作为SQL Expression的一个参数，序列化后传入后表的Chunk Server，过滤后表数据。无等值连接条件时，Bloom Filter为空，后表进行全表扫描。
3. Join处理：按照不同规则进行等值Join操作。
4. FILTER处理：按照不等值条件进行过滤，进而输出最终结果。

2.4 Bloomfilter Join算法

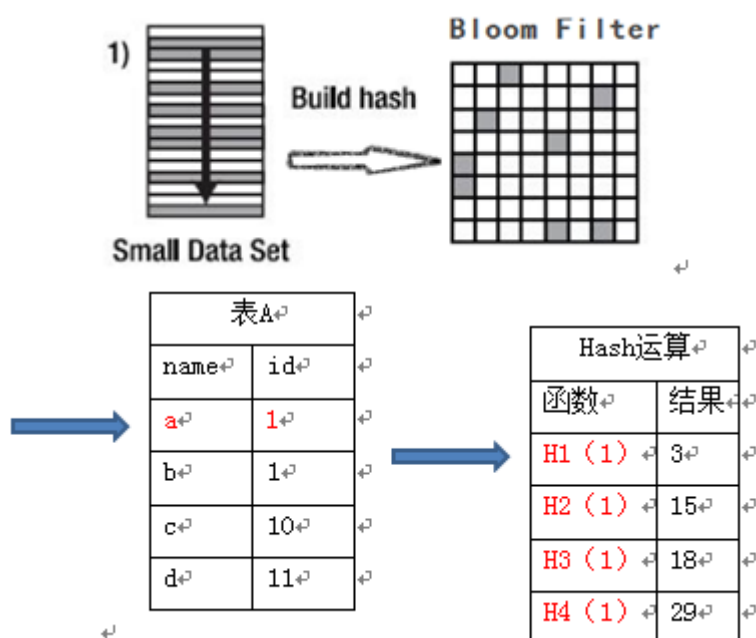
假设join操作: `select * from A left join B on A.id=B.id`

先通过表达式过滤，将符合条件的表A数据取到Merge Server




name	id
a	1
b	1
c	10
d	11

对于内连接和左连接，将表A的Join列做Hash运算，生成Bloom Filter并加入后缀表达式中，对于全外连接和右连接，Bloom Filter为空，即做全表扫描

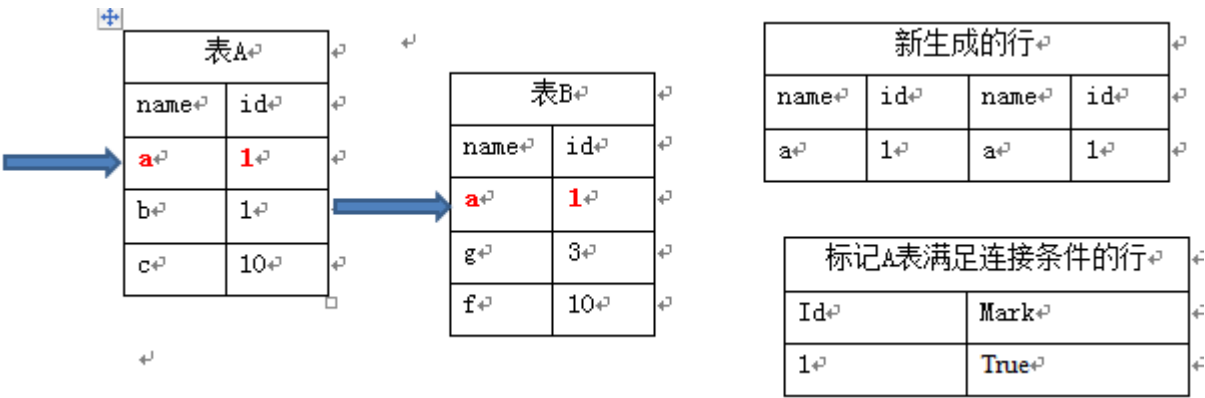


根据表A生成的表达式，到表B中取符合条件的数据到Merge Server

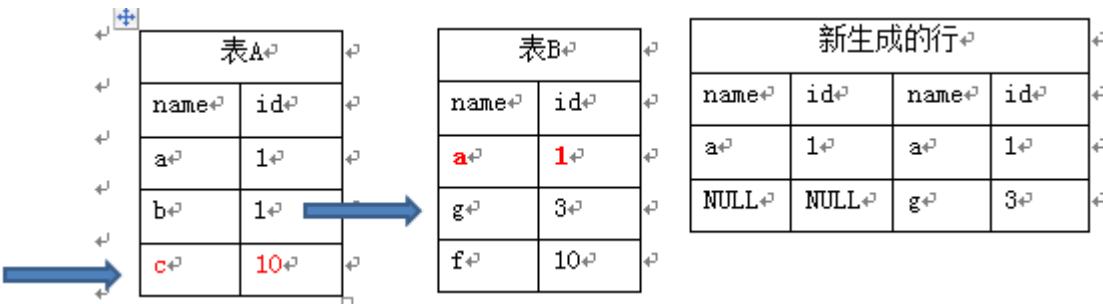


name	id
a	1
g	3
f	10

用B表的某一行，到A表中依次检查A表的各个行。将满足连接条件的A表的行和B表的行join成新行，对于左连接和全外连接，标记A表满足连接条件的行数



如果A表中没有一行和其满足等值连接条件，对于全外连接和右连接，则将相应位置为NULL之后join成新生成的行



对于左连接、全外连接，将表A与标记A表每条记录对比，将未在标记A表中出现的行的数据全部置为NULL之后输出

来自右表的数据置NULL后输出			
name	id	name	id
b	1	NULL	NULL

对于新生成的行，用不等值条件进行过滤。对于内连接，抛弃不满足不等值连接条件的行；对于左连接，将不满足不等值连接条件的行中来自右表的数据全部置为NULL之后输出

来自右表的数据置NULL后输出			
name	id	name	id
a	1	NULL	NULL

对于右连接，将不满足不等值连接条件的行中来自左表的数据全部置为NULL之后输出

来自左表的数据置NULL后输出			
name	id	name	id
NULL	NULL	b	1

三. 概要设计

3.1 原Join的流程

词法分析，语法解析：把用户输入的 SQL 语句解析成语法树

生成逻辑计划：根据语法树生成中缀表达式和其他信息，存到逻辑计划中

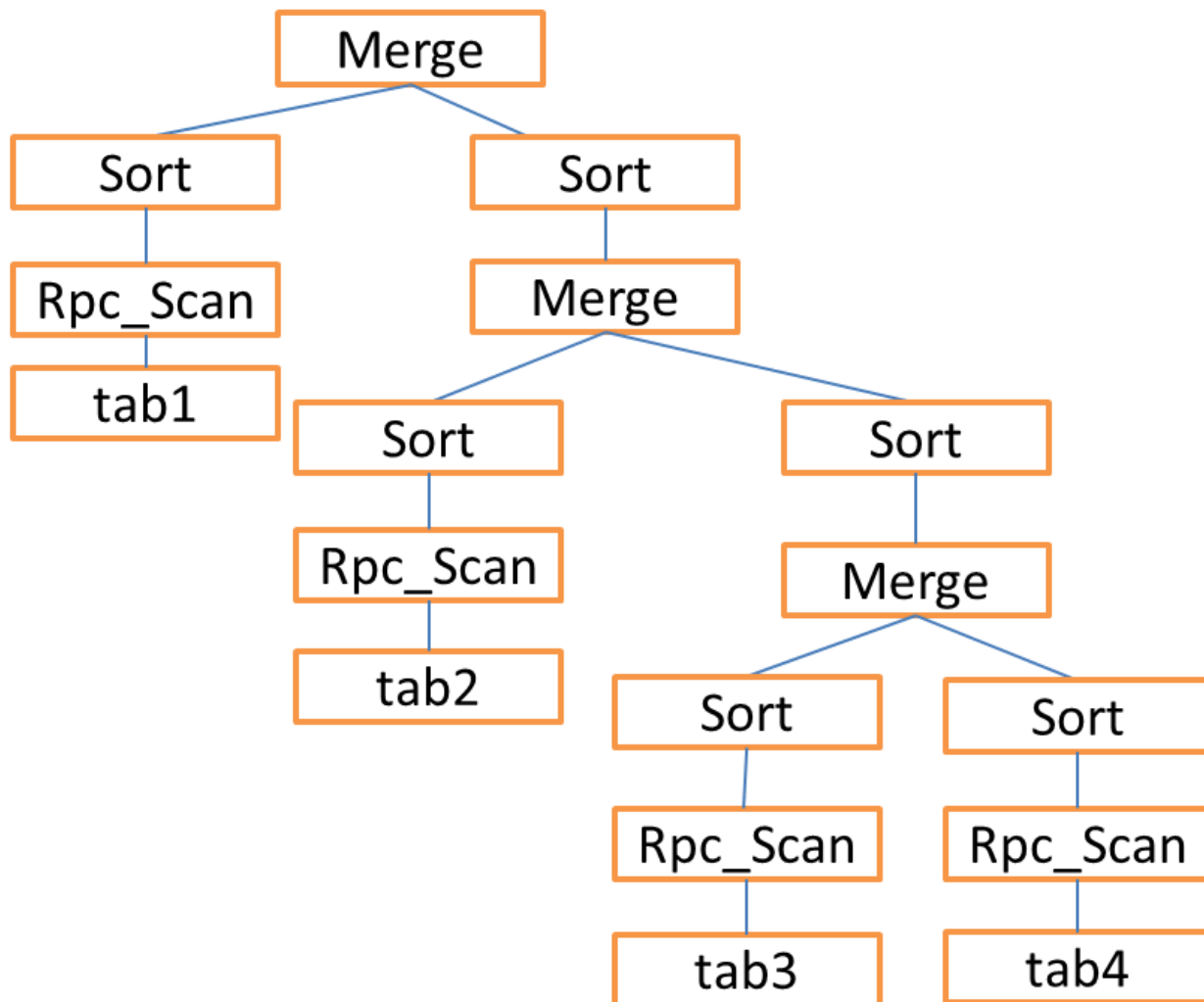
生成物理计划：根据逻辑计划里面存的相应信息，生成物理操作符，并确定操作符之间的父子关系。从逻辑计划中取出中缀表达式，转成后缀表达式存到相应的物理操作符里面。分为两种执行方式

```
select * from tab1,tab2,tab3 where tab1.col1=tab2.col2 and tab1.col1=tab3.col1
```

- 逻辑计划中无join运算符匹配，所有表执行Inner Join
- 生成物理计划所有表执行：gen_phy_joins(logical_plan, physical_plan,

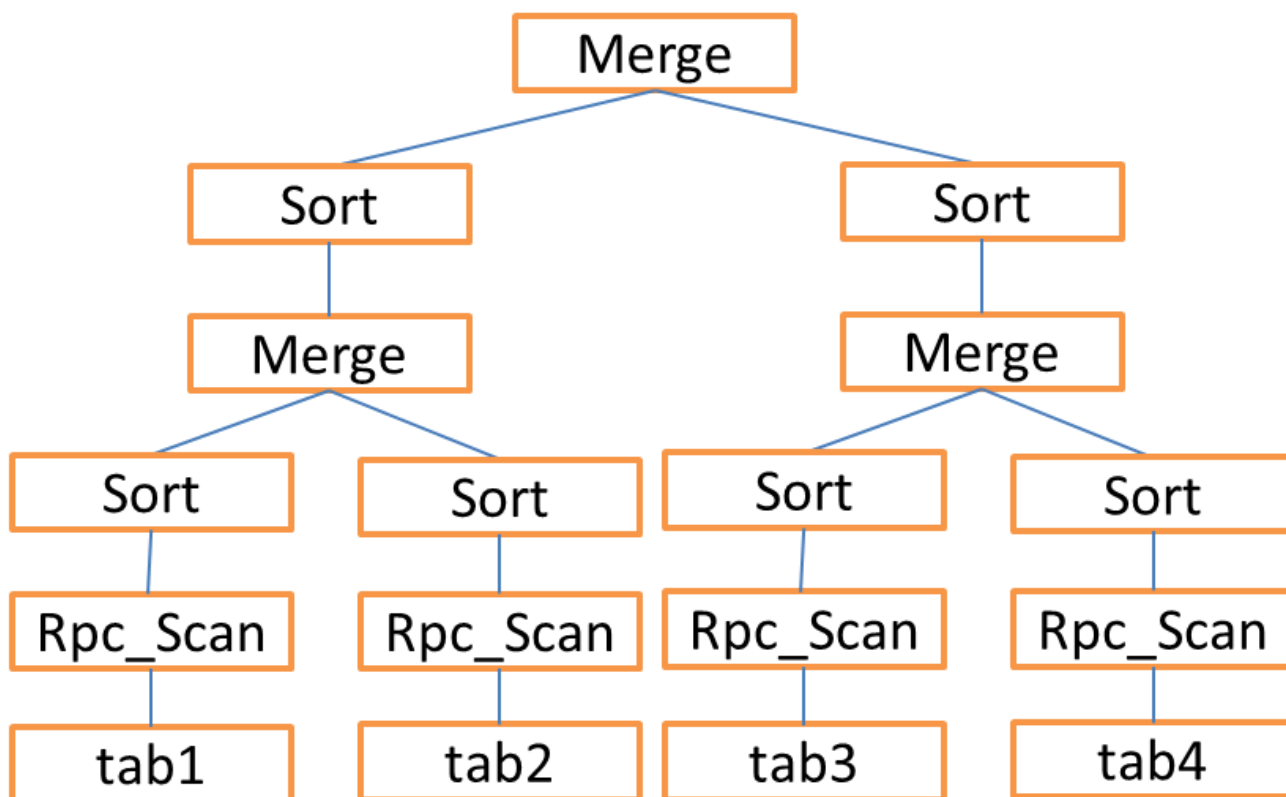
err_stat, select_stmt, ObJoin::INNER_JOIN, phy_table_list, bitset_list, remainder_cnd_list, none_columnize_alias)

- 此时,四张表及以上的内连接会生成的查询树会产生两种结构
- select * from tab1,ta2,tab3,tab4 where tab3.c1=tab4.c1 and tab3.c2=tab2.c1 and tab4.c1=tab1.c2

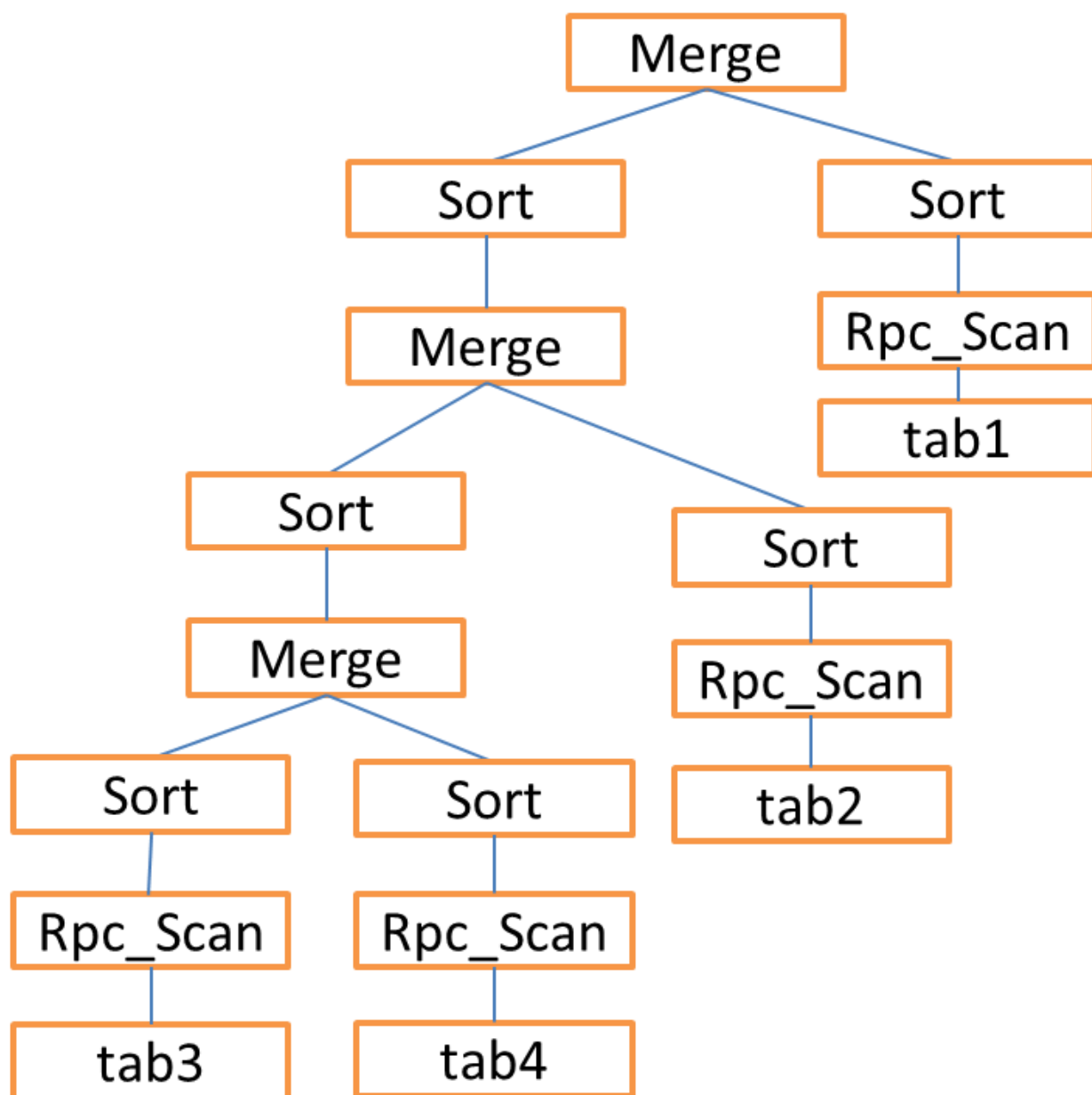


- select * from tab1,ta2,tab3,tab4 where tab3.c1=tab4.c1 and tab1.c2=tab2.c1 and

tab4.c1=tab1.c2



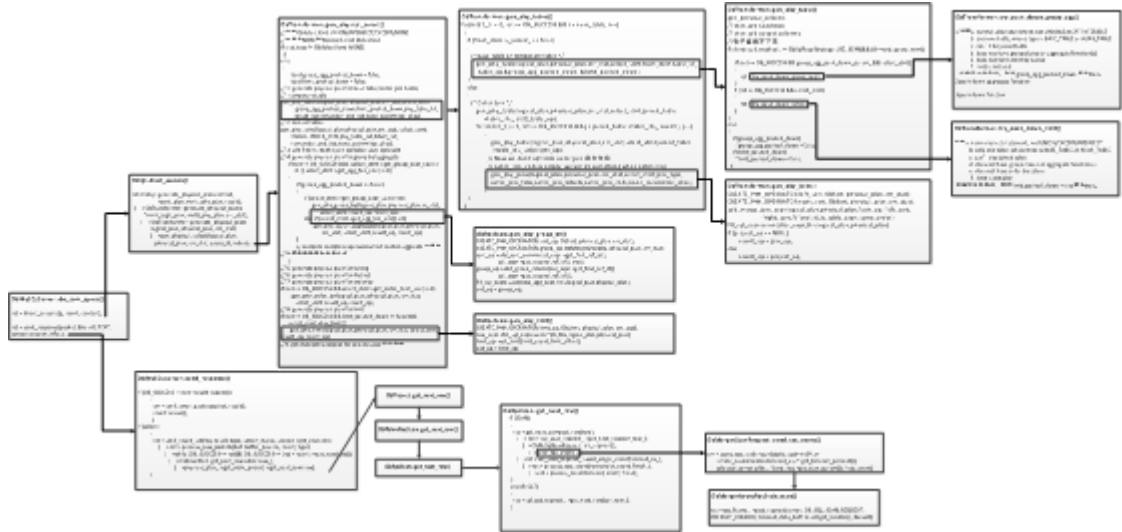
- `select * from tab1 left join tab2 on tab1.col1=tab2.col2 right join tab3 on tab2.col1=tab3.col1`
 - 逻辑计划匹配FULL_OUTER_JOIN/ LEFT_OUTER_JOIN/ RIGHT_OUTER_JOIN/ INNER_JOIN，每两张表确定一种Join运算
 - 生成物理计划时每两张表执行一次：`gen_phy_joins(logical_plan, physical_plan, err_stat, select_stmt, join_type, outer_join_tabs, outer_join_bitsets, outer_join_cnds, none_columnlize_alias)`
 - 此时,只会产生一种查询树结构



物理计划执行

Merge Join->Open()：构造查询列参数、输出列参数、过滤条件，left_op->Open()、right_op->open()，打开子运算符ObSort->open()，最终调用Rpc_Scan->Open()或Table_Mem_Scan->Open()。Rpc_Scan->Open()将这些信息发送到CS，CS根据这些信息来拉取数据。

物理计划Merge Join->get_next_row()：向CS发送请求，接受到CS返回的数据，两表做Merge Join，处理后返回给客户端。



具体流程请看流程图：select流程-Join的实现。

3.2 增加Bloomfilter Join功能后的Join流程

3.2.1 词法分析，语法解析

把用户输入的 SQL 语句解析成语法树。并增加对Hint中指定的Join算法进行解析。Hint中指定两两表之间的Join规则，例如

```
Select /*+join(Bloomfilter_join,merge_join,Bloomfilter_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(Bloomfilter_join,merge_join,Bloomfilter_join,merge_join)*/括号内需要符合“join_type (,join_type) ”规则，按照join顺序，表明两表之间的Join使用的算法规则

对于select * from tab1,tab2 where tab1.col1=tab2.col2;此类Join查询需要DBA改为为Inner Join后再指定Hint，完成Bloomfilter Join；否则将执行原Merge Join操作。

3.2.2 生成逻辑计划

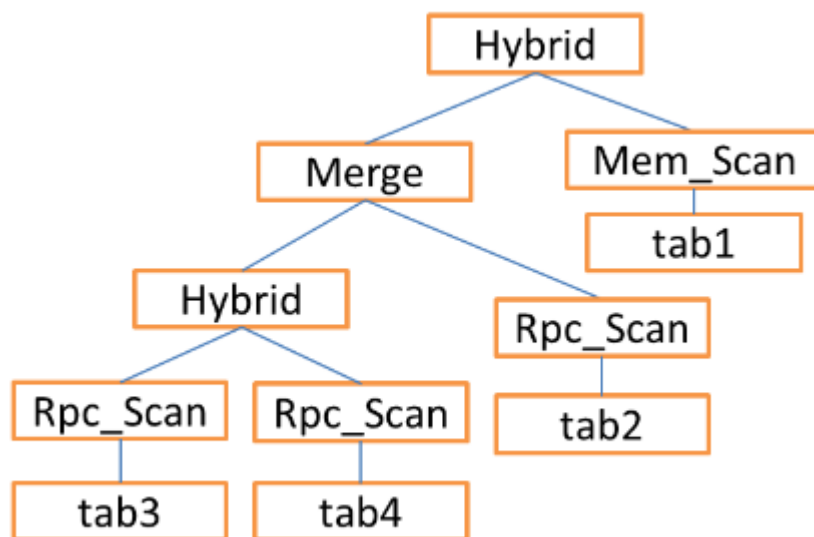
根据语法树生成中缀表达式和Hint信息，判断hint中指定的join信息是否符合规则，存到逻辑计划中

```
int resolve_hints(
    ResultPlan * result_plan,
    ObStmt* stmt,
    ParseNode* node)
{
    //读取ParseNode* hint_node = node->children[i]; 中的Hint信息
    //存储到ObQueryHint& query_hint 中，供生成物理计划时使用。
}
```

3.2.3 生成物理计划

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运算符和**Bloomfilter 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

由于两两表之间指定了Join的算法（无默认的Inner Join），只会产生一种结构



四、详细设计

4.1 Bloom Filter

Bloom Filter位于ObSqlExpression的成员(ObPostfixExpression) post_expr内。common::ObBloomfilterV1 bloom_filter。

ObPostfixExpression内有对Bloom Filter的序列化和反序列化处理。

假定误报率为0.01，位利用率50%，利用Bloom Filter的相关计算公式，可得1000000条记录需要6.44个hash函数及9200000bit（1150KB）（小于OB单个数据包2MB的上限，libeasy架构制限）。

备注：Bloom Filter相关公式。p：误判率，m：位数组大小，n：总数据数目，k：所需哈希函数数目。

对后表查询引入的Bloom Filter的功能：CS使用Bloom Filter初步过滤数据。因为Bloom Filter的误报（不会漏报），保证CS取得的数据集是符合条件的集合的超集。

综合上述，OB初步设置的Bloom Filter包含6个相互独立的hash函数，固定大小（9200000）bit数组，即在误判率0.01左右时，最大仅支持后表查询结果集值域不大于1000000。注意此时单个Bloom Filter为1150KB，在单个请求包2MB的限制下成立。但是本文OB做了处理，一旦大于上限，Bloom Filter失效，不再传送至其它Server过滤数据，当前的MS直接销毁。

4.2 Hash Map

BloomfilterJoin引入Bloom Filter后，MS从各个CS收到的是符合主查询的最终集合的超集。因此MS必须进行一次的严格的过滤以取得最终查询结果集。

OB既存的设计有两种Hashmap，如下所示：

1. common/hash/ob_hashmap.h文件内声明的class ObHashMap。这个ObHashMap的桶数由外部传入，每个桶内的数据个数不固定，桶内的数据以链表的形式组织

2. updateserver/ob_lighty_hash.h文件内声明的class LightyHashMap。这个LightyHashMap的桶数固定，每个桶内的数据个数也固定（不大于65535）

本文使用的HashMap是第一种ObHashMap的实例对象。由于OB没有预估程序衡量子查询结果集数目，因此默认设置桶的数目为100000个。在最优情形下1000000万条记录均匀分布到100000个桶内，每个桶10个节点，查找一个节点最低时耗 $O(2)$ [一次哈希计算，一次比对]，最大时耗 $O(11)$ [一次计算，十次比对]。

HashMap的构建实现方案：

- 前提：复用OB中已实现的ObHashMap功能。默认ObHashMap的桶为100000个，桶内数据个数没有限制
- 方案
 1. 使用OB的ObHashMap接口创建ObHashMap1，接口传入桶数目100000
 2. 每一条记录都会被转换成Obj数组
 3. 使用选中的单个哈希函数计算2中的Obj数组，得到单个哈希值
 4. 哈希值对ObHashMap1的总桶数取余，假设为L，然后将Obj数组作为value值保存进第L-1个桶
 5. 循环2、3、4这三步，不断将记录添加进ObHashMap1

HashMap利用ObHashMap提供一个接口函数，目的是判断传入的参数是否包含在内。参数就是一条原始记录转换后的Obj数组。

Semi Join 优化

一、概述

OceanBase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。业务中存在大量的多表连接查询，在处理多表连接时，目前Ocean Base进行Merge Join运算，在Merge Server（MS）分发请求给相应的Chunk Server（CS），CS把所有数据返回给MS，MS进行排序后做Merge操作。若对大表进行查询，CS将传输大量不会产生连接关系的无效数据到MS，浪费了大量的网络传输时间和无效数据的排序时间等，导致查询缓慢。

若两张表连接查询，一张小表，一张大表，小表对于大表的区分性较好，则使用Hash Join的查询效率会优于Merge Join；若两张表数据量都很大，而且对左右表的过滤较少，则目前实现的Merge Join需要将两表数据存入内存，会导致内存不足。目前CBase1.2中提供的Bloomfilter Join可以解决左右表数据量很大的问题，但毕竟是少数情况。如果左表数据量不大，则Bloomfilter Join不适用。因此Semi Join就是这样一种方案，可以满足在左表数据量不大，且右表返回数据过大的情况下，根据左表返回给MS的结果集来构建过滤条件，发送到CS上，并用于过滤右表的数据，大大减少了数据传输量，减少了数据传输的网络延时。

二、详细设计

2.1 Merge Join和Bloomfilter Join带来的启发

（1）Semi join的操作符open执行时机。

原有的Merge Join在其open阶段先后打开左右两个Sort操作符。这意味着在Merge Join的open阶段已经将左右两张表的数据都保存在Sort操作符中，并且数据是有序的。因此，如果在Semi Join的open阶段同时打开左右操作符，我们就无法将左表的数据用于过滤右表。在Semi Join的设计中，右操作符需要在左操作符获取到左表数据，并构建好filter，传给Rpc_Scan操作符后，才可以执行open函数。这样才能保证对右表的数据进行过滤，而不是全表返回。

（2）Semi join的内存管理。

原有的Merge Join在内存管理上很合理，其在open阶段先后打开左右操作符。完成open后，左右操作符会各持有左右表排好序的完整数据副本，底层的Rpc_Scan操作符中不再持有数据缓存。也就是说，在整个物理计划执行过程的open阶段内存中只有这1个左右的数据副本。此后上层操作符调用Merge Join的get_next_row，在此阶段会依次从Sort操作符的数据缓冲区中取数据进行合并返回结果行。当所有数据都join完成后，执行Merge Join的close，并依次释放各个物理操作符的内存。这就保证了在整个物理计划执行过程中，对于Merge Join而言，左右子树一共申请了两个左右数据副本的内存。

Bloomfilter Join是对原有Merge Join进行的优化，但是其对于内存的管理并不如原有的Merge Join。在Bloomfilter Join的open阶段，其先打开左操作符（也就是Sort操作符），这个时候左子树就申请了1个数据副本左右的内存，一个是rpc scan中的无序副本，一个是sort中内存中的有序副本。其后它又循环调用了左操作符的get_next_row用来构造Bloom filter，并将数据放到一个新申请的内存中作为副本。这样其左子树一共申请了两个数据副本左右的内存。再加上右子树的1个数据副本左右的内存，一共就有差不多4个数据副本左右的内存。显然高于原有的Merge join，但是其通过Bloom filter过滤了右表的数据，使得右表的数据量减少，内存使用也相应的减少。

2.2 Semi Join 流程

Semi Join的Open阶段：

在semi join的open阶段，首先打开左子操作符（sort）。这时做操作符异步填充数据，并排序，等待semi join执行get_next_row获取数据。接下来循环调用左子操作符的get_next_row来构造用于过滤右表的filter，并将构造好的filter传给Rpc_Scan，这时重新打开左子操作符（sort）进行数据填充。然后打开右子操作符（sort）进行数据填充。最后一次调用左右子操作符的关闭子操作符，注意不是关闭左右Sort操作符，而是关闭sort的子操作符。如果在这个阶段关闭sort，那么在get_next_row将会得不到数据。而关闭sort的子操作符作用在于释放不必要的内存，因为数据已经保存在sort操作符中了。

其他操作符的open与原Merge Join一致。

Semi Join的Get_next_row阶段：

其他操作符的open与原merge join一致。

Semi Join的Close 阶段：

其他操作符的open与原merge join一致

2.3 Semi Join 的open阶段详细流程

具体步骤如下：

- 1) 首先打开左表操作符，这时左表的数据正异步传输回ms，并且在sort操作符中进行排序。
- 2) 接下来判断是否需要使用semi join，这是在逻辑计划转物理计划时对semi join做的一次优化，目的在于判断是否需要使用semi join。具体场景将在下面分析。如果需要使用semi join则进入步骤4，否则进入步骤3。
- 3) 将左表的数据从sort操作符中取出保存在left cache中，用于上层调用get_next_row取出数据。
- 4) 为右表构造filter，如果左表返回数据为零，则设置OB_INDEX_NUM_IS_ZERO 为true，如果filter大小超过2M包大小的限制，则将is_can_use_semi_join设置为false，默认为true。
- 5) 如果is_can_use_semi_join为true并且OB_INDEX_NUM_IS_ZERO为false，则将filter加入到rpc scan中。
- 6) 判断OB_INDEX_NUM_IS_ZERO是否为true，如果为true，则进入8，否则进入7。
- 7) 右表操作符打开，但是不进行排序，并且进入end。

- 8) 判断是否可以使用semi join，如果不可以则设置返回错误码，并进入end。否则进入9。
- 9) 打开右操作符，并进入end。
- 10) End。

2.4 Semi Join 的优化点

(1)将左表返回数据作为右表过滤条件，可以减少右表返回数据量。

(2)对于连接条件中除了等值条件还有其他非等值连接条件的sql语句，原merge join对于这类sql的处理，是不会将其他连接条件作为过滤条件，在获取数据时就对数据表进行过滤，而是在左右两个表的数据全部返回后再进行非等值连接条件的判断。对于left join、right join、full join上述操作是情有可原的，但是对于inner join，由于会将不符合非等值连接条件的结果过滤，因此完全可以将非等值连接条件作为左右两表的过滤条件。因此semi join在实现时会将这种情况作为一个优化项。

(3)对于边界值的处理。对于数据量较大表，单次查询的速度较慢，少则几秒，多则几十秒。而Merge Join在左右表没有过滤条件的情况下，是全表返回的。比如，`select * from t1 join t2 on t1.id=t2.id where t1.id < 0` 这类sql查询，而t2数据较多，而又恰好没有过滤条件，而t1中id列中没有满足小于0的数据，因此左表返回的结果是空。但是这个时候右表仍在返回数据（假设数据比较大，则返回时间较慢），并不能及时的返回为空集的结果给客户端。而对 `select * from t1 join t2 on t1.id=t2.id and t1.id<0`，那就更慢了，因为这样的sql相当于左表没有过滤条件，也是返回全表后再进行过滤。

2.5 Semi Join 应用场景分析

1.两表join

根据左右两表全表数据以及结合查询时的过滤条件，考虑到以下四种返回MS的数据情况：

- (1)左表有数据，右表无数据。使用右表数据去过滤左表。
- (2)左表有数据，右表有数据。这种情况将会在后文有进一步讨论。
- (3)左表无数据，右表无数据。采用Merge Join。
- (4)左表无数据，右表有数据。使用左表数据过滤右表数据。

2. 三表join

当有三张表进行join时，有如下两种办法表与表之间的join顺序：

- (1)程序内部优化，即通过统计每张表相应的过滤条件的多少，然后进行降序排序，最后以过滤条件较多的表依次作为左表，以此表序生成物理计划。但是这样简单的通过每张表的过滤条件的多少来规定连接顺序可能起不到优化的效果，甚至可能其反作用。
- (2)由SQL改造人员来指定join顺序。

3. 表的join顺序不可改变

由于表的join顺序不可改变，因此无法通过改变连接顺序来提高执行效率。对此有以下几方面的考虑：

- (1)如果a表没有过滤条件，而b表有过滤条件，并且b表的过滤条件对于b表来讲过滤性很好，那么即使你使用了hint指定用semi join，也应该判断在这种情况下不应该由a表来过滤b表。因为如果这样做，那就是在浪费时间。
- (2)如果a表有过滤条件，而b表也有，那么无法确定使用哪张表来过滤，因为无法知道哪个表返回的结果多。因此这种情况下，默认不使用a表来过滤b表。

(3)如果a表有过滤条件，而b表没有，则用a表来过滤b表，这是最简单的处理方法。对于left join来说，最后join完成的结果集中，右表数据一定是小于等于左表的。那么在用等值连接条件来过滤上层Semi join操作符的右表的时候，就要考虑左右表完成连接后相应等值列（上层Semi join的等值相关列）的值哪个多、哪个少了。对于semi join来说，用可以产生较多列值的表数据来过滤上层Semi join的右表是最好的。

4. 其他场景分析：

假设有四张表test1、test2、test3、test4，表的Schema如下：且假设每张表的大小都为1000万数据量，数据是随机生成的。

field	type	nullable	key	default	extra
id	int32	0	1	NULL	
k	int32	0	0	NULL	
c	varchar(120)	0	0	NULL	
pad	varchar(60)	0	0	NULL	

1). 场景1：

Sql语句： `select /*+JOIN(SEMI_JOIN)*/ * from test1 a join test2 b on a.id=b.id where a.id<2;`

执行时间： 1 row in set (0.03 sec)

原因：左表有过滤条件，返回ms只有一条数据，因此构建的filter中也仅有一项数据，右表根据这个filter过滤出相应的数据返回ms。

2). 场景2：

Sql语句： `select /*+JOIN(SEMI_JOIN)*/ * from test1 a join test2 b on a.id=b.id and a.id<2 and b.id<10;`

执行时间： 1 row in set (0.01 sec)

原因：原来的merge join流程中的inner join，对于连接条件中除了等值连接条件以外的其他连接条件，是不会添加到相应的表的filter中的，这就导致左右两表全表返回，响应时间增加。Semi join在连接类型为inner join的情况下，将on表达式中的除了等值连接条件以外的连接条件分别传到左右表，进行数据过滤。

3). 场景3：

Sql语句： `select /*+JOIN(SEMI_JOIN)*/ * from (select * from test1) a join (select * from test2) b on a.id=b.id where a.id<100;`

执行时间： 超时

原因：semi join 不支持join左右两边为子查询，会转换成merge join执行。如果想要提升效率，可在子查询中适当添加过滤条件。

4). 场景4：

Sql语句： `select * from (select /*+JOIN(SEMI_JOIN)*/ * from test1 join test2 on test1.id=test2.id and test1.id<20);`

执行时间： 1 row in set (0.03 sec)

原因：如果子查询里面使用join，需要在子查询的作用范围内，增加hint。

5). 场景5：

Sql语句： `select /*+JOIN(SEMI_JOIN)*/ * from test1 a join test2 b on a.id=b.id where a.id<0;`

执行时间： Empty set (0.03 sec)

原因： semi join在处理边界值问题时，如果左表返回结果集为空，那么其右表就不会再打开，继而返回空集。缩小响应时间。

三、Semi Join的使用

Semi join 的使用需要借助Hint功能，下面分别介绍使用二级索引和不使用二级索引的使用方法。使用Hint功能，需要在连接数据库时要在连接语句后加入-c，例如：“mysql -h 10.10.10.1 -P2828 -uadmin -padmin -c”。这样才能让客户端识别带Hint的sql语句。

在Hint中的关键字主要有SI和SB。其中SI是根据左表全表数据构建过滤器发送到CS进行过滤，SB是跟左表的数据构建between表达式（与between功能过滤效果相同）来过滤右表数据。

由于SI和SB作用相同，以下介绍用SI作为例子，同样也是用SB关键字。

3.1 两张表做连接（无索引）

例如：`select * from t1 join t2 on t1.id = t2.id where t1.id=1;`

两张表之间做join，并且id为两张表的主键。如果想要使用semi join需要在select 后面加入 `/*+JOIN(SI)/*` 关键字，上面的sql就变成了如下所示的sql：`select /*+JOIN(SI)*/ * from t1 join t2 on t1.id = t2.id where t1.id=1;`

可以使用explain语法来查看物理计划，确定使用的是否为semi join。

3.2 多张表做连接（无索引）

对于多表连接，需要指定每个连接是否使用semi join，如果使用则加入SI关键字，否则加入MERGE_JOIN关键字。

例如：`select * from t1 join t2 on t1.id = t2.id join t3 on t2.id = t3.id where t1.id=1 and t2.id=1;`

如果t1和t2的join不想用Semi join，则需要用merge_join关键字来补位，t2和t3用Semi join，则使用SI关键字。切记，不管是否使用Semi join，Hint中SI和MERGE_JOIN关键字的总个数需要与sql语句中的join关键字的个数相同。使用Semi Join的sql如下：`select /*+JOIN(MERGE_JOIN,SI)*/ * from t1 join t2 on t1.id = t2.id join t3 on t2.id = t3.id where t1.id=1 and t2.id=1;`

注意事项： 如果表达式放在where中报错的话，可以移到on表达中。比如以下sql：`select /*+JOIN(SI)*/ * from t1 join t2 on t1.id = t2.id where t1.id=1;`

可以写成：`select /*+JOIN(SI)*/ * from t1 join t2 on t1.id = t2.id and t1.id=1;`

3.3 多张表做链接（有索引）

如果需要join的表上建立了索引，并且要使用semi join，那么如果不在hint中指定要使用的索引，就会默认使用第一个索引表。

例如：`select /*+JOIN(SI,SI)*/ * from t1 join t2 on t1.k = t2.k where t1.id=1;`

其中k分别为t1和t2的索引表中的主键。这类sql无需在Hint中指定使用哪张索引表。但是如果使用Hint如：

`select /*+JOIN(SI,SI)+INDEX(t2 t2_index_k)*/ * from t1 join t2 on t1.k = t2.k where t1.id=1;`

那么索引表的原表必须是join关键字右边的表，并且使用的索引表t2_index_k中的主键必须是on表达式中右表的列，在上述sql中，索引表必须是t2的索引表，索引表t2_index_k的主键必须是k，也就是on表达式（t1.k = t2.k）中t2的列k。

Sequence

一、SEQUENCE的功能及使用

SEQUENCE的功能是生成一个序列，这个序列可以递增、递减或为一个不变的常数。根据客户端请求返回相应的数值。具体的使用方法如下。

1.1创建SEQUENCE

- 语法规则

