**OceanBase SQL物理运算符详细设计**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **编号** | **文档版本** | **修订章节** | **修订原因** | **修订日期** | **修订人** |
| **1** | 0.1 |  | 新建文档 | 2012/4/24 | 竹翁  天官 |
| **2** | 0.2 |  | 增加Distinct运算符；HashGroupBy中增加对支持count(distinct)的说明 | 2012/4/24 | 竹翁 |
| **3** | 0.3 |  | 增加集合运算符；增加外排或Hash桶转储时的外存文件结构描述；MergeDistinct功能由Sort运算符实现；讨论后的其他细节修订；各内存结构之间访问使用指针； | 2012/5/2 | 天官、  竹翁 |
| **4** |  |  |  |  |  |
| **5** |  |  |  |  |  |
| **6** |  |  |  |  |  |

目录

[1 目标 3](#_Toc323741503)

[2 名词解释 3](#_Toc323741504)

[3 物理运算符接口 3](#_Toc323741505)

[4 查询转换实例 6](#_Toc323741506)

[5 物理运算符 8](#_Toc323741507)

[5.1 TableScan 8](#_Toc323741508)

[5.2 Filter 8](#_Toc323741509)

[5.3 Limit 8](#_Toc323741510)

[5.4 Project 9](#_Toc323741511)

[5.5 Sort 9](#_Toc323741512)

[5.5.1 数据收集阶段 9](#_Toc323741513)

[5.5.2 输出阶段 12](#_Toc323741514)

[5.6 Scalar Aggregate 13](#_Toc323741515)

[5.7 GroupBy 13](#_Toc323741516)

[5.7.1 MergeGroupBy 13](#_Toc323741517)

[5.7.2 HashGroupBy 13](#_Toc323741518)

[5.8 Distinct 15](#_Toc323741519)

[5.8.1 MergeDistinct 15](#_Toc323741520)

[5.8.2 HashDistinct 15](#_Toc323741521)

[5.9 Join 16](#_Toc323741522)

[5.9.1 MergeJoin 16](#_Toc323741523)

[5.10 Union 21](#_Toc323741524)

[5.10.1 MergeUnion 22](#_Toc323741525)

[5.11 Intersect 24](#_Toc323741526)

[5.12 Except 25](#_Toc323741527)

[附录 26](#_Toc323741528)

[CS端的TabletScan伪代码 26](#_Toc323741529)

[MS上TableScan的伪代码 36](#_Toc323741530)

[GroupBy接口的伪代码 39](#_Toc323741531)

[MergeGroupBy的伪代码 40](#_Toc323741532)

[HashGroupBy的伪代码 41](#_Toc323741533)

[Join操作接口伪代码 42](#_Toc323741534)

本文档是淘宝OceanBase( <http://oceanbase.taobao.org> )开源数据库中执行SQL查询时所需物理运算符的详细设计。

# 目标

OceanBase要全面支持SQL。为了支持SQL查询，在进行SQL解析优化之后，查询处理器要生成物理执行计划（Physical Query Plan）。物理查询计划是由若干物理运算符（Physical Operator）组成的树形结构。本文定义了物理运算符的接口，并包含所有物理运算符的详细设计。

# 名词解释

**Update Server**：OceanBase的增量更新子系统，记录用户的修改，通常由一主一备或一主多备构成；简写为**UPS**；

**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；简写为**CS**；

**MergeServer**：OceanBase的用户接口子系统，接收用户的查询请求，并从对应ChunkServer上获得基线数据以及从UpdateServer上获得更新的增量，然后融合二者后返回用户；简写为**MS**；

**RootServer**：OceanBase的管理服务器，用于管理UpdateServer、ChunkServer和MergeServer等；简写为**RS**；

**列：**一个列由列ID(column\_id)及其值(column\_value)组成；

**行：**一个行由若干列组成，有些时候其中的部分列构成主键(row key)并且整个表按主键顺序存储，有些表，如select指令的结果，可以不包含主键；

**schema**：表的列的类型、值范围等以及该表与其他表的join等关系称为表的schema；

**表**：一个表由若干列(在schema中定义)和任意行组成，有些表的每一行都存储了schema中定义的每一列，这样的表是稠密的，如静态(基线)数据的表一般是稠密的；另外一些表行只存储了部分的列，例如UpdateServer中的表的修改增量，这样的表是稀疏的。

# 物理运算符接口

所有物理运算符都要遵循相同的接口。通过get\_next\_row方法，可以获得该物理运算符输出的下一行数据。输出是行可以作为下一个物理运算符的输入，也可以直接作为查询结果输出。

