**MergeServer功能升级复合列简介**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 | 2011-7-1 | 晓楚、无施、解伦 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

随着推广的进行，产品线对oceanbase提出了更多功能上的需求。本版本设计主要针对复合列。

# 复合列

**基本列**：OB Schema中定义的列称为基本列。注：聚合列(Aggregated Column)也看作是一种基本列，列名由as决定。

**复合运算符**：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。这里将优先级运算符(“(“, “)”)也当做复合运算符的一部分。

ObObj中的各个类型所支持的复合运算：

ObNullType, // 空类型

ObIntType, (ALL – {LIKE})

ObFloatType, (ALL – { AND，OR，NEG, LIKE})

ObDoubleType, (ALL – { AND，OR，NEG, LIKE})

ObDateTimeType, (LT，LE，EQ，NE，GE，GT)

ObPreciseDateTimeType, (LT，LE，EQ，NE，GE，GT)

ObVarcharType, (LIKE)

ObSeqType, (-)

ObCreateTimeType, (LT，LE，EQ，NE，GE，GT)

ObModifyTimeType, (LT，LE，EQ，NE，GE，GT)

ObExtendType

仅仅ObIntType、ObFloatType、ObDoubleType之间，ObCreateTimeType、ObModifyTimeType之间支持复合运算。

条件运算的结果是ObIntType;逻辑运算的结果是布尔型，在OB中用ObIntType表示，取值为0/1。

**复合列**：指的是一个或多个基本列与常数之间通过复合运算得到的列。

## 查询

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，通过在查询中支持复合列来实现。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等，具体的用户API接口请参考《C++客户端v1.4》，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：如果将条件运算符和逻辑运算符、算术运算符同等看待，可以发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逻辑值：TRUE/FALSE。

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

# 复合列求值

## 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：

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

后缀表达式求值过程：

|  |  |  |  |
| --- | --- | --- | --- |
| 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 ;

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  
                           );  
};

# 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. **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。
   3. 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\_
   4. 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. **having属性过滤**，在groupby之后的结果上执行属性过滤（风险：由于having属性过滤必须等到全部GroupBy计算完成后才能进行，为了得到having属性过滤后的结果需要增加一次拷贝）
6. OrderBy：执行orderby
7. 按照limit返回结果

# 序列化

## ObScanParam协议

在ObScanParam增加以下内容：(待定：返回列和非返回列如何区分较好？)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 基本参数标识 | 是否缓存 | Scan version Border flag | begin version | | end version | | 表名或表ID | Scan Range Border Flag | | Begin rowkey | | end rowkey | | 扫描顺序 | 大小限制 |
| 列参数标识 | 列名或者ID | | | | | | | | | | | | | | |
|  | 列名或者ID | | | | | | | | | | | | | | |
| … | | | | | | | | | | | | | | |
| 简单属性过滤参数标示 | left\_operand  列名或者ID | | | | | LogicOpType | | | | | right\_oprand  const Obj | | | | |
|  | left\_operand  列名或者ID | | | | | LogicOpType | | | | | right\_oprand  const Obj | | | | |
| … | | | | | | | | | | | | | | |
| GroupBy参数标示 |  | | | | | | | | | | | | | | |
|  | GroupBy列参数标示 | | | groupby column1: obj(column\_id/column\_name) | | | | | | | | | | | |
|  | | | groupby column2: obj(column\_id/column\_name) | | | | | | | | | | | |
| … | | | | | | | | | | | |
| Group返回列标示 | | | groupby return column1: obj(column\_id/column\_name) | | | | | | | | | | | |
|  | | | groupby return column2: obj(column\_id/column\_name) | | | | | | | | | | | |
|  | | | … | | | | | | | | | | | |
| aggregate function参数标示 | | | aggregate function type | | | | | as column name | | | | Aggregate function 作用的列（column\_id / column\_name) | | |
|  | | | aggregate function type | | | | | as column name | | | | Aggregate function 作用的列（column\_id / column\_name) | | |
| … | | | | | | | | | | | |
| 排序参数标识 | 列名或者ID | | | | | | 排序顺序 | | | | | | | | |
|  | 列名或者ID | | | | | | 排序顺序 | | | | | | | | |
| … | | | | | | … | | | | | | | | |
| 分页参数标识 | 起始偏移 | | | | | | 返回结果条数 | | | | | | | | |
| 数据结束标识 | | | | | | | | | | | | | | | |

|  |  |  |
| --- | --- | --- |
| 复合列参数标示 | comp\_column1: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… | |
|  | comp\_column2: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… | |
| … | |
| where复合属性过滤标示 | comp\_column1: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… | |
|  | comp\_column2: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… | |
| … | |
| GroupBy参数标示 |  | |
|  | GroupBy列参数标示 | groupby column1: obj(column\_id/column\_name) |
|  | groupby column2: obj(column\_id/column\_name) |
| … |
| Group返回列标示 | groupby return column1: obj(column\_id/column\_name) |
|  | groupby return column1: obj(column\_id/column\_name) |
| … |
| 聚集函数参数标示 | olap function参数1: obj(function type), obj(column\_name), obj(src\_column\_id/src\_column\_name) |
|  | olap function参数2: obj(function type) , obj(column\_name), obj(src\_column\_id/src\_column\_name) |
| … |
| 复合列参数标示 | comp\_column1: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… |
|  | comp\_column2: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… |
| … |
| having复合属性过滤标示 | comp\_column1: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… |
|  | comp\_column2: obj(column\_name/column\_id) obj(expr) obj(expr)obj(expr)… |
| … |

其中，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。

所有复合列相关字段都使用了相同的结构：

obj(column\_name/column\_id), obj(expr) obj(expr) obj(expr)…

如果第一个obj为column\_name则using\_name\_为真，后面的obj为varchar类型，表示一个原始的字符串型的**中缀表达式**；否则第一个obj为column\_id，using\_id\_为真，后面的obj数组表示经过解析的**后缀表达式**。Obj数组的个数可以任意，直到最后遇到一个obj.get\_ext()不为零的元素表示结束。

对于原始的字符串型的中缀表达式需要解决列名与字符串之间的二义性问题。定义一组规则如下：

* **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

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

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。