```
CREATE [OR REPLACE] SEQUENCE sequence-name [AS data-type / AS INTEGER] [START WITH numeric-constant / START WITH 1] [INCREMENT BY numeric-constant / INCREMENT BY 1] [MINVALUE numeric-constant / NO MINVALUE] [MAXVALUE numeric-constant / NO MAXVALUE] [CYCLE / NO CYCLE] [CACHE integer-constant / NO CACHE / CACHE 20] [ORDER / NO ORDER] [QUICK / NO QUICK]
```

注：创建SEQUENCE时，以上选项均为可选项，对于用户未输入的参数均采取默认值选项，具体的默认选项见选项含义。

- 选项含义

OR REPLACE

此选项无默认值，若要新建的SEQUENCE的sequence-name已经存在，则调用OR REPLACE，对已经存在的SEQUENCE进行重新定义。

AS data-type

此选项用于指定SEQUENCE的数据类型，可以为：INTEGER、DECIMAL。其中INTEGER包括：SMALLINT、INTEGER、INT、BIGINT等整型，DECIMAL的范围（scale）为1~31，精度（precision）必须为0。默认为INT类型，即32位整型。

START WITH

此选项用于设定SEQUENCE的起始值，正数、负数、零均可但必须为整型。

当不指定START WITH值时：

- 若SEQUENCE为递增序列，且指定了最小值MINVALUE，则默认的START WITH值为MINVALUE；若未指定最小值，则默认的START WITH值为1。
- 若SEQUENCE为递减序列，且指定了最大值MAXVALUE，则默认的START WITH值为MAXVALUE；若未指定最大值，则默认的START WITH值为-1。

注：起始值可以不在最大最小值所限定的范围之内。

INCREMENT BY

此选项用于设定SEQUENCE每次自增或自减的步长。该值为正数时，SEQUENCE为递增序列；该值为负数时，SEQUENCE为递减序列；该值为0时，SEQUENCE为常数序列。默认为1。

MINVALUE or NO MINVALUE

此选项用于设定SEQUENCE的最小值。若用户在创建SEQUENCE时未指定MINVALUE，或设定为NO MINVALUE，则采取默认值。

- 递增序列中：若指定了START WITH的值，则MINVALUE的默认值为START WITH的值；若未指定START WITH的值，则MINVALUE的默认值为1（此时的START WITH默认为1，两者仍相等）。

- 递减序列中：MINVALUE默认为data_type能表示的最小值，例如：SEQUENCE为INT类型，则32位整型所能表示的最小值为-2147483648，此时MINVALUE默认为-2147483648。

注：递减序列中，未指定最大值MAXVALUE，则最大值默认为START WITH的值（START WITH取值参考相关章节），此时若指定MINVALUE，则MINVALUE必须小于等于MAXVALUE。

例如：

```
mysql> create or replace sequence seq1 increment by -1 minvalue 0;
ERROR 5001 (42000): Minvalue must equal or small than maxvalue.
```

MAXVALUE or NO MAXVALUE

此选项用于设定SEQUENCE的最大值。若用户在创建SEQUENCE时未指定MAXVALUE，或设定为NO MAXVALUE，则采取默认值。

- 递增序列中：MAXVALUE默认为data_type能表示的最大值，例如：SEQUENCE为INT类型，则32位整型所能表示的最大值为2147483647，此时MAXVALUE默认为2147483647。
- 递减序列中：若指定了START WITH的值，则MAXVALUE的默认值为START WITH的值；若未指定START WITH的值，则MAXVALUE的默认值为-1（此时的START WITH默认为-1，两者仍相等）。

注：递增序列中，未指定最小值MINVALUE，则最小值默认为START WITH的值（START WITH取值参考相关章节），此时若指定MAXVALUE，则MAXVALUE必须大于等于MINVALUE。

例如：

```
mysql> create or replace sequence seq1 increment by 1 maxvalue 0;
ERROR 5001 (42000): Minvalue must equal or small than maxvalue.
```

CYCLE or NO CYCLE

此选项用于设定SEQUENCE到达边界（即MINVALUE与MAXVALUE所确定的范围）后是否重新开始循环，若用户不指定则默认为NO CYCLE，NO CYCLE表示当SEQUENCE到达边界后无法再继续生成值。

当指定CYCLE时：若SEQUENCE为递增序列，则到达最大值后，从最小值MINVALUE开始重新生成序列值，而不是从START WITH开始；若SEQUENCE为递减序列，则到达最小值后，从最大值MAXVALUE开始重新生成序列值，也不是从START WITH开始。默认为NO CYCLE。

CACHE or NO CACHE

此选项用于设定SEQUENCE是否在内存中缓存值，此选项主要用于提高效率，若用户在客户端未输入此命令，则默认为CACHE 20；若用户输入NO CACHE则不缓存；此外用户可以自行确定缓存值的数量。默认为CACHE 20。

ORDER or NO ORDER

在多重分区或pureScale环境下，当两个以上用户共用一个SEQUENCE时，由于各自在内存中缓存了一部分值（例如缓存了20个值），在调用时会出现申请到1，21，2，22这样的情况。此选项用于设定SEQUENCE是否严格按顺序生成，此时选择ORDER，SEQUENCE将以严格的顺序生成。默认为NO ORDER。

QUICK or NO QUICK

该字段为OB特有的一个字段，当不指定该字段时，默认为NO QUICK，该sequence的使用将保持严格的独占（排它锁）递增（递减），同一时刻只能有一个session独占该sequence，其他session处于等待状态。当指定该字段时，该sequence的使用将不考虑sequence数据的一致性，多session下，所有session共享该sequence，如果所有session同时使用该sequence，可能出现sequence值重复情况，但是如果用户能保证使

用时，避免多session同时使用，则该字段将大大提高使用效率。

- 应用举例

```
CREATE SEQUENCE or REPLACE test_seq AS INTEGER START WITH 1 INCREMENT BY 2 MAXVALUE 20 NO CYCLE  
CACHE 5 NO ORDER QUICK;
```

1.2 修改SEQUENCE

- 语法规则

```
ALTER SEQUENCE sequence-name [RESTART / RESTART WITH numeric-constant] [INCREMENT BY  
numeric-constant ] [MINVALUE numeric-constant / NO MINVALUE] [MAXVALUE numeric-constant /  
NO MAXVALUE] [CYCLE / NO CYCLE] [CACHE integer-constant / NO CACHE] [ORDER / NO ORDER]  
[QUICK / NO QUICK]
```

- 选项含义

RESTART

此选项用于使SEQUENCE重新从开始生成。若用户输入RESTART，则从START WITH设置的值开始生成；若用户输入RESTART WITH numeric-constant，则SEQUENCE将从numeric-constant的值开始重新生成。

注：此选项无法修改创建SEQUENCE时确定的START WITH值。

其余选项

其余选项的含义与创建时的参数含义保持一致，在此不再赘述。需要注意的是ALTER SEQUENCE语句与UPDATE语句类似，是对SEQUENCE的修改，未输入的参数保持不变。

- 应用举例

```
ALTER SEQUENCE RESTART INCREMENT BY 1 MINVALUE 1 MAXVALUE 100 NO CYCLE NO CACHE NO ORDER QUICK;
```

1.3 删除Sequence

- 语法规则

```
DROP SEQUENCE sequence-name RESTRICT
```

- 选项含义

RESTRICT

用于检查是否有存储过程等正在依赖要删除的SEQUENCE，若有依赖项则禁止删除。若未输入该参数则可以直接删除相应的SEQUENCE。

注：目前未实现，语法也不支持。

- 应用举例

```
DROP SEQUENCE test_sequence;
```

1.4 INSERT语句中调用Sequence

- 语法规则

```
INSERT INTO table-name(column_1,...) VALUES(NEXTVAL / PREVVAL FOR sequence-name,...)
```


- 选项含义

NEXTVAL

此选项使FOR关键字指定SEQUENCE生成一个新值。

PREVVAL

此选项调用SEQUENCE的当前值。

说明：

在DB2中，当插入多行数据时，所有出现NEXTVAL的列必须保证严格的对齐（包括格式与内容），例如：

1、INSERT INTO TEST VALUES (NEXTVAL FOR SEQ1, ...), (NEXTVAL FOR SEQ1 +2, ...)..是不可以的，这些行中出现NEXTVAL字段的列必须保持完全的一致。在OB中无此限制。

2、在插入列中可以使用PREVVAL（产生序列的当前值）这个功能的，需要注意的是，在一次插入过程中，所有的PREVVAL共享一个值。简单说就是同一个INSERT语句中同一个SEQUENCE的所有PREVVAL的值相同。

- 应用举例

```
INSERT INTO *test_table*(id) VALUES(NEXTVAL FOR *test_sequence*);
```

1.5 INSERT语句中调用Sequence

- 语法规则

```
SELECT NEXTVAL / PREVVAL FOR sequence-name FROM table-name
```

- 选项含义

NEXTVAL

此参数用于生成并返回FOR关键字后sequence-name指定的SEQUENCE的下一个值。

PREVVAL

此参数用于生成并返回FOR关键字后sequence-name指定的SEQUENCE最近一次生成的值。因此，在一个SEQUENCE未调用NEXTVAL之前，无法使用PREVVAL关键字。

FOR

FOR关键字后面的sequence-name用于指定SEQUENCE。

FROM

FROM后的table-name必须存在，此表有多少行数据则该条SQL语句就输出多少行NEXTVAL或者PREVVAL。

- 应用举例

- 查询当前sequence值：

```
SELECT PREVVAL FOR *test_sequence* FROM *table_sequence*;
```

- 查询后续sequence值：

```
SELECT NEXTVAL FOR *test_sequence* FROM *table_sequence*;
```

注：查询一次SEQUENCE的NEXT VALUE意味着该SEQUENCE生成一个值，而生成后的值无法再被使用。

1.6 UPDATE语句中使用Sequence

- 语法规则

```
UPDATE table_name SET c1= NEXTVAL/PREVVAL FOR sequence_name... WHERE id = PREVVAL FOR  
sequence_name2...
```

- 选项含义

NEXTVAL FOR

该关键字组将会生成一个FOR后sequence名字所对应的序列号

PREVVAL FOR

该关键字组将会生成一个FOR后sequence名字所对应的序列号，该序列号是该sequence最后使用过的NEXTVAL的值，也就是当前有效next值的前一个值。

说明：在update的set子句中，可以出现NEXTVAL/PREVVAL这两种形式中的任意一种，而且可以进行任意的混合计算（一般人不会这么干）。

在update的where语句中，只可以出现PREVVAL这种形式。

以上OB和DB2是保持一致的。OB目前已经支持批量的update，对应的sequence也支持，用户可以在where中使用范围条件的sequence进行批量更新。

- 应用举例

```
UPDATE *sequence_lijq* SET *c1* = NEXVAL FOR *seq1* WHERE *id* **<***=* PREVVAL FOR *seq2* ;
```

1.7 DELETE语句中使用Sequence

- 语法规则

```
DELETE FROM table_name WHERE col = PREVVAL FOR sequence_name...;
```

- 选项含义

PREVVAL

该字段是SEQUENCE表达式的一种，表示取名字为sequence_name的当前最新有效值的前一个值，类型为用户在定义SEQUENCE时指定的类型，不存在小数部分。

注意：OB中在DELETE语法结构中，where条件只可以使用PREVVAL这种形式，不能使用NEXTVAL这种形式，这个和DB2是保持完全一致的。OB目前实现了批量删除，sequence也是对应的进行了支持。

2 注意事项

2.1创建SEQUENCE

- OR REPLACE选项的功能是当要创建SEQUENCE的sequence-name已经存在时，重新创建此SEQUENCE，且所有客户端未输入的选项将均采取默认值。例如：名为test_sequence的序列的INCREMENT BY值为2，使用OR REPLACE重新创建该SEQUENCE时未输入INCREMENT BY，则重新创建的test_sequence的Increment By值将会变为默认值1。
- SEQUENCE的起始值，即START WITH的值，可以超出MINVALUE与MAXVALUE所限定的范围。例如：名为test_sequence的序列的最小值为1，最大值为5，步长（INCREMENT BY）为1，而序列的起始值可以为-5。该SEQUENCE的值从-5开始生成，直到最大值5。

- CYCLE选项用于设定SEQUENCE在到达边界值之后是否重新开始生成值。例如：名为test_sequence的递增序列，即步长为正数，到达最大值后，若选择了CYCLE，则从最小值开始循环；若选择了NO CYCLE则停止生成值，此时该SEQUENCE无法被使用。

注：CYCLE选项使SEQUENCE从最小值（而非起始值）开始重新生成。

- 当用户手动设定MINVALUE时，需要注意最小值必须**小于等于**最大值。
- 当用户手动设定MAXVALUE时，需要注意最大值必须**大于等于**最小值。
- CACHE的数量有限制，不能超过2147483647。（此为DB2的既存制限）
- 对于同一个用户，不同session中调用相同的SEQUENCE时共享cache；对于不同的用户，在得到授权之后使用某个SEQUENCE时共享cache。（**不支持**）

2.2 修改Sequence

- ALTER SEQUENCE相当于修改SEQUENCE的参数信息，与OR REPLACE不同，在修改SEQUENCE时未输入的参数不会改变。
- 在ALTER SEQUENCE中使用参数RESTART WITH numeric-constant使SEQUENCE从numeric-constant开始重新生成值，但创建该SEQUENCE时定义的START WITH值不变。
- ALTER SEQUENCE语句无法修改SEQUENCE的数据类型。
- 修改SEQUENCE的范围时，即最大值与最小值时，注意不能超出数据类型的表示范围。

2.3 调用SEQUENCE

- OB中使用INSERT语句插入多行数据时，使用NEXTVAL或PREVVAL的列的**格式和内容**可以任意的组合。
- OB中SEQUENCE的作用域是所有session，即所有session共享sequence信息，对于prevval值，一旦合法使用过nextval值，则prevval将对所有session有效使用（DB2中prevval为session内有效，当前session的prevval其他session看不见），nextval与DB2保持一致。
- 对于新建的sequence：对于当行数据中的sequence，在未指定**QUICK**有效的情况下，**如果先使用prevval，再使用nextval**，则将失败，例如：INSERT INTO sequence_test VALUES(PREVVAL FOR SEQ, NEXTVAL FOR SEQ, 222);则将视为非法使用，同时sequence的信息将保持不变，必须在合法使用NEXTVAL 之后才可以成功；**如果先使用nextval，再使用prevval**，则会给用户反馈使用不合法提示，但是再次执行该语句将会成功，OB遵循优先消费nextval的原则，一旦优先使用nextval，则该sequence的信息将会被更新，例如：INSERT INTO sequence_test VALUES(NEXTVAL FOR SEQ, PREVVAL FOR SEQ,222), (NEXTVAL FOR SEQ ,PREVVAL FOR SEQ ,333);执行将是失败的，但是nextval的值将被更新一次（虽然插入值有多行），因为此时第一行的prevval值还没有生成。原因是nextval的值是行有效的，并且优先使用的，一旦某行执行失败，则将当前的sequence信息写回系统表，供所有session同时再次竞争使用，防止长时间独占，这点请特别注意。
- 使用INSERT语句插入多行数据时，同一行中使用NEXTVAL多次调用SEQUENCE，这些NEXTVAL的值均相同。简单说就是NEXTVAL值是行共享的。
- 使用INSERT语句插入多行数据时，PREVVAL的值保持不变，NEXTVAL的值随行数变化。简单说PREVVAL值是语句共享的。例如：

```
INSERT INTO table_sequence_1(id, num) VALUES (NEXTVAL FOR test_seq, PREVVAL FOR test_seq),
(NEXTVAL FOR test_seq, PREVVAL FOR test_seq), (NEXTVAL FOR test_seq, PREVVAL FOR test_seq)
```

此时，id列的值为14,15,16，而num列的值则为13,13,13。

- 使用SEQUENCE时可能会出现在两个表中使用相同的数值作为唯一主键的情况，例如：

```
INSERT INTO table_sequence_1(id, age) VALUES (NEXTVAL FOR test_seq, 22);