/// 一个行可以包含的最大元素/列数

**static** **const** int64\_t OB\_ROW\_MAX\_COLUMNS\_COUNT = 512;

/\*\*

\* 行描述

\*

\*/

**class** ObRowDesc

{

**public**:

/\*\*

\* 根据表ID和列ID获得该列在元素数组中的下标

\*

\* @param table\_id 表ID

\* @param column\_id 列ID

\* @param [out] 该列的下标

\* @return OB\_SUCCESS或错误码

\*/

int get\_idx(**const** uint64\_t table\_id, **const** uint64\_t column\_id, int64\_t &idx) **const**;

/\*\*

\* 根据列下标获得表ID和列ID

\*

\* @param idx

\* @param table\_id [out]

\* @param column\_id [out]

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** get\_tid\_cid(**const** int64\_t idx, uint64\_t &table\_id, uint64\_t &column\_id) **const**;

/// 一行中列的数目

int64\_t get\_column\_num() **const**;

/// 添加下一列的描述信息

**int** add\_column\_desc(**const** uint64\_t table\_id, **const** uint64\_t column\_id);

};

对于某个完成初始化的特定物理运算符的实例，它输出的行的RowDesc对象是不变的。所以，一般的，每个运算符对象包含一个RowDesc对象，每输出一个行前，只需要设置行的RowDesc指向运算符内部的RowDesc对象。add\_column\_desc用于构造RowDesc对象。

**class** ObRow

{

**public**:

ObRow();

~ObRow();

/// 赋值。浅拷贝，特别的，对于varchar类型不拷贝串内容

**void** assign(**const** ObRow &other);

/\*\*

\* 设置行描述

\* 由物理操作符在构造一个新行对象时初始化。ObRow的功能要使用到ObRowDesc的功能，多个ObRow对象可以共享一个ObRowDesc对象。

\* @param row\_desc

\*/

**void** set\_row\_desc(ObRowDesc &row\_desc);

/\*\*

\* 根据表ID和列ID获得cell

\*

\* @param table\_id

\* @param column\_id

\* @param cell [out]

\*

\* @return

\*/

**int** get\_cell(**const** uint64\_t table\_id, **const** uint64\_t column\_id, ObObj \*&cell);

/\*\*

\* 设置指定列的值

\*/

**int** set\_cell(**const** uint64\_t table\_id, **const** uint64\_t column\_id, **const** ObObj &cell);

**private**:

DISALLOW\_COPY\_AND\_ASSIGN(ObRow);

**private**:

**static** **const** int64\_t MAX\_COLUMNS\_COUNT = OB\_ROW\_MAX\_COLUMNS\_COUNT;

// data members

ObObj cells\_[MAX\_COLUMNS\_COUNT];

int16\_t cells\_count\_;

int16\_t reserved1\_;

int32\_t reserved2\_;

ObRowDesc \*row\_desc\_;

};

/// 物理运算符接口

**class** ObPhyOperator

{

**public**:

ObPhyOperator();

**virtual** ~ObPhyOperator();

/// 添加子运算符，有些运算符（例如join）可能有多个子运算符。叶运算符无子运算符。

**virtual** **int** add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator);

/// 打开物理运算符。申请资源，打开子运算符等。

**virtual** **int** open() = 0;

/// 关闭物理运算符。释放资源，关闭子运算符等。

**virtual** **int** close() = 0;

/\*\*

\* 获得下一行的引用

\* @note 在下次调用get\_next或者close前，返回的row有效

\* @pre 调用open()

\* @param row [out] 调用者不能修改行的内容

\*

\* @return OB\_SUCCESS或OB\_ITER\_END或错误码

\*/

**virtual** **int** get\_next\_row(const ObRow \*&row) = 0;

/\*\*

\* 打印物理运算符类型及其所有参数到字符串中，用于调试SQL

\* 注意，需要负责调用子运算符的to\_string()

\* @return OB\_SUCCESS或OB\_BUF\_NOT\_ENOUGH或其他错误码

\*/

**virtual** **int** to\_string(**char**\* buf, **const** int64\_t buf\_len) **const**;

**private**:

DISALLOW\_COPY\_AND\_ASSIGN(ObPhyOperator);

};

物理查询计划构造好后，执行过程为：

{

ObPhyOperator root\_operator = root\_operator\_; //根运算符

root\_operator->open();

ObRow \*row = NULL;

**while** (OB\_SUCCESS == root\_operator->get\_next\_row(row))

{

Output(row); //输出本行

}

root\_operator->close();

}

# 查询转换实例

对于查询：

Select t1.c1, t2.c1, sum(t2.qty)

