二级索引设计文档

修订历史

版本	修订日期	修订描述	作者	备注
0.1	2015-01-06	增加包括系统设计/IndexControlUnit子模块设计等部分内容	翁海星	无
0.1	2015-01-06	增加包括create/show/drop/select/global stage等部分内容	隆飞	无
0.1	2015-01-06	增加包括insert/delete/update/insert/local stage等部分内容	茅潇潇	无

1 系统设计

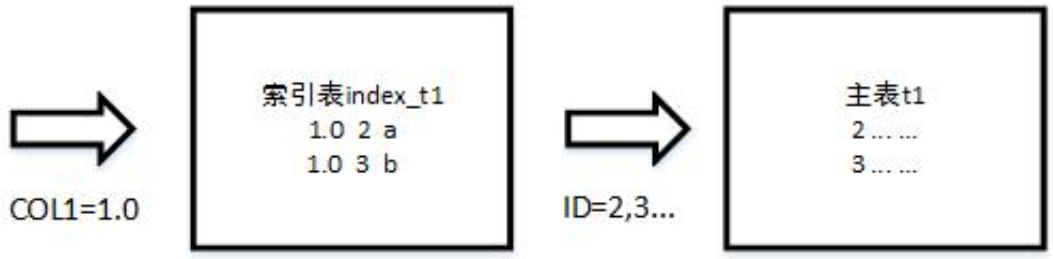
1.1 综述

ecnu数据库是一款分布式数据库，支持高并发的数据查询与数据高可用。目前阶段，能够通过实现自动化的索引存储机制提升非主键查询效率。我们将索引以表的形式存储在数据库系统中，这个表存储了索引列，主键列和storing列(冗余列，用以提升查询性能)，其设计原理和用法大致如下：

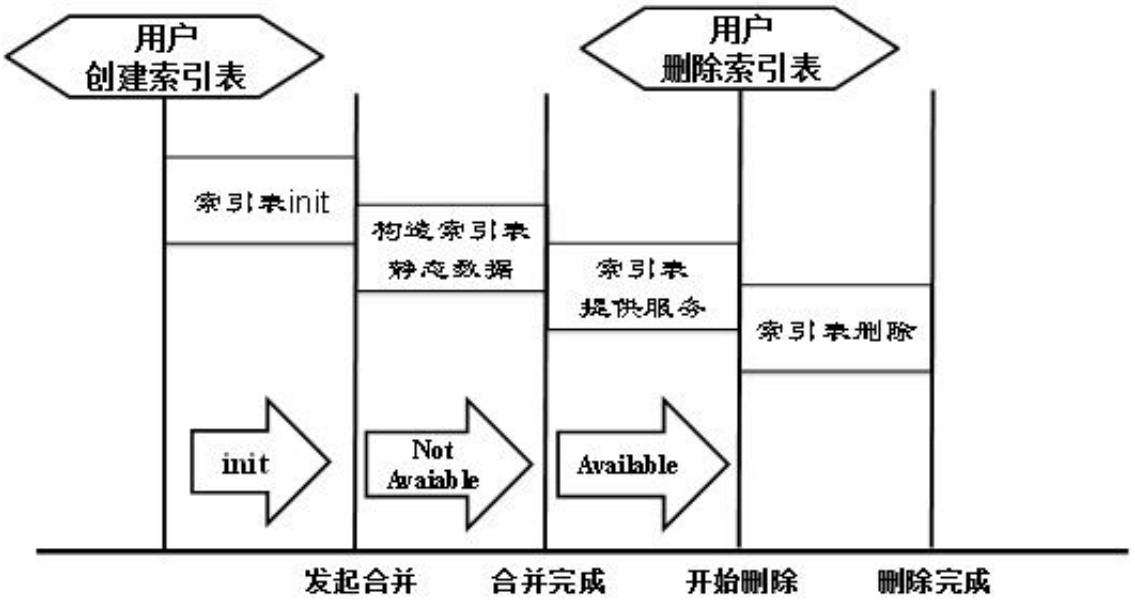
.	主表 t1	索引表 index_t1
主键列	ID	COL1 + ID
非主键列	COL1,COL2…	COL2(STORING)

SELECT * FROM t1 WHERE COL1 = 1.0;

1. 查找索引表index_t1表格，找到COL1为1.0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



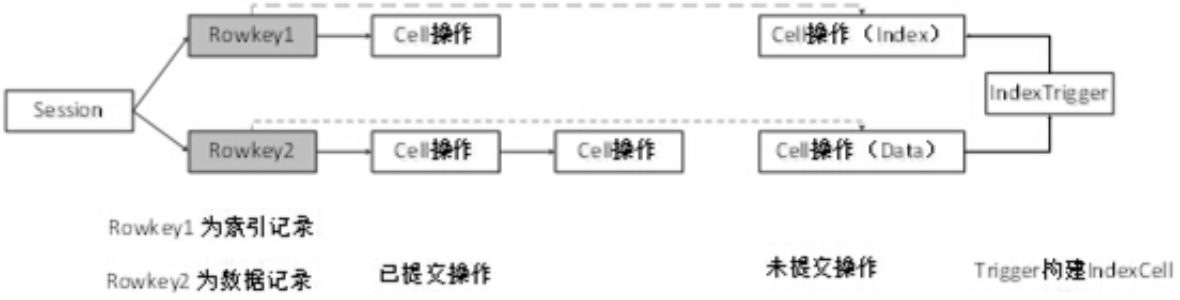
由于索引表也是一张存储的表，所以其也有相应的静态数据与增量数据，相同的，前者存储在CS上，后者存储在UPS的内存表当中。创建索引的时候不但要记录增量数据，也要针对既存数据创建索引。索引将在所有既存数据的索引记录构建完成后变为可用的状态。从索引被创建，直至其可用的过程如下图所示：



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. 首先等普通的每日合并完成，主表所有的tablet写到磁盘。
2. RootServer选择部分ChunkServer(根据实际情况决策)，通知其需要对哪些tablet根据索引列进行排序，这个过程称之为构建局部索引。
3. 局部索引构建完成后，ChunkServer对其进行采样，发送一个直方图给RootServer。RootServer根据接收到的直方图信息，对索引表tablet切分，决策并通知部分ChunkServer去完成一部分的tablet构建。
4. 被选中的ChunkServer互相交换局部索引的tablet数据，完成全局索引的tablet写磁盘。
5. 对创建完成的索引tablet构建副本，将索引状态置为可用。

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。XXBase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。为了实现这个同步更新，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。



1.2 名词解释

对本文档出现的缩写和特殊词汇进行解释

- **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，OceanBase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**：OceanBase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。

1.3 功能

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）则会将对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。索引建立语句的语法规则满足

  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>：已经存在的数据表。
  <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：将特定主表上的索引删除，删除语句的语法规则为：

  drop index [<索引表名>,..] on <主表名>

<索引表名>：已经存在的索引表，可缺省，表示删除主表上的所有索引。
<主表名>：索引表的的主表

- 显示索引：将一张表上的索引列出

  show index on <主表名>

  <主表名>：索引表的的主表

- 数据更新：语法与更新普通表相同

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

- 数据查询：现版本支持两种索引查询计划，一是自动检测索引列来判断是否使用索引，也支持用户显式要求对索引表查询。

A.自动使用索引表，语法与SQL语句的select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。

B.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 c
1=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“__idx_idxname”，idxname在创建索引表时由用户指定。

- 数据删除：

  1. 索引表数据与索引表同步，删除数据以主表数据删除为主。
  2. 使用语法与原delete语句相同。

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。考虑到如果主表已有既存数据，新建的索引无法实时维护索引记录，因此，在索引表没有在合并后生效的时间里不提供服务。
- 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收集完全部直方图并决策索引的分片。

2 模块设计

总体的模块设计和框架描述

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。OceanBase中，新创建的索引需要等待一次每日合并后才生效。

2.1.1 语法格式

```
CREATE INDEX [IF NOT EXISTS] index_name ON table_name (columnname_l
ist1) STORING(columnname_list2) [index_options_list];
```

使用上述的SQL语句创建一张索引表，索引表名字由index_name指定。原表名字由table_name指定，索引列由columnname_list1指定。其中columnname_list2为冗余列，其作用为当SQL查询该数据表的冗余列时，能从索引表直接返回冗余列的数据，从而减少一次回表查询。几点说明如下：

- 使用“IF NOT EXISTS”时，即使创建的表已经存在，也不会报错，如果不指定时，则会报错。
- 本语句建立索引的排列方式为：首先以 columnname_list 中第一个列的值 排序；该列值相同的记录，按下一列名的值排序；以此类推。
- SQL语句允许创建一张没有Storing列的索引表，因此，下面的SQL语句是能够成功被执行的：CREATE INDEX idx1 ON test(col1);
- 执行“SHOW INDEX语句可以查看创建的索引。
- “index_options_list”内容和Oceanbase普通数据表中的table_option是基本一致的。请参见下表，各子句间用“,”隔开。index_option尽量与原表保持一致。考虑到索引表和原数据表之间的一致性问题，本文实现不支持用户设置索引表的过期条件。