INSERT INTO table_sequence_2 (id, income) VALUES (PREVVAL FOR test_seq, 8600);
```

此时两个表中的id列的值相等。

- 当SEQUENCE要生成的值超过了SEQUENCE的最大值与最小值确定的范围，并且指定该SEQUENCE不循环，此时系统会提示出错。此时需要用户使用ALTER语句对SEQUENCE的范围进行修改，或者使用DROP语句删除SEQUENCE并重新创建一个SEQUENCE，并设定范围足够大的数据类型。

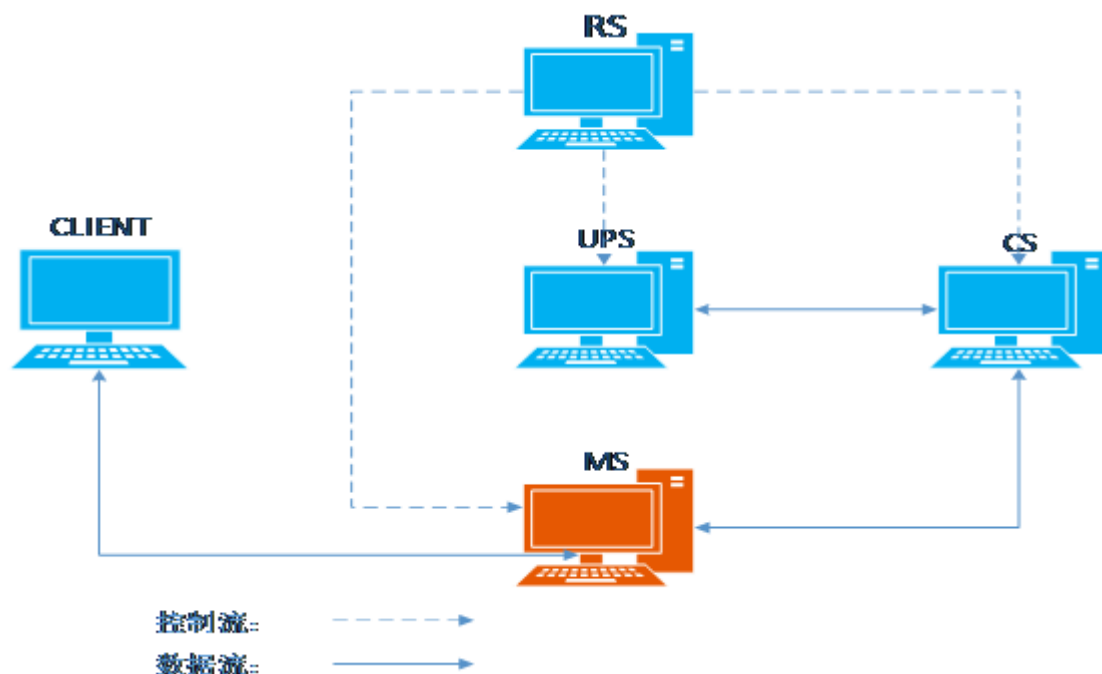
使用SEQUENCE时，对于一个新建的SEQUENCE,在其没有使用prevval时，需要使用过一次nextval只才可以使用prevval，OB中，如果在一行插入值中先使用了新建的prevval，而后使用nextval，则会报用户使用不合法错误，但是nextval值会更新，所以当用户下次再调用相同的语句就可以成功执行。

批量(插入，更新，删除)

一、总体设计

1.1 CBASE insert批量插入整体设计

OB的整体架构、数据流及控制流如下图所示;



备注：上图中的红色图形表示要对应的模块。下文中的红框均表示要对应模块或功能。

OB划分为如下几个模块：

- 客户端：用户使用OB的方式和MySQL数据库完全相同，支持JDBC、C客户端访问。
- RootServer：管理集群中的所有服务器，子表数据分布以及副本管理。
- UpdateServer：存储OB系统的增量数据。
- ChunkServer：存储OB系统的基线数据。
- MergeServer：接收并解析用户的SQL请求，经过词法分析、语法分析、查询优化等一些列操作后转发给相应的ChunkServer。如果请求数据分布在多台ChunkServer上，MergeServer还需对多台ChunkServer返回的结

果进行合并。

根据OB各个模块的功能设计，并结合OB源码进行分析，得出：mergeserver对insert语句进行解析，将解析后的select子查询发送到CS执行，得到插入的数据，并根据select子查询的主键值构造过滤器，根据过滤器从CS取得select子查询出的各行是否存在的信息，然后将这些信息和执行插入的物理计划一起发送到updateserver进行插入操作。因此，insert语句相关的模块为UpdateServer、MergeServer和ChunkServer，当然还包括这三个模块间的序列化及反序列化。

为了最大限度的利用OB已实现功能代码，尽可能降低实现复杂度，本文给出了如下的insert批量插入的整体设计。

MergeServer端：

- 实现Insert 批量插入语句的词法语法解析。（OB已实现）
- 生成Insert 批量插入语句的逻辑计划。（OB已实现）
- 生成Insert 批量插入语句的物理计划。（OB已有select子查询的物理计划，缺少insert into ... select ...整体的物理计划的生成）
 1. 新增insert语句整体物理计划的对应处理，以实现insert主查询计划和select子查询计划的正确组合。
 2. 新增物理操作符ObBindValues，以实现1)主查询物理计划和子查询物理计划的无缝组合2)完成单批次插入。
- 执行insert子查询SQL语句的物理计划。

ChunkServer端：

- 执行insert...select...语句中的select子句，取得需要插入的静态数据。
- 根据取得的select子句的查询结果的各行构造filter，通过ObTableRpcScan操作符到CS获得这些行是否已经存在的信息。

UpdateServer端：

- 接受insert...select...语句的插入物理计划和从CS获得的信息，执行实际的插入操作。

MergeServer和ChunkServer间通信：

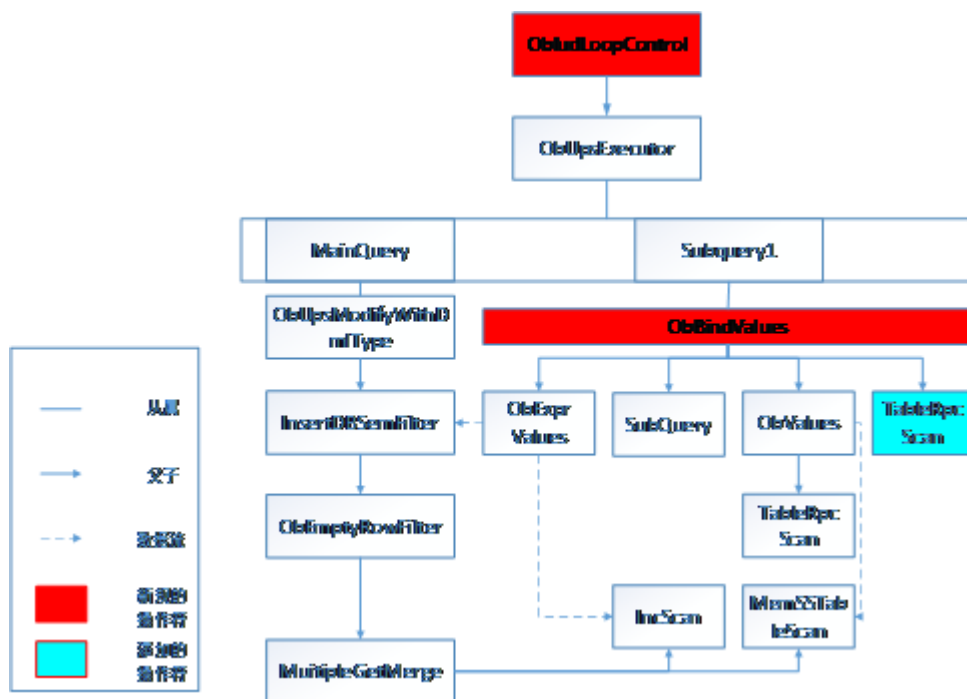
执行select子查询和获取插入的各行是否在CS中存在的信息时需要MS与CS之间的通信。

MergeServer和UpdateServer间通信：

Insert主查询的部分物理计划需要序列化发送到UpdateServer进行反序列化，然后执行。

1.2 OB insert批量插入整体物理计划设计

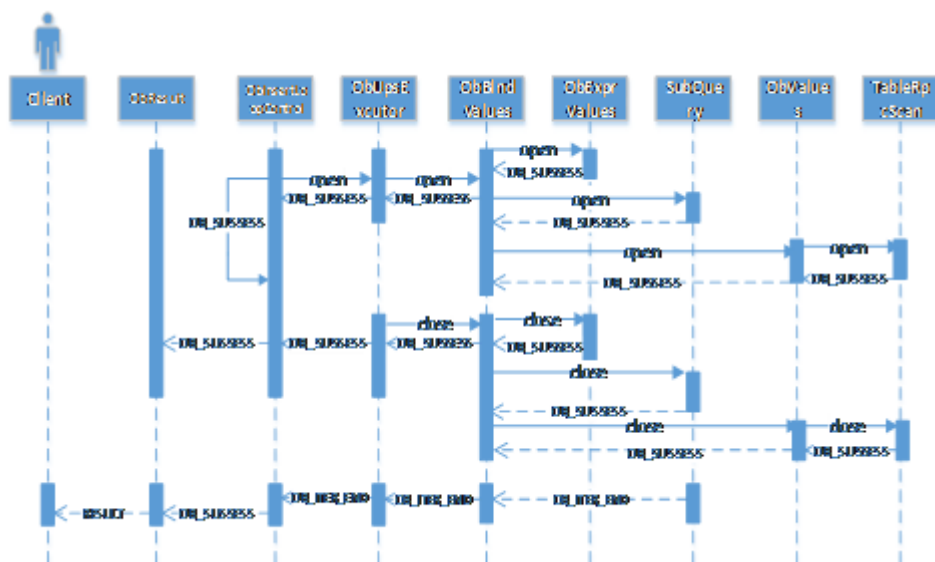
把OB自身的insert物理计划和select查询的物理计划正确地结合在一起，就可以实现insert批量查询。以 `insert into table_a(col1,col2,col3) Select col1,col2,col3 from table_b where col1<1000000` 句式进行说明。下图为insert批量插入物理计划。



为实现insert批量，新规两个操作符：ObludLoopControl和ObBindValues。ObludLoopControl作用为控制对批次的正确插入，通过多次调用ObUpsExecutor，完成多批次插入任务，其中每运行一次ObUpsExecutor，完成一批次的数据插入。ObBindValues作用为从SubQuery中读取一批需要插入的数据，并将读取的数据格式转换为主表的格式，然后填充到ObExprValues和ObValues操作符中。

1.3 OB insert批量插入物理计划执行设计

实现insert批量插入后的OB物理计划的执行规则如下图所示。



规则：

- OB首先打开ObResultSet操作符，由ObResultSet打开主查询ObludLoopControl，由ObludLoopControl打开ObUpsExecutor。ObludLoopControl控制整个分批插入的循环操作。
- ObUpsExecutor操作符打开新规的操作符ObBindValues，由ObBindValues依次打开其孩子，并在打开的过程中，将从SubQuery操作符中取得的select子查询的行转换为主表的数据格式，并填充到ObExprValues和ObValues操作符中。

- ObUpsExecutor操作符将物理计划和取得的静态数据发送到ups执行后，调用ObBindValues的close函数，由ObBindValues的close函数依次调用其孩子操作符的close函数。
- 当取完全部数据后，整个insert批量插入物理计划就执行完毕，将插入的行数返回给客户端。

二、批量更新/删除

2.1 既存delete/update语句功能描述

Delete

1. 句式：`DELETE FROM table_name [[as] alias_name] WHERE where_condition [when ROW_COUNT(statement) op expr]`
2. 功能：删除一条或零条数据。
3. 制限
 - 只支持指定全主键简单条件的单行删除
 - 不支持where条件中存在select子查询

Update

1. 句式：`UPDATE table_name [[as] alias_name] SET column_name1=column_value[,column_name2=column_value] ... WHERE where_condition [when ROW_COUNT(statement) op expr]`
2. 功能：更新一条或零条数据。
3. 制限
 - 不支持set语句中存在select子查询
 - 只支持指定全主键简单条件的单行删除
 - 不支持where条件中存在select子查询
 - 不支持set中带有聚集函数

2.2 既存delete/update存在场景描述(以delete语句为例)

1. 无显示事务
 - 单独存在
 - 存在于prepare语句。详细介绍请参见《Oceanbase 0.4.2 SQL 参考指南》
 - 存在于explain语句详细介绍请参见《Oceanbase 0.4.2 SQL 参考指南》
 - 存在于when语句详细介绍请参见《Oceanbase 0.4.2 SQL 参考指南》
2. 存在于事务中
 - 将“1.无显示事务”中涉及的环境放在start transaction...comit/rollback语句块中
3. 存在于并行环境中
 - 将1和2根据并行执行的相关性分布在不同的session中并行执行。

2.3 新规后delete和update语句功能描述

Delete

1. 句式：`DELETE [ob_hint] FROM table_name [[as] alias_name][WHERE where_condition] [when ROW_COUNT(statement) op expr]`
2. 功能：

- 若无hint语法，则会删除某个阈值以内数据，可书写任意where条件。
- 若书写hint语法：UD_MULTI_BATCH，理论上删除数据无上限，但需要根据机器性能进行超时时间的调整，可书写任意where条件。

3. 制限

- 无hint语法情况下，保证Ocenabase定义的事务，有hint语法，不保证Ocenabase定义的事务。
- 如果when条件后的statement是insert或replace，则不能删除新增的数据行
- 在大版本冻结时执行delete语句，会有问题，出问题的原因是在进行并行处理判断数据是否安全时，由于执行大版本冻结时，从CS读取的时间戳是有效的，但在UPS读取到的时间戳是无效的，导致不能正常更新。

Update

1. 句式: `UPDATE [ob_hint] table_name [[as] alias_name] SET column_name1 =column_value [column_name2=column_value] ... [WHERE where_condition][when ROW_COUNT(statement) op expr]`

2. 功能：

- 若无hint语法，则会更新某个阈值以内数据，可书写任意where条件。
- 若书写hint语法：UD_MULTI_BATCH，理论上更新数据无上限，但需要根据机器性能进行超时时间的调整，可书写任意where条件。
- 可更新主键列（此功能由上海侧新规）。

3. 制限

- 无hint语法情况下，保证Ocenabase定义的事务，有hint语法，不保证Ocenabase定义的事务。
- 如果when条件后的statement是insert或replace，则不能更新新增的数据行
- 不支持set语句中存在select子查询
- 不支持set中带有聚集函数

2.4 新规后delete/update存在场景描述

与“3.2.2 CBase既存delete/update存在场景描述(以delete语句为例)”相同。

2.5 新规后事务支持

目前CBase支持的隔离级别为RC，对于不加hint的批量，支持小于日志限制大小（目前是32M）的操作，保证事务，可以放在隐式或者显示的事务中。

对于加hint的批量，支持任何大小的操作，不保证整体语句的事务。如果加hint的批量放入显示或者隐式事务中，可以保证整体事务，但是数据量必须小于日志限制的大小。不放入事务中，不保证事务。

In子查询

一、需求概述

因交行业务对IN子句子查询功能的需求，需要对现有OB IN子句的功能进行改造和优化。

OB IN子句需新规功能：

- IN条件需支持单个子查询的结果集。

例如：`select ... from table_a where table_a.x in (select ...)`

- IN条件必须支持and或or的组合。

例如：`select ... from table_a where table_a.x in (select ...) and table_a.x in (select ...)`，即多个and的in并列。

`select ... from table_a where table_a.x in (select ...) or table_a.x in (select ...)`，即多个or的in并列。

and、or、in三者的组合也必须支持。

- IN条件支持的子查询可以是嵌套的子查询。

例如：`select ... from table_a where table_a.x in (select x from table_b where table_b.x in (select ...))`

- and和or的并列的in条件，可以同时是嵌套子查询。

例如：`select ... from table_a where table_a.x in (select x from table_b where table_b.x in (select ...)) and table_a.x in (select x from table_b where table_b.x in (select ...))`

`select ... from table_a where table_a.x in (select x from table_b where table_b.x in (select ...)) or table_a.x in (select x from table_b where table_b.x in (select ...))`

- IN字句必须支持子查询结果集是大数据的情形。
- OB IN字句新规子查询性能要求：
- //待添加，需综合其它数据库设置信息和交行的需求，从而进行权衡。

二、 制限

OB既存制限：

- OB非主键列没有索引，当IN绑定列为非主键时，全表扫描效率低。
- OB 单次查询超时上限为3秒。
- OB in操作符的操作数上限为510组（不大于）。

例如：`(col1,col2) in ((1,2),(3,4))`：(1,2)和(3,4)都是单独的一组。

`(col1) in (1,2)`：1和2也是单独的一组。

- OB内部通信单个数据包大小上限2M。OB绑定过滤条件的物理计划总体积不能超过上限。

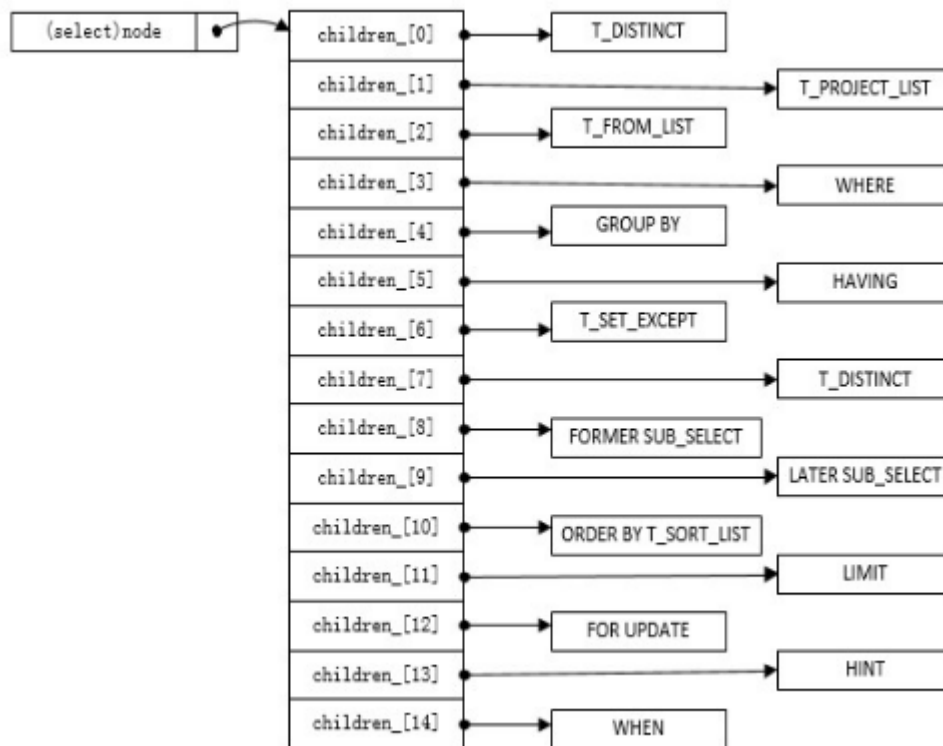
OB新规IN子查询制限：

- IN子查询的IN操作符的左表达式只能是同一张表的列名组合。
- IN子查询的IN操作符的右表达式只能是单一的一个查询语句，查询语句可嵌套。这条与mysql保持一致。
允许 `IN (select ...)` 类型语句。 禁止 `IN (select ..., ConstValue, ConstValue...)` 类型语句。 禁止 `IN (select ..., select ..., ...)` 类型语句。
- 支持IN子句子查询后，单个ObSqlExpression表达式内子查询数目不大于5（暂定，后期可商讨确定，但必须固定在程序里）。

三、 总体设计

3.1 OB既存语法树IN子树

OB的select语句经过词法语法解析后，生成的语法解析树如图表所示。

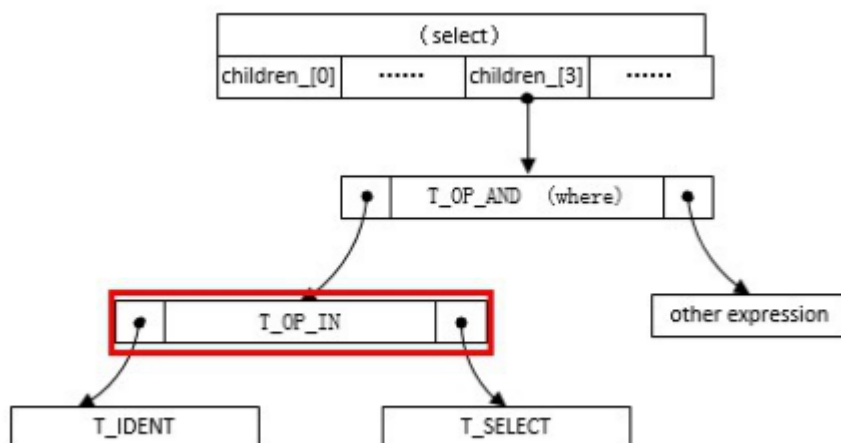


从上述图表可以看出，select语句的每一个语法成分都和语法树的一个固定的节点相对应。在select语句中，IN关键字位于where条件内。因此，IN操作的相关信息被唯一保存在以select语法树的第四个孩子（children_[3]）为根节点的子树（简称IN子树）内。

备注：select语法树的第一层孩子节点数目是固定的15个。如果select语句缺少相关的字段信息，则与字段信息相应的节点为空。

Where后跟的过滤条件可以有多种选择类型，本文只关注IN过滤条件。IN过滤条件可以和and、or操作符组合，亦可产生复杂过滤条件。为了简单明了介绍IN子树，本文以“select * from table_x Where table_x.y In (select ...) and ...”类型的SQL查询语句为例，并给出IN子树的结构图。

备注：OB词法语法部分已经正确识别IN过滤条件为子查询的SQL查询语句。



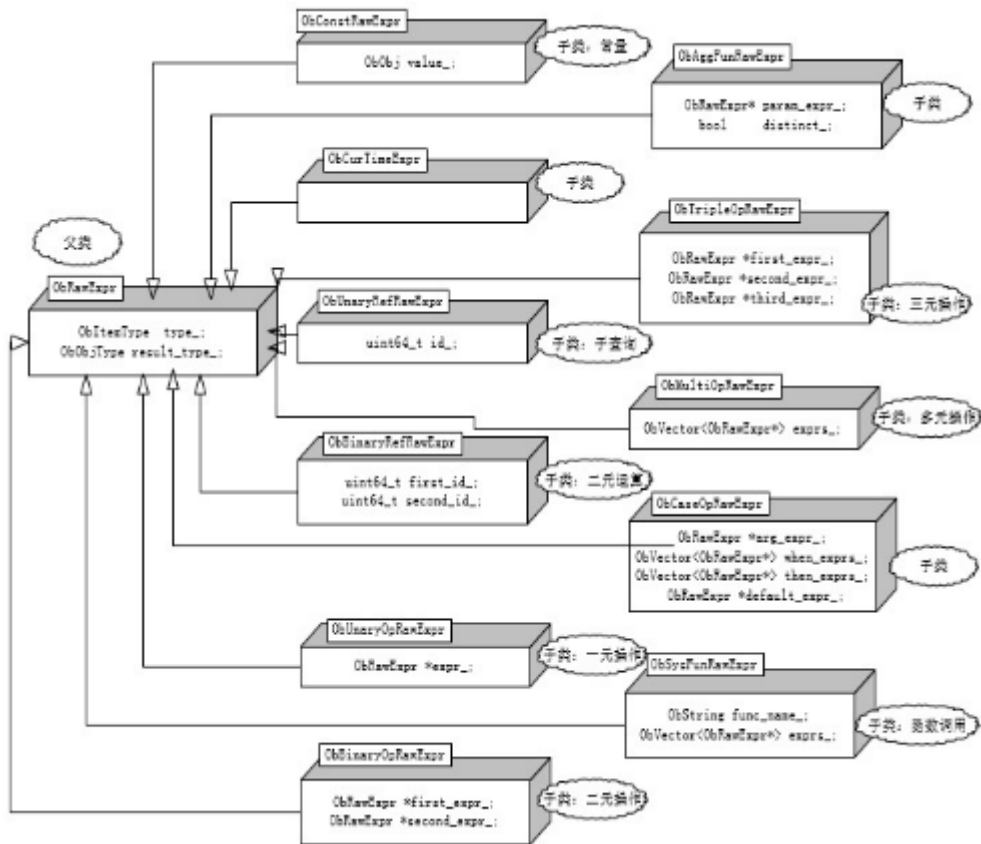
上图中，T_OP_IN对应IN操作符。因样例SQL的IN左表达式是table_x.y且唯一，OB内部用type：T_IDENT的节点来标识列名。样例SQL的IN右表达式是一个select查询语句，因此T_OP_IN的右孩子挂载的是一个以type：T_SELECT的节点为根节点的完整的语法树（OB词法语法解析程序同样将子查询语句按共同的select词法语法进行解析，只是子查询的语法树的根节点又被挂载在对应包含子查询的操作符的节点上）。

OB既存的词法语法解析程序已经实现识别IN过滤条件是子查询的功能。因此OB词法语法部分不需要对应。

3.2 OB既存逻辑计划IN相关部分

3.3.1小节指出OB既存的词法语法解析程序已实现识别IN子句子查询的功能。按OB的处理流程，根据select语法树生成逻辑计划紧随词法语法解析之后。

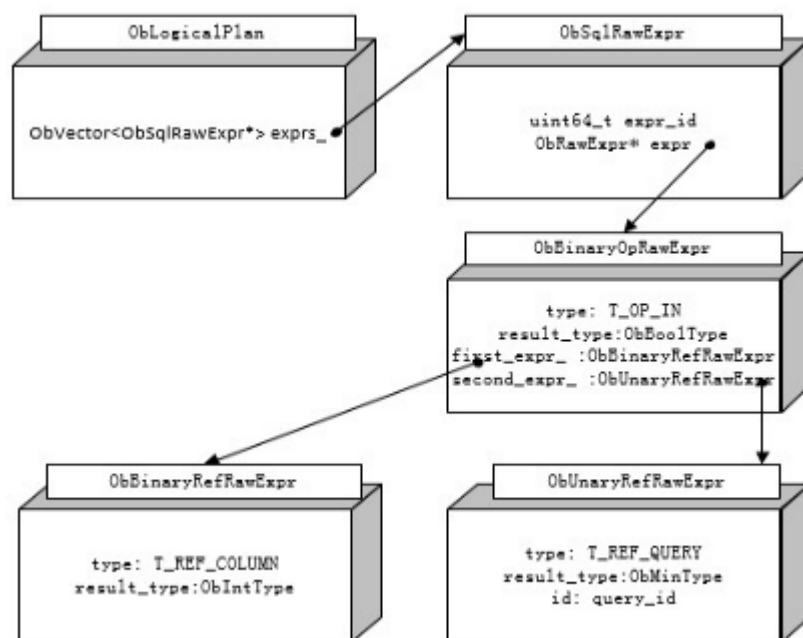
OB逻辑计划处理部分将SQL语句中非关键字按照自身的设计准则，将语法树的节点填充到一个个表达式内。OB表达式的所有相关类及类之间的关系如下图表所示。OB逻辑计划中的表达式的基类是ObRawExpr。OB真正的表达式值存储在基类ObRawExpr的各个对应子类内部。



根据本次开发的目的，本文只关注逻辑计划IN相关部分的处理。为了简单明了介绍逻辑计划IN部分，本文同样以3.3.1小节的 select * from table_x Where table_x.y In (select ...) and ... 类型的SQL查询语句 为例，给出了该语句的逻辑计划IN相关部分。

下图展示了OB内部该SQL查询语句的IN相关表达式、表达式之间的链接组织方式及IN相关表达式如何挂载到SQL逻辑计划。

OB既存的逻辑计划处理部分已经完全实现IN过滤条件是子查询的SQL查询语句的逻辑计划生成的功能。因此OB逻辑计划部分不需要对应。



3.3 OB既存物理计划IN相关部分

OB既存物理计划处理部分只对应了IN绑定具体数目确定值的情况。对于IN绑定子查询的情况未进行进一步处理，只是简单返回添加filter失败。

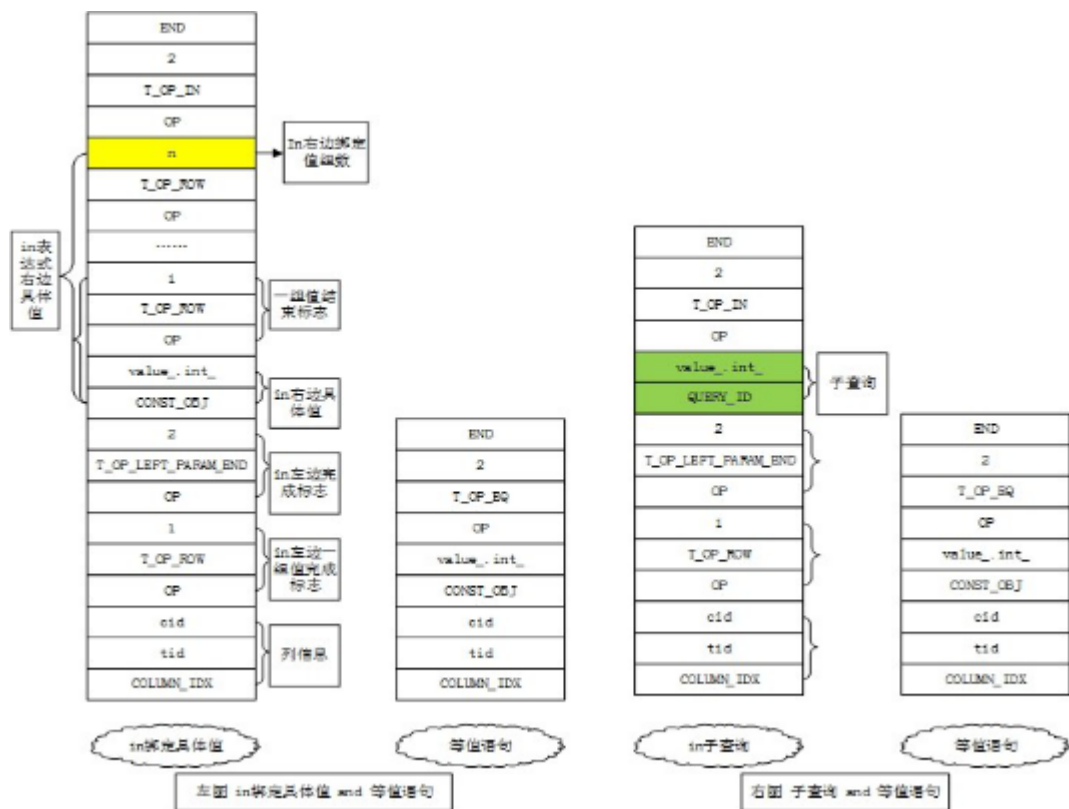
OB既存物理计划处理部分会根据逻辑计划中IN相关表达式的所属类，分别调用所属类的成员函数 fill_sql_expression，将每个表达式转换成ExprItem，然后再依次将ExprItem保存进ObSqlExpression类的成员 (ObPostfixExpression) post_expr 的成员 (ExprArray) expr 内部。最终会将这个ObSqlExpression添加进ObFilter内部。每个filter对应一个ObSqlExpression。OB内部的一个ObFilter操作符包含一个或多个filter。

OB既存的where条件转换成ObSqlExpression时，有以下规则：

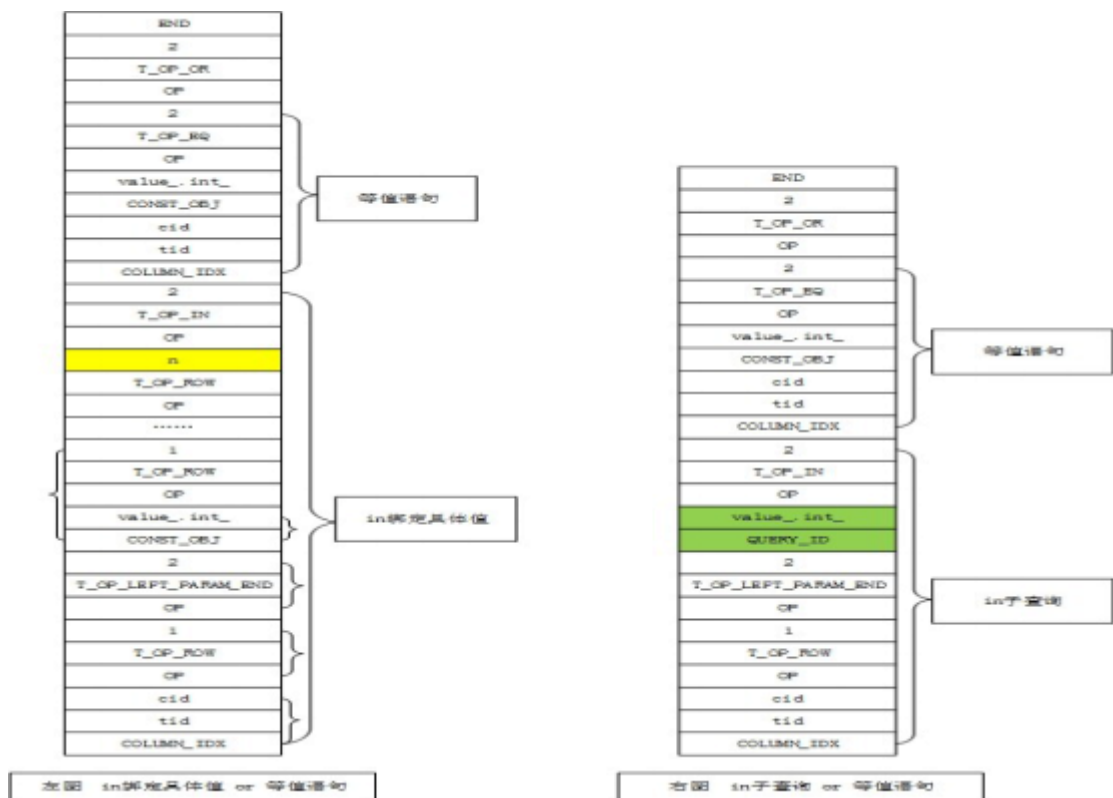
- 如果where后的过滤条件之间是并列“与”的关系，即多个过滤条件只采用and连接，则每个过滤条件最终被转换成单个ObSqlExpression表达式。
- 如果where后的过滤条件不完全是并列“与”，此时又有三种子情况如下：
 1. 过滤条件 [and 过滤条件] or 过滤条件 [and 过滤条件]，这种形式的where后的多个过滤条件被转换成唯一的ObSqlExpression表达式。
 2. 过滤条件 [and 过滤条件] ([and 过滤条件] or [and 过滤条件]) 过滤条件 [and 过滤条件]，这种形式下，括号内的表达式被转换成一个ObSqlExpression表达式，括号表达式和剩余的表达式完全是并列“与”，则每个剩余表达式被转换成一个ObSqlExpression表达式。
 3. 如果②中的剩余表达式和括号表达式不是完全“与”，即包含至少一个or，那么where后的多个过滤条件最终被转换成单个ObSqlExpression表达式。

为了说明上述OB的where条件与ObSqlExpression规则，本文给出以下两条where条件的ObSqlExpression表示式。

- Where table_x.id in (constvalue,...) and table_x.id1 = constvalue。该where条件的ObSqlExpression见下图表的左图。



- Where table_x.id in (constvalue,...) or table_x.id1 = constvalue。该where条件的ObSqlExpression见下图的左图。



OB既存的实现已经将IN子查询语句的物理计划生成，只是在把IN子查询物理计划挂载到主计划时，由于缺少对应处理，物理计划最终生成失败。

3.4 OB新规物理计划IN相关部分

3.3.3小节讲解了OB既存的IN实现，同时指出了IN子查询物理计划已经生成，只是缺少如何将IN子查询物理计划和filter关联的处理，最终造成物理计划生成失败。

为了生成最终的包含IN子查询的SQL查询语句的物理计划，本文新规了“IN子查询物理计划和ObFilter中的过滤条件ObSqlExpression关联的处理操作”。

在处理IN表达式的部分，新增IN子查询表达式处理。处理如下：

因为多个子查询过滤条件可能被转换成一个ObSqlExpression表达式，为了以后使用,新增加了以下函数，用以存取单个表达式内的子查询数目。

```
//位于：class ObSqlExpression内
Public：
    void add_sub_query_num(){sub_query_num_++;};
    int32_t get_sub_query_num(){return sub_query_num_};;
```

为了将IN子查询关联到过滤条件ObSqlExpression内，新规了以下函数，用以将子查询标识写入过滤条件ObSqlExpression内，实现关联。

```
//位于：int ObPostfixExpression::add_expr_item(const ExprItem &item)函数内
case T_REF_QUERY:
    item_type.set_int(QUERY_ID);//QUERY_ID是一个标识，类型为枚举类型值
    obj.set_int(item.value_.int_);
    if(OB_SUCCESS != (ret = expr_.push_back(item_type)))
    {}
    else if (OB_SUCCESS != (ret = expr_.push_back(obj)))
    {}
```

QUERY_ID位于(enum)ObPostExprNodeType内，标记表达式类型为需后期用子查询结果集动态绑定类型。

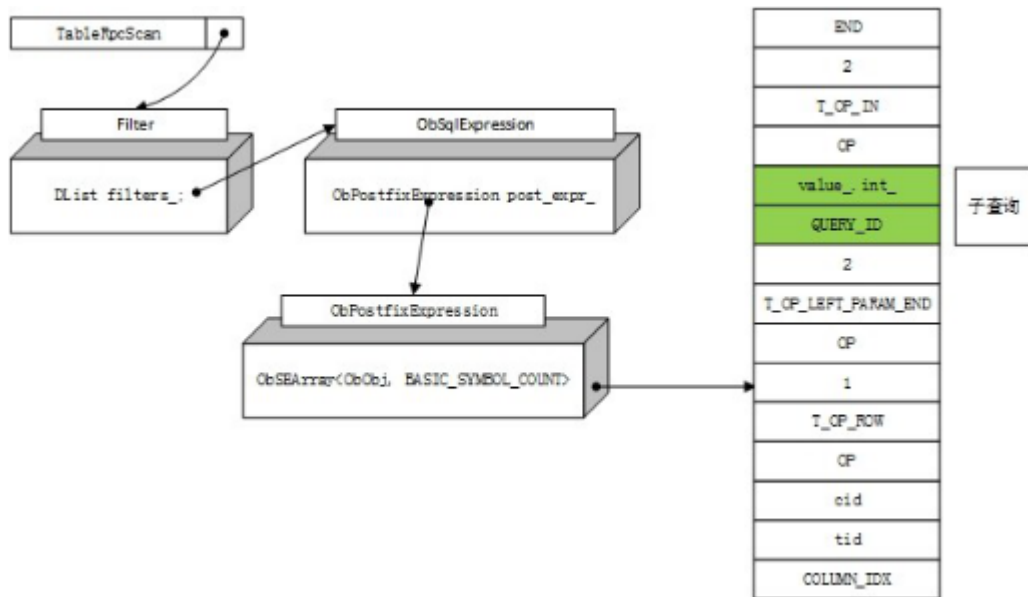
OB内部新规了IN子查询号和具体子查询计划关联的函数。