From t1, t2

Where t1.c2 = t2.c2

And t1.c3 > 5

And t2.c3 < sin10°

Group by t1.c1, t2.c1

Having count(t2.qty) < 3

Order by t2.c1

Limit 2, 5;

根据下一节定义的物理运算符，我们将产生如下物理查询计划：

TableScan(table=t1, col={c1,c2,c3},

filter={c3>5})

TableScan(table=t2, col={c1,c2,c3,qty},

filter={c3<10°})

Sort(col={t1.c2})

Sort(col={t2.c2})

MergeJoin(type=INNER, cond={t1.c2=t2.c2})

HashGroupBy(groupby={t1.c1, t2.c1},

aggr={sum(t2.qty), count(t2.qty)})

Filter(cond={count(t2.qty)<3})

Project(col={t1.c1,t2.c1,sum(t2.qty)})

Limit(offset=2, count=5)

Sort(col={t2.c1})

# 物理运算符

## TableScan

全表扫描操作TableScan扫描一个指定的基本表，它是最基本的物理运算符。在MS端，扫描表的操作通过把扫描操作发送给组成表的各个tablet所在的CS来完成，MS把CS扫描的结果按照tablet range拼接起来作为输出。MS端控制每个CS上的并发扫描数，保持每个CS上任一时刻属于同一个TableScan操作的扫描数不超过K个。参见MS上TableScan的伪代码。

CS端负责一个tablet范围内数据的扫描。CS把数据从sstable文件中读出，根据rowkey查询相应行在UPS上是否有修改，如果有，把修改融合进来。如果被扫描的表中包含内联join的另一个表，则根据join列去查询UPS获得被内联行的最新数据，根据最新数据修改本行。最终得到的结果作为一行输出。在有内联join的情况下，为了减少与UPS间通信次数，Get操作要批处理，我们聚集若干条Get操作后用一个MultiGet向UPS请求数据。参见CS端的TabletScan伪代码。

作为优化，TableScan操作还要集成Filter，Project/Compute，Limit等操作。

## Filter

Filter操作符没有自己的临时结果，它总是向自己调用的操作符申请一条数据，如果这条数据不满足过滤条件，就继续申请下一条数据，直至申请到一条满足过滤条件的数据为止，然后把该条数据返回给自己的调用者。

Filter中的过滤条件一定是针对本行数据的，给定一条输入，就能判断是否满足条件。

Row

取下一条数据

不满足过滤条件

作为结果输出

## Limit

Limit操作符只返回计数在[m, n]范围内元组

1. 如果m没有设定，[m, n] = [1, n];
2. 如果n没有设定，[m, n] = [m, +∞)；
3. 如果m和n都没有设定，[m, n] = (-∞, +∞)

Row

取下一条数据

不在[m, n]之间

作为结果输出

## Project

Project的操作符输出的Row的schema是可以与输入不同的，所有Project操作符要有自己的临时结果。Project每读入一行，就可以生成一个新行。输出表达式根据本表达式对输入column的引用，计算出新的column值，作为结果返回给自己的调用者。

计算

Row

取下一条数据

作为结果输出

输出列表达式

New Row

## Sort

Sort操作符考虑了内排和外排（内存放不下）两种情况：当内存可以放下全部数据时，排序只有一路；当内存不能放下全部数据时，排序会是多路，中间结果刷到外存上。Sort分两个阶段：1. 数据收集阶段；2. 输出阶段

### 数据收集阶段

Sort操作是一个阻塞操作，需要取回所有需排序的元组之后才能输出，所以首阶段需要收集所有的数据。在这个过程中，Sort操作符有三个辅助结构，分别是紧密存储结构、排序列结构、辅助结构。

**紧密存储结构**：每条读进来的元组（Row），Sort都需要对它进行缓存。Sort读入展开格式的元组，在存储结构内把它存成紧密格式（紧密格式的具体描述请参看《KV cache》）。存储结构不能通过(row\_id, column\_id)进行随机访问，只能通过偏移或指针读出一条元组。如果排序的列为复合列，要把对应表达式的计算结果添加到这条元组中来。

比如输入的元组为：

|  |  |  |
| --- | --- | --- |
| UserId | Normal\_cost | Additional\_cost |
| 10001 | 100 | 786 |

Extend:

(展开

格式)

Order by: normal\_cost+additional\_cost asc, normal\_cost asc

那么中间存储的紧密格式的元组为：

|  |  |  |  |
| --- | --- | --- | --- |
| UserId | Normal\_cost | Additional\_cost | Normal\_cost+Additional\_cost |
| 10001 | 100 | 786 | 886 |

