**复合列功能设计与实现**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **文档版本** | **修订章节** | **修订原因** | **修订日期** | **修订人** |
| **1** | 0.1 | 全文 | 新建 | 2011-8-17 | 晓楚、无施 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

目录

[2 概述 2](#_Toc301359165)

[3 复合列基本概念 2](#_Toc301359166)

[3.1 词汇定义 2](#_Toc301359167)

[3.2 复合列用途 2](#_Toc301359168)

[3.2.1 复杂运算 2](#_Toc301359169)

[3.2.2 条件过滤 2](#_Toc301359170)

[3.2.3 例子 3](#_Toc301359171)

[4 复合列求值 3](#_Toc301359172)

[4.1 表达式解析 4](#_Toc301359173)

[4.1.1 词法分析 4](#_Toc301359174)

[4.1.2 语法分析 5](#_Toc301359175)

[4.2 求值 6](#_Toc301359176)

[4.2.1 概述 6](#_Toc301359177)

[4.2.2 数据结构 7](#_Toc301359178)

[5 复合列与现有代码整合 8](#_Toc301359179)

[5.1 客户端 8](#_Toc301359180)

[5.2 mergeserver反序列化 9](#_Toc301359181)

[5.3 mergeserver请求处理流程 9](#_Toc301359182)

[5.3.1 复合列作用于原始行 10](#_Toc301359183)

[5.3.2 复合列作用于aggregate行 12](#_Toc301359184)

[5.4 ChunkServer 13](#_Toc301359185)

[5.5 其它细节问题 14](#_Toc301359186)

[5.5.1 查询不返回 14](#_Toc301359187)

[5.5.2 复合列index管理 14](#_Toc301359188)

# 概述

本文档的主要目的包括：（1）说明复合列的基本概念、（2）说明复合列实现方法、（3）说明在MergeServer中整合复合列的方法。

# 复合列基本概念

## 词汇定义

**基本列**：原始列(直接从oceanbase中读出的数据)和groupby之后的aggregate列统称基本列。

**运算符**：包括逻辑运算符（AND，OR，NEG，IS）、数学运算符（ADD, SUB, MUL, DIV）、条件运算符（LT，LE，EQ，NE，GE，GT，LIKE）等。运算符之间具有一定的优先级，括号可以用来调整运算符优先级。

**复合列**：指的是一个或多个基本列与常数之间通过运算符进行运算得到的列。

## 复合列用途

从作用对象来说，复合列可以作用于原始row(直接从oceanbase中读出的数据)，也可以作用于groupby之后的数据，叫做aggregate列。

从目的来说，复合列有两个目的：复杂运算，条件过滤。条件过滤作用于原始列的时候，对应了sql语言的where条件；当条件过滤作用于aggregate列的时候，对应于sql语句中的having。

### 复杂运算

传统SQL中支持将多个列之间的复合计算结果作为一个新列来返回给用户，OB 0.3中支持类似功能。根据上面复合列的定义，查询中支持将多个简单列进行复合运算当作查询结果，形如：

(a+b\*c) as result1

(SUM(a) + SUM(b) \* 0.2) as result2

具体的用户API接口请参考最新的《C++客户端》，API使用示例请参考《C++客户端应用 - SQL语句与OB语句转化示例》。

### 条件过滤

0.3版本的条件过滤实现方式与0.2有较大不同。0.3版本充分利用了复合列支持来提供丰富的条件过滤功能。在0.3版本中，条件表达式求值可以利用复合列实现：条件表达式与其它数学表达式本质是相同的，只不过它的求值结果是一个TRUE/FALSE逻辑值。

一个filter中支持多个逻辑操作，逻辑操作之间可以通过AND ，OR连接；支持多个filter串联（即多次调用add\_filter方法），多个filter之间是and的关系。例如：

Filter1 = columnA < columnB **AND** column A > columnC **OR** columnA = 0

Filter2 = columnA < as\_column1 **OR** columnA > columnD

Filter3 = columnD like columnE

Final\_Filter = Filter1 **AND** Filter2 **AND** Filter3

### 例子

成绩单（Score）格式：

**学号 班级 姓名 语文 数学 英语**

001 01 王二 90 80 70

002 01 张三 80 80 80

003 02 李四 70 80 90

select 班级, sum(语文)/count(语文) as 语文平均分

from 成绩单group by 班级 having 语文平均分 > 80;