```
//位于：class ObSqlExpression内
void set_sub_query_idx(int32_t idx,int32_t index){sub_query_idx_[idx-1]= index};;
```

本文调用此接口函数，顺序保存了表达式内多个子计划的索引，实现如下。

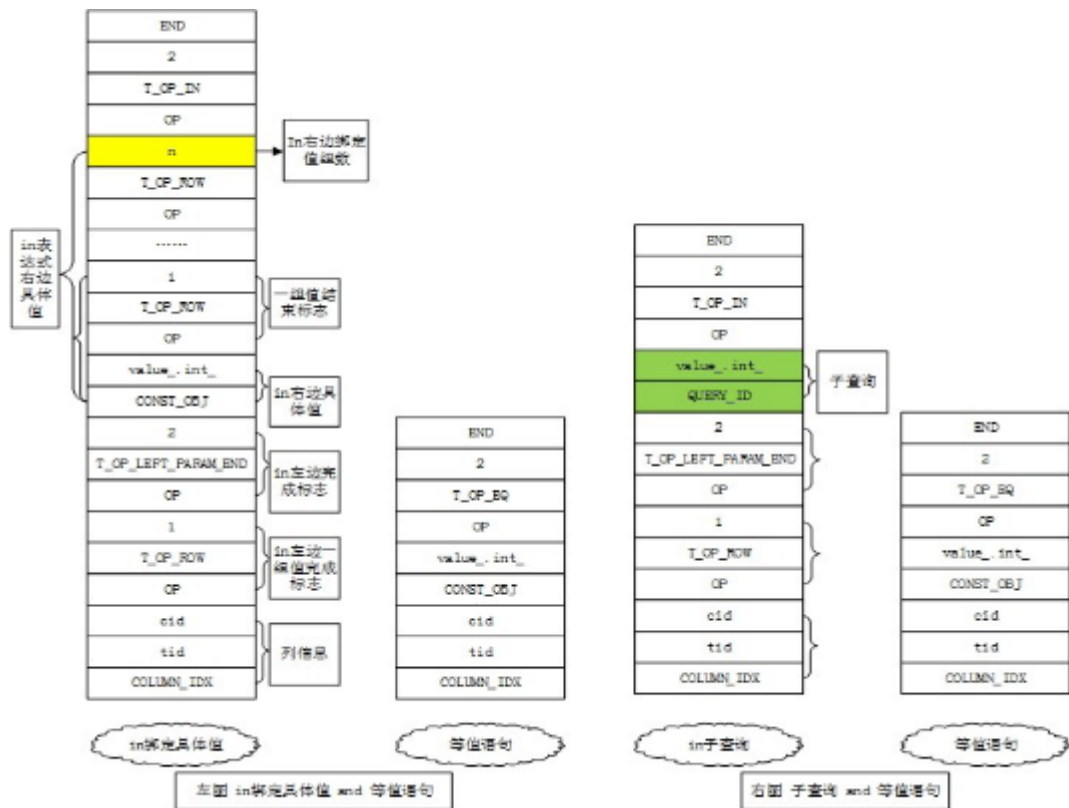
```
//位于：int ObUnaryRefRawExpr::fill_sql_expression(
//    ObSqlExpression& inter_expr,
//    ObTransformer *transformer,
//    ObLogicalPlan *logical_plan,
//    ObPhysicalPlan *physical_plan) const 函数内
inter_expr.add_sub_query_num();
inter_expr.set_sub_query_idx(inter_expr.get_sub_query_num(),index);
ObSelectStmt *sub_query=
dynamic_cast<ObSelectStmt*>(logical_plan->get_query(id_));
item.value_.int_ = sub_query->get_select_item_size();
```

添加上述代码后，以 table_x.x in (simple select 语句)生成的filter过滤条件如图表:

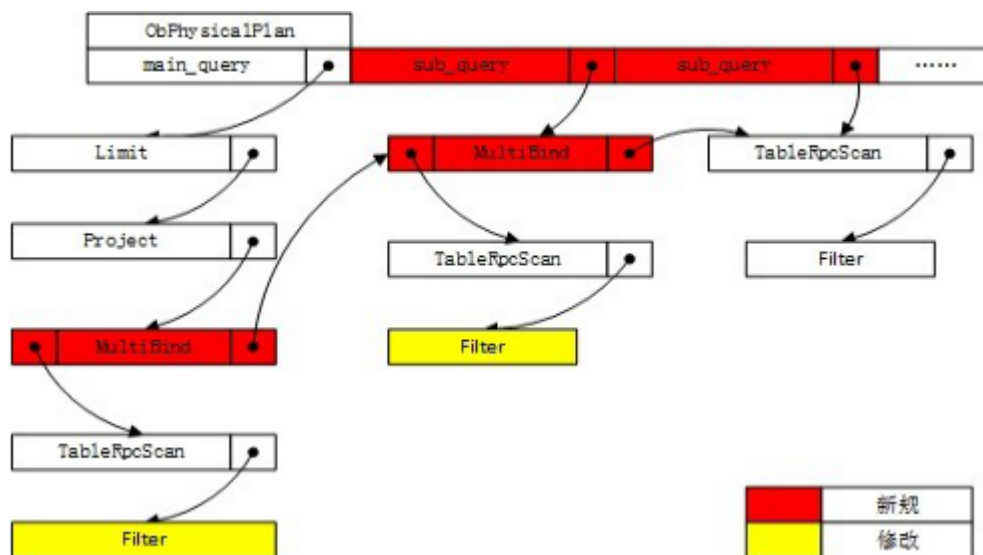


为了展现出修改后的filter表达式，本文给出以下两条where条件的ObSqlExpression表示式。

- Where table_x.id in (简单子查询) and table_x.id1 = constvalue。该where条件的ObSqlExpression见图表的右图。



- Where table_x.id in (简单子查询) or table_x.id1 = constvalue。该where条件的ObSqlExpression见图表的右图。



其它嵌套、并列的子查询，虽然和以上两种略有不同，但是是以上两种的组合或者简化，因此不再详细介绍。

3.6 IN子句新规物理操作符

OB既存的设计缺少将子查询计划挂载到主计划的处理。由于OB的物理计划是一串链接的物理操作符，缺少指向子查询计划的指针，整个物理计划就不能正常执行。因此为将IN子句子查询物理计划挂载到主计划和主计划动态绑定子查询结果集的功能，OB新规了1个物理操作符ObMultiBind。因为主计划可以有多个子查询计划，所以该操作符继承操作符ObMultiChildrenPhyOperator，以支持挂载多子查询物理操作。

该类的实现和OB已实现的操作符类似，重要成员如下：

```
class ObMultiBind: public ObMultiChildrenPhyOperator
{
    /* *sub_result用来存储多个子查询的结果集 */
    common::ObRowStore *sub_result_;
    /* *sub_query_map_ 用来存储多个子查询结果集的hashmap ,
        ObRowkey由子查询的相关列组成*/
    common::hash::ObHashMap<
        common::ObRowkey, common::ObRowkey,
        common::hash::NoPthreadDefendMode> *sub_query_map_;
};
```

ObMultiBind类中的子查询结果集的动态绑定策略如下：

前提：

- OB in操作符的操作数最多只能绑定510组值。
- OB只允许in一个简单select语句或一个嵌套的select语句。
- OB内部通信包单个上限2MB。

假设i=1，where后有k个子查询

策略：

1. ObMultiBind调用成员函数打开第i个 IN子查询物理计划，初始化num=0，设置结果默认保存标记 SFLAG=TRUE；
2. 如果i大于k，则转 1. 执行。

3. 读取一条子查询结果，并根据当前结果构建Hashmap、Bloomfilter。如果SFLAG==TRUE 则保存结果，反之，则不保存。随后num++；
4. 判断num是否大于510，按以下四种情况处理：
 - 如果num不大于510且当前子查询执行完毕，则释放当前子查询的Hashmap和Bloomfilter，随后i++，转 1.执行；
 - 如果num不大于510且当前子查询没有结束，则转 3. 执行；
 - 如果num大于510且子查询没有结束，则释放掉当前子查询的保存结果集，将SFLAG=FALSE，随后i++，转 3. 执行；
 - 如果num大于510且子查询执行完毕，则释放掉当前子查询的保存结果集，将SFLAG=FALSE，随后i++，转 1. 执行；
5. 循环遍历k个子查询，语句保存的是结果集还是Bloomfilter和Hashmap，动态的绑定到主计划的filter内，即替换物理计划中的有关子查询的填充标识处。
6. 如果绑定filter后的tableRpc包大小超过2M，则将filter相关全置NULL，即扫描表所有数据到ms；如果包不大于2M，则绑定成功，发送执行行。

新规ObMultiBind物理运算符内新增了两个重要的数据结构，Bloomfilter和Hashmap。

3.7 IN子句新规Bloomfilter

新规的Bloomfilter位于ObSqlExpression的成员(ObPostfixExpression)

post_expr内。因声明“单个表达式内的子查询不大于5”，因此在post_expr内新建一个数组：
common::ObBloomFilterV1 bloom_filter[MAX_SUB_NUM] (MAX_SUB_NUM=5)。

ObPostfixExpression内增加对Bloomfilter的序列化和反序列化处理。

OB既存的IN操作符的操作数上限为510组确定值，但是交行的实际需求要求OB支持IN绑定子查询结果集是大数据（数万条，乃至百万条）的情形。在OB既存架构下，如果采用循环绑定，代价不可忍受。

以 `select count(*) from table where table.x in (select ...) and table.y in (select ...)` 为例。假设每个子查询结果集包含100000条数据。采用循环绑定，每次绑定500条，主查询共计要执行 $(100000/500)* (100000/500)=40000$ 次。再假定索引存在，即采用USE_GET读取。OB采用USE_GET时，启用20个线程同时处理。不考虑比对等额外开销，最终可得到单次执行耗时 $1.5ms(3/(40000/20))$ 左右（OB默认超时上限为3秒）。实际OB执行IN单次绑定500确定值的SQL的耗时至少比 $1.5ms$ 高1个数量级，而且循环绑定还牵涉到子查询结果集/主查询结果集的保存等一系列复杂难题。总之，循环绑定没有实现的价值。

因此，IN子句子查询引入了大数据集合判定利器Bloomfilter。假定误报率为0.1，位利用率50%，利用Bloomfilter的相关计算公式，可得100000条记录需要3.2个hash函数及460000bit（57.5KB）。每一组待绑定值只需4.6bit即可。两个IN条件同时绑定也只需115KB（远远小于OB单个数据包2MB的上限，libeasy架构制限）。一次绑定规避了主查询结果集的保存等难题。OB，可以借鉴参考。总之Bloomfilter可以较好的支持IN字句子查询的实现。

备注：Bloomfilter相关公式。 p ：误判率， m ：位数组大小， n ：总数据数目， k ：所需哈希函数数目。

IN子句子查询引入的Bloomfilter的功能：CS使用Bloomfilter初步过滤数据。因为Bloomfilter的误报（不漏报），保证CS取得的数据集是符合IN子句条件的集合的超集。

综合上述，OB初步设置的Bloomfilter包含3个相互独立的hash函数，固定大小（4600000）bit数组，即在误判率0.1左右时，最大仅支持子查询结果集大小不大于1000000条。注意此时单个Bloomfilter为575KB，在单个请求包2MB的限制下，最多只能有并列的3个子查询。（但是OB物理计划只能包括有5个子查询，如果2MB全部用来保存子查询的Bloomfilter。在现有的Blommfilter下，即误判率0.1，3个hash函数，单个子查询最多支持绑定数据集上限71万左右，实际中数据包中还有一些其它的信息，因此实际中会小于71万的上限）。但是本文OB新规做了处理，一旦大于上限，Bloomfilter失效，不再传送至其它Server过滤数据，当前的MS直接销毁。

新规的Bloomfilter所选用的3个哈希函数如下：

- murmurhash64A。
- FNV-1A。
- BKDR。

以上3个哈希函数，murmurhash64A和FNV-1A是OB既存的实现。OB选用的这两个哈希函数都具有快速、低冲突率的特点。因此新规的Bloomfilter继续使用。根据资料显示：BKDRHash无论是在实际效果还是编码实现中，效果都是最突出的。因此Bloomfilter的第三个哈希函数选用BKDRHash。

备注：hash函数的调查详见文档《Bloomfilter_Hash函数调研.txt》。

新规的Bloomfilter的构建实现方案：

前提：

复用OB中已实现的Bloomfilter功能。OB默认误判率0.1，OB实现的Bloomfilter只需传入总数据集数目即可。新规的Bloomfilter默认子查询结果集数据不大于1000000个。

方案：

- ① 使用OB的Bloomfilter接口创建Bloomfilter1，接口传入总数据集数目1000000；
- ② 每一条记录都会被转换成Obj数组；
- ③ 分别使用选中的三个哈希函数计算②中的Obj数组，得到三个哈希值；
- ④ 三个哈希值分别对Bloomfilter1的总bit位数取余，依次为L、M、N，然后分别将Bloomfilter1内的位数组的下标为L、M、N位置为1
- ⑤ 循环②③④这三步，不断将记录添加进Bloomfilter1。

新规的Bloomfilter重新向外提供一个接口函数，目的是判断传入的参数是否包含在内。接口函数的参数个数唯一，参数就是一条原始记录转换后的Obj数组。

3.8 IN字句新规HashMap

IN字句引入Bloomfilter后，MS从各个CS收到的是符合主查询的最终集合的超集。因此MS必须进行一次的严格的过滤以取得最终查询结果集。

OB既存的设计有两种Hashmap，如下所示

第一种：common/hash/ob_hashmap.h文件内声明的class ObHashMap。这个ObHashMap的桶数由外部传入，每个桶内的数据个数不固定，桶内的数据以链表的形式组织。

第二种：updateserver/ob_lighty_hash.h文件内声明的class LightyHashMap。这个LightyHashMap的桶数固定，每个桶内的数据个数也固定（不大于65535）。

本文新规的HashMap是第一种ObHashMap的实例对象。由于OB没有预估程序衡量子查询结果集数目，因此默认设置桶的数目为10000个。在最优情形下1000000万条记录均匀分布到10000个桶内，每个桶100个节点，查找一个节点最低时耗 $O(2)$ [一次哈希计算，一次比对]，最大时耗 $O(101)$ [一次计算，100次比对]。

本文新规的HashMap的构建实现方案：

前提：

复用OB中已实现的ObHashMap功能。默认ObHashMap的桶为10000个，桶内数据个数没有限制。

方案：

- ① 使用OB的ObHashMap接口创建ObHashMap1，接口传入桶数目10000；
- ② 每一条记录都会被转换成Obj数组；
- ③ 使用选中的单个哈希函数计算②中的Obj数组，得到单个哈希值；
- ④ 哈希值对ObHashMap1的总桶数取余，假设为L，然后将Obj数组作为value值保存进第L-1个桶；
- ⑤ 循环②③④这三步，不断将记录添加进ObHashMap1。

新规的HashMap利用ObHashMap提供一个接口函数，目的是判断传入的参数是否包含在内。参数就是一条原始记录转换后的Obj数组。

备注：总体处理不变，后续可能修改记录的转换类型，当前暂定Obj数组。

引入hashmap的优势（大数据情形下）：

- 支持超大子查询结果集，打破OB单个操作符的操作数510制限。
- 规避了多次绑定结果集的操作，节省时间，降低了实现复杂度。

引入hashmap的劣势（大数据情形下）：

- 耗费内存

主键更新

更新主键功能指通过update语句更新主键列的值，其中主键例不限于单一主键和联合主键。在本文档中：

- （1）旧行：即原行值，特指更新主值前的行；
- （2）新行：特指更新主键值后的行；
- （3）单行更新：特指非批量更新。

1 DB2中更新主键功能的介绍

DB2支持完整的更新主键功能，可同时更新多列多行的主键值，更新后的新主键值不允许与现有的主键重复，且不允许一条update语句更新多行的主键为相同的值。

2 OB中支持更新主键概述

基本思想：分为两大步骤1）MS生成物理计划时，将新行（如果存在）和旧行所有列的基线数据一起发送至UPS；2）UPS执行物理计划时，融合基线与增量数据，判断新行是否已存在，若已存在，说明主键重复，更新操作失败；否则，根据新主键的赋值表达式和旧行数据，计算出新主键值，和旧行的其它列组合成新行，插入至Memtable，并删除旧行。

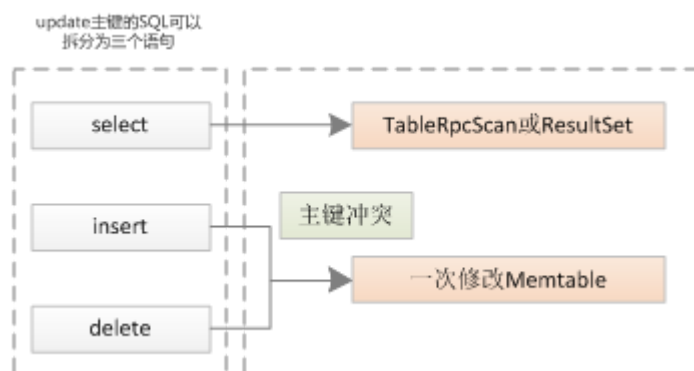


图1-1 更新主键的原理

基于DB2对更新主键功能的支持情况和OB当前版本特点，设计并实现OB中更新主键的基本功能有：

- (1) 更新主键值与非主键值方法完全相同，即原来支持的update语句在没有主键冲突的情况下均可更新主键值；
- (2) 当更新涉及多行时，若多行更新中存在多行修改为相同的主键，或任意行新主键值已存在，该批量更新不再执行，返回错误"Duplicate entry"；
- (3) 在高并发的情况下，若在物理计划发送到UPS的过程中原行数据被修改，则该次更新操作被取消；
- (4) 支持批量更新、二级索引和Sequence功能等。

```
UPDATE table_name SET idcolumn1=columnvalues [idcolumn2= columnvalues] ... WHERE condition;
```

其中：column_values 可为[纯数值]或[列名+纯数值]。

本文对原OB系统的修改只涉及update操作的物理执行计划在MS的生成和在UPS上的执行。

3 Update的物理执行计划

由于update操作涉及二级索引和批量更新所生成的物理执行计划不同，因此物理执行计划有多种组合的形式。新增更新语义过滤操作符（ObUpdateDBSemFilter）用于在UPS端检查主键冲突，在多种物理执行计划的组合中，需要保证ObUpdateDBSemFilter操作符是MultipleGetMerge操作符的直接父亲节点，确保基线数据和增量数据没有经过任何过滤（Filter）操作而不完整。在“3.3 批量更新的物理计划”的小节中重点说明各操作符的改造情况 and 作用。

判断是否更新主键

在生成物理计划前，判断从逻辑计划获取的赋值表达式列表（update语句中的set子句）是否包含主键列，若包含，则说明该update操作正在使用更新主键值的功能，只有此次判断为真，才会进入生成支持更新主键功能的物理计划的代码逻辑。

单行更新的物理计划

单行更新不含二级索引的物理计划

单行更新不含二级索引的表对应的物理执行计划如下图所示，其中ObFilter为非必定存在的操作符。

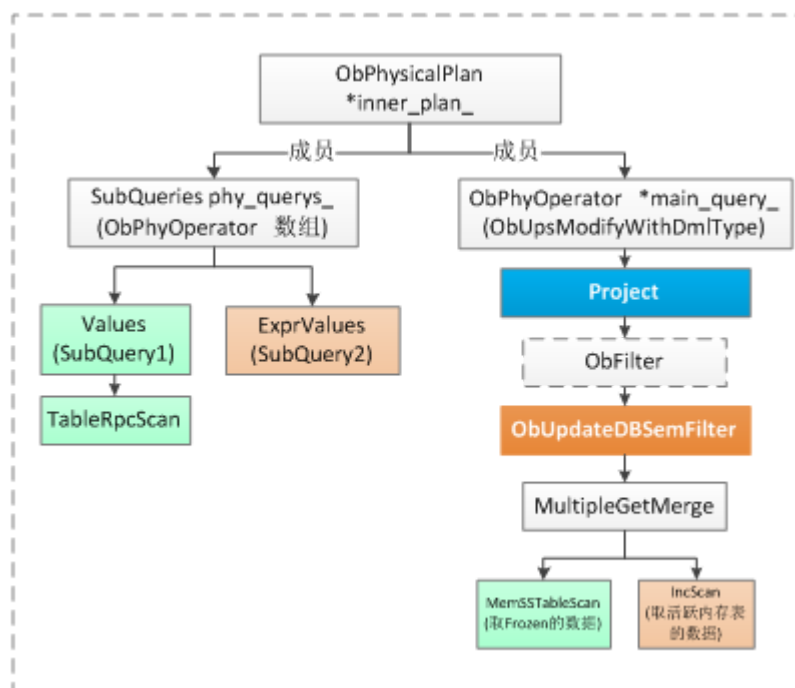


图1-2 单行不含二级索引更新操作的物理执行计划

单行更新含二级索引的物理计划

单行更新含二级索引的表对应的物理执行计划如下图所示，其中ObFilter为非必定存在的操作符。

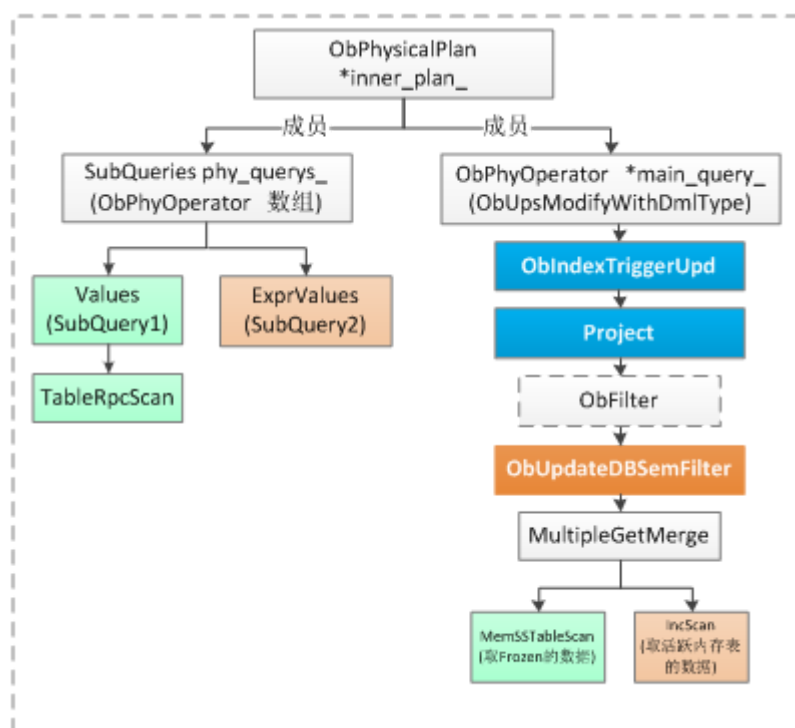


图1-3 单行含二级索引更新操作的物理执行计划

批量更新的物理计划

批量更新不含二级索引的物理计划

更新不含二级索引的表对应的物理执行计划如下图所示，其中：

- (1) **ObFillValues** : 在MS端执行open操作，查询出所有涉及的旧行数据，构造批量更新的IN表达式。在该操作符中计算新主键值，特别的，提前获取了Sequence的值。检查同批更新的行中是否有多个行更新为相同的主键值。IN表达式中新行与旧行交替出现，保证某行的新行在旧行前，确保在UPS端更新操作不会对查询增量数据产生影响。
- (2) **Values** : 包含一个ObTableRpcScan操作符，用于查询和保存基线数据。
- (3) **ExprValues** : 用于在UPS查询增量数据，新行和旧行的顺序与Values中IN表达式中主键的顺序一致，若是新行，将该新行的所有非主键列值设置为OB_NEW_ROW，而旧行所有非主键列值为NULL，用以区别新行和旧行。
- (4) **ObProject** : columns_列表保存了所有更新列的赋值表达式，包括新主键列的赋值表达式，原ObProject操作符不允许添加相同ID的列赋值表达式，改造后的ObProject去掉了这个限制。
- (5) **ObUpdateDBSemFilter** : 新增的操作符，检查基线数据与增量数据融合后新行是否存在。
- (6) **InScan** : 查询增量数据，并根据查询参数中的OB_NEW_ROW的标记区分出新行和旧行。

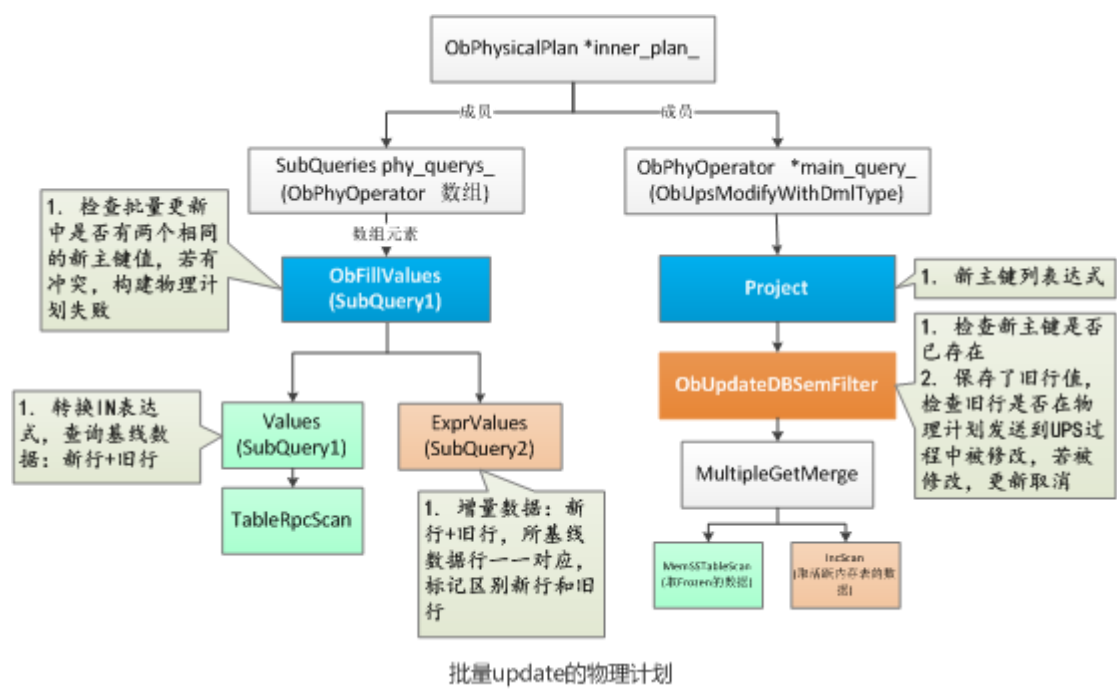


图1-4 批量不含二级索引更新操作的物理执行计划

批量更新含二级索引的物理计划

单行更新含二级索引的表对应的物理执行计划如下图所示。

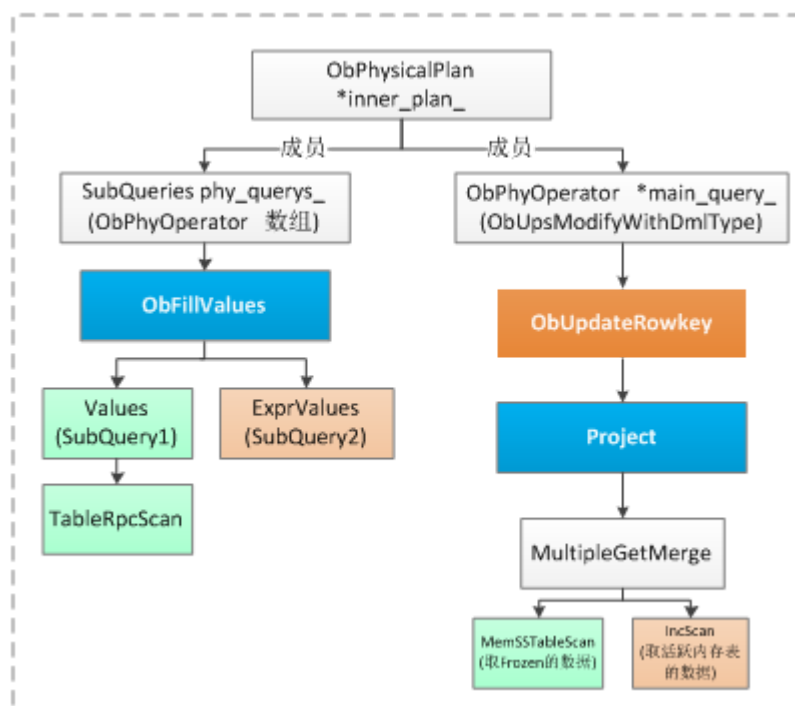


图1-5 批量含二级索引更新操作的物理执行计划

4 检查主键冲突的实现

主键冲突指由于数据库表中主键唯一性的约束，更新后的主键值不能与数据库已有的行主键重复，并且批量更新不能将多行更新为相同的主键值。特殊的，OB中主键值可为NULL，并允许多行的主键值同时为NULL。

基本流程

由于OB中基线数据与增量数据的分离，在检查主键冲突时，首先需要在MS端读取CS上的基线数据，并将基线数据发送给UPS，UPS融合基线数据和增量数据后，若行不存在，则不存在主键冲突；否则说明主键存在冲突，更新失败。

检查主键冲突的流程如下图所示，在MS端生成物理执行计划时，分为“单行更新”和“批量更新”两种情况，其中单行更新通过一次策略为ObSqlReadStrategy::USE_SCAN，一致性为STRONG的ObTableRpcScan操作获取旧行所有列的数据，以旧行为基础计算出新主键的值，并将原查询基线数据的RpcScan中的等值表达式转换为IN表达式，将新行和旧行的基线数据一起发送UPS，同时在查询增量数据的表达式中加入新主键行，并标记新行和旧行。特别的，若主键值实际没有改变，则只添加旧行。批量更新的流程中已经查询出旧行的数据，且构造了一个IN表达式，因此只需要计算新主键的值，加入到已有的IN表中。在此过程中，检查多个新主键是否重复，若重复说明存在主键冲突，更新失败。

在UPS端执行物理计划时，融合新行和旧行的基线数据和增量数据，若新行已存在，则说明存在主键冲突，更新失败。同时比较融合后的旧行数据是否被修改，若被修改，说明在MS计算得到的新主键值失效，更新失败。

在UPS端执行物理计划时检查主键冲突，而非MS生成物理计划时检查的原因有：

- （1）在MS生成物理计划时，检查到新主键已存在，可能在UPS执行物理计划时，该新主键被修改为其它值，此时，实际应不存在主键冲突；
- （2）在MS生成物理计划时，检查到新主键不存在，可能在UPS执行物理计划时，某行的主键被修改为新主键值，或插入了新主键值的行，此时，实际存在主键冲突；

在MS生成物理计划时检查批量更新主键重复，而非在UPS执行物理计划时检查的原因有：

- (1) 在MS端生成物理计划，计算得到的多个新主键重复，在UPS执行物理计划时再次计算得到的新主键值必定也重复，否则说明旧行数据在此过程中被修改，更新依然失败；
- (2) 在MS端生成物理计划，计算得到的多个新主键不重复，在UPS执行物理计划时再次计算得到的新主键值必须也不重复，否则说明旧行数据在此过程中被修改，更新失败；

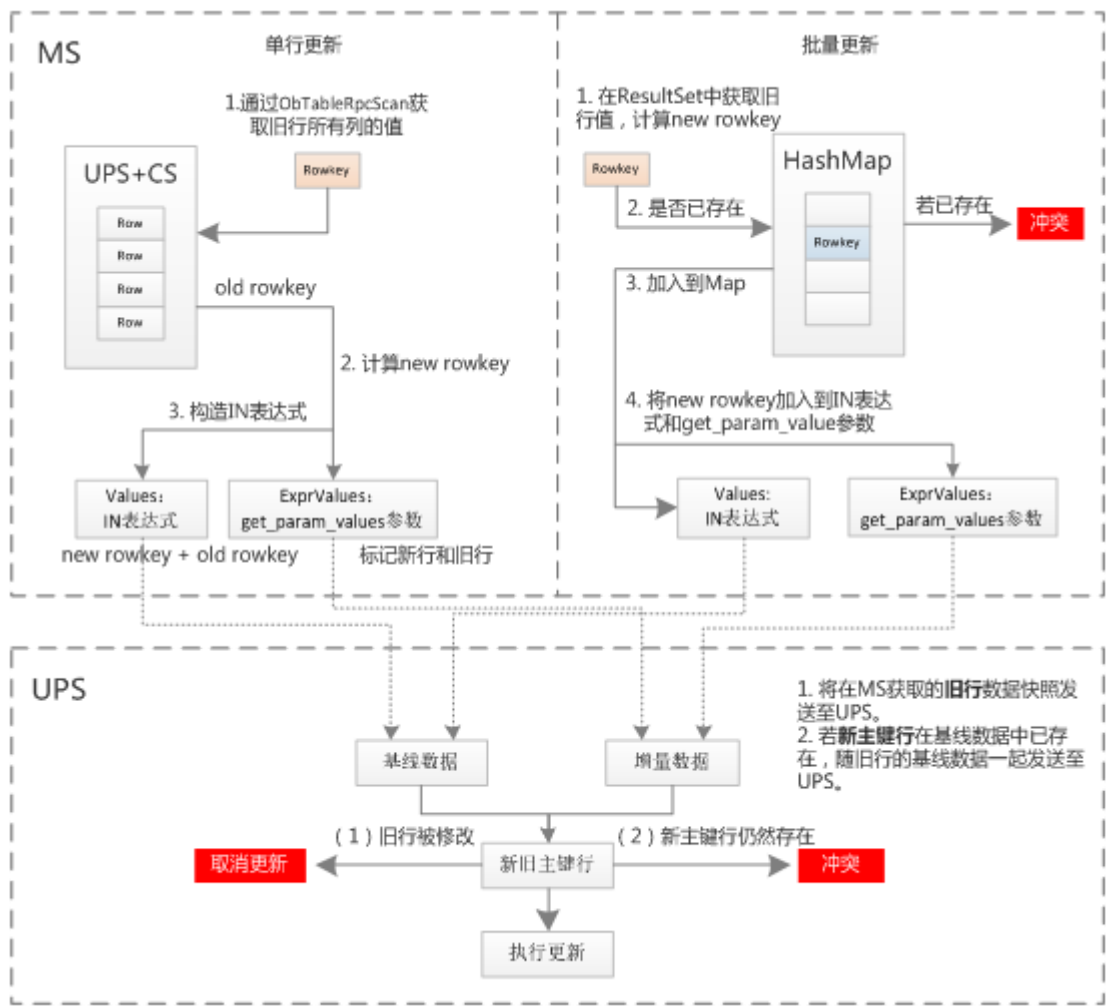


图1-6 MS和UPS检查主键冲突的流程

MS查询基线数据和检查主键冲突

MS需要查询出新行和旧行的基线数据，并且构造获取增量数据的表达式参数，由于MS对单行更新和批量更新的处理流程不同，因此查询基线数据分为单行更新和批量更新两种情况。特别的，在批量更新主键值时，检查同一批更新中是否有主键冲突，若存在冲突，则生成物理计划失败，不再发送给UPS执行。

单行更新

假设存在table ID为3001的表，其主键为联合主键，同都是int类型，column ID分别为16和17，通过全主键条件更新column ID为16的主键值为3。以下为修改前后查询基线数据和获取增量数据方式的对比。