参数	含义	举例
EXPIRE_INFO	过期条件	暂不支持设置，索引表直接继承原数据表的过期条件

TABLET_MAX_SIZE	索引表的Tablet最大尺寸，单位为字节，默认256MB	TABLET_MAX_SIZE = 268168356
REPLICA_NUM	NDEX表的tablet副本数，默认为3	REPLICA_NUM = 1
COMPRESS_METHOD	数据压缩算法[none(默认值，不作压缩) | lzo_1_0 | snappy_1_0]	COMPRESS_METHOD = 'none'
USE_BLOOM_FILTER	USE_BLOOM_FILTER[0:默认值,不使用 | 1:使用]	USER_BLOOM_FILTER = 0
COMMENT	注释信息	COMMENT = 'this ia an index table'
INDEX_TABLE_ID	指定索引表的ID，否则由系统分配。如果ID小于1000，需要打开rootserver的配置项开关"ddl_system_table_switch"	index_table_id = 1000

2.1.2 索引表规范和索引表及其他注意事项

- 上述的SQL语句当中，index_name是INDEX的索引名，而存储在OceanBase当中的索引表，表名是按照[原表name+idx+索引名]的规则生成，被称作索引表的内部表名。内部表名对用户是不可见的，作用相当于C++中的命名空间，使得我们可以在不同的table上建立相同名字的索引表。同一张数据表上不可建立同名索引表。基于上述规则索引表内部表名实际形式如下：

```
___[tablename]__idx__[indexname]
```

- 索引表其实在OB当中是特殊的一类数据表。其主键列为索引列+数据表的主键列。当创建索引表的时候不指定数据表的主键列为索引列，本文实现也会将原数据表的主键列加入到索引表中，充当主键的一部分。因此，在计算索引表的主键的时候，不要忘记系统本身添加的原数据表的主键列。
- 索引表数量限制:目前定的限制是一张数据表最多建立五张索引表。暂时不可以根据应用场景动态调整。
- 索引表列数的限制:索引列的数目限制，取决于数据表的主键数目。因为在Oceanbase当中，任何一张表的主键数是不能超过16的。因此，创建的索引列+数据表的主键列之和不能超过16。特殊的，如果索引列中包含了数据表的主键列，那么这个列在所有的列数计算之中只记录一次。
  再者，索引表的总列数也有限制。现在暂时定的限制为索引表的所有列数加起来不超过100.这个数字并不是一成不变的，可在源码中进行更改。暂不支持根据应用场景动态调整。
- 允许使用SQL语句对数据表的某一个主键列建索引，也就是说，如果有一张表test2，其主键列是col1以及col2的联合主键。那么如下的SQL语句是能够成功被执行的：
  CREATE INDEX i1 ON test2(col2) STORING(col3);

- 索引表的冗余列不能是数据表的主键。因为数据表的主键是用来构造索引表的复合主键的，放在storing列里没有意义。如果查询到非前缀的主键，经过索引表之后也是不用回表的。如下的SQL语句是不能通过的：
  CREATE INDEX i1 ON test2(col2) STORING(col1);

2.1.3 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，在新增内部表中添加了两个新的字段：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
  回答额外两个字段在什么地方添加？Oceanbase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Oceanbase0.4.2（OB）设计的对于OB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，就是填充操作符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中有一个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，去填充一个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别OB中的数据表和新增的索引表。

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。

2.2.1 语法格式

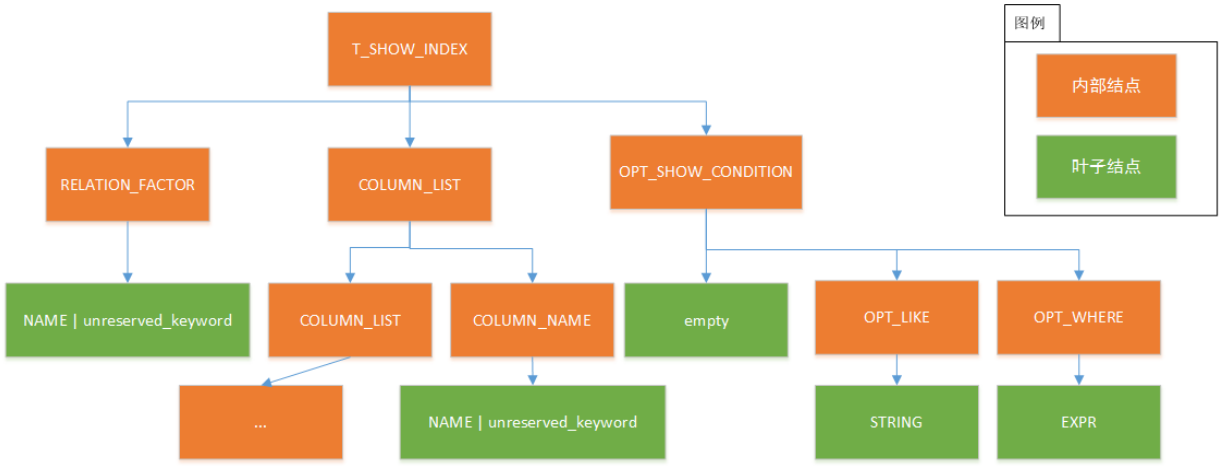
```
SHOW INDEX ON relation_factor opt_show_condition;
```

使用上述格式查看一张数据表上所建立的全部索引表的信息。其中relation_factor指定原表，opt_show_condition使用like或where条件对查询结果进行过滤。

2.2.2 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。



2.2.3 SHOW INDEX结果

在OceanBase中存在一张DUAL的虚拟表，它可以视为一个一行零列的表。当我们不需要从具体的表来取得表中数据，而是单纯地为了得到一些我们想得到的信息，并要通过SELECT完成时，就要借助一个对象，这个对象就是DUAL。一般可以使用这种特殊的SELECT语法获得用户变量或系统变量的值。当SELECT语句没有FROM子句的时候，语义上相当于FROM DUAL，此时，表达式中只能是常量表达式。比如说我们在OceanBase中执行：SELECT 2*3;或者SELECT 2*3 FROM DUAL;的时候，我们会得到如下结果：

```
+------+
| 2*3  |
+------+
|    6 |
+------+
1 row in set (0.00 sec)
```

我们可以新增一张虚拟表来存储showindex查出的结果集。这张表中不存储实际数据，只是使用其schema填充逻辑计划/物理计划/结果集的信息。这张表的schema信息由ObShowSchemaManager管理，所以要在ObShowSchemaManager添加该表的shema信息。因此，我们会获得一张类似于上面的表:

```
mysql> show index on obst;
+--------------+----------+-----------------------------------------+
| index_name   | status   | index_column                            |
+--------------+----------+-----------------------------------------+
| __obst_obsi1 | AVALIBLE | obstcol2  obstcol1                      |
| __obst_obsi2 | AVALIBLE | obstcol3  obstcol4  obstcol1  obstcol2  |
| __obst_obsi3 | AVALIBLE | obstcol2  obstcol3  obstcol1            |
| __obst_obsi4 | AVALIBLE | obstcol2  obstcol7  obstcol8  obstcol1  |
| __obst_obsi5 | AVALIBLE | obstcol1  obstcol2                      |
+--------------+----------+-----------------------------------------+
5 rows in set (0.00 sec)
```

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。

2.3.1 语法格式

```
DROP INDEX opt_if_exists index_list ON table_name;
```

使用上述的SQL语句删除一张索引表，索引表名字由index_list指定。原表名字由table_name指定。

- 使用“IF EXISTS”时，即使创建的表不存在，也不会报错，如果不指定时，则会报错。
- 同时删除多个索引时，用逗号隔开。不指定索引表名字时，删除table_name对应数据表上建立的所有索引表，但不会删除数据表。
- 需要特别注意的是，当删除一张数据库中的一张原表的时候，会删除该表上的所有的索引表。