班级 语文平均分

01 85

ObGroupByParam groupby\_param;

groupby\_param.add\_groupby\_column("班级");

groupby\_param.add\_aggregate\_column("语文", "a", SUM);

groupby\_param.add\_aggregate\_column("语文", "b", COUNT);

groupby\_param.add\_column("a/b", "语文平均分");

groupby\_param.add\_filter("语文平均分 > 80");

ObScanParam scan\_param;

scan\_param.add\_column("班级");

scan\_param.add\_column("语文",false);scan\_param.set\_group\_by\_param(groupby\_param);

其中“sum(语文)/count(语文)”为一个复杂运算，“语文平均分 > 80”为一个条件过滤。

# 复合列求值

复合列提供给客户端的接口为字符串接口，需要通过表达式解析和表达式求值两步才能实现复杂运算和条件过滤。

## 表达式解析

假设用户调用了下面一个函数来获得一个复合值：

scan\_param.add\_column("`Col1` - ((`Col2` + `Col3`) / `Col4`)");

我们需要将其解析成便于计算机处理的式子。表达式解析是指将人类可读的中缀表达式转化成便于计算机运算的后缀表达式。例如：

|  |  |
| --- | --- |
| 中缀表达式 | Col1 - ((Col2 + Col3) / Col4) |
| 后缀表达式 | Col1 Col2 Col3 + Col4 / - |

为了实现这个过程，首先要用词法分析识别出中缀表达式中的每一个操作符和操作数，然后再用语法分析将中缀表达式按照运算优先级的要求转化成后缀表达式。

### 词法分析

词法分析的主要用途是从左至右逐个扫描输入字符串，产生一个个单词序列，用以语法分析。执行词法分析工作的程序称为词法分析程序。

词法分析程序可以有两种设计方法：一种是一次性将所有单词序列分析出来并存储在中间文件中，传递给语法分析程序；另一种是将词法分析程序设计成一个子程序，每当语法分析需要一个单词时就调用此程序。第二种方法可以省掉中间文件。

OB中词法分析程序的输出包括：

|  |  |
| --- | --- |
| 输出类别 | 输出内容 |
| 标识符 | 列名 |
| 常数 | 整数，浮点数，true，false，null，字符串 |
| 运算符 | +, -, \*, /, 比较运算符, is, like, AND, OR |
| 分界符 | #, `, ‘, 空格 |

为了消除二义性，便于词法分析，特定义如下规则：

|  |  |
| --- | --- |
| 规则 | 说明 |
| 列名规则 | 必须用反引号(`)括起来，列名内仅不允许出现反引号，例如 `$\_column\_1` |
| 字符串规则 | 必须用单引号(')括起来，字符串内出现单引号需用反斜线(\)**转义**，反斜线自身亦用反斜线转义，例如'尿不湿'， 'it\'s me' (表示it's me) 'C:\\dir' (表示C:\dir) |
| 时间串规则 | 必须用井号(#)括起来，年月日用横线(-)相连，时分秒用冒号(:)相连，年月日和时分秒之间用空格分隔，例如 #2012-23-14 10:32:01# |
| 数字 | 仅支持十进制整数和浮点数，例如 10， 0.5 |
| 符号 | 支持<, <=, =, !=, >, >=, like, is, (, ), +, -, \*, /, and, or， 不区分大小写 |
| 二进制串 | 以b/B开头，数字部分为十六进制，并用单引号括起，不区分大小写。例如 b 'efA0231a' B'efA0231a' |

表达式规则稍加总结一下：**列名**用``括起来，**字符串常量**需要用单引号(')括起来，**datetime**用##括起来，**数字**直接书写。例如:

select `my\_int` from `test` where `my\_time` > #2010-02-11# and `my\_char`='a' and `my\_int`>10 or `my\_binary`=b'1a1aa1'