```
update <3001> set <16> = 3 where <16> = 1 and <17> = 2;
```

(1) 构造IN表达式

原单行更新用于查询基线数据的过滤器表达式为多个等值表达式，为了同时查询出新行和旧行的基线数据，需要将上述等值表达式转换为IN表达式，其中新主键在旧主键前面，如下所示：

原单行更新的等值表达式	更新主键的IN表达式
Filter(filters= [expr<NULL,65517>= [COL<3001,16> int:1 EQ<2>], expr<NULL,65516>= [COL<3001,17> int:2 EQ<2>],]) 表示查询出主键值为 (1, 2) 的行	Filter(filters=[expr<NULL,18446744073709551615> = [COL<3001,16> COL<3001,17> ROW<2> LEFT_PARAM_END<2> int32:3 int32:2 ROW<2> (新主键) int32:1 int32:2 ROW<2> (旧主键) ROW<2> IN<2>],]) 表示查询出主键值为 (3,2) 和 (1, 2) 的行

若更新后的主键值实际没有改变，则IN表达式中只有旧主键值。

(2) 修改获取增量数据的get_param参数

ExprValues操作符中包含用于获取增量数据的get_param参数，在将物理执行计划发送给UPS前，ExprValues中的值 (values) 会转换为get_param参数，在更新主键的ExprValues的值中加入新行的主键值，其新行和旧行的相对位置与IN表达式中新旧主键值一一对应，把新行置于旧行之前的目的是避免查询到更新后的新行的增量数据，将新行的非主键列值设置为int:20 (OP_NEW_ROW的值) 表达该行为更新后的行。

原ExprValues值	更新主键的ExprValues值on
values= < 0:expr<3001,16 >=[int:1]> < 1:expr<3001,17 >=[int:3]> < 2:expr<3001,18 >=[null:]>	values= < 0:expr<3001,16>=[int32:3]> < 1:expr<3001,17 >=[int32:3]> < 2:expr<3001,18 >=[int:20]> (20表示新行) < 3:expr<3001,16 >=[int:1]> < 4:expr<3001,17 >=[int:3]> < 5:expr<3001,18 >=[null:]>

若更新后的主键值实际没有改变，则ExprValues的值中只有旧行。

批量更新

假设存在table ID为3001的表，其主键为联合主键，同都是int类型，column ID分别为16和17，通过column ID为16的主键列值等于1的条件更新column ID为16的主键值为3。

update <3001> set <16> = 3 where <16> = 1;

(1) 检查批量主键冲突

批量更新可能同时更新多行的主键，需要判断多行的新主键值是否有冲突。在计算出每行的新主键值后，使用标准库中的哈希集合std::set<int64_t>，判断该行新主键的哈希值在集合中是否已存在，若已存在则说明有主键冲突，更新操作失败。（没有使用ObHashSet或ObDuplicateIndicator数据结构的原因是不能事先确定批量更新涉及多少行，也就不能给哈希表分配合适的大小）。

(2) 保存旧行值的快照

将旧行的完整的值保存到ObUpdateDBSemFilter操作符中，用于在UPS端检查旧行在物理计划从MS发送到UPS的过程中是否被修改。在MS计算得到的新主键值是以当时时刻的旧行数据为基础的，若旧行值中途被修改，则说明新主键值已失效，因此，需要将MS计算新主键时的旧行值的快照一起发送至UPS。

(3) 构造IN表达式

批量更新查询基线数据的Filter也是一个IN表达式，因此只需要将新主键值分别加入到对应旧主键值的前面。同样，若更新后的主键值实际没有改变，则IN表达式中只有对应行的旧主键值。

批量更新的IN表达式	更新主键的IN表达式
Filter(filters=[expr<NULL,18446744073709551615>=[COL<3001,16> COL<3001,17> ROW<2> LEFT_PARAM_END<2> int32:1 int32:2 ROW<2> int32:1 int32:3 ROW<2> ROW<2> IN<2>],]) 表示查询出主键值为（ 1, 2 ）和（ 1,3 ）的行	Filter(filters=[expr<NULL,18446744073709551615>=[COL<3001,16> COL<3001,17> ROW<2> LEFT_PARAM_END<2> int32:3 int32:2 ROW<2> （ 新主键 ） int32:1 int32:2 ROW<2> int32:3 int32:3 ROW<2> （ 新主键 ） int32:1 int32:3 ROW<2> ROW<4> IN<2>],]) 表示查询出主键值为（ 3,2 ）、（ 1,2 ）和（ 3,3 ）、（ 1,3 ）的行

(4) 修改获取增量数据的get_param参数

与单行更新类似，批量更新主键的ExprValues的值与IN表达式一一对应。同样，若更新后的主键值实际没有改变，则ExprValues的值中只有对应的旧行。

批量更新的ExprValues值	更新主键的ExprValues值
values=<0: [int32:1], [int32:2], [null:]> <1: [int32:1], [int32:3], [null:]>	values=<0: [int32:3], [int32:3], [int:20]> （ 20表示新行 ） <1: [int32:1], [int32:2], [null:]> <2: [int32:3], [int32:3], [int:20]> （ 20表示新行 ） <3: [int32:1], [int32:3], [null:]>

UPS检查主键冲突

UPS在ObUpdateDBSemFilter操作符中检查主键冲突，ObUpdateDBSemFilter的惟一子节点是ObMultipleGetMerge操作符，ObUpdateDBSemFilter判断ObMultipleGetMerge融合基线和增量后的数据中是否存在新行，若存在，则说明有主键冲突。

融合基线和增量数据

ObMulitpleGetMerge在融合基线和增量数据时，需要根据增量数据参数中“新行”的标记识别出融合后数据中的新行和旧行。

检查主键冲突

ObMultipleGetMerge在open时获取融合后所有行的数据，若发现任一带“新行”标记的行出现，则说明存在主键冲突，不允许更新。

检查原行是否被更新

ObMultipleGetMerge在open时获取融合后所有行的数据，与从MS携带过来的旧行数据快照数据——对比，若发现任一列值不一致，则说明在MS端计算的新主键值失效，检查主键冲突的操作也无效，不允许更新。

5 更新主键值的实现

UPS以追加的形式更新增量数据，每行以主键列值为键值，即若当前行已存在，则追加到该行，否则插入新行，继续在新行上追加数据，因此，在检查主键无冲突后，需要插入新行，并标记删除旧行。

基本流程

更新某一行的主键值，实质上是删除旧行，插入新行的过程，而新行需要包含旧行所有的列值，即使该列值没有更新。更新主键值分为两个步骤：1）在MS端查询出所有列的基线数据；2）在UPS端插入由新主键值和旧行的其它列值组成的新行，并删除旧行。

在MS端查询出所有列的基线数据，只需将所有非主键加入到列行描述信息ObRowDesc中，所得本次更新涉及所有列。在UPS端插入新行删除旧行的流程如下图所示：新主键值的赋值表达式保存在ObProject操作符中，在执行物理计划时，根据融合后的旧行数据计算出新主键值，首先插入新行，然后删除旧行，最后将所有非主键列值追加到新行。特别的，若新主键值实际没有改变，则不插入新行，仍有原行中追加数据。

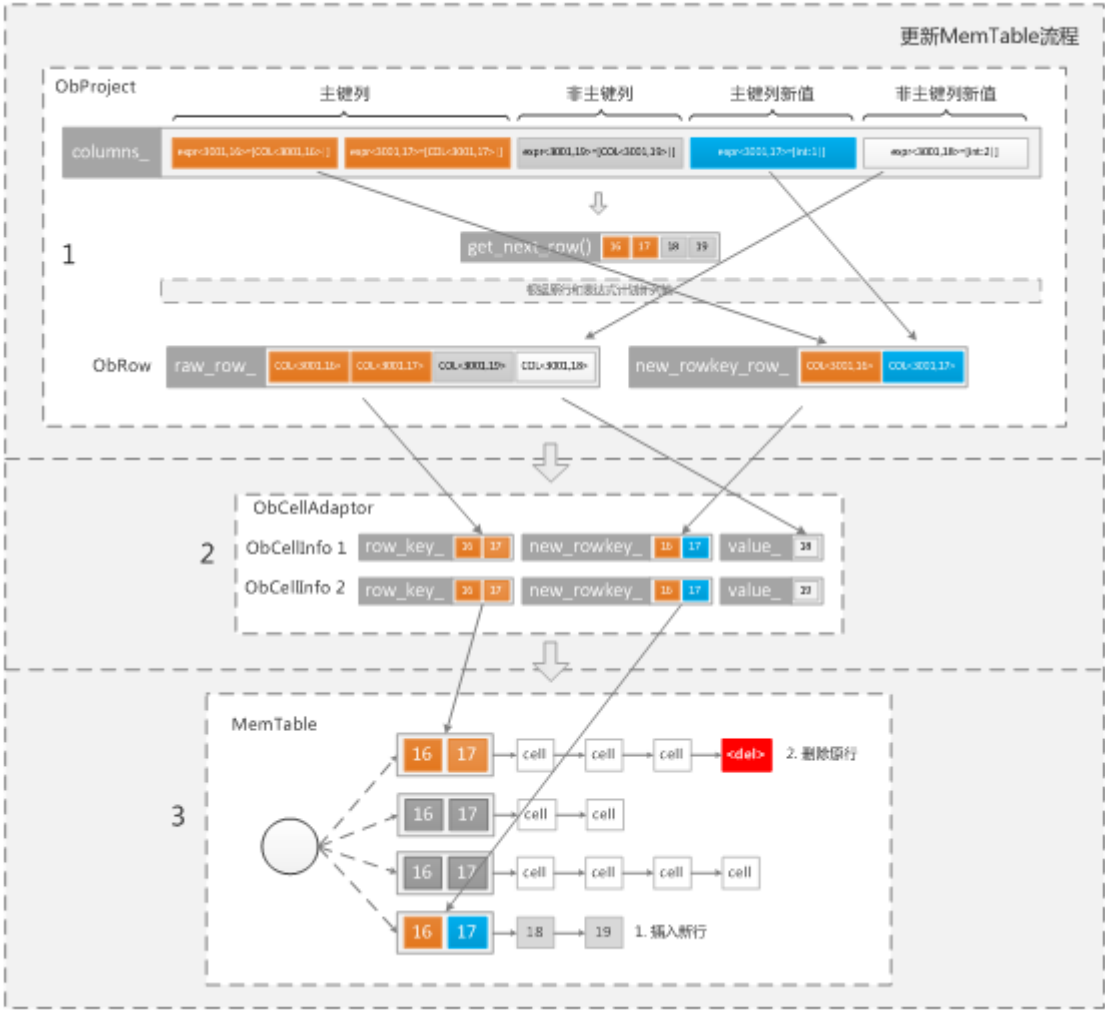


图1-7 UPS更新主键值的流程

MS查询全部列的基线数据

在OB系统中，某行的数据分为基线数据和增量数据，分别存储于CS和UPS，而增量数据是对部分列的最近更新值，因此UPS上的增量数据包含的列值对整行的数据来说是不完整的。在原update流程中，只会查询出主键列和该次更新操作涉及的列的基线数据，而当更新主键时，需要在UPS的增量数据中插入一个完整的新主键列和旧行其它列组成的完整的新行。因此，MS生成的物理执行计划应该查询出涉及行的所有列的基线数据。

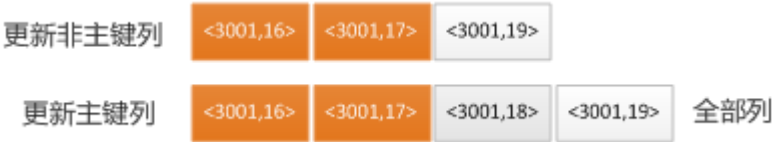


图1-8 更新主键的行描述信息

UPS更新主键值

更新主键值分为三个步骤1) 在ObProject中计算新主键值；2) 在迭代器中构造Cell；3) 将Cell追加到Memtable。

计算新主键值

更新主键列的赋值表达式与更新非主键列的赋值表达式一起保存在ObProject操作符中，ObProject根据行描述信息和更新语句生成输出表达式列表columns_，该列表按顺序包含了4个部分：

- (1) 原主键的引用表达式；
- (2) 原非主键的引用表达式；
- (3) 更新的主键表达式和非主键表达式。

增加ObRawRow类型的成员保存新行的完整的主键值，即若某个主键未被更新，则保存为旧行的主键值，否则保存为更新后的主键值，用于在Memtable中插入新行。赋值表达式列表如下图所示：



图1-9 更新主键的赋值表达式列表

由于原ObProject不允许表达式列表出现相同column ID表达式，因此，对这一限制进行了修改，允许主键列的赋值表达式最多出现一次。

修改Memtable

Memtable是UPS存储和查询增量数据的内存结构，由多行的Cell键表组成，每次更新操作都会在Memtable中产生一条记录，即在相应的列上追加Cell。

(1) 插入新行

每行都至少包含一个非主键列，在追加新行的非主键值Cell时，会自动插入一个新行。特别的若Memtable已存在被标记为删除的新主键行，则不会生成新行，继续追加至该行末尾。

(2) 删除旧行

若追加新行的第一个cell成功，即插入新行成功，则标记删除旧行。

备UPS更新主键

主UPS以操作日志形式将更新操作发送至备UPS执行，而操作日志由多个Mutator组成，只需在操作日志中加入删除旧行的Mutator，即可在备UPS执行相同的更新主键操作。

事务回滚与原子性

对原有的事务回滚机制没有修改，由原有的机制保证操作的原子性。

6 对既有模块的影响

本文对更新流程的修改，主要涉及update物理计划的生成和执行，对部分基础数据结果增加了一些属性，主要对原update、批量更新、二级索引和Sequence模块进行了修改。

对原update功能的影响

（1）原流程会主动拦截更新主键

原更新操作不支持更新主键的操作，在生成物理计划时，若检查到更新主键，会报OB_ERR_UPDATE_ROWKEY_COLUMN的错误，需要移除相关的检查代码逻辑。

（2）构造所有列的行描述信息

行描述信息是整个更新流程的基础，涉及的每一行的数据都包含了完整的列值，因此也影响了二级索引和Sequence，需要对部分代码进行修改。

（3）原ObProject不允许在赋值表达列表中添加相同column ID的表达式

允许添加新主键的表达式。在open操作时通过表达式列表构造行描述信息，而行描述信息不允许出现相同column ID的列，因此，在ObProject中增加了一个记录主键列ID的哈希表，在构造行描述信息时跳过新主键的赋值表达式。

（4）新增ObUpdateDBSemFilter操作符

改变了update物理计划树的结构。

（5）在追加新行的过程中删除旧行

将1次追加Cell的操作解释成追加2个Cell（新行的某列值Cell + 旧行的删除标记Cell）。

对批量更新功能的影响

（1）在IN表达式加入新主键值

先将主键值保存到一个列表，在计算出所有行的新主键值后，再交替加入到IN表达式中，可能会增加基线数据的数据量。

（2）在get_param中加入新行

先将旧行保存（深度复制）到一个列表，在计算出所有行的新主值后，再交替加入到get_param中。

（3）增加了旧行快照数据

增加了发送到UPS的数据量。

（4）增加检查批量更新主键值是否重复

增加了个std::set<int64_t>哈希表检查多个新主键值是否重复。

对Sequence功能的影响

(1) 修改了ObProject的表达式

Sequence根据ObProject的表达式来计算seq的实际值，在ObProject增加了非主键列的表达式，需要在计算seq值忽略该表达式，如下图所示，即应跳过expr<3001,19>的表达式：

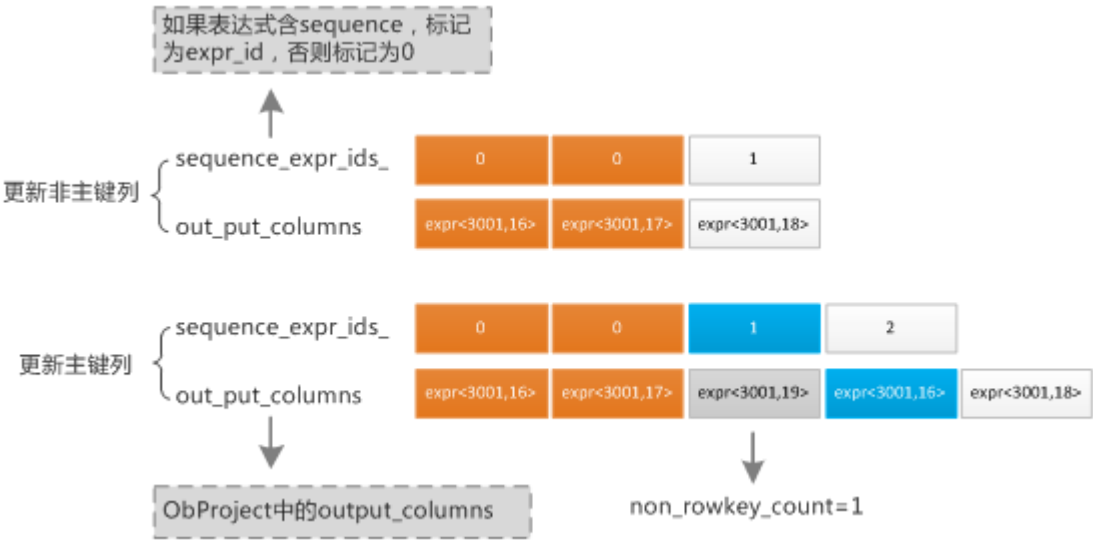


图1-10 计算Sequence的实际值

其中：<16>和<17>为主键列，更新主键表时设置列<16>和<18>为Sequence值。

(1) 在MS获取Sequence的值

在查询出旧行数据后，计算新主键值时，若新主键值使用到了Sequence，则替换为实际值。

对二级索引功能的影响

更新二级索引时需要更新索引表的主键列，都是更新主键列，两种方案的区别在于物理计划携带更新后的新行全部列值的方式不同，更新索引表时增加了一个新的操作符用以携带这部分数据，而本方案中只是在原有的操作符中增加一部分信息，使得同时更新二级索引和主键时不会产生冲突。

(1) 修改了二级索引的表达式列表和转换值列表

二级索引新增了ObIndexTriggerUpd操作符来更新索引表，该操作符中的set_index_columns更新索引表各列的赋值表达式，cast_obj用于转换计算后的实际值的类型，更新主键时由于已经有了新主键的表达式，因此不添加旧主键的表达。

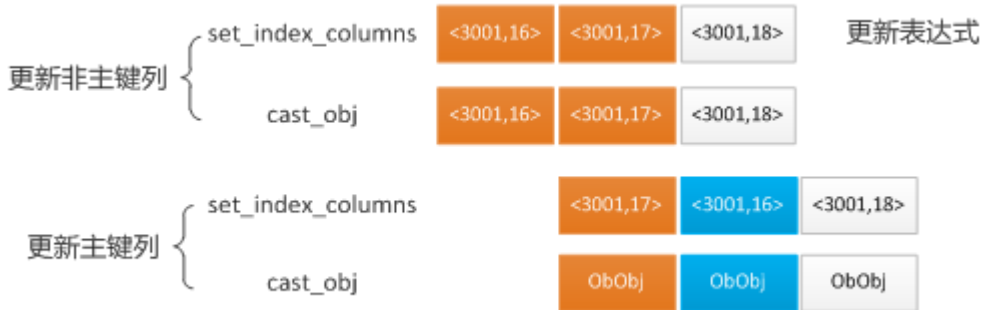


图1-11 含二级索引的表更新非主键列和主键列的对比

（2）在基线数据中增加了多余的列

原二级索引更新时只会使用索引表涉及的列的基线数据，如下图所示，更新主键列时基线数据中会包含非主键列值，更新二级索引时需要跳过这部分列值。

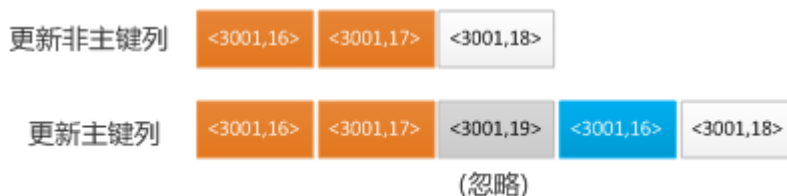


图1-12 含二级索引的表更新非主键列和主键列的基线数据对比

集群管理

1 名词解释

系统各节点纵向组成多个小集群，横向分为四个层次：

- RootServer层：每个小集群可指定0或多个RS，在分布式选举过程中，若没指定RS的优先级，各RS地位平等，即不修改原有的RS选举流程；
- UpdateServer层：每个小集群可指定0或多个UPS，主UPS平等的将日志同步至本集群内和集群外的备UPS，即不修改原有的UPS日志同步流程；RS选取主UPS时，在最大日志号和任期时间戳相同的情况下，优先选择本集群的UPS为主；
- MergeServer层：每个小集群可指定0或多个MS，MS将写请求发送给主UPS，优先将读请求发送给本集群的CS；
- ChunkServer层：每个小集群可指定0或多个CS，负载均衡以小集群为单位，每个Tablet在各个小集群内的副本总数相同（若某个集群存在CS，则该集群拥有数据库完整的基线数据），CS优先将副本迁移至本集群的CS；CS按UPS流量比例配置将读请求发送给主备UPS。

2 概要设计

CBASE系统的负载分为存储负载和服务负载。存储负载均衡指数数据表的Tablet在各ChunkServer上分配均衡，即每个Tablet的副本数目达到一定值，每张表在每个CS的Tablet副本数目在一定范围内。

数据由基线数据和增量数据组成，负载均衡针对于基线数据。基线数据中每张Table（表）按照主键排序并且划分为数据量大致相等的Tablet（子表），每个子表包含多个副本，分布在多台CS上，RootTable（根表）记录了每个Tablet在多台CS上存储的位置信息。子表迁移指某个子表副本从源CS复制至目标CS上，然后源CS将该子表副本删除。

RootServer包含一个独立线程（ObRootBalancerRunnable）定期执行子表复制和负载均衡任务。

ObChunkServerManager管理一个迁移任务队列，当检查线程检测到需要复制子表或均衡负载时，触发子表复制或迁移，生成一个迁移任务加入到列表，同时将命令发送给相应的目标CS执行，该命令标记源CS是否需要保存子表副本，若需要保存即为子表复制，否则为子表迁移，目标CS在主动向源CS拉取子表副本完成后，通知RS从迁移任务列表中移除相应的任务，同时，若源CS不需要保存子表副本，则通知源CS删子表副本。

检查线程在进行下一轮检测前，会一直等待上一轮检测生成的迁移任务全部完成或超时；在生成子表迁移任务前，会检查每个CS正在迁入的子表数目、正在迁出的子表数目不超过阈值，且整个系统同时可进行的迁入迁出子表总数不得超过迁移任务列表（数组）的最大长度。

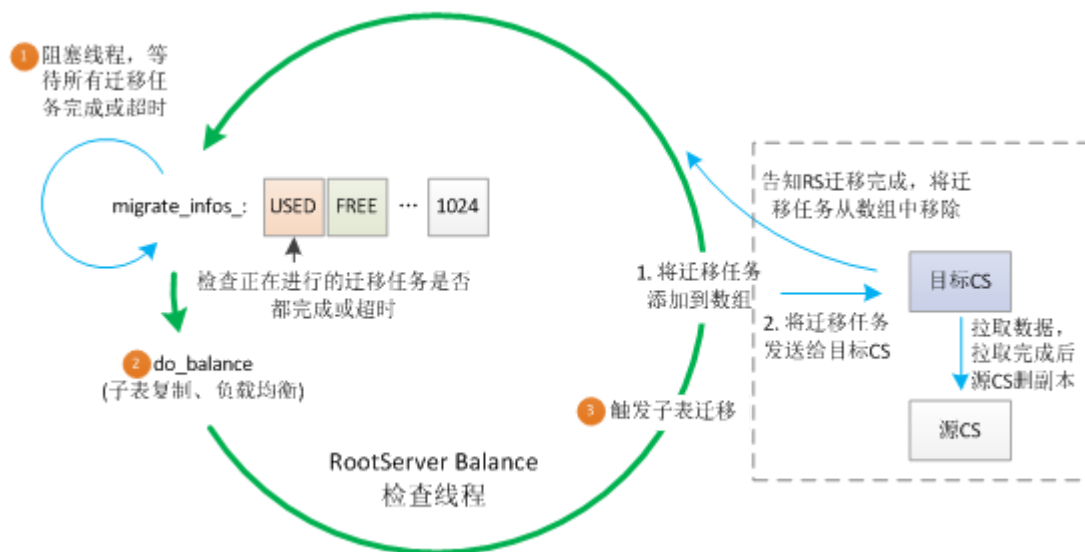


图3-1 负载均衡总流程

负载均衡共三个步骤，对原有总流程不修改：

- 1) 等待上一轮迁移任务全部完成或超时；
- 2) 开始下一轮检测子表复制和负载均衡；
- 3) 触发子表迁移，分发迁移任务给各CS执行，更新迁移任务列表。

3 详细设计

3.1 集群号约束实现

为方便内部实现，规定一个OB系统最多只可划分为6个集群，每个集群号可指定为1、2、3、4、5、6中六个数字的任意值，可不连续，例如：可部署三个集群，集群号分别指定为1、2、5。

定义以下常量：

```
//既有常量，最大集群数目为6

static const int64_t OB_MAX_CLUSTER_COUNT = 6;

//规定集群号指定范围为1-6

const int32_t OB_MAX_CLUSTER_ID = OB_MAX_CLUSTER_COUNT;

//用于定义集群相关的数组，下标0不使用

const int32_t OB_CLUSTER_ARRAY_LEN = OB_MAX_CLUSTER_COUNT + 1;

//标记其它任务集群的集群号

const int32_t OB_EXTERNAL_CLUSTER_ID = -1;

//建表时默认的副本数，RS替换为实际可用的最大副本数
```

```
const int32_t OB_DEFAULT_COPY_COUNT = -1;

//内部表__all_cluster_config集群副本数字段名称前缀

const char* const OB_CLUSTER_REPLICA_NUM_PREFIX = "cluster_replica_num_";
```

集群号及含义说明：

cluster id	含义
-1	代表大集群内的其他某小集群作用：标记从其他集群中复制子表
0	代表所有集群作用：标记访问所有集群的资源
1-6	有效集群号
其他	无效值

3.2 副本数目管理器

新增ObRootClusterManager类保存和管理集群副本数目，系统启动时，根据集群数目初始化各集群副本数目，若某表使用自定义的副本数目，则根据默认集群副本数的比例重新计算各集群的副本数目。由于副本存放在CS中，副本分布涉及到的集群指的是有在线CS的集群。

```
class ObRootClusterManager
{
private:
    //各集群的副本数，总副本数<=6
    int32_t replicas_num_for_each_cluster_[ OB_CLUSTER_ARRAY_LEN];
    //主集群的集群号
    int32_t master_cluster_id_;
};
```

3.2.1 初始化集群副本数目

集群默认的副本数目需要根据含CS的集群数目计算，计算结果如下表示：

集群数目	主集群默认副本数目	备集群默认副本数目	副本总数
1	3	0	3
2	3	3	6
3	2	2	6
4	3	1	6
5	2	1	6
6	1	1	6

在创建内部表和系统表时，需要使用集群默认的副本数目，因此，在创建表前初始化集群副本数目。（初始化集群副本数=6/集群数，多余的追加至主集群）。

3.2.2 计算集群副本数目

负载均衡过程中，检查某张表的副本总数，若等于默认的副本总数，则直接使用各集群默认的副本数，否则，根据自定义的副本总数，计算各集群实际分配的副本数目。计算方法如下：

```
For Each cluster
    1. 计算该集群默认副本数目与默认总副本数的比例
    2. 该集群的副本数 = 自定义的副本总数 * 比例
    3. 若计算得到的副本数等于0，则修改为1，即每个集群至少有一个副本
End For
4. 若计算得到总副本数小于自定义的副本总数，则将该差值增加到主集群，保证总副本数一致
```

3.3 存储负载均衡实现

负载均衡功能实现主要在类ObRootBalancer中，检查线程单线程顺序的迭代所有Table，检查每个含在线CS的集群内的子表副本数是否符合要求；再迭代所有Table，检查每个集群内的CS副本负载是否符合要求。

3.3.1 子表复制

1. 计算子表的集群副本数目

由于检查线程是单线程顺序的执行，因此每迭代一张表，则读取该表的总副本数目信息，计算出在每个集群分布的副本数，保存在成员变量数组中。

```
class ObRootBalancer
{
    private:
        int32_t balance_replica_num_[OB_CLUSTER_ARRAY_LEN];
}
```

2. 按集群复制子表

检查每个集群的子表副本数是否符合要求，若过少，则复制到本集群的其它CS上，必要时，可从其它集群的CS上复制子表；若过多，则从本集群中选择一个副本删除。

3.3.2 负载均衡

1. 计算每个集群内CS的平均负载

统计每个集群内某张表的子表副本总数，再除以该集群内CS的数目，得到每个集群内CS的平均负载。

2. 按集群均衡负载

判断每个集群内的每个CS上的负载是否符合集群平均负载的一定范围，若过高，则迁移至本集群的其它CS上。

3.4 服务负载均衡

小集群服务负载控制指RS管理集群内部的UPS、MS和CS，控制各节点间的相互通信，避免数据访问和服务跨跃多个集群。

目标节点	消息类型	说明
RS	所有类型	所有请求发送至主RS
UPS	写请求	MS发送至主UPS
UPS	强一致读	CS发送至主UPS
UPS	非强一致读	主UPS所在集群的CS按设定的流量比例发送至本集群的主备UPS；若本集群没有备UPS，则只发送至主UPS；其他集群的CS按等分的流量比例发送至本集群的备UPS；若集群没有UPS，则发送至主UPS所在集群的UPS
MS	SQL	系统内部所有SQL发送至主UPS所在集群的MS，若无MS，则选择任意集群的MS
CS	读请求	MS优先读取本集群的CS的数据，若本集群没有数据，则读主UPS所在集群的数据，若该集群没有数据，则读任意集群的数据

4 外部工具命令

- 查看子表分布信息，运行rs_admin命令：