2.3.2 举例

- OB中添加二级索引功能以后，如果使用如下SQL语句删除一张表：

```
DROP TABLE t1;
```

如果这张表上建立了一些索引，那么这些索引在表被删除的时候也要同时被删除。

- 同时，还应该支持用SQL语句单独删除索引，比如:

```
DROP INDEX i1 on t1;
```

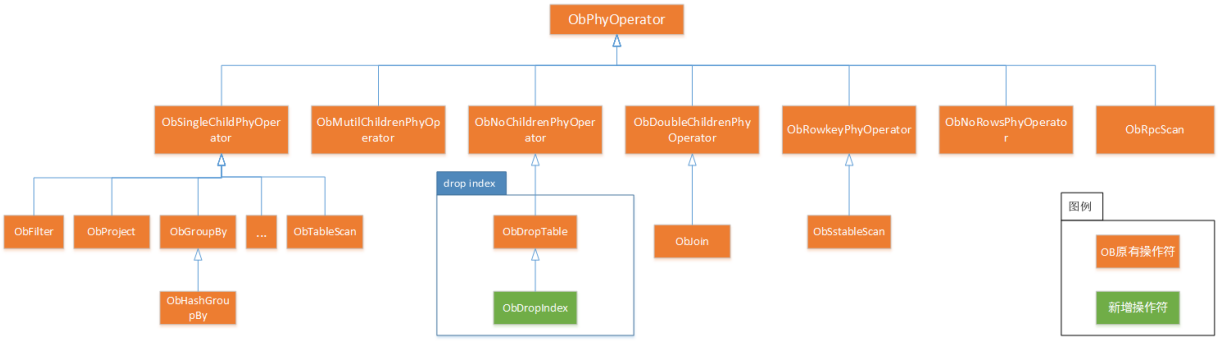
那么表t1上的一张索引表”“__t1__idx__i1”“将会被删除。

2.3.3 设计原理

索引表本质上是OceanBase中的一张数据表。但是索引表和数据表也存在着一些区别。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，由于要保证索引表对应的原表和索引表之间数据的一致性。因此，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

2.3.3.1 删除索引表

为了实现Drop index去删除一张索引表，我们新增一个继承于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。此操作符用于完成对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)…所以这些可以使用drop_table来删除。至于索引表特有的信息，可以使用drop index来删除。

2.3.3.2 删除数据表

删除带索引的数据表。开源的OceanBase 0.4.2版本当中还没有实现Secondary Index。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的类中增加一个私有的成员变量：

```
private:
  // data members
  bool if_exists_;
  common::ObStrings tables_;
  mergeserver::ObMergerRootRpcProxy* rpc_;

  // add longfei [drop index] 20151028
  bool has_indexs_; ///< table has index?
  common::ObStrings all_indexs_; ///< store all indexs on all table
s
```

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indexs_是否为空。

1. 索引表为空，这种情况跟原来OB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&2.3.3.1说的流程: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table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。

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。OceanBase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当中的索引使用规则说明部分。

2.4.1 语法格式

参考Oceanbase0.4.2 SQL使用指南 select部分。

2.4.2 规则算法

2.4.2.1 使用索引表

原select流程（指的是OB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。

2.4.2.2 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。

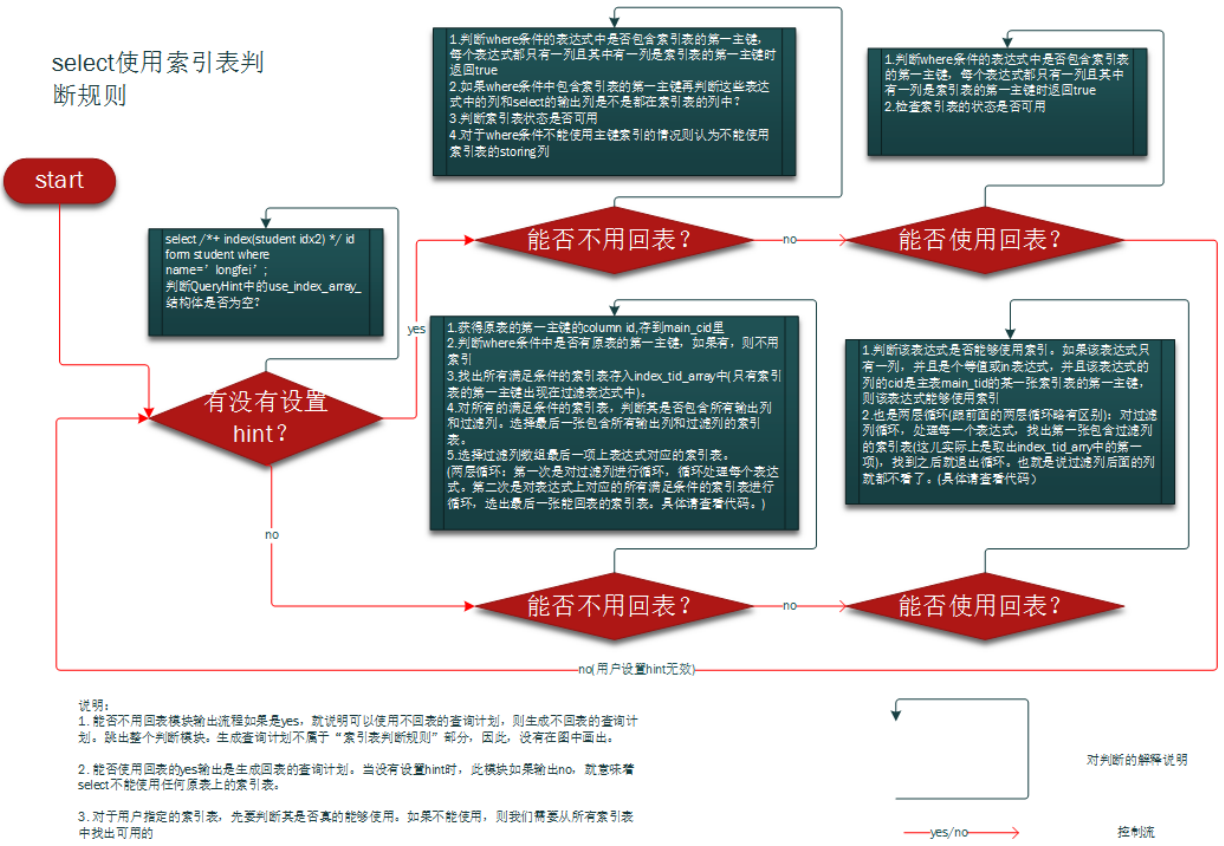
判断sql语句能够使用索引表的不回表的索引规则：

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。

2.4.2.3 流程



2.4.3 物理计划生成

2.4.3.1 不回表的物理计划的生成

函数名：gen_phy_table_not_back
原理：在设置物理计划的rpc_scan 的时候将table_id改为索引表的tid，使物理计划查询索引表。

2.4.3.2 回表的物理计划的生成

函数名：gen_phy_table_back
原理：回表的使用，底层使用两次查询来实现。只有在回表的情况下，才会有两次查询。根据上面生成的回表情况下的ObTableRpcScan操作符，在这个操作符open的时候，生成了对索引表的scan_param，并把scan_param存到sql_scan_request里面。根据select原来的流程实现，它会在这个操作符get_next_row的时候，调用sql_scan_request的open函数，向cs拉取数据。ObTableRpcScan在get_next_row的时候，会调用孩子操作符ObRpcScan的get_next_row函数。
ObRpcScan的get_next_row函数根据查询的方法（scan或是get）来进行不同的处理。这里我主要修改了查询方法为scan的情况下，ObRpcScan的get_next_row函数。使之能够支持二次查询。（因为第一次查询索引表的方法肯定为scan）

实现思路：
对scan方法，ObRpcScan调用了get_next_compact_row()方法来获得数据，我们主要修改了此函数。具体流程如下：

1. 如果不是回表，则按照原来的实现走。
2. 如果是回表，则先充值一下sql_get_request（这个sql_get_request和

sql_scan_request都是ObRpcScan的成员，在前面open的时候，由于方法为scan，故只对sql_scan_request设置了scan_param，sql_get_request没有用到），重置完之后，生成get_param。

3. 对于get_param。我写了函数create_get_param_for_index来专门生成它。在函数create_get_param_for_index里面，我调用了sql_scan_request的get_next_row函数，因为sql_scan_request的scan_param是针对索引表生成的，故这里获得的数据为索引表的数据。（这里就是对第一次scan索引表的实现）。在scan索引表的数据结束后，我处理下得到的索引表的数据，把它转化成主表的主键，然后把所有的主键全都存到get_param里面。
4. get_param生成完之后，我再把get_param存到sql_get_request里面，调用sql_get_request的open函数向cs发送请求，再调用sql_get_request的get_next_row函数，得到返回的一行数据。最后将返回的一行数据作为该函数的输出。（这里就是对第二次get原表的实现）