### 语法分析

语法分析有自底向上分析法和自顶向下分析法两种。自底向上分析法是一种移进-规约过程，当分析的栈顶符号串形成句柄时就采取规约动作，规约动作比较简单，因而自底向上分析法的关键问题是在分析过程中如何定句柄。自顶向下分析法也称为面向目标的分析法，也就是从文法的开始符号出发企图推导出与输入的单词串完全相匹配的句子，若输入串是给定文法的句子，则必能推出，反之必然出错。

算符优先分析法是一种简单直观、特别方便于表达式分析，易于手式实现的方法。算符优先法只考虑算符之间的优先关系，它是**一种自底向上的归约过程**，但这种归约未必严格按照句柄归约。它是一种不规范归约法。

算符优先分析法可有效地将中缀表达式转换成后缀表达式。其关键在于确定运算符的优先级：

OB运算符优先级表

|  |  |  |
| --- | --- | --- |
| 优先级 | 内容 | 备注 |
| 1 (最高) | () |  |
| 2 | \* / | 暂不支持% |
| 3 | + - |  |
| 4 | > >= < <= = !=  IS LIKE | is运算符主要用于判断is NULL条件 这里=用于where子句，等价于C语言的比较运算符== |
| 6 | AND | 为了兼容MySQL，AND优先级高于OR |
| 7 | OR |  |

MySQL运算符优先级参考：  
<http://dev.mysql.com/doc/refman/5.0/en/operator-precedence.html>

暂未支持的内容包括：

正数、负数表示，如-3, +12，解决方法是写成（0-3）, (0+12)

取模运算，如a % 3

逻辑非运算，如!23.0

位运算，如^,~,>>, <<, &, |等

为了实现方便，为每一种运算符和操作数分配了内码：

内码表

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算符 (sym) | 内码 (code) | 优先级 (prio) | 算符 (sym) | 内码 (code) | 优先级 (prio) | 算符 (sym) | 内码 (code) | 优先级 (prio) |
| （ | 1 |  | > | 11 |  | 列名 | 50 |  |
| ） | 2 |  | >= | 12 |  | 字符串 | 51 |  |
| \* | 3 |  | is | 13 |  | 时间 | 52 |  |
| / | 4 |  | like | 14 |  | 整数 | 53 |  |
| + | 5 |  | and | 15 |  | 浮点数 | 54 |  |
| - | 6 |  | or | 16 |  | Null | 55 |  |
| < | 7 |  | not | 17 |  | True | 56 |  |
| <= | 8 |  | % | 20 |  | False | 57 |  |
| = | 9 |  |  |  |  |  |  |  |
| != | 10 |  |  |  |  |  |  |  |
| 备注：支持二进制串(ObString)，内码同字符串。 | | | | | | | | |

有了以上内容后，就可以根据算符优先(Operator Precedence)算法实现对表达式的解析。

为了在解析过程中发现类型的不匹配情况，还可以对算符优先算法稍加扩展，实现一个辅助的类型检查模块，每次规约时都进行一次类型匹配检查。

## 求值

### 概述

后缀表达式（又称逆波兰式）是表达式求值中经常采用的一种数据结构。它的核心思想是将带有运算优先级的普通的表达式（中缀表达式）表达成对计算机友好的，没有运算优先级的，可以从左到右顺序计算的表达式。具体参考维基百科相关词条（英文：<http://en.wikipedia.org/wiki/Reverse_Polish_notation> 中文：[http://zh.wikipedia.org/wiki/逆波兰表示法](http://zh.wikipedia.org/wiki/%E9%80%86%E6%B3%A2%E5%85%B0%E8%A1%A8%E7%A4%BA%E6%B3%95)）。

复合列求值本质就是表达式求值。首先将复合列表达式按照运算规则转化成后缀表达式，然后采用后缀表达式求值。采用后缀表达式求值有两方面的优势：1、后缀表达式易于序列化 2、易于支持新的操作符，可扩展性好。