```
bin/rs_admin -r 10.10.10.1 -p 12220 dump_balancer_info -o table_id=1
```

在主RS的系统日志中可看到如下输出信息：

```

===== Balancer Info [table id: 1] =====
<-- Rereplication -->
[Tablet]:1
  [Cluster]:1 [CS]: <182.119.80.57:12225@1,1> <182.119.80.52:12225@1,1> <182.119.80.53:12225@1,1>
  [Cluster]:2 [CS]: <182.119.80.56:12225@2,1> <182.119.80.55:12225@2,1>
<-- Balance -->
[Cluster]:1
  [CS: 182.119.80.57:12225@1]: 1
  [CS: 182.119.80.52:12225@1]: 1
  [CS: 182.119.80.53:12225@1]: 1
  [CS: 182.119.80.54:12225@1]: 0
[Cluster]:2
  [CS: 182.119.80.56:12225@2]: 1
  [CS: 182.119.80.55:12225@2]: 1
=====

```

表示table id=1的表的子表分布信息，该Table共有1个Tablet，编号为1。输出信息分为两部分：

1. Rereplication副本分布信息：

该唯一的子表共有5个副本，其中集群1中有3个副本，集群2中有2个副本。[10.10.10.2:12225@1,1](#)表示CS的IP为10.10.10.2，端口号为1225，所属集群号为1，子表版本号为1。

2. Balance CS负载信息：

表示每个集群内各CS每张表的子表副本总数。

- 查看集群副本数目：

有两个方法可查看集群副本数目。

1. 系统表：

通过查询_all_cluster_config表可查看各集群默认的副本数目。

2. rs_admin命令：

```
bin/rs_admin -r 10.10.10.1 -p 12220 get_replica_num
```

该命令可查看所有在线或配置副本数不为0的集群的副本数目。

- 修改集群副本数目，运行rs_admin命令：

```
bin/rs_admin -r 10.10.10.1 -p 12220 set_replica_num -o cluster_id=1,replica_num=3
```

使用限制：

- 修改后的总副本数不得超过OB_MAX_COPY_COUNT；
- 设置replica_num>0时，该集群内必须存在在线的CS。

直接修改内部表_all_cluster_config内的副本数目，不能立即生效，只有发生主RS切换从新从内部表中恢复数据，才会使用内部表中的数据，因此不建议直接修改内部表。

- 设置主集群ID，运行rs_admin命令：

```
bin/rs_admin -r 10.10.10.1 -p 12220 set_replica_num set_master_cluster_id -o
new_master_cluster_id=3
```

使用限制：

只有主RS能够对主集群ID进行修改，修改后会同步到__all_cluster表中；

- 查询RS保存的主集群ID，运行rs_admin命令：

```
bin/rs_admin -r 10.10.10.1 -p 12220 stat -o master_cluster_id
```

使用说明：

1. 主RS需要实时保存正确的主集群ID，与__all_cluster表同步，以记录负载均衡的根据；
2. 备RS在切换为主以后，从__all_cluster表中读取主集群ID，重置自己保存的主集群ID。

UPS日志同步及优化

1 原架构描述

OceanBase 0.4.2中在发生日志同步的UPS主要有：主集群的主、备UPS和备集群的主UPS。主UPS与主备UPS、备主UPS之间通过操作日志实现同步。正常情况下，主UPS作为数据写入和日志同步的发起者，备UPS进行日志同步和日志回放，主UPS由RS选主产生。主备UPS通过较弱的一致性日志同步方式，即主UPS向所有备异步发日志，如果备UPS没有回复则会将这个UPS剔除日志同步列表。即使所有备UPS同步日志失败，主节点仍然会写入日志并继续作为主节点执行操作。RS与UPS之间保持心跳链接，RS上有管理UPS的线程，一旦UPS出现故障，RS会及时检测出并重新选出主UPS。

2 存在问题及方案设计

2.1 主备日志不一致

生产环境中，使用OB0.4.2架构出现过主备UPS日志不一致导致备UPS启动失败的场景。主要是由于主UPS的操作日志直接写入了commit_log，当网络故障或者server故障无法同步给备UPS时，备UPS被选为主UPS后继续写日志，会出现与旧主UPS日志不一致的问题。

在双集群架构中，参与UPS同步的一般为3个，分别为server 1, server 2和server 3。其中server 1作为集群的主UPS，接受系统的写请求。举例如下图所示。（同种颜色编号相同代表是同一日志，编号相同颜色不同代表由不同的主UPS生成的日志，如蓝色6号日志由server 1生成，黄色6号日志由server 2生成）

主1	1	2	3	4	5	6	7	8	9				
备2	1	2	3	4	5								
备3	1	2	3	4									

旧主1	1	2	3	4	5	6	7	8	9				
新主2	1	2	3	4	5	6	7	8	9	10			
备3	1	2	3	4	5	6	7	8	9				

备1	1	2	3	4	5	6	7	8	9	10			
主2	1	2	3	4	5	6	7	8	9	10			
备3	1	2	3	4	5	6	7	8	9				

当失效的旧主server 1再次启动时，6-9号日志与新主server 2不一致，但是这部分日志已经写入server 1的磁盘。重启时，server 1会直接将6-9号日志应用到memtable中，而不是检查6-9号日志是否与新主server 2不一致。当server 1收到新主server 2的10号日志并进行回放时，进行校验和（checksum）检验，发现checksum与新主UPS不一致，server 1会kill self，导致异常结束进程，重启失败。

2.2 初步解决方案

上面的例子，解决的方案就是旧主server 1要删掉6-9号日志，并从6号开始接受新主的日志，旧主必须知道哪些日志是应该被删除的（例1中6-9号日志应该被删除），哪些日志是不应该被删除的（1-5号日志不用删除）。

参照Paxos[]的日志同步理论，把写入多数派（ $> m/2$ 表示UPS个数）UPS磁盘的日志（即例1中的1-5号日志，其中1-4号日志3个server都有，而5号日志2个server有）看作commit_log，即提交日志。其余的日志（6-9号）看作temp_log，即临时日志。重新选主之后，临时日志需要跟新主UPS的日志比对，如果是新主UPS没有的日志则应该删除。

因此，一条日志只有在多数派UPS写入磁盘成功后，才能在主UPS上提交对应的事务。

备UPS收到日志写盘后不能立即应用到memtable，要等待这条日志在主UPS上已经提交后才能应用到memtable。

以上修改主要涉及到：主UPS发日志、备UPS收日志、备UPS回放日志。

2.3 UPS选主问题

为了防止数据丢失，RS在进行UPS选主时，会选择数据数据较新的，即日志号较大的UPS作为新主。上述初步解决方案仍然存在UPS选主的问题，如例2所示：

主 1	1	2	3	4	5	6	7	8	9				
备 2	1	2	3	4	5								
备 3	1	2	3	4									

旧主 1	1	2	3	4	5	6	7	8	9				
新主 2	1	2	3	4	5	6	7	8	9	10			
备 3	1	2	3	4	5	6	7	8	9				

备 1	1	2	3	4	5	6	7	8	9				
旧主 2	1	2	3	4	5	6	7	8	9	10			
备 3	1	2	3	4	5	6	7	8	9				

接例1，新主server 2工作一段时间后也宕机了。这时，我们重启server 1，集群处于无主UPS状态，不能进行日志比对。RS需要在server 1和server3中选出主UPS，它们都将自己拥有的日志信息发送给RS，其中，server 1拥有的6-9号日志是server 1宕机之前的日志，即server 1还是主UPS时期产生的日志，为未提交日志；server3拥有的6-9号日志是最新的日志，即server 2是主UPS时期产生的日志。面对两份相同日志号的临时日志，RS可能会选server 1为主UPS，显然这是不可取的，因为server 3的日志是更新的日志，因此，应该选择server 3为主UPS。

2.4 最终解决方案

面对着两份日志号相同的临时日志，RS无法选择，所以在日志号的基础上，需要增加另一个日志属性——日志term，用来标记相同日志号日志的新旧。重新选举产生的主UPS拥有一个新的term号——一般使用时间戳表示。日志写盘时，不仅要写日志号和日志内容，还需要把term号一起写入磁盘，代表该日志是当前主UPS作为主时产生的。

9	<i>term = time1</i>
9	<i>term = time2</i>

经过这样修改上图旧主写的蓝色的日志是term = time1，新主写的橙色的日志是term = time2。再次选主时，server 1拥有term = time1的9号最大日志，server 3拥有term = time2的9号最大日志，从而选择term号较大的server 3为主节点。

以上修改主要涉及到：RS选主UPS。

- 能保证以下几点

以上设计的最终方案能够保证UPS上的数据满足一下几点：

1. 回复客户端成功的事务，对应的日志一定在多数派UPS上落盘成功。
2. 所有UPS存有的commit_log一定保持一致且连续。
3. 不同UPS上的日志，若其日志号、term号相同，则其日志一定一致。因为这是同一个主UPS（term号）发来的同一个编号（日志号）的日志。

3 UPS日志复制

3.1 日志复制

为了解决第二章所述的主备UPS日志不一致和UPS选主问题，提高集群可用性，我们根据Raft一致性协议重新设计了UPS的日志同步流程，保证主UPS宕机时，集群中多数UPS存在时，能够选出一个新主UPS，且日志一致连续。

3.1.1 算法概述

为了保证事务的原子性，分布式数据库常使用两阶段提交协议。该协议将事务的提交过程分为两个阶段。1) 主事务处理节点预处理事务并将事务日志写入磁盘后，向其它备事务处理节点发送prepare消息。备节点根据日志进行事务预处理并把日志写入磁盘，之后回复主节点。2) 若主节点根据收到全部备节点的回复信息，则提交该事务并向备节点发送commit消息。否则向备节点发送abort消息。备节点按照主节点的指令来commit或abort事务。这样，便保证了各个节点要么全部commit事务，要么全部abort事务。本文提出的日志复制机制在保证事务原子性方面采用了类似于两阶段提交协议的方法。

由于commit_log文件中只能记录已经确定能够提交的事务日志，我们增加一个新的日志文件temp_log，用来存储两阶段提交协议中第一阶段需要写入磁盘的日志，当第二阶段成功后，该temp_log将成为commit_log。

注：在具体实现中只有一种日志文件，commit_log日志和temp_log日志都写在一个日志文件中。我们另外使用一个文件commit_point文件，该文件内容只包含一个日志号，该日志号之前的日志为commit_log，之后的为temp_log。为了理解方便，我们将其描述为两种日志文件。

日志复制的流程如下所示：

Step1: 主UPS将事务应用到memtable后将对应日志发送给所有备UPS，无论发送成功或失败，主UPS都会把日志写入temp_log。

Step2: 备UPS收到日志后，先写入temp_log，成功后回复主UPS已经成功接收到日志的消息。

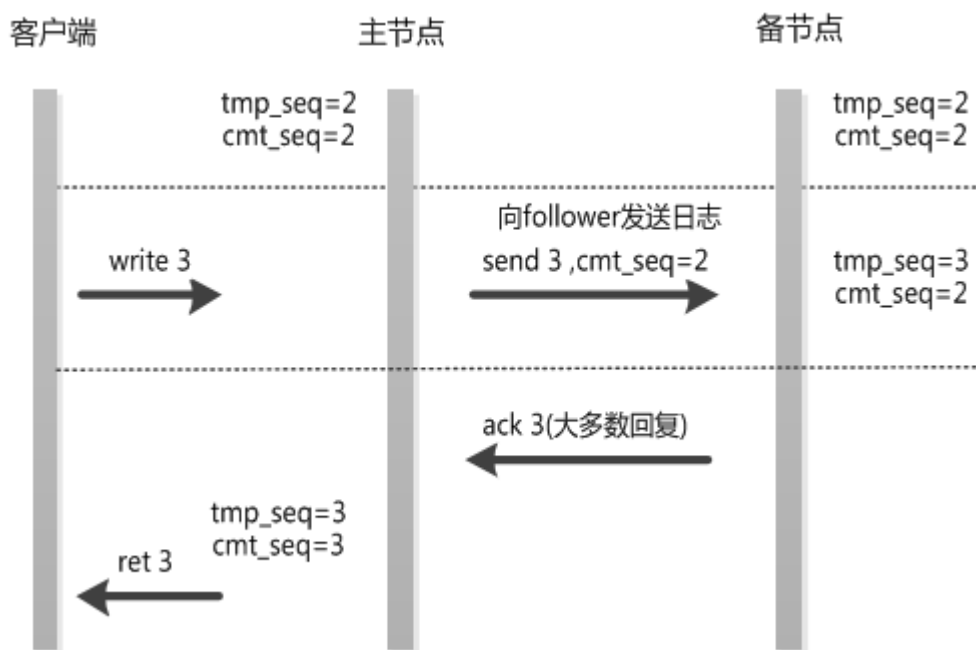
Step3: 主UPS根据收到回复的情况，如果有大多数（加上主UPS超过半数）备UPS回复，则主UPS将日志提交，并回复客户端。

Step4: 主UPS下一次发送日志时，会携带主UPS已提交最大日志号的信息，备UPS根据主UPS的这个日志号来将temp_log中对应的日志回放。

注：备UPS将日志内容应用到memtable称为回放，回放流程为：1) 将日志内容提交给applyworker--多线程将日志内容应用到memtable。2) 提交日志。

该日志复制机制允许主UPS连续发送日志，不会因为某条日志没有收到大多数备UPS回复而导致不能发送下一条日志，这提高了发送日志的效率，也缩短了事务执行时间。此外需要说明的是，如果主UPS收到了多数派回复，主UPS可以立即提交事务并响应客户端，而备UPS需要等待接收下一条日志时才能提交该事务。为了避免没有新的日志产生而导致备UPS无法提交之前的事务，我们令主UPS定期发送空日志(nop日志)，来确保备UPS的事务能够提交。这种机制减少了一次网络通信，提高了事务处理效率。

接下来我们将以一个具体事例解释上述过程，图1显示了主UPS向备UPS同步3号日志的流程。其中tmp_seq表示temp_log的最大日志号，cmt_seq表示commit_log的最大日志号。



主UPS发送3号日志前，临时日志号tmp_seq和提交日志号cmt_seq都为2。发送时，主UPS将3号日志写入本地的temp_log，更新临时日志号tmp_seq = 3，然后主UPS将3号日志发给每个备UPS，发送时携带当前最大提交日志号cmt_seq = 2一同发送给备UPS，之后等待备UPS的回复。

备UPS收到日志3和cmt_seq后，首先将日志3写入本地的temp_log，更新临时日志号tmp_seq = 3，并回复给主UPS ack3，表示已经将3号日志写盘成功。然后提交日志号小于等于cmt_seq的日志。

若主UPS收到了多数UPS回复的ack 3，则提交3号日志，并回复客户端ret 3。至此，3号日志对应的事务提交成功。

若备UPS没有收到主UPS的新日志，则无法提交3号日志，为了避免没有后续日志的情况，我们让主节点定期检查在这期间是否发过日志，如果没有发过日志，则发送一条空日志给备节点，由于这条日志携带者主节点已经提交日志号的信息，备节点收到后就可以提交3号日志了。

3.1.2 异常处理

◦ 网络异常

网络异常主要是指由于网络抖动、延迟或网络包丢失等问题引起日志复制偏离3.1.1所述的正常流程的情况。

对于主UPS来说，网络异常可能导致主UPS无法收到备UPS回复的日志写盘成功的消息，而不能提交相应事务；对于备UPS来说，网络异常可能导致备UPS接收到主UPS发送的日志。

如果主UPS发现日志Q没有收到多数派回复，此时，备UPS可能收到了日志Q也可能没有收到，主UPS不会提交相关事务，也不会重试发送。当主UPS发送下一条日志Q+1到备UPS后，备UPS会查看日志的序号。如果发现日志不连续，即日志Q由于网络丢包备UPS并未收到，则备UPS会向主UPS同步请求日志Q，补全之后，备UPS才会向主UPS回复。如果日志连续，则备UPS将日志Q+1写盘后回复主UPS。此时主UPS若收到大多数Q+1回复，则主UPS会将日志Q+1之前的事务一起提交。由于这种备UPS补全机制的保障，主UPS收到多数个日志号为N的回复，就可以将日志号N之前的日志一起提交。

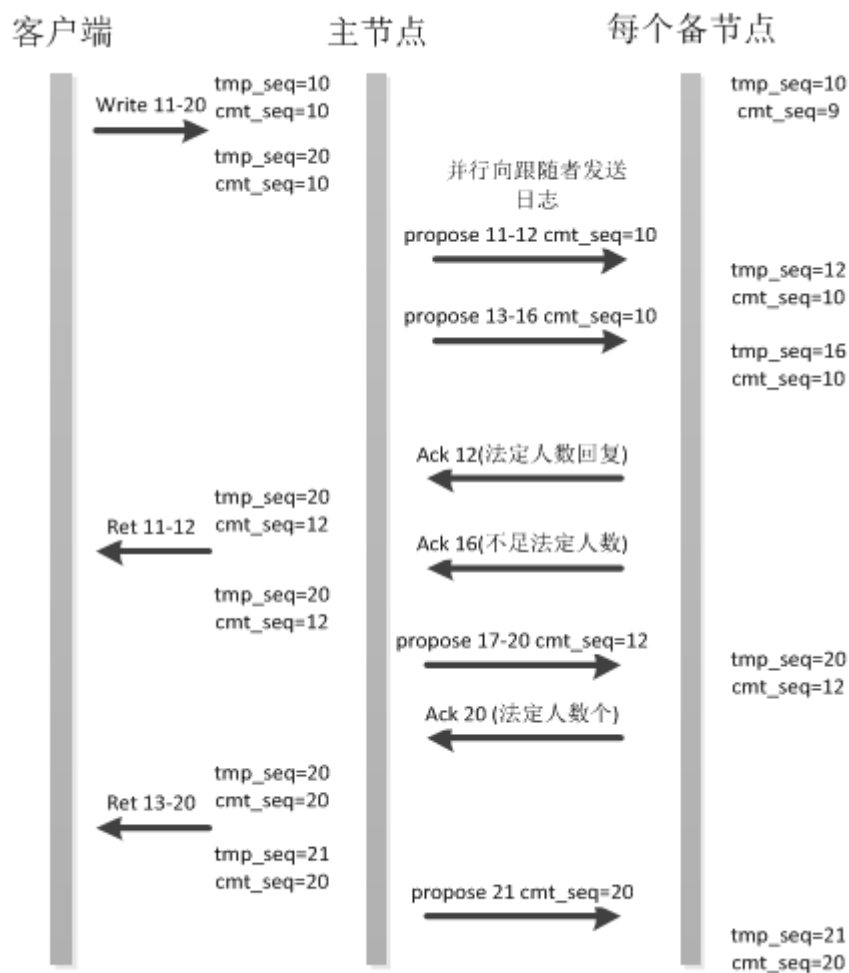
图5中是主节点发送11-20号日志时发生网络异常的一个例子：

主UPS发送日志给备UPS之前，临时日志号tmp_seq和提交日志号cmt_seq都为10。发送时，主UPS将11-20号日志写入temp_log，临时日志号tmp_seq变为20，假设主UPS将11-20号日志分成11-12、13-16和17-20三组日志发给每个备UPS，每发一组对应一次日志同步的过程。发送时连同当前最大提交日志号cmt_seq一同发送给备UPS，之后等待备UPS的回复。

备UPS收到主UPS的日志11-12和cmt_seq = 10后，首先把发来的日志写入temp_log，更新临时日志号tmp_seq = 12，并回复给主UPSack12，表示12号之前的日志已经落盘。接着备UPS根据主UPS已经提交的日志号cmt_seq提交自己的日志，并修改自己的提交日志号cmt_seq = 10。

假设主UPS正常收到了多数派备UPS回复的ack12，主UPS此时可以提交12号之前的日志，并回复客户端ret11-12。11-12这个日志包没有发生网络异常，日志复制正常结束。之后主UPS开始发送第二组13-16号日志。

假设在发送第二个组日志时由于网络故障或者备UPS的故障，导致主UPS接收的回复ack16不足多数派。原因可能是以下两种：1）备UPS未收到主UPS发送的日志；2）备UPS收到了日志，但是回复给主UPS的ack消息丢失在了网络里。主UPS无法提交13-16号日志。当下一组日志17-20正常得到备UPS的多数派回复时，主UPS把20号之前的日志，即，13-20号日志一起提交并回复客户端ret13-20。



此外，需要说明的是，若备UPS没有收到主UPS的新日志，则无法提交13-20号日志，为了避免没有后续日志的情况，我们让主UPS每过500ms检查在这期间是否发过日志，如果没有发过日志，则发送一条NOP日志给备UPS，由于这条日志携带者主UPS已经提交日志号的信息，备UPS收到后就可以提交13-20号日志了。

由此可见，备UPS的临时日志号会始终大于已提交日志号。备UPS的提交日志号会始终小于等于主UPS的提交日志号。

o 节点故障

处理事务时，UPS会将已提交的事务以日志的形式持久化到commit_log中，而未提交事务写入temp_log。UPS重启时按照日志写入的顺序依次回放，即可恢复数据在宕机前的最新状态。节点故障恢复需要保证系统正确地恢复到故障前最新状态，而且不会出现数据丢失的严重问题。下面分别就三种情况进行说明。

a) 备UPS故障重启

系统正常运行状态下，一台备UPS发生故障后重启，此时该备UPS中的已提交日志号cmt_seq可能落后于主UPS，但是这些日志的内容一定是与主UPS保持一致的。故备UPS重启之后，首先会回放已提交日志号cmt_seq之前的日志，即commit_log。

临时日志temp_log中的日志不能确定是否与主UPS一致，所以该部分的日志是否能提交并不能确定，所以备UPS需要逐条检查自己的temp_log中的日志是否与主UPS相应日志号的日志一致，如果其中某个日志项Q与不一致，则Q之后（包括）的日志都是无效的，需要清除掉，Q之前的日志可以保留，至此temp_log的检查就结束了。备UPS此时可以接受主UPS发来的同步日志M，Q与M之间可能有间隙，备UPS向主UPS同步拉取日志的方式补全。

b) 主UPS故障重启

主UPS发生了故障宕机，此时RS检测到主UPS租约超时，会重新对UPS进行选主。选主算法见3.2节。

新主UPS选出之后要将temp_log日志提交，在提交这部分日志之前需要等待多数备UPS拉取到这部分日志。

故障的节点在重启后的身份为备UPS，它把自己的cmt_seq和tmp_seq汇报给RS，RS会返回主UPS的地址，之后的恢复过程与情况a)备UPS故障重启策略相同。

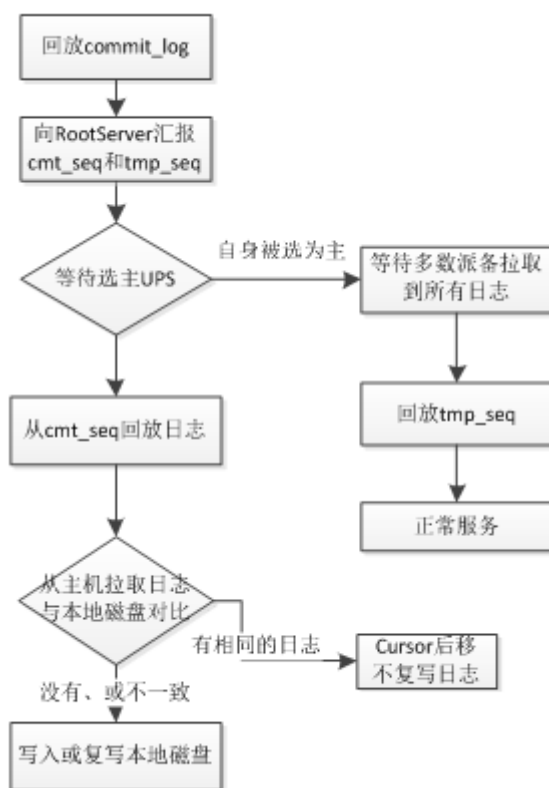


图3描述的是UPS宕机系统重启恢复的流程，主备UPS对应不同的恢复策略。主UPS先回放已提交日志commit_log，接着等待多数派备UPS拉取到临时日志temp_log，接着回放临时日志temp_log，开启服务。备UPS先回放已提交日志commit_log，接着从cmt_seq开始和主UPS的日志进行一一比对，若备UPS出现了与主UPS不同的日志，则从该日志开始复写，与主UPS保持一致。

4 日志强同步详细设计

日志同步在主UPS上入口与大致流程为：

Step1: bool TransExecutor::handle_write_trans_

工作线程执行写操作，将修改应用到内存memtable中后，调用TransCommitThread::push(&task) 将task push到commit 单线程的task_queue_中。等待commit单线程处理任务。

Step2: void TransExecutor::handle_commit(void *ptask, void *pdata)

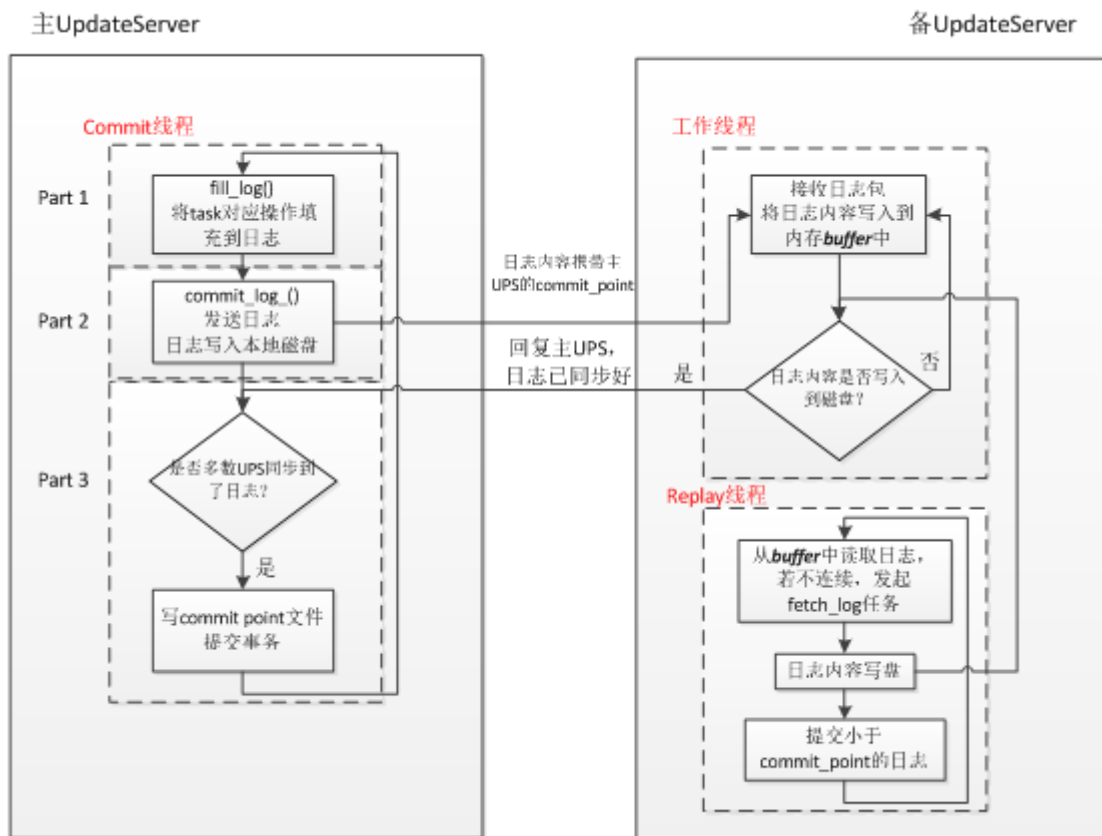
该函数是commit单线程处理任务的入口。commit单线程的功能是将task修改memtable的操作日志写到磁盘上，并将日志同步给备UPS，这两个操作完成后，主UPS将该task对应的事务提交，即将session结束，将应用到内存memtable中的修改对外可见。以上功能对应commit线程的三个函数。

(1) int TransExecutor::fill_session_log_，主要作用是将task对应的操作填充为日志的形式。

(2) int TransExecutor::commit_log_()，主要作用将日志刷盘并同步到备UPS上。

(3) int TransExecutor::handle_flushed_log_()，判断日志是否同步成功，成功后提交事务。

UPS日志强同步主要分为三个模块：主UPS日志发送模块、备UPS日志接收模块和备UPS日志回放模块。总体流程如图5所示。



接下来的流程涉及到三个方面：1) 主发日志；2) 备接收日志；3) 备回放日志。

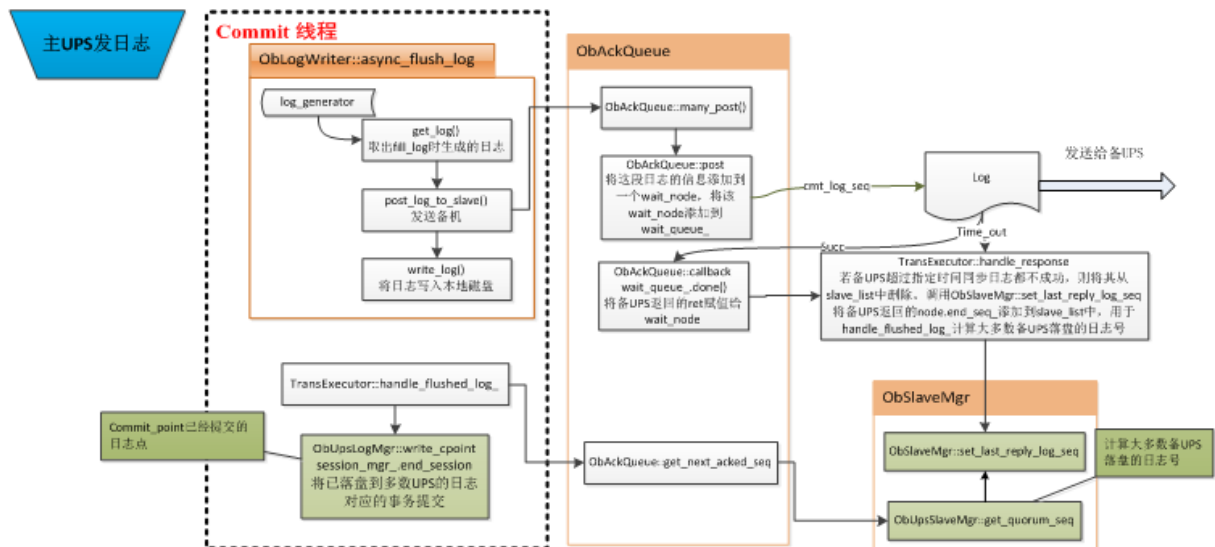
4.1 主UPS发日志

如图6所示。commit线程调用int ObLogWriter::async_flush_log()函数用于主UPS向所有的备UPS发送日志并将日志写入本地磁盘。

int ObLogWriter::async_flush_log()流程中主要包括三个函数：

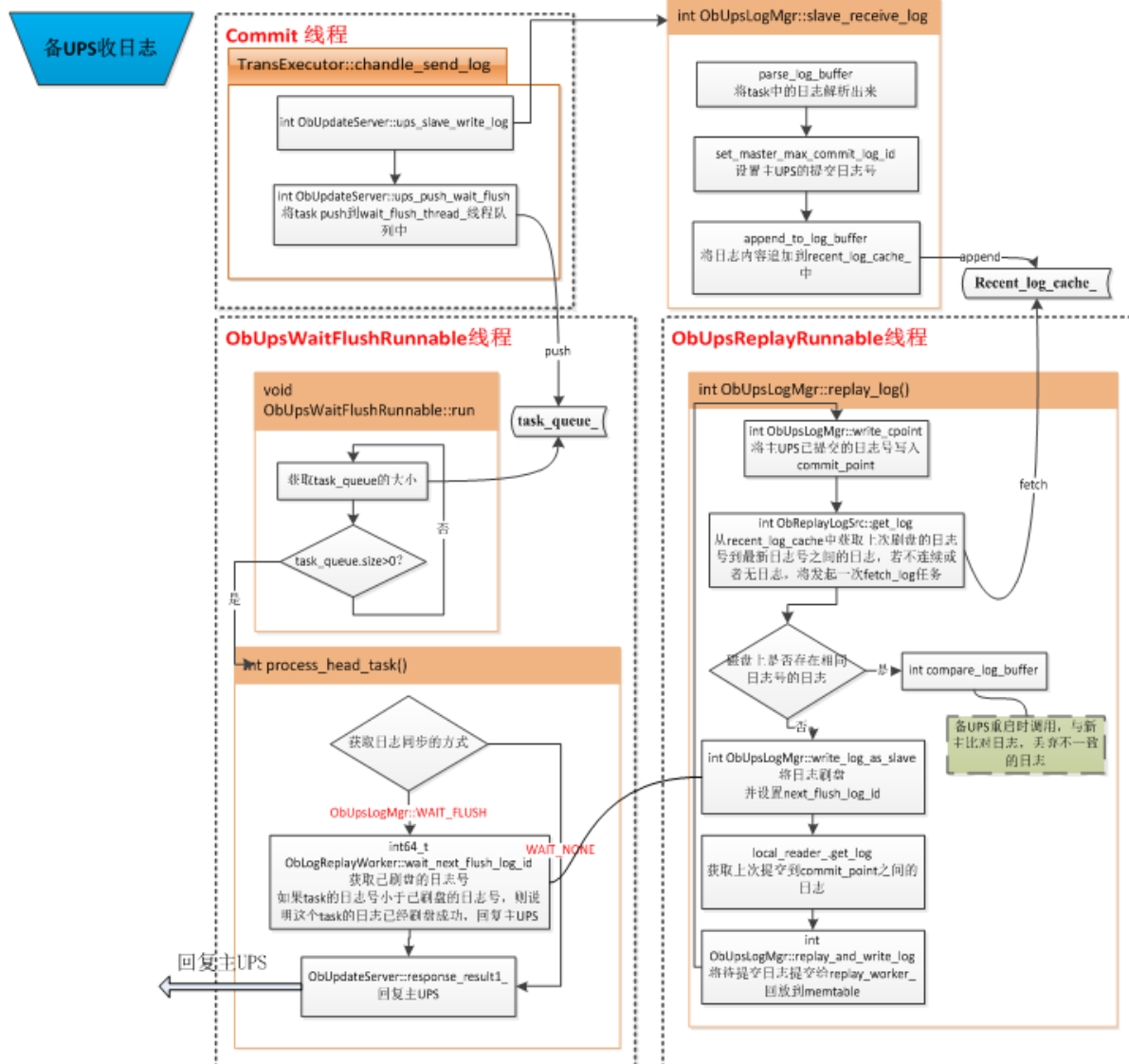
ObLogGenerator:: get_log(); int ObUpsSlaveMgr::post_log_to_slave(); int ObLogDataWriter::write()

首先向int ObLogGenerator:: get_log()取出要发送的日志（可能多条日志），接着调用int ObUpsSlaveMgr::post_log_to_slave()将携带主UPS提交日志号commit_point和日志包OB_SEND_LOG发送给备UPS，备UPS成功回复后调用回调函数中的int TransExecutor::handle_response()，记录每个备UPS写磁盘的日志号。在int TransExecutor::handle_flushed_log_()方法中调用ObAckQueue::get_next_acked_seq()计算出多数UPS回复的日志号作为可提交日志号，并将其写入commit_point文件作为commit_log日志号的标志，并提交可提交日志号之前的事务。



4.2 备接收日志

备节点接收到OB_SEND_LOG类型的包后，ObUpsLogMgr::slave_receive_log()处理接收到的日志，备UPS将日志放到recent_log_cache_中去。并将该任务push到ObUpsWaitFlushRunnable线程的处理队列中去。如图7所示，日志任务在该队列中等待刷盘后回复主UPS，这样做的目的是不阻塞commit线程处理任务。



4.3 备处理日志

UPS启动时会开启ObUpsReplayRunnable线程，循环调用replay_log()这个函数。该函数的作用为检测到接收到新日志时将日志写盘并回放。

这个函数包括两段。第一段是从远程和recent_log_cache_获取从上次刷盘日志号到最新日志号之间的日志，写入磁盘；第二段是从本地缓冲区或磁盘读取上次回放的日志号到主UPS的commit_point之间的日志由ReplyWorker多线程回放日志。写盘成功后设置next_flush_log_id，此时ObUpsWaitFlushRunnable中的任务获取到next_flush_log_id，如果task的日志号小于next_flush_log_id，则备UPS将该任务回复给主UPS。

4.4 主备UPS切换

主要涉及三个函数：