对于表别名的处理：

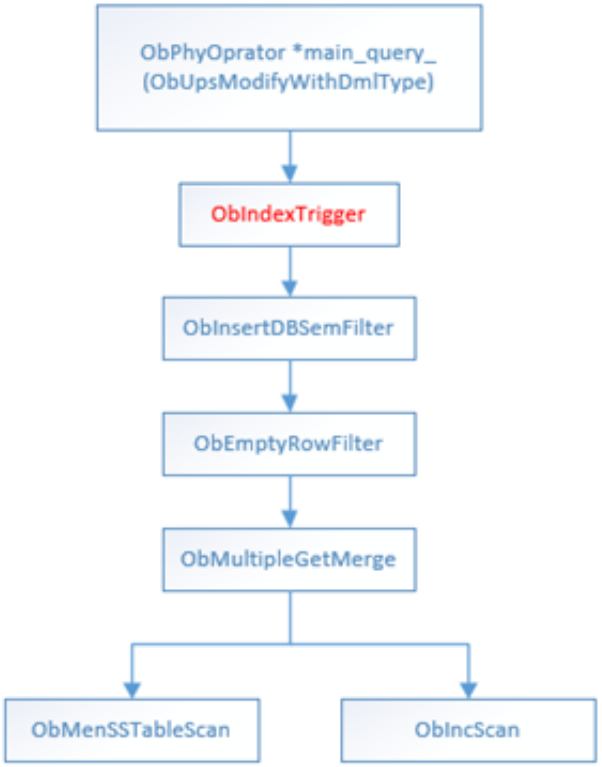
OB原有的处理：对于表别名，OB会给别名取一个tid，sql语句中所有涉及到该表的信息里存的都是表别名的tid。包括返回的数据，每一列的数据的tid也是表别名的tid。OB在生成操作符ObTableRpcScan的时候，会把别名的tid和实际表的tid都存起来。

现在的处理：在不回表的查询中，如果有表别名，则别名的tid不变，实际表的tid我把它改成了索引表的tid。然后在存储filter和输出列的时候，如果有表别名，则不用修改相应表达式.在回表的查询中，如果出现了表别名。则别名的tid不变，实际表的tid我把它改成了索引表的tid。

2.5 insert子模块设计

2.5.1 流程

在原insert的物理计划里面新增了一个物理操作符：ObIndexTrigger。所有对索引表的插入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：



2.5.2 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

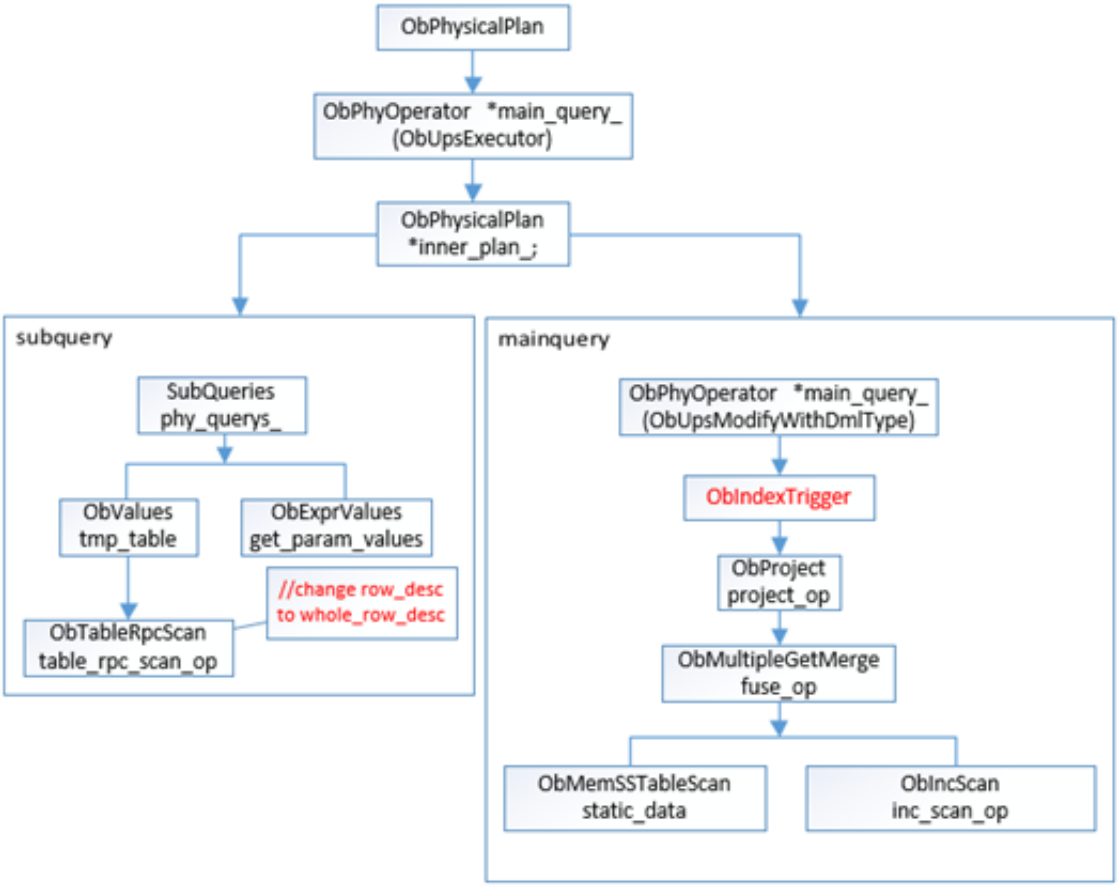
函数handle_one_index_table()：

1.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

2.6 delete子模块设计

2.6.1 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：



2.6.2 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）。

3 取数据时直接从ObProject的子操作符中获取数据，以获取不仅仅是主表主键列的数据，而是主表所有所需列的数据。

UPS在接收到MS发过来的物理计划之后，会delete的物理计划open。 Open完之后，会调用apply函数将原表的数据删除。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将原表的数据从内存表里删除，之后再调用操作符IndexTriggerIns的函数handle_tringger()，在函数handle_tringger()里将索引表的数据从内存表里删除。

函数handle_tringge():

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

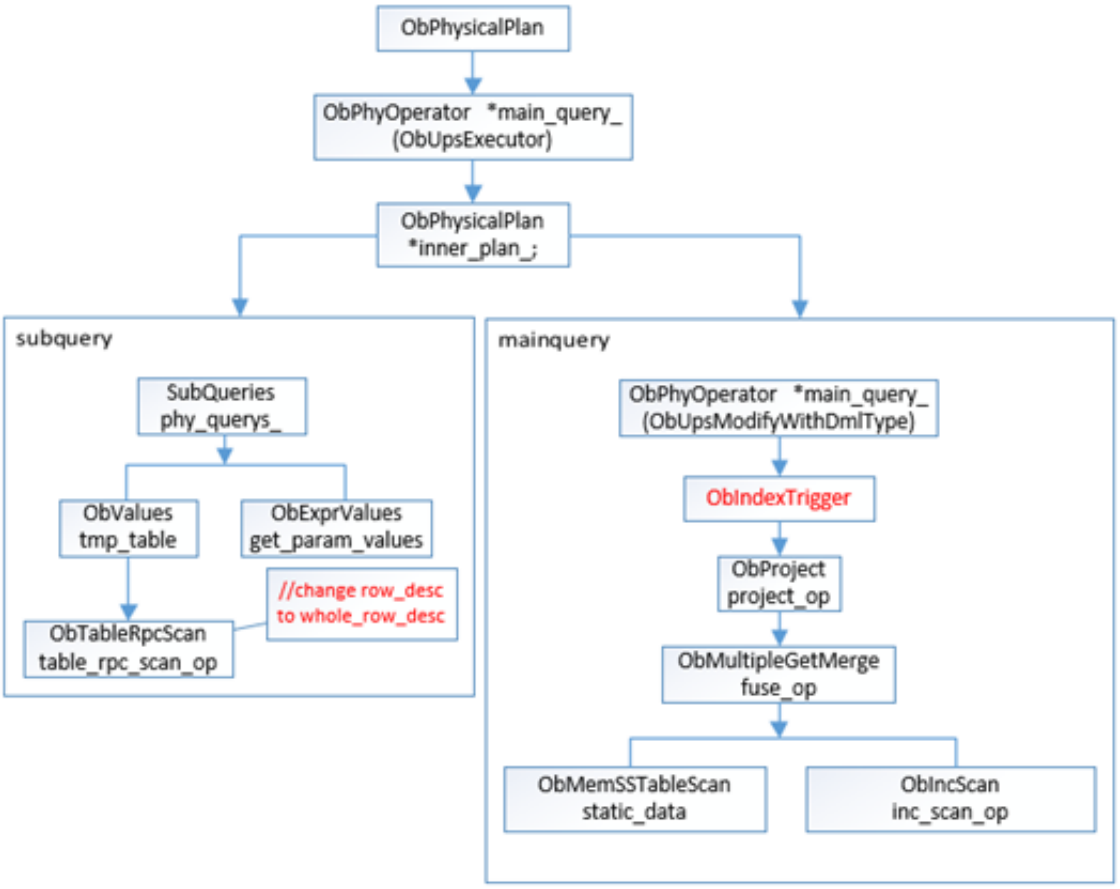
函数handle_one_index_table():

1.识别当前sql语句的类型
2.如果sql语句为delete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语句中原表需要删除的数据，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.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

2.7 update子模块设计

2.7.1 流程

在用UPDATE语句修改数据表达的时候，判断修改的列是否被其索引表所引用，如果不是，则物理计划不需要做任何改动，按照原本的流程对数据表进行修改。如果是，需要新增操作符ObIndexTrigger。另外在修改索引表的时候应当考虑到，有时候改动到的列对于索引表来说是主键，OB现在不能直接对主键列进行修改，所以，在修改索引表的时候，应该使用先删除，再插入的办法更新索引表，也就是，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。具体的物理计划如下：