下面举例说明如何用后缀表达式对复合列进行求值：

对于前面出现的

Col1 Col2 Col3 + Col4 / -

其后缀表达式求值过程如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 操作 | 堆栈 | 注释 |
| Col1 | 入栈 | Col1 |  |
| Col2 | 入栈 | Col1, Col2 |  |
| Col3 | 入栈 | Col1, Col2 ,Col3 |  |
| + | 加法运算 | Col1, (Col2+Col3) | Col2 Col3出栈，运算结果(Col2+Col3)入栈 |
| Col4 | 入栈 | Col1, (Col2+Col3) ,Col4 |  |
| / | 除法运算 | Col1, ((Col2+Col3)/Col4) |  |
| - | 减法运算 | Col1-((Col2+Col3)/Col4) |  |

### 数据结构

后缀表达式求值类ObPostfixExpression负责存储和计算复合列表达式。

/\* 后缀表达式 \*/

class ObPostfixExpression

{

/// 记录column、常数、字符串之间的运算关系  
/// column中包含有row\_index信息。  
public:

  ObPostfixExpression();

  ~ObPostfixExpression();

  ObPostfixExpression(const ObObj \*expr);

  int set\_expression(const ObObj \*expr)();

**int** calc(ObObj &composite\_val,   
           const ObCellArray & org\_cells,  
           const int64\_t org\_row\_beg,   
           const int64\_t org\_row\_end  
 );  
};

中缀表达式负责存储中缀表达式，并支持将中缀表达式转化成后缀表达式

/\* 中缀表达式 \*/

**class** ObInfixExpression  
{  
public:

/\* @param expr:未经解析的中缀字符串表达式 \*/

ObInfixExpression(const ObString &expr);

ObInfixExpression();

inline int set\_expression(const ObString &expr);

inline int get\_expression(ObString &expr);

~ObInfixExpression();

};

/\* 复合列 \*/

**class** ObCompositeColumn  
{  
public:

ObCompositeColumn();

~ObCompositeColumn();

/\*构造复合列，其内容为中缀表达式，用于客户端与服务器的交互\*/

ObCompositeColumn(**const** ObString & expr,

constObString & as\_column\_name,

constint64\_t as\_column\_idx = -1);

/\*构造复合列，其内容为后缀表达式，用于服务器之间的交互\*/

ObCompositeColumn(constObObj \*expr,

constObString & as\_column\_name,

constint64\_t as\_column\_idx);

int set\_expression(constObString & expr,

constObString & as\_column\_name,

constint64\_t as\_column\_idx);

int set\_expression(constObObj \*expr,

constObString & as\_column\_name,

constint64\_t as\_column\_idx);

inline int  get\_infix\_expression(ObString &expr);

inline int64\_t get\_as\_column\_idx()const ;

inline ObString get\_as\_column\_name()const ;  
 /\* 计算本复合列的值 \*/

**int** calc\_composite\_val(ObObj &composite\_val,                           const ObCellArray & org\_cells,  
                           const int64\_t org\_row\_beg,   
                           const int64\_t org\_row\_end  
                           );  
};

# 复合列与现有代码整合

## 客户端

客户端提供如下复合列相关接口：

**class** ScanParam  
{  
  **public**:  
    /// defautl is\_return is true  
    **int** add\_column(**const** **char**\* expression, **const** **char**\* as\_name);  
    **int** add\_column(**const** **char**\* expression, **const** **char**\* as\_name,   
 **const** **bool** is\_return);  
  
    **int** add\_filter(**const** **char**\* expression);  
};

**class** ObGroupByParam  
{  
  **public**:  
    /// defautl is\_return is true  
    **int** add\_column(**const** **char**\* expression, **const** **char**\* as\_name);  
    **int** add\_column(**const** **char**\* expression,   
 **const** **char**\* as\_name, **const** **bool** is\_return);  
  
    **int** add\_filter(**const** **char**\* expression);  
};  
expression、as\_name、is\_return等参数序列化后传递给MergeServer。

## mergeserver反序列化

MergeServer接到客户端请求后，首先deserialize所有数据结构，然后在decode阶段遇到复合列标示时如果发现表达式为中缀字符串，则先将其转化成后缀表达式，再将表达式中的列名转成列index。