```

int ObUpdateServer::start_service()
int ObUpdateServer::master_switch_to_slave()
int ObUpdateServer::switch_to_master_master()
  
```

我们增加了submit_replay_tmp_log()方法用于提交一次回放temp_log日志任务。增加replay_tmp_log()方法用于回放临时日志。

启动一个UPS时，首先调用`int ObUpdateServer::start_service()`，在该函数中，首先提交一次回放`commit_log`的任务，回放提交日志。接着获取本地的最大日志号向主RS注册。该函数中还存在一个`while`循环，用于不停地检测UPS的状态，如果UPS的身份是MASTER的话，并且状态是`replaying_log`，代表该UPS被选为了新主。该UPS首先要等待多数备UPS已拉取到了`temp_log`，确保`temp_log`在多数UPS上落盘后提交一个回放`temp_log`日志任务。日志回放完成后将该UPS的状态切换为ACTIVE。

UPS身份的切换涉及到两个流程：主切备和备切主流程。

在UPS更新租约`int ObUpdateServer::ups_update_lease()`时，如果检测到RS已经选出了新主UPS，则调用`ObUpdateServer::switch_to_master_master()`方法，该方法中，首先需要等到`replay`线程停止，接着将UPS的身份修改为MASTER，将状态切换为`replaying_log`，并添加设置`term`值为切换为当前时间戳。从而使得`int ObUpdateServer::start_service()`中可以回放主UPS的临时日志。

同样在`int ObUpdateServer::ups_update_lease()`中，如果主UPS发生了切换，旧主UPS检测到时需要将自己切为备UPS，执行`int ObUpdateServer::master_switch_to_slave()`。在该函数中，需要首先将UPS的写线程暂停，接着等到`session`都结束，接着将`log_cursor`移动到`commit_log`指向的点，后面的日志都作为临时日志处理，需要与新主UPS进行比对。接着将`replay`线程开始。完成主UPS切换到备UPS。

RS选举设

1 目前主备RS架构设计

- 架构描述

OceanBase0.4 版本中支持一主一备双RS架构。主备RS之间通过操作日志实现热备。

正常情况下，主RS管理整个集群的工作，而备RS仅以最大可用模式同步主RS发送来的操作日志。主备RS通过Linux HA (High Available) 机制保证RS的高可用，OceanBase使用Corosync软件提供完整的HA功能，它可以将若干节点统一管理，控制虚拟ip (vip) 漂移。主备RS共享一个vip，vip所在的机器为主RS。当主RS宕机不可用，甚至服务器毁坏时，vip漂移到备RS所在服务器，备RS检测到vip后，将自动切换为主，接管集群，从而保证了集群内总有RS来提供服务。如果主RS检测到自己的vip漂移了，则将自动下线。

- 存在问题

生产环境中，出现过主备RS切换失败，最终集群中RS全部宕机的情况。原因如下：Corosync通过调用`ob_ping`不断向主RS发送信息，如果`ob_ping`超时，则认为主RS发生故障，然后Corosync会kill主RS，并将vip漂移到备RS所在服务器。这里存在一个问题是，即使主RS没有故障，但只要`ob_ping`超时，就会引起vip的漂移，并且正常运行主RS会被停止。如果此时又发生vip漂移失败，最后就会出现主RS停机，同时备RS未检测到vip也停机这种情况。归根结底，第三发工具承担集群重要节点的主备切换是存在风险和隐患的。

2 基于Paxos的RS选举设计

为了消除第三方工具在控制RS切换时存在的安全隐患，提高集群可用性，同时简化集群部署步骤，我们根据Raft一致性协议重新设计了RS的主备架构及切换流程，保证主RS宕机时，只要多数个RS存在，就可以自动选择唯一——一个新的主RS接管集群。

2.1 Raft一致性协议介绍

Server States Raft算法将服务器划分为3种角色：

1. Leader

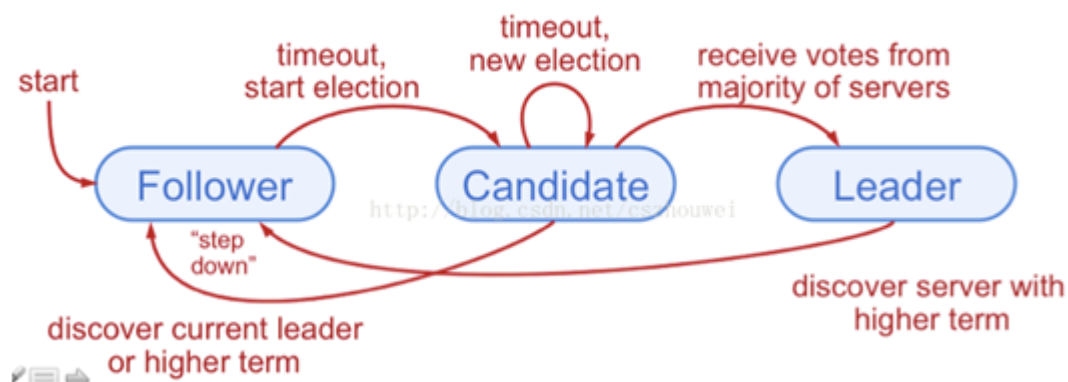
功能：负责管理集群，同一时刻系统中最多存在1个Leader

2. Follower

功能：被动响应请求RPC，从不主动发起请求RPC

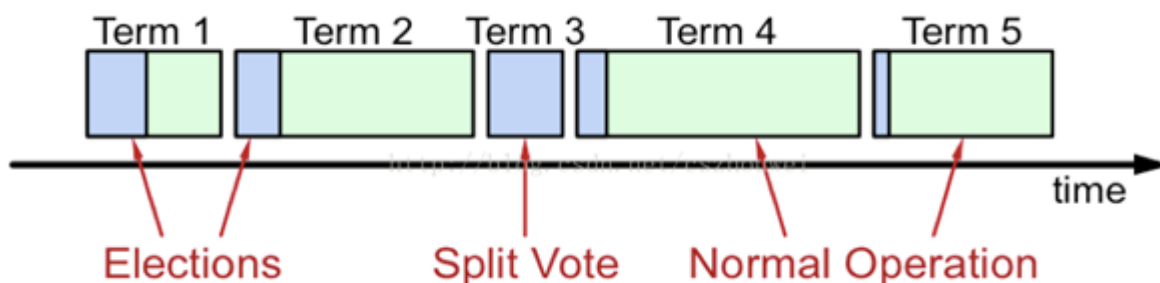
3. Candidate

功能：由Follower向Leader转换的中间状态



众所周知，在分布式环境中，“时间同步”本身是一个很大的难题，但是为了识别“过期信息”，时间信息又是必不可少的。Raft为了解决这个问题，将时间切分为一个个的Term，可以认为是一种“逻辑时间”。如下图所示：

1. 每个Term至多存在1个Leader
2. 某些Term由于选举失败，不存在Leader
3. 每个Server在本地维护currentTerm



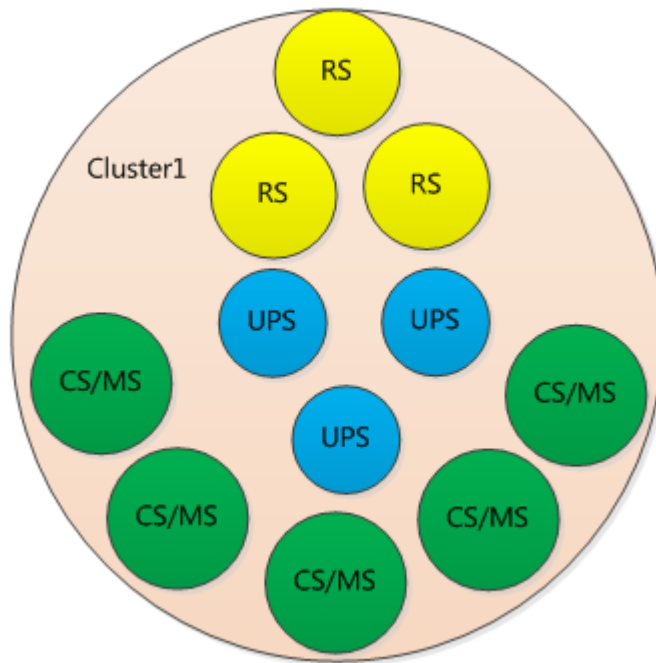
1. 所有的Server均以Follower角色启动，并启动选举定时器
2. Follower期望从Leader或者Candidate接收RPC
3. Leader必须广播Heartbeat重置Follower的选举定时器
4. 如果Follower选举定时器超时，则假定Leader已经crash，发起选举

细节补充：

1. Candidate在等待投票结果的过程中，可能会接收到来自其它Leader的RPC。如果该Leader的Term不小于本地的currentTerm，则认可该Leader身份的合法性，主动降级为Follower；反之，则维持Candidate身份，继续等待投票结果
2. Candidate既没有选举成功，也没有收到其它Leader的RPC，这种情况一般出现在多个节点同时发起选举（如图Split Vote），最终每个Candidate都将超时。为了减少冲突，这里采取“随机退让”策略，每个Candidate重启选举定时器（随机值），大大降低了冲突概率

2.2 RS选举功能设计

修改RS的HA架构，不再通过Corosync控制主备切换的时机，同时取消VIP。实现单集群内，基于Raft协议的RS选举机制，架构图如下：



在一个集群内部署三台RS、三台UPS以及多台CS/MS。其中UPS间实现基于Paxos的主备日志同步，这部分内容在《UPS日志同步》中介绍。

进行功能设计时，我们在Raft协议基础上进行了修改，选举的逻辑和流程较之原始Raft有所区别。整个功能可以分解为如下几个部分：

2.1.1 RS选举

集群正常运行过程中，如果主RS宕机，且集群中仍有多数个RS存在（例如3个RS中有2个正常，5个RS中有3个正常），则能够在一定时间内选举出新的主RS来接管集群。而如果集群中没有多数个RS存在，则无法选出主RS，系统处于异常无主状态。

在选举过程中不存在主RS，选举结束后，整个集群的状态需要与选举开始前保持一致。各个RS都会记录各自的选举开始和结束时间，并计算选举耗时，不同RS统计出的选举耗时可能不同。用户查看选举时间时可以以Leader上统计的时间为准。

RS能在日志中显示当前的选举状态。RS_ELECTION_DONING表示正在进行选举，RS_ELECTION_DONE表示选举已经结束。

- RS将在以下情况发生时，启动选举流程，记录选举开始时间：
 1. 备RS检测到自身租约过期。
- RS将在以下情况发生时，结束选举流程，记录选举结束时间：
 1. RS收到多数投票，并成功广播多数RS。
 2. RS收到Leader的心跳，或者广播。

2.1.2 RS启动

RS启动时通过启动命令中的-R参数确定主RS，被指定的主RS将以master角色启动，备RS以slave角色启动，并向主RS注册。当主RS检测到集群中所有RS已经启动并成功向自己注册时，就开始接管集群，此时所有RS形成一个Paxos组，开始正常工作。

需要指出的是：部署集群时，必须先启动主RS，再启动备RS。启动主RS时，主RS需要所有RS注册后才能正常开始自动流程；如果启动备RS时向指定的Leader注册失败，备RS将退出进程，启动失败；如果注册成功，备RS会在等待Leader正常后开始正常启动。

首次启动后需要进行bootstrap，原实现中，主RS会通过日志将bootstrap信息发送给备RS，备RS回放该日志生成first_tablet_meta文件。但是由于我们取消了RS之间的同步日志，使得bootstrap后备机上没有生成first_tablet_meta文件。因此，我们在bootstrap时，令主RS向备RS直接发送相关信息，并当所有已注册的备机成功生成first_tablet_meta文件后才确认bootstrap成功。

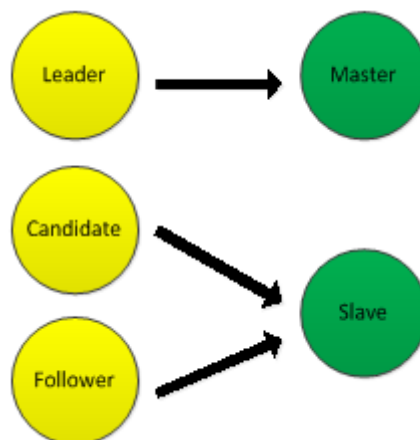
如果用户需要部署多台RS，可以通过启动参数-U设置需要部署的RS个数，这些RS构成一个Paxos组；用户可以通过启动参数-u设置集群部署的UPS个数。允许用户在集群运行过程中修改这两个参数。

无论RS是否是首次启动，都需要用户通过长命令方式启动，-r参数指向当前集群的Leader地址。

2.1.3 RS主备切换

目前版本中，RS通过检查VIP是否在本机上来决定是否进行切换。如果是主RS，当发现VIP不在本机时，那么主RS会停机，进程退出；如果是备机，当发现VIP处于本机时，那么备RS会执行slave -> master的切换。

在基于Raft的RS选举设计中，我们不再通过检测VIP来决定切换与否，而是通过选举角色的转变来确定。我们使用leader/candidate/follower作为节点角色（逻辑角色），与ob原有的master/slave角色（物理角色）对应关系如下：

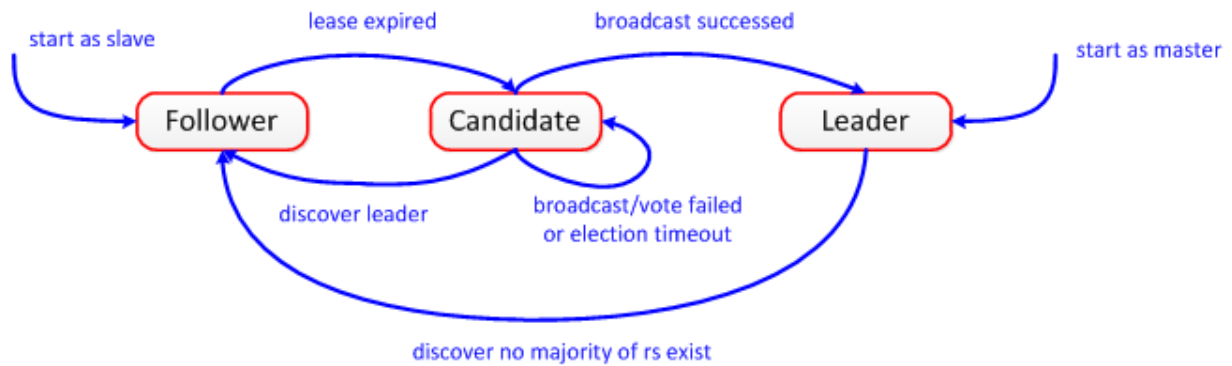


在选举过程中，如果发生选举角色的转变，那么主备角色也改变，并执行对应的切换流程。目前ob中已经实现了Slave -> Master 切换，我们需要增加Master -> Slave的切换功能，并且不再令主RS检测到角色发生改变时就退出。

RS逻辑角色的基本切换条件为：

1. Leader发现包括自己在内的RS总数不足多数。 Leader -> Follower
2. Follower发现租约过期。 Follower -> Candidate
3. Candidate发现广播已经得到多数派回应。 Candidate -> Leader
4. Candidate发现Leader已被选出或者收到更高的term。 Candidate -> Follower

转化图如下：



RS物理角色的切换条件为：

1. 如果Leader -> Follower，则Master -> Slave
2. 如果Candidate -> Leader，则Slave -> Master

2.1.4 RS心跳机制

RS增加心跳机制，用来维持各个RS之间的联系。主RS通过发送心跳包，更新其它RS的租约，并统计与自己保持通信的备RS个数。而目前版本中的租约检查机制被取消。

2.1.5 MS/CS/UPS的注册和RS信息维护

由于取消了VIP，MS/CS/UPS中维护的主RS地址将不会一成不变。它们需要在内存中维护所有RS的信息列表，并且当主RS发生改变时，能够遍历RS列表尝试向RS注册，直至成功。

2.1.6 __all_server中维护各个RS的信息

目前all_server表中只维护了RS的vip信息。我们需要将所有RS信息记录在该表中，并标明其主备角色（svr_role字段），使用1表示master，2表示slave，0表示异常。当RS角色发生改变时，系统能够实时更新all_server表。

2.1.7 修改系统RS个数配置

系统的RS个数（名称是rs_paxos_num）是初次部署时确定的，RS切换过程中根据该值计算多数派。因此，用户不能随意修改该参数，以避免切换时多数派计算错误而最终导致双主等问题。

增加rs_admin命令，提供安全的查看和修改方式。如果是增大RS个数配置，其值不能超过以当前配置为多数派的值；如果是减小RS个数配置，其值不能超过当前RS个数配置的多数派值。例如，若当前RS个数配置为5，那么用户设置的新值范围为(2, 10)；若当前最大Paxos个数为3，那么用户设置的新值范围为(1,6)。

由于任何配置值的多数派值都不是1，因此当需要将RS个数配置修改为1，或者从1修改为其他值时，应特殊处理。只支持配置值1和2之间的修改。

当从2减小为1时，先设置rs_paxos_num为1，主RS之外的其他RS将自动下线。

当从1增大为2时，先增加机器，再设置rs_paxos_num为2。

2.1.8 设置选举优先级

支持用户设置RS的选举优先级，控制选举中，指定RS当选Leader的概率。选举优先级取值范围是[1,5]，其中1表示最高优先级，5表示最低优先级，RS的默认优先级为1。

用户通过sql语句可以修改RS的选举优先级：

```
alter system set rs_election_priority=<priority_value> server_type=rootserver server_ip=, server_port=;
```

2.1.9 强制设置Leader

用户可以通过rs_admin命令指定某台RS为Leader，原主RS自动切为备。

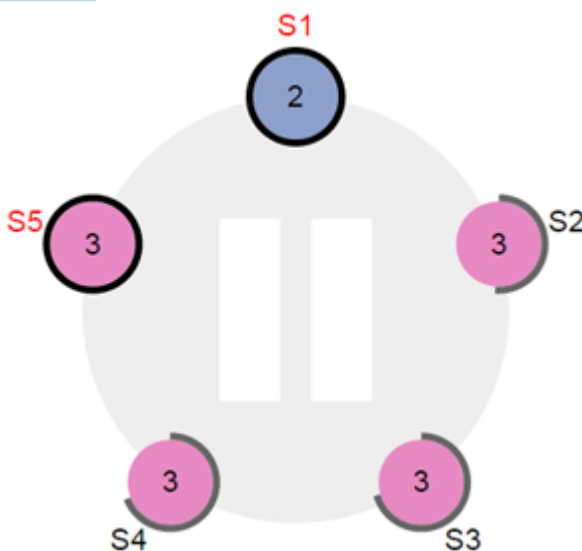
```
rs_admin -r -p set_rs_leader -o rs_ip=<rs_ip>,rs_port=<rs_port>
```

2.2 RS选举详细设计

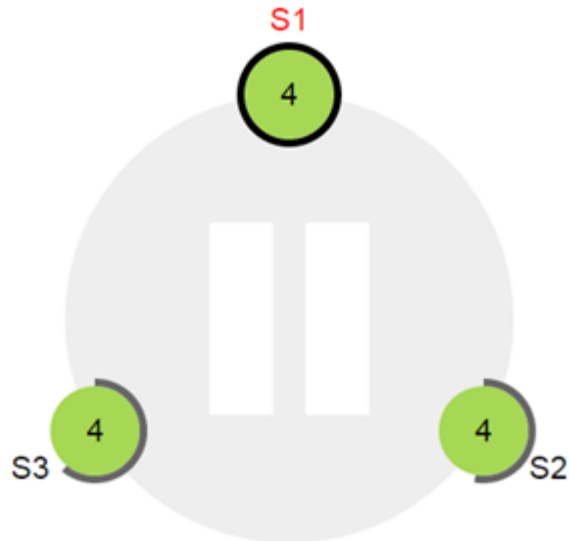
2.2.1 RS选举

我们参考了Raft选举机制实现RS间的自动选举。在Raft中，网络故障可能会导致脑裂和频繁选举。

脑裂，是指由于网络等异常情况，集群错误认为原主节点已经失效，而让另一个节点作为主节点管理集群，此时原主节点又恢复了与集群其他节点的通信，最终导致集群内存在两个主节点，使得集群发生混乱。如图所示，S1与其它节点存在网络故障，S2-S5四台机器中选举出S5作为新的主节点。此时S1和S5均是主节点，都可能接收到客户请求。但是由于节点在处理请求时需要满足多数派才能提交事务，因此发往S1的请求会失败，这样不会对整个集群的正确性造成影响。可见脑裂对于Raft来说并没有太大影响。但是在Oceanbase中，RootServer是集群的管理节点，同一时刻有且仅能有一个RS对外服务，不允许出现脑裂，我们在设计选举方案时必须予以避免。



频繁选举，是指集群中经常有节点发起选举，并且每次都能选出新的主节点，并且伴随着一段时间的脑裂现象，这种情况通常也是由网络异常引起的。如图所示，S1和S2之间网络断开，目前S1是主节点。S2一直不能接收到S1的心跳包来更新自己的租约，一段时间之后S2发现租约过期，则升高term号并发起选举。由于S2和S3之间网络正常，最终S2会成功变为主节点，此时S1也是主节点，出现脑裂。等S1变为普通节点且租约过期时，S1也会再次发起选举，最终也能够变为主节点。如果S1和S2间的网络一直无法恢复，那么频繁选举、切主的过程将会一直持续。Raft对于这种情况并没有做处理，因为这对集群的状态没有太大影响，也不会发生数据错误。但是对于Oceanbase来说，每次切换RS都会导致一段时间内客户端请求失败，RS频繁选举将严重影响数据库性能，无法满足数据库高可用的需求。



为了避免上述两种情况，我们在设计时对Raft进行了修改，在出现网络分区时，保证在Follower发起选举之前，原Leader就会放弃主的身份，这避免了脑裂；另外，我们令RS在认为集群内存在Leader时，拒绝任何投票相关的信息，这避免了频繁切主的问题。

2.2.1.1 选举模块设计

- 增加ObRootElectionNode结构体，保存RS作为一个选举节点的所有状态信息。

```
struct ObRootElectionNode
{
    enum ObRsNodeRole  // RS的逻辑角色
    {
        OB_INIT = -1,
        OB_FOLLOWER = 0,
        OB_CANDIDATE = 1,
        OB_LEADER = 2,
    };
    common::ObServer my_addr_; // rs addr (ip, port)
    common::ObServer vote_for_; // the rs addr that I voted for
    ObRsNodeRole rs_election_role_;
    bool is_alive_; // rs is alive or not
    bool has_leader_; // leader exist or not
    int64_t lease_;
    int64_t current_rs_term_; // election term
    int64_t hb_resp_timestamp_; // for leader, value is -1; for others, value is timestamp
    int32_t vote_result_; // -1 unknown, 0 vote for, 1 vote against
    int32_t bd_result_; // -1 unknown, 0 recv broadcast, refuse broadcast
    int32_t paxos_num_;
    int32_t quorum_scale_;
}
```

- 增加ObRootElectionNodeMgr类，用于实现选举模块，负责处理RS选举相关的状态维护和流程控制，是选举设计的核心部分。下面的类中仅包含了成员变量和部分重要的成员函数。

```
class ObRootElectionNodeMgr
{
```

```

public:
    int grant_lease();//Leader向其他RS授租
    int refresh_inner_table(...); //刷新内部表
    int slave_init_first_meta(const ObTabletMetaTableRow &first_meta_row); //Leader向其它RS发送first_meta信息，用以生成first_meta文件
    int32_t add_rs(const common::ObServer &rs); //在other_node_array中增加RS
    int32_t get_real_rs_count();//获取当前系统中实际的RS个数
    int leader_broadcast(bool is_first_start = false); //发送广播信息
    int request_votes(); //发送投票信息
    int excute_election_vote(...); //进入投票阶段
    int excute_election_broadcast(...); //进入广播阶段
    int check_lease(); //检查RS状态、租约等并做相应处理，不同角色的RS会执行不同的检查和处理逻辑。
    bool check_most_rs_alive();//Leader用来检查集群中是否有多个RS在线
    int get_random_time(); //获取随机时间
int handle_rs_heartbeat(...); //处理心跳包
int handle_rs_heartbeat_resp(...); //处理心跳回复包
    int handle_leader_broadcast(...); //处理广播包
int handle_leader_broadcast_resp(...); //处理广播回复包
    int handle_vote_request(...); //处理投票请求包
    int handle_vote_request_resp(...); //处理投票请求回复包
    bool need_leader_change_role(const int64_t now); //Leader用以判断是否需要放弃主
    void start_election();//设置选举开始时间和选举状态
    void end_election();//设置选举结束时间和选举状态
    ... ..
private:
    static const int64_t MAX_CLOCK_SKEW_US = 200000LL; // 200ms
    // data members
    ObRootAsyncTaskQueue * queue_;
    ObRootServer2 *root_server_;
    ObRoleMgr *role_mgr_;
    ObRootRpcStub &rpc_stub_;
    ObRootElectionNode other_nodes[OB_MAX_RS_COUNT - 1]; //store other rs info
    ObRootElectionNode my_node_; //store local rootserver info
    ObRsElectStat election_state_; //选举状态
    const int64_t &revoke_rpc_timeout_us_;
    int64_t no_hb_response_duration_us_; //无心跳回复的最长允许时间
    int64_t lease_duration_us_; //备RS租约时间
    int64_t leader_lease_duration_us_; //主RS租约时间
    int32_t vote_count_; // agreeable votes count
    int64_t rs_hb_timestamp_array[OB_MAX_RS_COUNT - 1]; // only for sort, only include ts
of rs in other node array
    int64_t begin_time_;// 选举开始时间
    int64_t end_time_; // 选举结束时间
    const int64_t &rs_paxos_number_;
    ... ..
};

```

2.2.1.2 选举流程设计

RS启动时会开启ObRsCheckRunnable *rs_election_checker_线程，循环调用check_lease()线程检查RS状态或租约，时间间隔为50ms

- 若RS逻辑角色为Follower

调用check_lease()时，发现租约过期，则开始发起选举流程。首先RS设置选举开始时间、选举状态以及相关节点信息，然后将自己的角色置为Candidate，之后会进入下一轮check_lease()，再根据RS最新的逻辑角色进行处理。

- 若RS逻辑角色为Candidate

如果check_lease时发现RS的逻辑角色为Candidate，此时RS会执行选举流程。选举包括两个阶段，投票阶段和广播阶段。投票阶段中，RS会向其它在线的RS发送同步投票请求，请求的内容为“选自己为Leader”，并收集投票结果。如果得到多数RS同意（包括自己，自己肯定会同意），则认为投票成功，进入广播阶段，否则认为本轮选举失败，进入下一轮check_lease()；广播阶段中，RS会向其它在线的RS发送同步广播，内容为“已经收到多数投票，将切换为Leader”，并收集广播的确认结果。如果得到多数RS确认（包括自己，自己肯定会确认），则认为广播成功，此时RS将逻辑角色置为Leader，选举结束。

RS根据vote_request_doing和broadcast_doing判断是处于投票阶段还是广播阶段。如果正在投票阶段，首先判断投票是否成功，若成功，则发起广播，否则发起下一轮投票；如果正在广播阶段，首先判断广播是否成功，如果成功，则将角色设置为Leader，否则重新发起投票。每次发起投票都将增加term。

发起投票后，RS会设置下一次发起投票的时间点，在这个时间点之前不会再次发起投票，为了使得不同RS发送投票的时间错开，减少选举冲突，计算时间点时除了基本时间election_timeout_us_（200ms），还需要加上一个随机时间（选举优先级越高，随机时间的取值区间越高），计算方式为