2.7.2 关键算法

在用UPDATE语句修改数据表达的时候，如果修改的列被其索引表所引用，那么在物理计划里需要添加操作符ObIndexTrigger，对索引表的插入和删除操作都封装在这个操作符里面。我们还修改了ObTableRpcScan的row_desc参数，原来的row_desc仅仅是主表主键列和修改列的row_desc，现在需要将其修改为主表所有所需列的row_desc（包括索引表所涉及的列）。最后，我们再取数据的时候直接从ObProject的子操作符中获取数据，以获取不仅仅是主表update相关列的数据，而是主表所有所需列的数据。

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

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

2.8 replace子模块设计

2.8.1 问题

REPLACE语句作用的效果有两种形式：插入和替换。如果REPALCE结果为插入，那么插入数据中的主键一定不存在于表中；如果REPALCE结果为替换，那么插入数据中的主键一定存在于表中。在未实现二级索引的OceanBase下，REPLACE语句只需要修改数据表。因此，不需要向ChunkServer请求静态数据，通过插入数据中的主键直接更新MemTable，更改操作为REPLACE，而无视主键是否已存在表中（即，插入和替换均可同等对待）。

实现二级索引的OceanBase中，REPLACE语句不仅需要修改数据表，同时也完成对索引的更新操作。如果REPALCE结果为插入，那么只需把新纪录同时插入数据表和索引即可；如果REPALCE结果为替换，那么直接修改数据表的MemTable可以达到目的，但是直接修改索引表的MemTable，会导致索引表中出现脏数据。举例说明（加粗标注为主键字段）：

table name	***c1***	***c2***	c3	c4	c5
test	***1***	***2***	3	4	5

index name	***c3***	***c1***	***c2***	c4
idx1	***3***	***1***	***2***	4

此时，执行replace into test values(1,2,4,4,5)

得到结果：

able test (***1***,***2***,4,4,5)

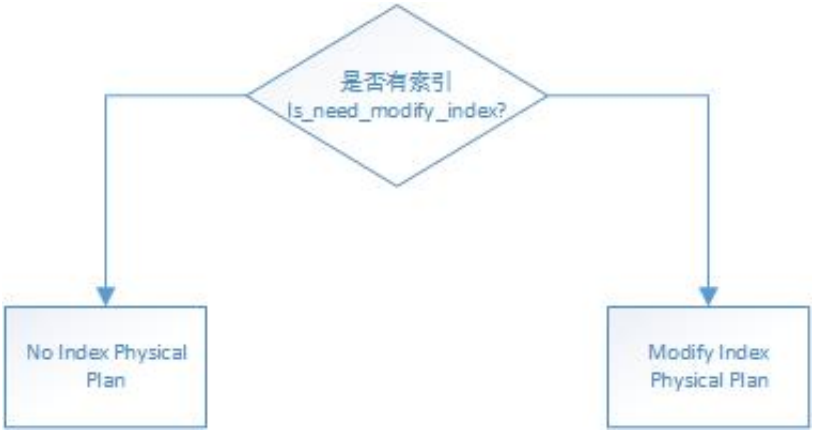
Index idx1 (***3***,***1***,***2***,4), (***4***,***1***,***2***,4)

结果，数据表test一切正常，索引表idx1中会出现两条记录，其中，(***3***,***1***,***2***,4)为脏数据，是应该删除的。

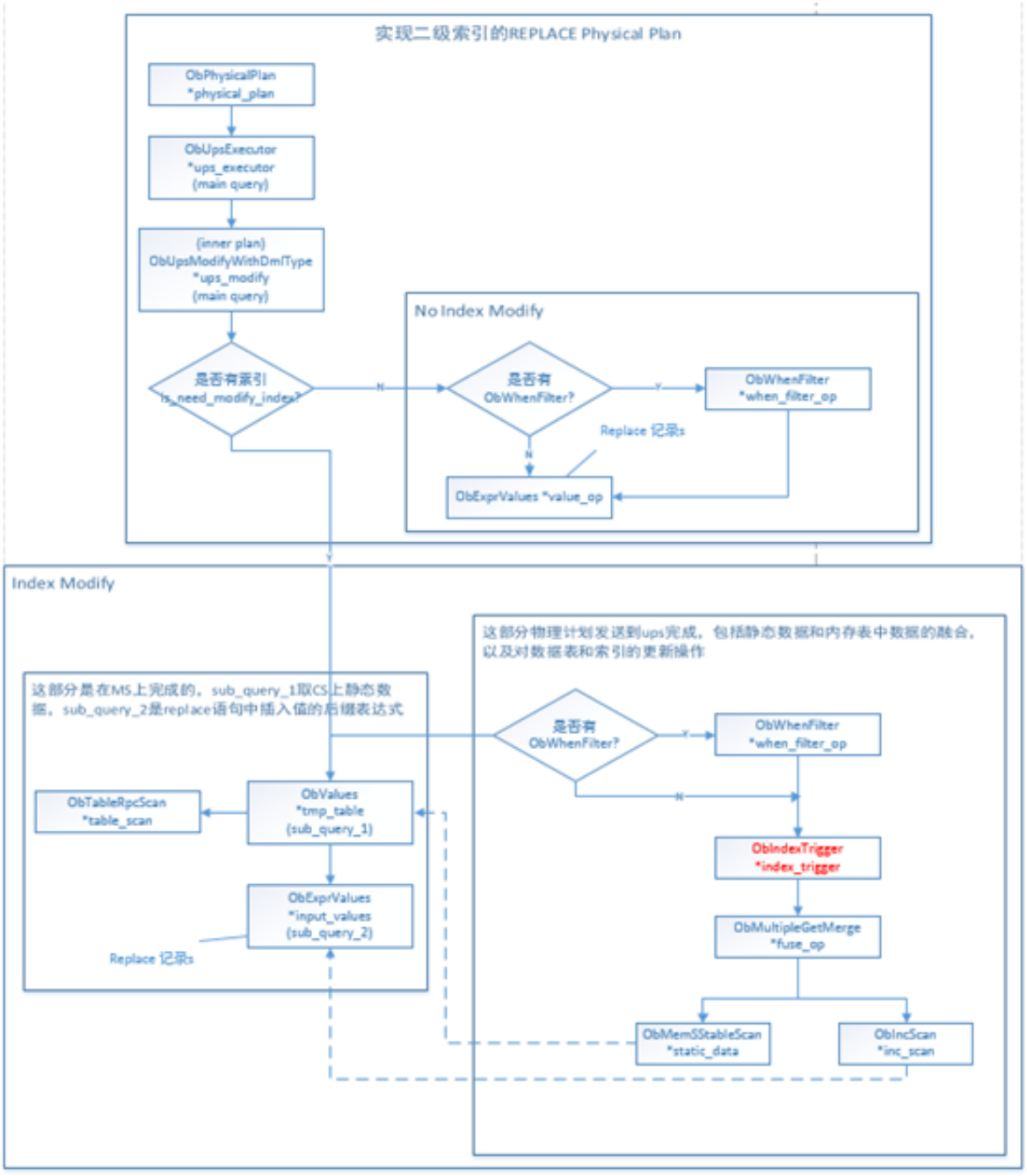
结论：增加二级索引后，REPALCE也需要向ChunkServer获取静态数据，用以更新索引表。同时，增加静态数据的获取，也增加REPLACE的开销，进而导致REPLACE的性能接近于INSERT。

2.8.2 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。



2.8.3 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

2.9 静态数据构建子模块设计

2.9.1 概述

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OB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，就必须完整的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。

本设计文档将静态数据构造过程分为了以下几个阶段：

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。

索引构建是在cs上完成的，但是需要rs来调控。cs通过在心跳包中加入有关索引构建的信息来调控整个过程，携带信息的结构体如下所示：

```
    //add longfei [cons static index] 151120:b
    // index beat is a type of heart beat, can trans some informati
on between rs and cs
    struct IndexBeat
    {
      uint64_t idx_tid_; ///< 索引表id
      IndexStatus status_; ///< 索引目前的状态
      int64_t hist_width_; ///< 直方图的宽度
      ConIdxStage stage_; ///<  索引构建阶段
      IndexBeat()
      {
        idx_tid_ = OB_INVALID_ID;
        status_ = ERROR;
        hist_width_ = 0;
        stage_ = STAGE_INIT;
      }
      void reset()
      {
        idx_tid_ = OB_INVALID_ID;
        status_ = ERROR;
        hist_width_ = 0;
        stage_ = STAGE_INIT;
      }
      NEED_SERIALIZE_AND_DESERIALIZE;
    };
    //add e
```