## mergeserver请求处理流程

mergeserver处理一个请求涉及以下过程：merge/join、属性过滤、group by、order by、limit。如果顺序的执行这些流程需要把所有的原始数据都存放在mergeserver的内存中。举个极端的例子：count全表的记录。如果顺序执行，首先把所有的记录都执行merge、join，那么需要把全表的数据读入内存。另外一个例子是按照主key排序，取limit，如果把所有的记录都读入再limit，显然内存也是会撑爆的。所以，这里必须针对不同的请求执行不同的路径。仅仅存在orderby的情形下才需要读入数据的全集。

ms请求处理流程如下：

1. merge/join，得到的结果是2M大小的ObCellArray，merge\_join\_result\_
   1. 这里可以加入判断，如果请求中没有属性过滤、没有group、没有orderby，并且scan的范围已经结束，那么请求就可以判定为执行结束。减少后续的内存拷贝。加入这个判断可以减少最朴素请求的一次内存拷贝。暂时先不做这样的优化。
2. **复合列计算1**：计算所有取值与GroupBy无关的复合列（note: 为与SQL中的运算优先级where > 聚合函数 > having兼容，属性过滤阶段就需要计算merge join filter中的复合列）
3. **where属性过滤**：从merge\_join\_result\_中迭代出每一行，判断是否通过属性过滤；按照这种方法实现，不需要在内存中保存所有的原始记录的内容
4. GroupBy：把通过属性过滤的行加入group\_by\_result\_中，借用hash表
   1. 如果原始请求中有groupby参数，这一步将执行olap计算；
   2. 如果原始请求中没有groupby参数，这一步仅仅是把当前行从merge\_join\_result\_拷贝到group\_by\_result\_
   3. 在每次将通过属性过滤的行加入group\_by\_result\_后，会判断请求处理是否结束，如果没有orderby只有limit的情况，那么通过这一步判断可以终止后续的读操作（如果发现没有orderby仅有limit，在满足limit条件的时候可以终止迭代merge\_join\_result\_中的结果）。
5. **复合列计算2**：计算groupby之后结果的复合列，需要计算所有取值与GroupBy有关的复合列。
6. **having属性过滤**，在groupby之后的结果上执行属性过滤
7. OrderBy：执行orderby
8. 按照limit返回结果

### 复合列作用于原始行

#### 接口与协议

复合列通过以下接口作用于原始行：

class ObScanParam

{

public:

/// complicated calculation

int add\_column(const ObString & expression, const ObString & as\_name,

const bool is\_return = true);

/// prop filter

int add\_filter(const ObString & expression);

};

属性过滤和复杂运算，对于内部而言都是相同的，只是属性过滤的结果一定不返回，而复杂运算可能返回可能不返回。因此，内部实现将add\_filter(const ObString & expression);替换为add\_column(expression, as\_name, false);这样ObScanParam可以使用一个数组管理作用于原始行的复合列（包括属性过滤）。属性过滤交给0.2版本的ObSimpleFilter来完成。这样，复合列就负责运算，ObSimpleFilter负责过滤。这样，调用add\_filter(const ObString & expression);转换为两个步骤：add\_column(expression, “\_\_\_ob\_basic\_comp\_filt\_i”, false)其中i表示当前查询请求中的第几列；add\_cond(“\_\_\_ob\_basic\_comp\_filt\_i”，EQ, TRUE)。这样把计算的逻辑和过滤的逻辑分开，而过滤的逻辑可以使用已有逻辑。为了保证兼容性，ObSimpleFilter必须保留。

ObObj增加Bool类型。

协议如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 复合列参数标示 | comp\_column1: obj(column\_name/column\_id) | obj(is\_return) | obj(expr) obj(expr)obj(expr)… |
|  | comp\_column2: obj(column\_name/column\_id) | obj(is\_return) | obj(expr) obj(expr)obj(expr)… |
| … | | |
| where复合属性过滤标示 | comp\_column1: obj(column\_name/column\_id)  \_\_\_ob\_basic\_comp\_filt\_1 | obj(is\_return) =false | obj(expr) obj(expr)obj(expr)… |
|  | comp\_column2: obj(column\_name/column\_id)  =\_\_\_ob\_basic\_comp\_filt\_2 | obj(is\_return) =false | obj(expr) obj(expr)obj(expr)… |
| … | | |