```
election_vote_lease_us_ = now + election_timeout_us_ + random_time
```

发起广播后，RS会设置下一次发起广播的时间点，在这个时间点之前不会再次发起广播，计算这个时间不需要加随机时间，计算方式为

```
election_broadcast_lease_us_ = now + election_timeout_us_
```

RS发送投票请求和广播的过程为异步。rs_election_checker线程只负责检查当前RS的角色，并依据角色发出不同的执行命令。rs_election_checker线程会将命令包放入单线程队列ob_root_election_queue中，由负责该队列的线程执行发送投票请求包的动作。我们将选举相关的所有包，不论是接收的还是发送的，都放入ob_root_election_queue中处理，这样选举中对于相关变量的修改就是由单线程进行，避免了大多数的加锁操作。

对于接收方RS来说，需要根据该RS的逻辑角色(Leader, Candidate or Follower)决定处理逻辑。同时，接收者会根据has_leader的值来作为处理包的判断条件之一，如果has_leader为true，则不能接受投票请求和广播，通过这种方式避免频繁选主。选举开始时，RS会将has_leader_置为false；选举结束时置为true。

选举相关包主要是下面六种：

```
OB_RS_RS_HEARTBEAT = 608, //心跳包
OB_RS_RS_HEARTBEAT_RESPONSE = 609, //心跳回复包
OB_RS_VOTE_REQUEST = 610, //投票请求包
OB_RS_VOTE_REQUEST_RESPONSE = 611, //投票请求回复包
OB_RS_LEADER_BROADCAST = 612, //广播包
OB_RS_LEADER_BROADCAST_RESPONSE = 612, //广播回复包
```

RS对于投票和广播相关包的处理逻辑：

1. 处理投票请求包

- 角色为Leader

拒绝同意投票请求。

- 角色为Candidate

如果has_leader_ == true，说明RS处于广播阶段，所以拒绝投票请求。

如果`has_leader == false`，且本节点`term`小于接收到的`term`，则同意投票请求。并且将角色变为Follower。

其它情况拒绝投票请求。

- 角色为Follower

如果`has_leader_ == true`，说明集群中存在Leader，所以拒绝投票请求。

如果`has_leader == false`，且本节点`term`小于接收到的`term`或`vote_for`无效，则同意投票请求。

其它情况拒绝投票请求。

1. 处理投票请求回复包

只有Candidate处于投票阶段时才处理该回复包。RS接收到包后，将`other_nodes`数组中相应的RS信息更新，标明是否为赞同票。然后遍历`other_nodes`数组中所有RS信息，统计与自己`term`相等的RS的赞同票，并将结果存入`ObRsVoteResult vote_result_`数据结构中。

如果是反对票，且投票回复中`term`大于自身`term`，则将自己的角色切为Follower

1. 处理广播包

- 角色为Leader

报错，并将进程退出。

- 角色为Candidate

直接确认广播包发送者为Leader，并根据包内容更新本节点信息。

- 角色为Follower

如果`has_leader == false`，直接确认广播包发送者为Leader，并根据更新本节点信息。

如果`has_leader == true`，且之前没有收到同一个RS的广播包，拒绝确认广播发送者为Leader。

1. 处理广播回复包

只有Candidate处于广播阶段时才处理该回复包。RS接收到包后，将`other_nodes`数组中相应的RS信息更新，标明是否为赞同票。然后遍历`other_nodes`数组中所有RS信息，统计与自己`term`相等的RS的赞同票，并将结果存入`ObRsBroadcastResult broadcast_result_`数据结构中。

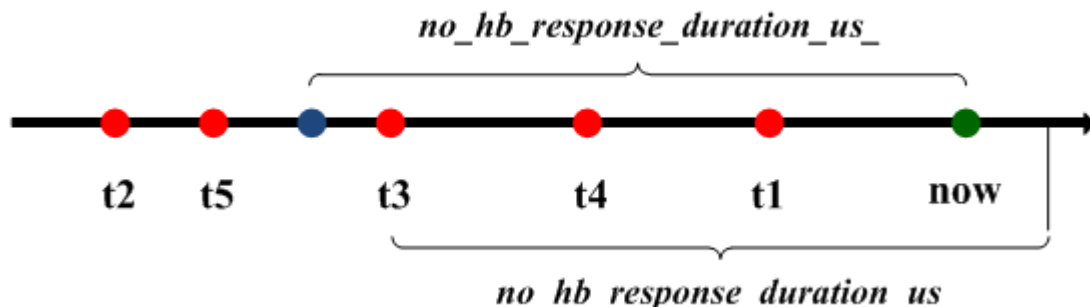
- 若RS逻辑角色为Leader

为了避免脑裂现象，我们通过设计机制保证在选举发生之前，集群中不存在Leader。考虑到只有当Follower租约过期时才会发起自动选举，因此只要令Leader在Follower的租约过期前退出即可。一般来说，只要Leader与集群内大多数RS（包括自己）都保持通信，那么即使有某个Follower发起选举，由于选举无法得到多数派同意，将不会成功，因此不会出现脑裂。所以，我们只要判断Leader在最近一段时间内（这段时间即为`no_hb_response_duration_us_`，要小于Follower租约时间，我们设计为租约时间的一半）是否与多数RS保持通信即可。

为此，我们利用RS间的心跳机制来解决这个问题。选定Leader后，Leader每隔100ms向其它RS异步发送心跳，由其它线程接收并处理心跳回复包。Leader将记录每个心跳回复包的接收时间戳，保存在各个RS的节点信息结构中。Leader每次进行`check_lease`时，会将这一时刻的各个RS心跳回复时间戳（以Leader接收的时间为准，且Leader的时间戳默认是最大的）升序排序，并取中间值`ts`。如果`ts + no_hb_response_duration_us_ > now`，则说明Leader与多数保持通信，此时Leader重置租约，反之，Leader不做任何动作。当Leader发现租约过期后，将切换为Follower，并置`has_leader_`为`false`，同时切换物理角色`master -> slave`。

如果集群中只配置唯一——一个RS作为Leader，那么不会执行上述判断，Leader将一直保持角色不变。

如下图所示，红点t1~t5表示Leader接收到心跳回复包的时间戳，已经按从小到大顺序排好，其中t1为Leader的时间戳（t1总是被认为是最大的），绿点表示当前系统时间，蓝点表示距离当前时间前no_hb_response_duration_us的时间点。可以看出，只要位于中间的时间点t3加上no_hb_response_duration_us大于now，那么就可以确认RS在最近no_hb_response_duration_us的时间内收到多数个心跳回复包，即能够与多数RS保持通讯。



为了能够让用户修改no_hb_response_duration_us参数，我们将其添加进Rootserver的系统参数中，可以通过alter system param语句修改。

2.2.2 RS启动

RS启动时需要用户指定主RS，不支持通过自动选举方式选主。

原OB的主备RS启动时，会将-r参数设置的vip地址与本机地址（如果vip在RS上，那么该RS的本机地址就是vip地址，否则本机地址是节点物理地址）进行比较，如果一致，则设置为Master并调用start_as_master()启动RS；如果不一致，则设置为Slave并调用Slave并调用start_as_slave()启动RS。

在我们的设计中，取消vip的功能。RS启动时可以确定是作为Master还是Slave节点，启动命令中，-r参数设置为RS自己的ip，-R参数要设置为主RS的地址ip。如果-r和-R参数设置的值一致，说明该RS是用户指定的主节点；反之，该RS是备节点，并向-R指定的主RS节点注册。我们增加了两个函数start_as_master_leader()和start_as_master_slave()，分别替代原来的start_as_master()和start_as_slave()函数执行RS启动流程。

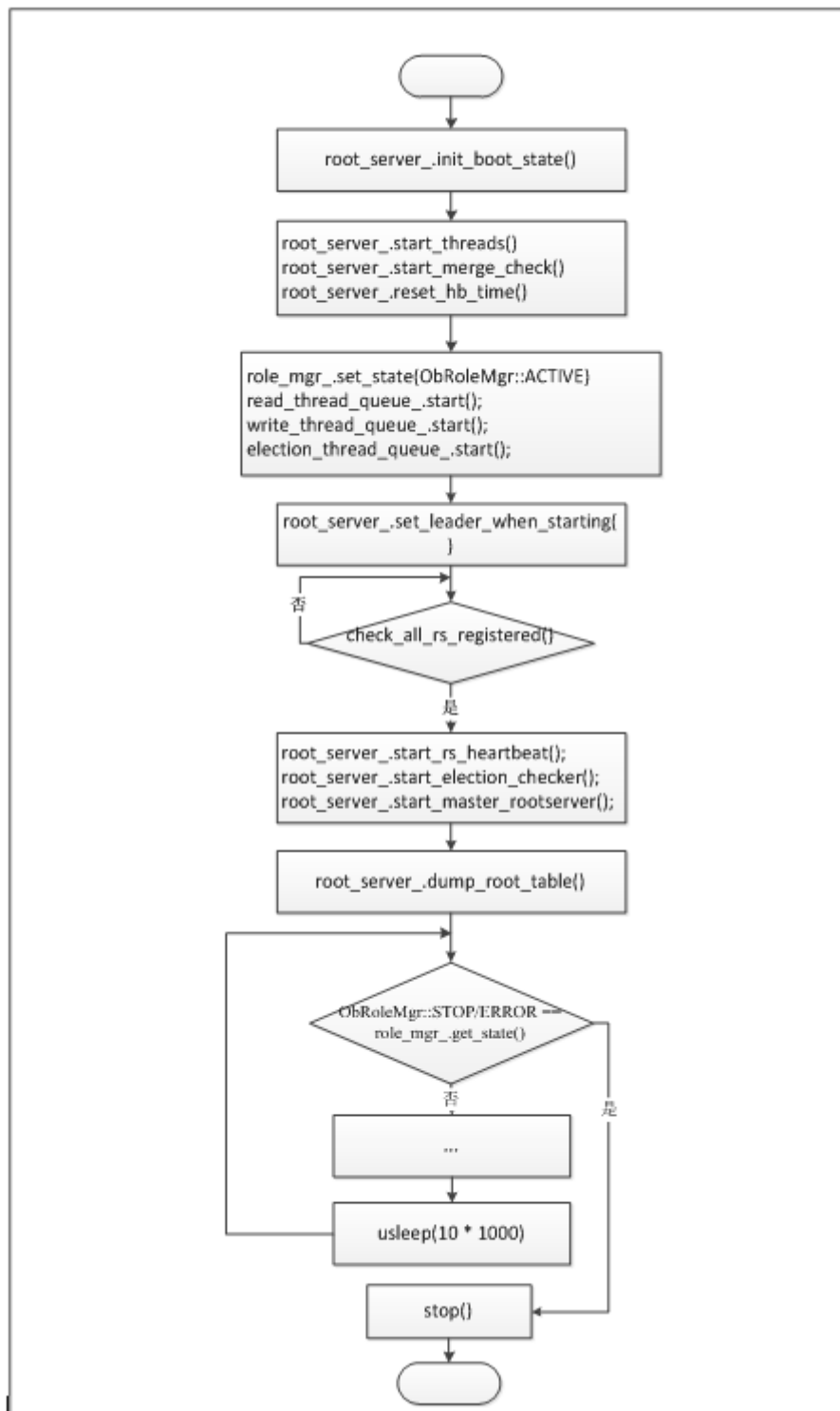
- RS启动流程

1. RS作为Master启动

为了实现启动RS时指定Master的功能，我们令主RS启动时等待所有RS成功注册，再执行后续启动流程。如果不这样设计，而是允许Leader启动时直接执行启动Master的过程，那么很可能导致Leader在检查自己是否与多数RS保持通信时失败，最终发起选主，这样无法完成用户指定Leadr启动的功能。为了方便修改，我们增加start_as_leader_master()函数，它包含了start_as_master()去除日志回放部分的全部代码，并增加了等待多数RS注册的功能。

```
while(err == OB_SUCCESS){
    if(!root_server.check_all_rs_registered()){
        usleep(10 * 1000);
    }
    else{
```

为了能在正式启动前处理网络包，我们会先启动election_thread_queue和read_thread_queue、write_thread_queue，前者是我们新增的选举队列，用于接收并处理选举相关包，后者是原有的读写队列。最后，启动流程中取消了replay_log相关代码。启动流程如下：



1. RS作为Slave启动

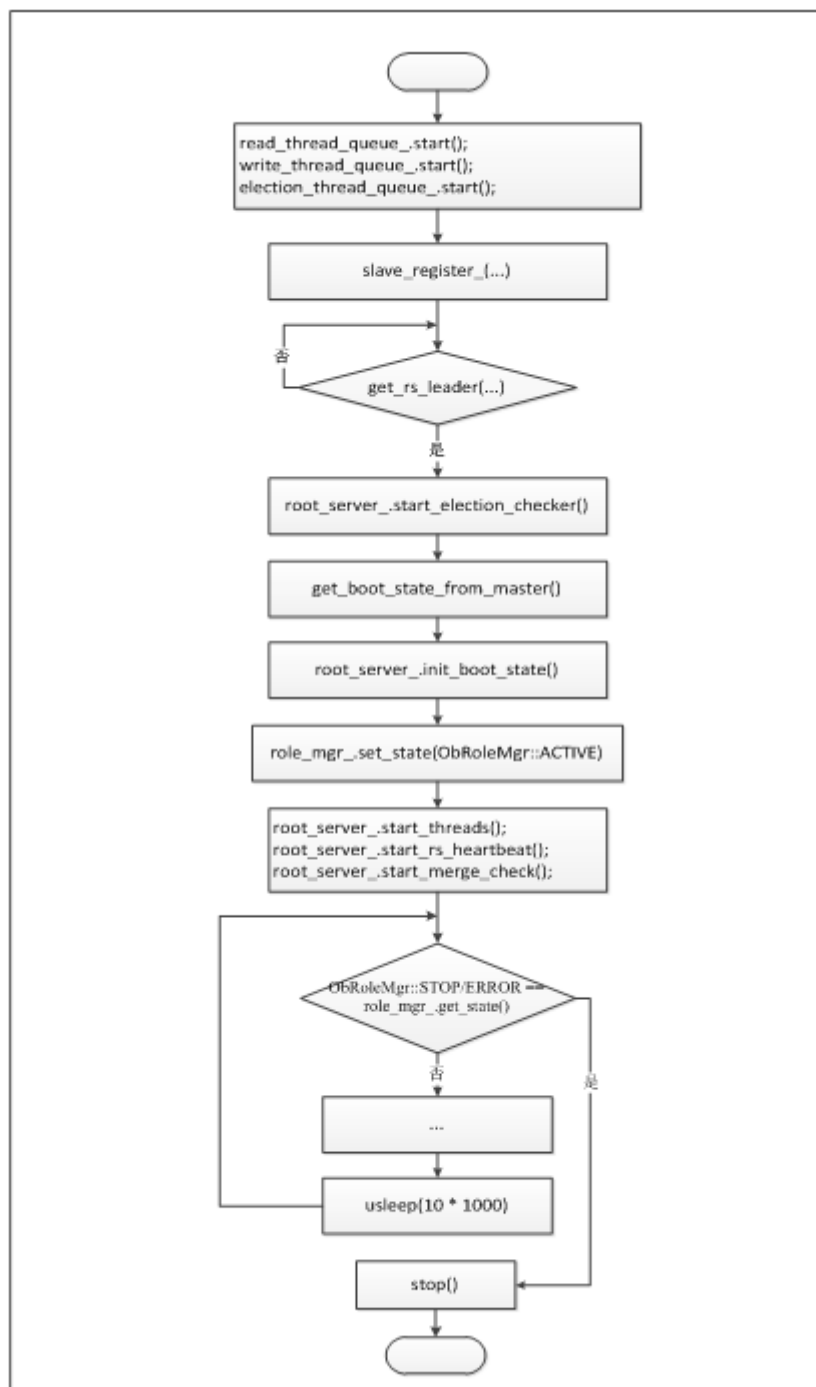
增加start_as_slave_follower()函数，它包括除了日志相关处理的start_as_slave()全部代码，并且增加了检测是否向主RS注册成功r的功能，只有确认注册成功，才能继续执行正常启动备RS的流程。另外，同启动主RS时类似，会首先启动election_thread_queue 和read_thread_queue、write_thread_queue。

```

Observer rs_leader;
ret = root_server_.get_rs_leader(rs_leader);
while (true) {
    if (rs_leader == rt_master_) {
        break;
    }
    ret = root_server_.get_rs_leader(rs_leader);
    usleep(10 * 1000);
}

```

启动流程如下：



1. Bootstrap

首次启动集群时，需要执行bootstrap命令初始化集群，并在各个RS上生成 first_tablet_meta文件。原来主RS通过操作日志的方式将first_tablet_meta文件内容同步到各个备机，备RS回放该日志即可在磁盘上生成文件。现在，我们取消了操作日志，而是通过rpc将文件内容发送给所有的RS，生成first_tablet_meta文件。package_code是OB_RS_SLAVE_INIT_FIRST_META

```
ObBootstrap::create_all_core_tables()
|--> init_meta_file(...)
|--> root_server_.get_first_meta()->init()
|--> log_worker_->init_first_meta()
```



```
ObBootstrap::create_all_core_tables()
|--> init_meta_file(...)
|--> root_server_.get_first_meta()->init()
|--> root_server_.slave_init_first_meta()
|--> for all RS
    rpc_stub_.rs_init_first_meta()
```

备RS接收到OB_RS_SLAVE_INIT_FIRST_META数据包后，调用rt_slave_init_first_meta_row()函数生成first_tablet_meta文件，生成文件的过程与原本RS解析日志后生成文件的过程一致。

2.2.3 RS主备切换

RS主备切换指的是物理角色之间的切换，即Master切为Slave，以及Slave切换为Master。物理角色的切换涉及到主备功能的转变，不像逻辑角色切换一样只需要盖面状态值即可。

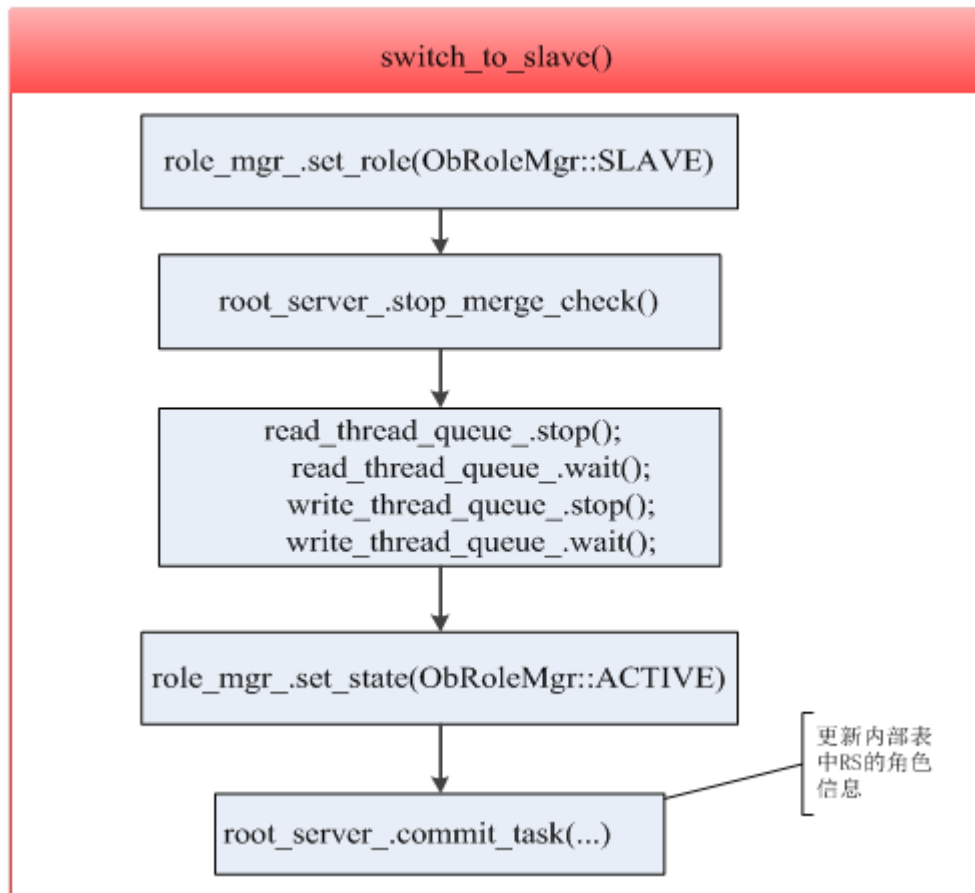
原OB中，没有Master -> Slave切换的实现，当Master出现问题时，只能kill self退出进程；有Slave->Master的切换流程，当备机检测到RS状态为SWITCHING时，则执行备切主流程。我们的架构中，RS可能在Follower, Candidate, Leader之间转换，需要实现Master->Slave切换功能，并且主备切换不能仅能执行一次。因此，我们增加了switch_to_master()实现备切主功能，增加switch_to_slave()实现主切备功能。

1) switch_to_master()

在原OB的start_as_slave()函数中已经包括备切主的流程，我们将这个流程包装成函数，使其可以被多次调用。另外修改了start_as_slave()函数，执行完备切主后，只要没有出错，就不会跳出while循环。这样RS可以不断进行主切备、备切主。

2) switch_to_slave()

原OB中没有相关实现，我们根据备RS启动时需要开启的线程、队列等，实现switch_to_slave。



2.2.4 RS心跳机制

原OB中的RS之间没有心跳机制，我们为了让Leader更新其它RS的租约时间，增加了该功能。

(1) 心跳包数据结构（只列出与心跳相关的成员变量）

```
struct ObMsgRsHeartbeat {
    static const int MY_VERSION = 1;
    ObServer addr_; // 心跳发送端server的ip, port
    int64_t lease_; //消息发出端授予接收端的租约(now + during_time)
    int64_t current_term_; // 心跳发送端的term
    int64_t config_version_;
};

struct ObMsgRsHeartbeatResp {
    static const int MY_VERSION = 1;
    ObServer addr_; // 心跳发送端server的ip, port
    int64_t current_term_; // 心跳发送端的term
};
```

(2) RS发送心跳

增加ObRsHeartbeatRunnable rs_heartbeat_thread线程，RS启动时开启。当RS为Leader时，每100ms调用grant_lease()函数异步向在线的RS发送心跳包。心跳包的内容包括Leader的地址、新的租约、Leader的term值和config版本。另外，还会将Leader上保存的所有RS信息也发送过去。如果Leader发现某RS不在线（election_node中的is_alive表示RS是否在线），则调用root_server->commit_task(..)将该RS从内部表中删除。

(3) RS接收心跳

- 角色为Leader

如果发送方的term大于等于本地term，则切换角色为Follower 否则，忽略该心跳

- 角色为Candidate

切换为Follower，并跟随作为发送方的RS，同时更新本RS的信息。

- 角色为Follower

如果has_leader_ == false，则跟随作为发送方的RS，同时更新本RS的信息。

如果has_leader_ == true且vote_for_ == hb.addr_，则更新本RS信息

如果has_leader_ == true且vote_for_ != hb.addr_，则报错

2.2.5 MS/CS/UPS的注册和RS信息维护

UPS/CS/MS不能再使用VIP访问RS，当主RS发生变化时，这些server需要能够感知并重新向新主RS注册，需要使其预先知道所有RS的地址。因此，需要在内存中缓存所有RS地址信息。

系统初次启动时，主RS将RS列表同步发送给各个UPS/CS/MS。在系统运行过程中，主RS也会通过定时任务向各个非RS节点发送RS列表。

当主RS切换后，新主RS向MS/CS发送命令，使其租约立即过期，缩短重新注册时间。

当UPS/CS/MS租约过期后，会尝试重新向自己保存的主RS注册。如果目标RS不可用或不是主RS，则向内存列表中保存的下一个RS注册

(1) 类ObRsAddressMgr用于管理UPS/MS/CS中Paxos组各个RS的地址信息。

```
class ObRsAddressMgr
{
public:
    int32_t find_rs_index(const common::ObServer &addr);
    int update_all_rs(const common::ObServer *rs_array, const int32_t count);
    void reset();
    int get_master_rs(common::ObServer& master);
    void set_master_rs(const common::ObServer &master);
    int32_t add_rs(const common::ObServer &rs);
    common::ObServer next_rs();
private:
    mutable tbsys::CThreadMutex rs_mutex_;
    common::ObSpinLock lock_;
    ObServer rootservers_[OB_MAX_RS_COUNT]; //store all rs info
    int32_t server_count_; //rootserver count
    int32_t current_idx_; //current index of rootserver for register
    int32_t master_rs_idx_; // array index of master rs,
}
```

(2) 修改各个server上的注册流程

- UPS上注册流程修改：

原注册流程：

```
int ObUpdateServer::register_to_rootserver(const uint64_t log_seq_id){
    ...
    set_register_msg(log_seq_id, msg_register);
    ups_rpc_stub_.ups_register(...)
    ...
}
```

注册流程修改点：

遍历rootservers_[]中各个rs，并向其注册。

如果向主RS注册成功则停止遍历。

如果向主RS注册返回失败，则遍历到下一个RS进行注册。

如果注册超时，则下次执行register_to_rootserver()时向里表中下一个RS注册。

- CS上注册流程修改：

原流程：

流程与UPS中类似。CS中原通过CS_RPC_CALL_RS向rootserver注册。

```
rc = CS_RPC_CALL_RS(register_server, chunk_server->get_self(), false,
                    status, cluster_id, server_version);
```

其中，

```
#define CS_RPC_CALL_RS(func, args...) \
    THE_CHUNK_SERVER.get_rpc_stub().func( \
    THE_CHUNK_SERVER.get_config().network_timeout, \
    THE_CHUNK_SERVER.get_root_server() , args)
#endif
```

如果OB_RESPONSE_TIME_OUT == rc 或者 OB_NOT_INIT == rc，那么cs会继续调用CS_RPC_CALL_RS向rs注册。

修改后流程：

本设计需要修改register_server函数，int ObGeneralRpcStub::register_server(...) const。在该函数中实现all_rs遍历并注册。其具体流程参考前面在ups中的实现。

- MS上注册流程修改：

遍历rs流程与ups中类似，但修改的位置为

```
ObMergerLeaseTask::runTimerTask(void)
    |-- service_->register_root_server()
```

RS选择主UPS方案设计

1 目前设计

rs在内存中维护ups_array[]数组，能够存储最多17个ups的信息，即可以支持最多17个ups在线。在rs中，ups有四种状态：

UPS_STAT_OFFLINE：表示ups处于下线状态。
UPS_STAT_NOTSYNC：表示ups日志不同步，不能服务。
UPS_STAT_SYNC：表示ups日志同步，可以服务。
UPS_STAT_MASTER：表示ups为主ups。

rs每100ms会检查集群中ups状况，包括两个方面：

- (1) 集群中是否有主ups，如果没有，则直接根据rs中已保存的ups最大日志号选择新的主ups。
- (2) 集中中各个ups的租约是否过期。如果租约过期，则将ups的状态设置为OFFLINE。如果发现主ups租约过期，那么rs还需要重新选择主ups。

rs通过函数select_ups_master_with_highest_lsn()选择主ups。这个函数首先确定日志号最大的ups，然后将这个ups状态置为UPS_STAT_MASTER，最后修改内部表_all_server中相应ups的记录项，并重置各个ups的流量分配。

通过心跳机制，rs会将主ups信息告知集群内所有ups，如果ups发现自己的主备角色变了，则进行主切备（函数master_switch_to_slave）/备切主（函数switch_to_master_master）。

如果发生RS切换或主RS宕机重启，主UPS如何保证重新注册后仍然是主？

2 Paxos架构下RS选择主UPS方案设计

2.1 功能设计

Paxos架构下，RS需要判断已注册的UPS是否满足多数派，必须在多数UPS成功注册的情况下才能够进行选主。

UPS间实现了基于Paxos的日志同步，RS在选择主UPS时应当选择拥有最新数据的UPS，这样才能保证UPS切换后数据的完整性，不会出现数据丢失和错误。我们使用election_term表示UPS的选举版本，每次选出主UPS后这个值会自增1。UPS上使用log_term表示日志是处于哪个选举版本上，每次收到RS发送来的election_term后，会设置log_term值为election_term。有最大log_term的多个UPS中，一定拥有最新的日志数据。在这个基础上，拥有最大日志号的UPS上的日志就是最新的。因此，RS在选择主UPS时，将选择拥有最大log_term和最大日志号的UPS作为主UPS。

为了确保获取各个UPS实时的最新日志情况，RS不会缓存UPS的日志号等信息，而是在每次选择UPS前获取多数UPS的最大日志号、election_term和log_term，然后再根据这些信息选择主UPS。

一般存在两种异常引起RS发起选主UPS的流程。其一，主UPS宕机；其二，主UPS未宕机，但与RS间的网络故障。第一种情况下，RS可以直接向UPS获取日志号等信息并选主、但是对于第二种情况，由于主UPS仍在提供服务，备UPS返回给RS日志号等信息后，可能继续接受到同步日志并写入磁盘，那么当新的主UPS被选出后，它实际上已经不是集群中拥有最新日志的UPS了。为了避免这种情况，我们需要在选UPS过成功，禁止UPS接收新的日志。

注意，election_term和log_term都用来表示选举版本，但是它们的作用是不同的，而且不能够相互替代。

2.2 详细设计

- 检查在线的UPS数目

在ObUpsManager类中增加函数check_most_ups_registered()，检查ups_array[]中状态为非OFFLINE的ups个数是否为多数。rs会在每次选主ups前循环调用该函数，只有在线的ups个数为多数时，才能执行选主。

- 选主UPS时禁止UPS处理新的日志

RS发起选举主UPS的流程前，首先向各个UPS同步发送OB_LOCK_ACTIVE_UPS包，令UPS不再接收新的同步日志。当锁住多数个UPS后，RS才开始进行选主。如果没有确认多数个UPS被锁住，那么RS在下一轮check_lease时会再次通知未锁住的UPS锁日志。

我们在心跳包中增加unlock_ups_，成功选出主UPS后将其置为true，UPS收到心跳后就会解锁接收日志的功能。

UPS收到OB_SEND_LOG包时，增加判断need_refuse_send_log来决定是否处理日志包，UPS锁日志即是将need_refuse_send_log置为true，这样就不会处理接收的日志包了。

```
case OB_SEND_LOG:
    if ((ObUpsRoleMgr::SLAVE == role_mgr_.get_role()
        || ObiRole::SLAVE == obi_role_.get_role())
        && ObUpsRoleMgr::FATAL != role_mgr_.get_state()
        && !need_refuse_send_log_) // if lock or not
    {
        trans_executor_.handle_packet(*req);
        ps = true;
    }
    .
```

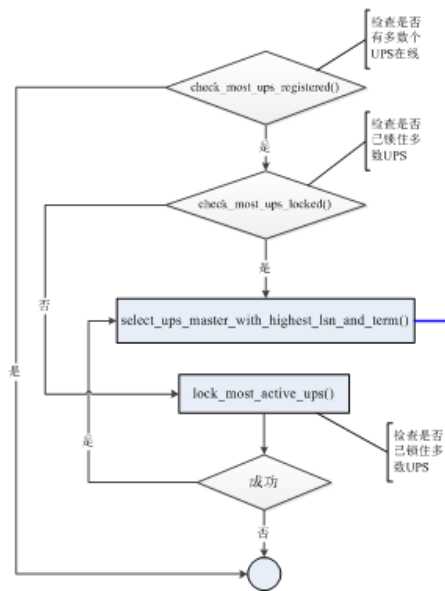
- 获取UPS最大日志号和选举版本，并进行选主

RS在锁UPS的时候，会同时从UPS获取到最大已写盘日志号、log_term和election_term，并保存在ups_array[]数组中，RS锁住多数个UPS的后，就可以直接根据这些信息选主UPS。首先找出log_term最大的若干个UPS，再从这些UPS中选出日志号最大的作为主UPS。

之后，将主UPS的election_term加1，作为这一次选举的选举版本，保存在RS内存中。RS会将该最新的term发送到各个UPS上，当多数UPS都收到并更新自己的election_term后，RS才确认选主UPS成功，否则认为不成功。RS还会通过心跳将最新的election_term发送个各个UPS。

具体流程如下：

ObUpsManager::select_new_ups_master()



ObUpsManager::select_ups_master_with_highest_lsn_and_term()