2.9.2 模块介绍

2.9.2.1 索引构建准备阶段

索引构建整个流程跟Oceanbase的每日合并流程较为相似。索引构建阶段任务是在每日合并完成之后启动，有关每日合并模块请参考Oceanbase源码doc目录下的《ChunkServer_detail_design》文档。

首先，构建索引任务开始之前，要启动多个构建索引的线程。这些线程是在cs启动的时候就被构建完成的。cs启动的时候初始化index_handle_pool，这是一个多线程池。cs启动的时候要为每一个线程分配空间。

```
        //add longfei [cons static index] 151117:b
        if(OB_SUCCESS == rc)
        {
          if (OB_SUCCESS != (rc = chunk_server_->get_tablet_manage
r().init_index_handle_pool()))
          {
            TBSYS_LOG(ERROR,"start index handler thread failed,re
t=%d",rc);
          }
        }
        //add e
```

在init_index_handle_pool()中调用类ObIndexHandlePool的init()函数，后者主要完成以下工作：

```
        // 创建两个hashmap,用来保存rs切分的range
        if (OB_SUCCESS != (ret = set_config_param()))
        {
          TBSYS_LOG(ERROR, "failed to set index work param[%d]", re
t);
        }

        // 调用tbsys::CDefaultRunnable的start函数来启动多个线程
        else if (OB_SUCCESS != (ret = create_work_thread(max_work_t
hread_num)))
        {
          TBSYS_LOG(ERROR, "failed to initialize thread for inde
x[%d]", ret);
        }

        // 为每一个handler(完成索引构建的类)分配空间
        else if (OB_SUCCESS != (ret = create_all_index_handlers()))
        {
          TBSYS_LOG(ERROR, "failed to create all index handle
r[%d]", ret);
        }

        // 初始化处理索引构建失败信息保存类
        else if (OB_SUCCESS != (ret = black_list_array_.init()))
        {
          TBSYS_LOG(ERROR, "failed to init black list array");
        }


        else
        {
          TBSYS_LOG(INFO,"init index handle pool succ");
        }
```

ObIndexHandlePool初始化时创建启动多个线程，调用run()运行construct函数并阻塞，等待唤醒。

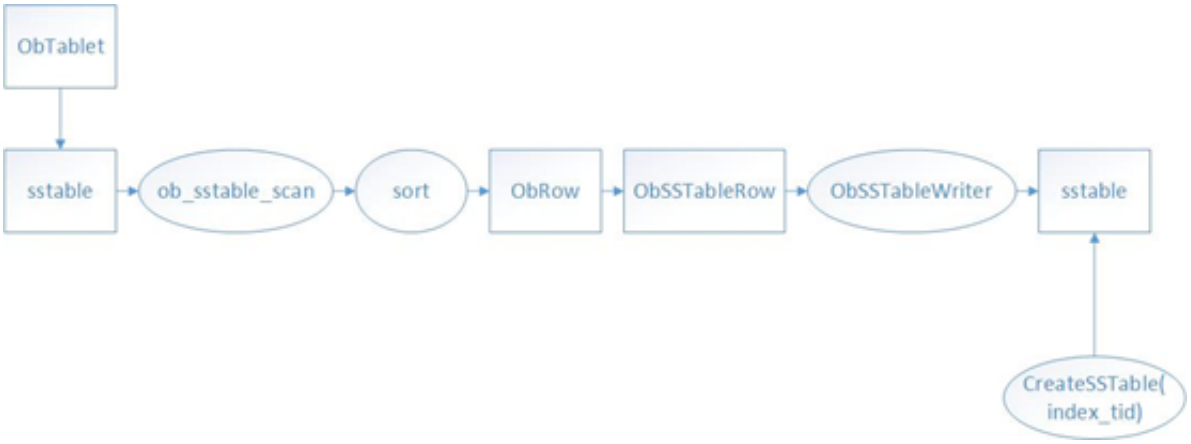
在多线程被唤醒之前，需要完成对于索引构建时期用到的信息进行准备，比如索引表对应的原表有哪些tabelt，这些tablet位于哪些地方，rs切分的range是什么？等等。构建前准备工作是由index handle pool里的start_round()函数来完成的。此函数中，根据不同阶段来完成不同的准备工作，比如说全局阶段如下所示：

```
        else if (GLOBAL_INDEX_STAGE == which_stage_)
        {
          if (OB_SUCCESS != (ret = fetch_tablet_info(GLOBAL_INDEX_STA
GE)))
          {
            TBSYS_LOG(ERROR, "start build index,round global error");
          }
        }
```

fetch_tablet_info()中会做两遍fetch_tablet_info,分别传原表的id,和索引表的id，解析取回来的信息，以此得到所有的range信息，保存在range_array中。当信息准备好了之后，发送广播通知阻塞的handler开始做实际的工作。handler通过从拿回的信息不同来区分处在哪一个阶段。当从准备信息中拿出的是一个range，则处在全局索引构建阶段。

2.9.2.2 局部索引构建阶段

在局部索引构建阶段进行的主要工作是拿到一个数据表的tablet后，读取这个tablet上的数据，根据索引列排序，将排序后的结果写进sstable，将新写的sstable加入到数据表的tablet当中。具体流程如下图所示。



读取一个Tablet的数据
使用ObSSTableScan操作符来得到一个Tablet的全部行、特定列的数据。

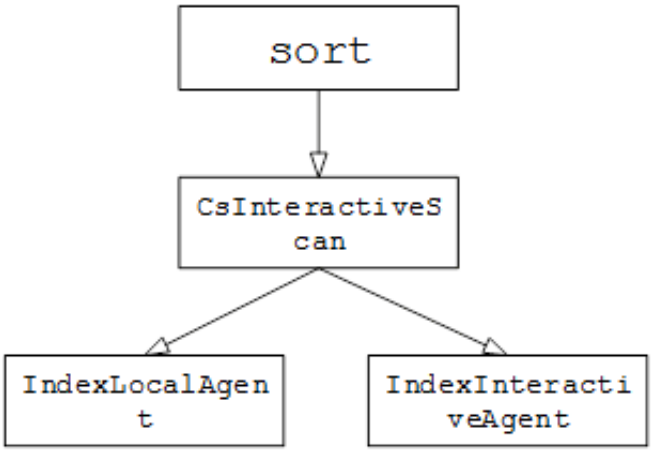
其中需要构造两个参数，一个是sstable_scan_param，包括需要的列id以及range，可以从当前schema和需要构建索引的tablet中得到。另一个是scan_context，即执行sstablescan的上下文，可以从tablet_manager中得到。

对读取出来的数据排序
使用ObSort操作符，并把ObSSTableScan作为ObSort的子操作符。ObSort在open的时候会读取所有的数据进行排序，然后调用ObSort的get_next_row()方法得到的数据就已经是排好序的数据。此操作符是用内存排序，每个tablet大小不超过256M。

写局部索引sstable文件
sstable的行数据是以ObSSTableRow的类型写入数据的，而用scan方法得到的是ObRow类型的数据，因此调用ObSSTableRow的add_obj方法把ObRow类型的数据转化为ObSSTableRow类型的数据，然后再调用save_current_row()函数来将当前sst_row_写入sstable。

2.9.2.3 全局索引构建阶段

全局索引构建过程中使用了两个scan操作符IndexLocalAgent和IndexInteractiveAgent，如下图所示：



由于在ob中，sort操作符只能有一个孩子结点，因此需要借助中间操作符作为过渡IndexLocalAgent和IndexInteractiveAgent分别负责本机上数据的取回和其他cs上的数据的取回。具体得说，rs根据局部阶段汇报的数据表的统计信息来重新切分range，符合这个range的信息可能是在本机上也有可能是在其他cs上。这两者之间的区别在于数据的取回方式，如果在本机上，我们使用封装sstable_scan的IndexLocalAgent来获取数据；IndexInteractiveAgent使用远程调用的方式将结果存放到ObScanner中，然后取出Scanner中的数据，构造成为本机的数据。但是这儿存在一个问题，要读取到CS上的数据，则需要首先确定数据所在的ObTablet，定位到Tablet所在的sstable，使用range定位到数据所在的block，才能读取到数据。而局部索引的sstable这个时候没有生成ObTablet。所以，此刻无法向正常的查询那样，问询RootServer数据所在的Tablet。也不能够对局部索引的sstable写tablet并加入到roottable中，因为这些局部索引sstable的主键范围是不连续的，无法用于查找定位。以，必须给我们创建的局部索引的sstable找到一个宿主tablet。实际上是将每个tablet最大sstable的数目从1更改为2。第一张sstable只用于索引的静态数据的构建，对于用户不可见。由于现在既需要找到tablet的位置，又需要找到rs给定的范围内的数据，因此，使用faker_range来描述局部sstable的范围。

判断是否是新的global阶段：
通过设置变量total_work_start_time_记录global阶段开始时间，当此值为0时意味着这是一个新的global的阶段。