第一列为as\_name,第二列为is\_return标记，第三列为expression。当数据为从client到ms时，expression为一个字符串，列中只有一个obj；当数据从ms到ups时，expression已经被转码为一组算符和算子，列中包含多个obj。

后缀表达式的序列化结构需要解决列id与常数项之间的二义性问题。经过了解析的后缀表达式由一组obj组成，其中可能包含列id（column\_id），常数项(const obj)和运算符(const obj)，需要定义一种规则以区分三者，否则无法进行后缀表达式求值。如果不以区分，列id用整数表示，常数项中也正好使用了整数，则会在后缀表达式求值阶段产生二义性。 最简单的后缀表达式序列化结构如下：Obj(type),Obj(value),Obj(type),Obj(value), Obj(type),Obj(value)… 也就是使用type-value对来描述每一个表达式中的元素，Obj(type)表示其后的Obj（value）是列id、常数还是操作符，Obj（value）则是具体的表达式元素。该设计的优点是简单清晰，并具有较好的扩展性，缺点是增加了一倍的存储。

#### ms代码整合：复合列作用于原始行

按照上面的处理方式，复合列对于ms而言，处理很简单，就是在执行merge、join后计算所有的复合列就ok了。Merge、join本身的逻辑并不涉及到太多的变化。修改点如下：

void oceanbase::mergeserver::ObMergeJoinOperator::move\_to\_next\_row(

const ObGetParam & row\_spec\_arr,

const int64\_t row\_spec\_arr\_size,

int64\_t &cur\_row\_beg,

int64\_t &cur\_row\_end);

修改为

void oceanbase::mergeserver::ObMergeJoinOperator::move\_to\_next\_row(const ObGetParam & row\_spec\_arr,

const int64\_t row\_spec\_arr\_size,

int64\_t &cur\_row\_basic\_beg,

int64\_t &cur\_row\_basic\_end,

int64\_t &cur\_row\_end);

其中cur\_row\_basic\_beg表示当前行basic列开始的偏移，cur\_row\_basic\_end表示当前行basic列结束的偏移，cur\_row\_end表示当前行basic列加上复合列结束的偏移。对于get和没有复合列的scan请求，cur\_row\_basic\_end == cur\_row\_end。

int oceanbase::mergeserver::ObMergeJoinOperator::merge\_next\_row(int64\_t &cur\_row\_beg, int64\_t &cur\_row\_end)；

改函数现有逻辑保持不变，merge部分的逻辑把对cur\_row\_end的依赖修改为cur\_row\_basic\_end。

在merge结束后，即函数的尾部，如果cur\_row\_end != cur\_row\_basic\_end，则剩余的部分填入value为null的cell，cell的column\_id初始化为OB\_INVALID\_ID。初始为OB\_INVALID\_ID可以保持join部分的逻辑不做任何修改。

int oceanbase::mergeserver::ObMergeJoinOperator::get\_next\_rpc\_result();

在调用do\_merge\_join结束后，（在函数的末尾）调用复合列的功能类计算所有行的复合列的值。

这样，作用于原始行的复合列计算就完成了。属性过滤交在prepare\_final\_result\_process\_intermediate\_result\_函数中给ObSimpleFilter来做。

### 复合列作用于aggregate行

#### ms代码整合：复合列作用于aggregate行

复合列作用于aggregate行的时候，表述方法与作用于原始行相同。通过修改ObGroupByParam实现。计算方式也相同。

整合部分涉及到以下修改点：

int oceanbase::mergeserver::ObScanMergeJoinAgentImp::prepare\_final\_result\_process\_intermediate\_result\_()；

在函数的末尾，也就是groupby计算结束后，遍历ObGroupByOperator中的所有行，里面通过hashtable的方式组织，计算每个aggregate行的复合列结果，并进行条件过滤。对于没有通过过滤的行，将该group的结果从hashtable中干掉。