2.9.2.4 索引构建RS端——IndexControlUnit子模块设计

IndexControlUnit**(以下简称ICU)**是RootServer控制索引构建，判断索引是否正确可用的处理模块。ICU在合并开始时启动，执行索引构建流程，确保索引成功建立后停止。

1. 合并开始时，启动ICU，检测合并进度。
2. 检测到所有数据表tablet合并完成后，ICU开始构建索引，与UpdateServer通信，拿到需要构建的索引清单。
3. 对清单中的每一个索引，都依次做如下工作：
   a. 访问RootTable，选择合适（该索引表对应的数据tablet所在）的ChunkServer用以构建索引。
   b. 向被选择的ChunkServer发送构建局部索引的信号，同时开始收集采样汇报信息并自我检测。
   c. 如果经检查得知局部索引的汇报信息收集齐，ICU开始根据汇报信息对索引tablet分片，并规划到合适的ChunkServer，发送构建全局索引的信号，同时开始收集全局索引的汇报信息并自我检测。
   d. 如果经检测得知全局索引的汇报信息收集齐，ICU开始检查数据和索引的列校验和，保证数据和索引的信息一致之后，将索引表置为某一状态。
4. 所有索引构建完成之后，等待索引的副本拷贝，其他集群的相同工作完成，之后将索引表置为可用状态。

2.9.2.4.1 结构

ICU的重要成员包括ChunkServerHandler，IndexService以及IndexDesigner，其核心功能如上图所示。

ChunkServerHandler的核心成员：

```
hash::ObHashMap<ObServer,bool,hash::NoPthreadDefendMode> cm_;//key
chunkserver, value is_alive
```

这个成员在索引开始创建前重置，并检测选择在线、合适的ChunkServer，发送对应的索引构建信号。同时，如果期间有Chunkserver下线，也会重新计算cm_，选择新的ChunkServer执行之后的索引构建。

IndexService的主要接口：

```
    ObScanHelper    *client_proxy_;   //interface to access interna
l system table
    nb_accessor::ObNbAccessor nb_accessor_; //use it to scan data,
and mutate system table to update param
```

这两个成员在ICU初始化的时候构建，用以查询内部表的数据，并且将内部表的增量更新以mutator的方式发送并实现。

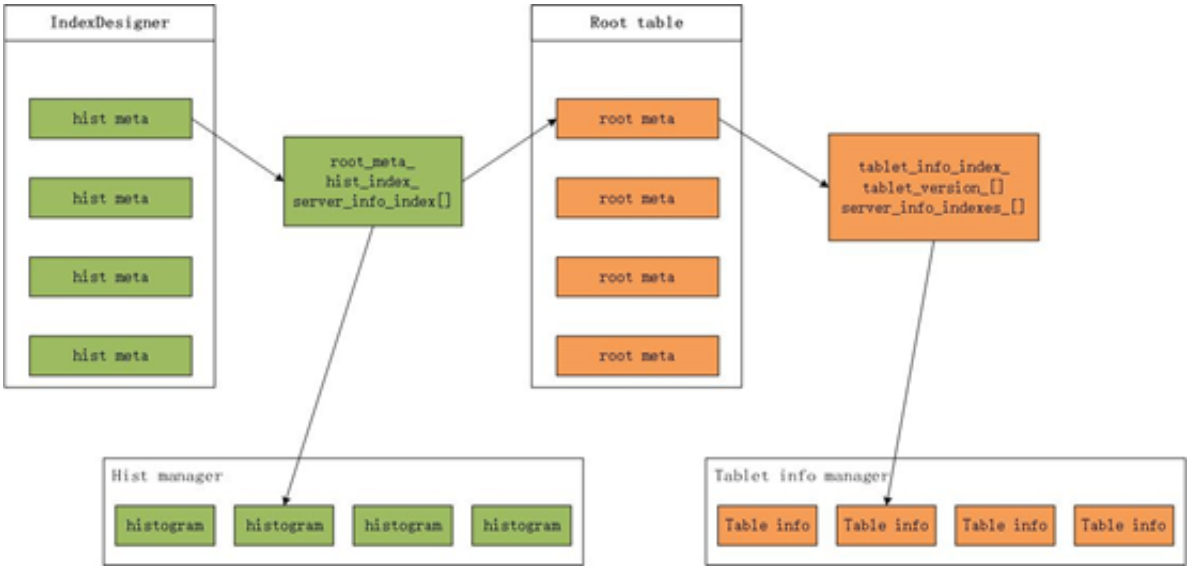
IndexService目前提供了如下几个关键的接口：

1. 获得特定集群下，指定索引表的可用状态
2. 修改内部表，将指定索引表的状态修改为可用/不可用/错误状态

IndexDesigner的核心成员：

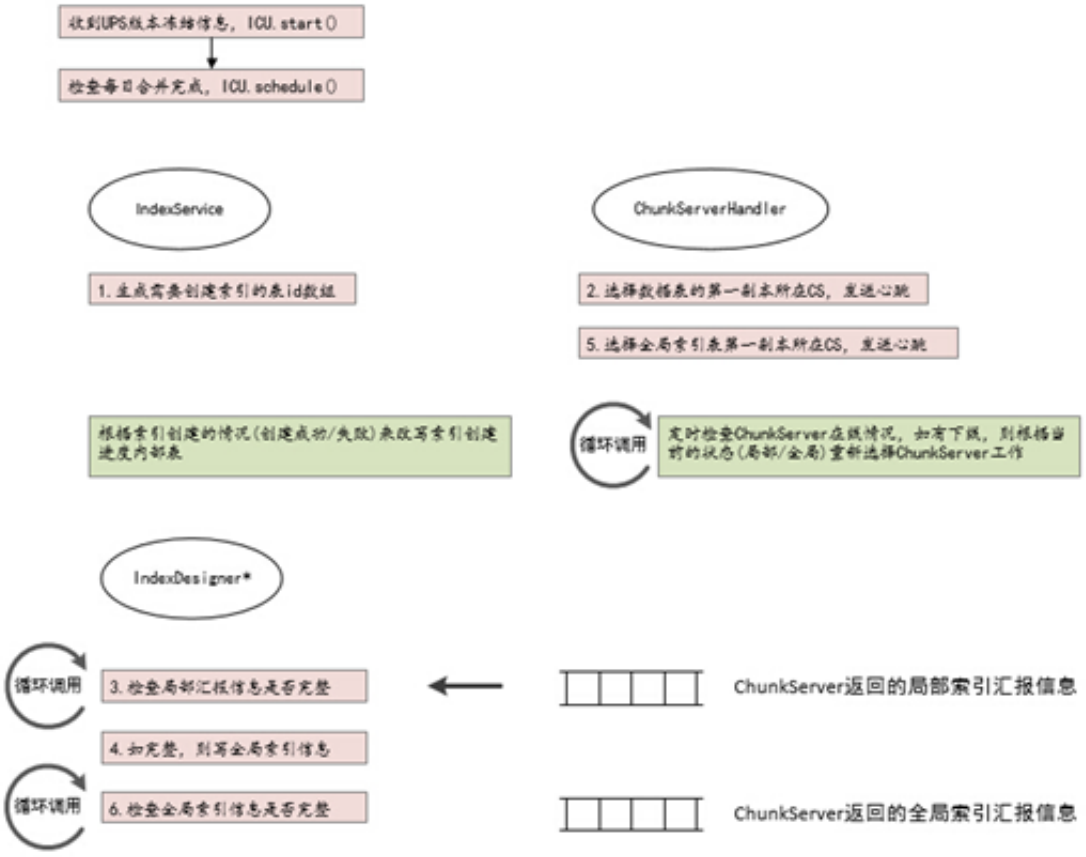
```
    ObTabletHistogramMeta data_holder_[ObTabletHistogramManager::MA
X_TABLET_COUNT_PER_TABLE];
    //store histogram report info data
    common::ObArrayHelper<ObTabletHistogramMeta> data_meta_;
    //meta of report info
    ObTabletHistogramManager *hist_manager_;
    //manage report info
    rootserver::ObRootServer2* root_server_;
    //use this link field to get root table lock
    uint64_t table_id_;
    uint64_t index_tid_;
```