执行聚合运算后结果都保存在ObGroupByOperator中，Having复合属性过滤需要基于ObGroupByOperator中的聚合结果进行，不符合条件的行都会从中删除，形成一张新的结果表，但是这样做会增加一次拷贝。为了避免拷贝，没有通过过滤的行只会从hashtable中删除，随后的orderby操作基于hashtable中的剩余行进行。

#### 接口与协议

ObGroupbyParam{

public:

/// complicated calculation

int add\_composite\_column(const ObString & expression, const ObString & as\_name,

const bool is\_return = true);

/// prop filter

int add\_filter(const ObString & expression);

};

Groupby后的属性过滤也交给ObSimpleFilter来完成。这样，复合列就负责运算，ObSimpleFilter负责过滤。这样，调用add\_filter(const ObString & expression)转换为两个步骤：add\_composite\_column(expression, “\_\_\_ob\_groupby\_comp\_filt\_i”, false)其中i表示当前查询请求中的第几列；add\_cond(“\_\_\_ob\_groupby\_comp\_filt\_i”，EQ, TRUE)。这样把计算的逻辑和过滤的逻辑分开，而过滤的逻辑可以使用已有逻辑。

ObGroupByOperator{

public:

/// filter row

int filter\_aggregated\_row(const ObCellArray &cells, const int64\_t row\_beg, int64\_t row\_end, const ObSimpleFilter & filter, bool &result);

};

Having复合属性过滤中出现的列引用必须出现在Groupby下的复合列参数标示中。而复合列参数标示中出现的列必须出现在聚集函数参数标示中。这也就规定了在Groupby中必须先计算聚集函数，整个表处理完后才能开始计算Having中出现的复合列，在复合列计算过程中可以同步进行having复合属性过滤。

协议如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| GroupBy参数标示 |  | | | |
|  | 复合列参数标示 | comp\_column1: obj(column\_name/column\_id) | obj(is\_return) | obj(expr) obj(expr)obj(expr)… |
|  | comp\_column1: obj(column\_name/column\_id) | obj(is\_return) | obj(expr) obj(expr)obj(expr)… |
| … | | |
| having复合属性过滤标示 | comp\_column1: obj(column\_name/column\_id)  =\_\_\_ob\_groupby\_comp\_filt\_1 | obj(is\_return) =false | obj(expr) obj(expr)obj(expr)… |
|  | comp\_column1: obj(column\_name/column\_id)  =\_\_\_ob\_groupby\_comp\_filt\_2 | obj(is\_return) =false | obj(expr) obj(expr)obj(expr)… |
| … | | |

## ChunkServer

复合列传入到ChunkServer时各列均已从字符串转成了对应的id/index。不使用name。

## 其它细节问题

### 查询不返回

查询不返回功能是0.3中新加入的，协议设计中需要在每一列中新加入一项指示是否返回。

查询不返回由

int oceanbase::mergeserver::ObScanMergeJoinAgentImp::next\_cell()；

成员实现。

用一个数组记录每行那些cell需要返回。并记录当前行已经返回到第几个cell。调用next\_cell的时候，跳过所有不需要返回的列。当前行访问结束后，把当前行访问到第几个cell的记录归0.

### 复合列index管理

每个基本列都会在当前行中有对应列，用于存储从ups和cs取得的数据。基本列通过index作为下标在当前行中寻址数据，index取值从0起，到查询包括的总基本列数basic\_column\_size为止，即0<=index<basic\_column\_size。复合列也在当前行中有对应列，用于存储复合计算结果。复合列也通过index作为下标在当前行中寻址数据，index取值范围为basic\_column\_size <= index < basic\_column\_size + composite\_column\_size。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 列名 | A | B | C | A+B  as C1 | A-B  as C2 | A\*B  as C3 | A+C1  as C4 | C2+C3  as C5 | C5-C4  as C6 |
| Index | 0 | 1 | 2 | **3** | **4** | **5** | **6** | **7** | **8** |

在decode阶段，通过名字查找的方式来翻译复合列计算中用到的column\_name，将其转换成index，此时的index取值范围可能为0 <= index < basic\_column\_size + composite\_column\_size。为了实现名字到index的翻译，可以使用一个hash map加速，为了简便起见当前版本不使用hash map，只是简单地使用顺序扫描查找。