data_meta和root_table存在对应的联系，以此检查汇报信息是否收齐，简单来说，其关系如下图所示：

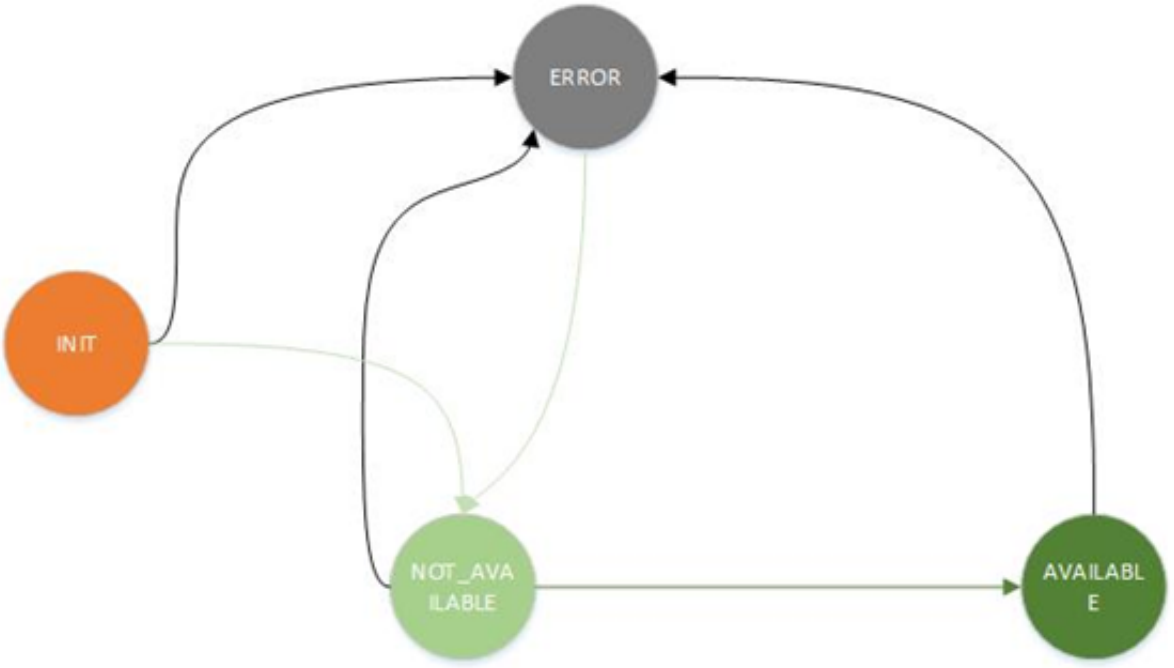


2.9.2.4.2 关键算法

2.9.2.4.3 流程



流程已在章节前作出描述，在整个索引的生存周期内，其状态可能发生如图的转化：



- INIT状态：刚创建完索引时，索引状态为INIT，需要等待一次合并完成后才会转换为ERROR或者NOT_AVAILABLE状态，从而间接转换为AVAILABLE状态；
    - NOT_AVAILABLE：每次列校验和检查成功，都会尝试把索引表的状态修改为NOT_AVAILABLE，NOT_AVAILABLE是一种临时状态，最后会由定时任务将状态改为AVAILABLE，或者ERROR状态(当发生错误时)

- ERROR：索引变成ERROR状态一般是由于建静态索引超时或者列校验和检查失败；构建完静态索引或者每次合并完成都会检查列校验和，如果索引表的列校验和和数据表的列校验和不一致，将索引状态改为ERROR；
- AVAILABLE：只能由NOT_AVAILABLE转换而来，处于该状态的索引才可用，select查询优化使用；

2.9.2.5 信息汇报子模块设计

2.9.2.5.1 局部索引信息汇报

收集信息

根据RS切分range的规则，需要在CS端按照采样频率来收集相应行数的range范围，以便RS端进行全局索引数据构建的range切分，所以需要在CS端收集按照索引列为第一主键的range信息并且构造样例的直方图信息发送给RS。

构建汇报信息

在构建局部索引时，会以索引列为第一主键进行排序，而这时候恰好可以按照采样频率收集相应的range范围，所以在从sort操作符中取出数据的时候就按照采样频率来取数据，构建相应的range并存入ObIndexReporter的ObTabletHistogram中。在一个tablet的数据都采样完毕之后，将对应的一个tablet采样信息存入ObIndexReporter的ObTabletHistogramReportInfo中。当所有的tablet都取完，再将相应的信息发送给RS。此处要注意的是tablet的range构造，从MIN到MAX。

汇报信息

当CS上所有的原表的tablet信息都收集完毕之后，就会进行信息汇报，将range信息发送给RS。

2.9.2.5.2 全局索引信息汇报

收集信息

在构建全局索引时，因为构建了新的索引表信息，增加了tablet，而这些tablet信息没有汇报到RS上面去，所以需要在构建全局索引的时候构建索引表tablet信息，并且汇报给RS。当汇报失败次数达到默认次数时，不进行重新汇报。

构建汇报信息

构建全局索引的时候会保存此时构建的tablet信息，这时候对这些信息进行收集并构建tablet，然后将其更新到TabletImage中去。

汇报信息

使用原来合并的汇报流程处理现在的汇报流程，但是为了减少网络交互，只需要将索引表的信息汇报出去。

2.9.2.6 列校验和子模块设计

2.9.2.6.1 结构

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。

考虑到索引表保存列只是原表的全部列的一个子集合，因此列校验和只保存三个部分：主键列，索引列，冗余列。此外，列校验和以字符串形式保存，形式如下：

column_id:checksum,coulumn_id:checksum,column_id:checksum……

其中包括索引列、主键列与冗余列的列ID号和对应的列校验和。

列校验和的保存采用内部表管理的方法。在实现时，提供了一张新建的内部表：

表名为：__all_column_checksum_info ，表id为801

第一主键为主表的id或索引表的id

第二主键为主表或索引表的集群id

第三主键为计算时的合并版本号

第四主键为对应的tablet的range

数据列column_checksum为计算得到的列校验和

2.9.2.6.2 关键算法

列校验和的计算主要分为原表列校验和计算与索引表列校验和计算，而且都可以分为两种情况。在第一次建立索引表时，由于索引表在合并前无法使用，也无法计算其对应的列校验和。因此，我们将第一次计算索引表列校验和的时机设定在构建索引表全局索引的时候。在确定全局索引tablet的range的情况下，可以在构建该tablet数据时一行一行地计算该tablet的列校验和。

而对于原表的列校验和的计算，由于考虑到在原表没有增量数据的情况下，无法将每一行的数据读出并计算，因此将该部分的计算保留在构建索引表的局部索引时。无论原表数据是否有变化，在构建局部索引的数据时都需要对其进行读取，使得计算原表的列校验和可以实现。

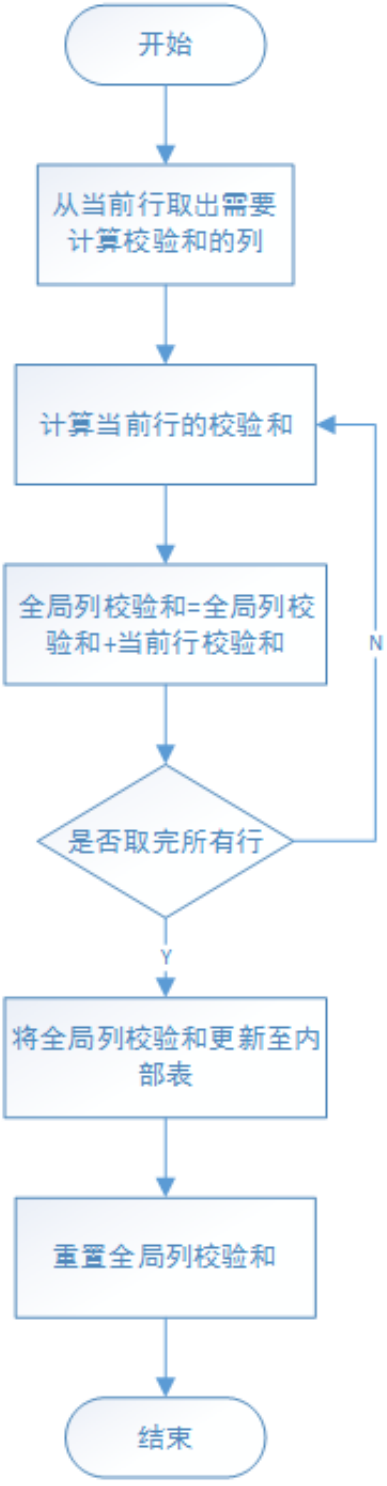
在第一次构建成功后，计算的流程则按原OB合并流程进行。

列校验和算后会通过构建mutator的方式更新至内部表。考虑到在tablet没有增量数据的情况下不会计算列校验和，因此，在没有增量数据的情况下将发送请求给RS，并通过nb_access接口将原表的列校验和检索回来进行更新，即只更新一个版本号version。

列校验和的校对安排在两个时机：（1）第一次完成列校验和后，在rootserver的IndexControlUnit中，在完成全局索引建立时进行判断。（2）在日常合并完成时，即在daily_merge_checker中进行列校验和的校对，确认索引表在本次合并后是否可用。

2.9.2.6.3 流程

二级索引列校验和的计算使用CRC校验，OB已经提供了计算接口（行校验和使用的也为该接口）。计算过程按行计算，每取到一个行则按上述定义，逐列检查，保留每一个元组的列校验和。而后将计算结果加到该表维护的全局列校验和上，得出最终结果。



3 模块接口

3.1对外接口

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=r
ootserver;
```

时间一秒作单位，默认限制是1800s。

4 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。