Compressed：

(紧密格式)

紧密格式会存储Row的所有信息，对于变长类型的数值，也不会有指针指向别处。

**排序列结构**：这个结构中存储的是一个展开的元组，其中的列为排序列和本条元组在存储结构中的指针。元组的第一列为第一排序列的值，第二列为第二排序列的值，以此类推。最后一列为本条元组在紧密存储结构中的偏移。排序列的结构允许通过下标进行随机访问。

**辅助结构**：辅助结构是个一维的数组，存储的是某个Row在排序结构中的指针。在排序过程中，Sort只调整指针在数组中的位置，而不改变Row在紧密存储结构和排序列结构中的相对位置。辅助结构每个元素的大小比较是根据排序列（排序列结构中的非指针列）比较而决定的，比如：

Order by: normal\_cost+additional\_cost asc, normal\_cost asc

排序列结构中的数据为：

|  |  |  |
| --- | --- | --- |
| Normal\_cost+Additional\_cost | Normal\_cost | Pointer |
| 10100 | 100 | …… |
| 500 | 500 |  |

辅助结构初始化为

|  |
| --- |
| 排序列结构第一条元组的指针 |
| 排序列结构第二条元组的指针 |

行比较时，(10100, 100) > (500, 500) 🡺 辅助结构的第一条元组 > 辅助结构第二条元组

**Step 1：读入缓存数据**

Sort每读入一条Row，1. 会把这条Row存入紧密存储结构中，如果排序列中有复合列，复合列的结果同样会存入紧密存储结构中；2. Row对应的排序列的值和Row在紧密存储结构中的偏移会以展开的格式添加至排序列结构中（方便辅助结构比较Row大小时的随机访问）；3. Row在排序列结构中的下标会添加至辅助结构中。如，

Select UserId, normal\_cost, additional\_cost

From consumption

Order by normal\_cost+additional\_cost asc, normal\_cost asc;

Scan consumption表的数据为：

|  |  |  |
| --- | --- | --- |
| UserId | normal\_cost | additional\_cost |
| 10001 | 100 | 10000 |
| 10002 | 500 | 0 |
| 10003 | 350 | 250 |
| 10004 | 150 | 200 |

紧密存储结构：

10001 100 10000 10100

0 500

10002 500

10003 350 250 600

10004 150 200 350

0

1

2

3

10000

100

500

500

600

350

350

150

A+N\_cost

Pointer

Idx

N\_cost

排序列结构：

辅助结构：

|  |
| --- |
| 排序列结构中第1条元组的指针 |
| 排序列结构中第2条元组的指针 |
| 排序列结构中第3条元组的指针 |
| 排序列结构中第4条元组的指针 |

**Step 2：排序**

当紧密存储结构的内存满了，或者排序列结构的内存满了，或者是所有的数据都已经收集完毕，Sort需要对内存中的数据进行排序，然后按顺序刷到外存。

排序过程只调整辅助结构中的数据，根据(Additional\_cost+Normal\_cost asc, Normal\_cost asc)的值（排序结构中可以随机访问），排序后得出辅助结构中的数据排列为：

|  |
| --- |
| 排序列结构中第4条元组的指针 |
| 排序列结构中第2条元组的指针 |
| 排序列结构中第3条元组的指针 |
| 排序列结构中第1条元组的指针 |

**Step 3：按序输出**

顺序扫描辅助结构，依次刷出紧密格式的结果。通过辅助结构中的指针，访问排序列的结构，获得Row在紧密存储结构中的指针，通过这个指针把这个Row刷至外层。每刷依次为一路结果。

仍延续上面的例子，如果到第四条数据，紧密存储结构满了，或者排序列结构满了，或者输入结束了，Sort操作按照辅助结构输出的结果为：

外存上的某路结果

10001 100 10000 10100

250 600

10002 500 0 500

10003 350

10004 150 200 350

在(Additional\_cost+Normal\_cost asc, Normal\_cost asc)上有序

|  |  |
| --- | --- |
| Additional\_cost+Normal\_cost | Normal\_cost |
| 350 | 150 |
| 500 | 500 |
| 600 | 350 |
| 10100 | 100 |

### 输出阶段

当Sort操作符内部已经保留了所有需要的数据，而且已经按结构存储，就可以对外输出了。这里有两种情况需要区分：1. 内存可以放下所有需要排序的数据，只需要内排序。2. 内存放不下所有的数据，必须外排序。

#### 内排序的输出

当数据收集结束时，Sort操作符都没有因为某种存储结构的内存不够而刷出数据，那么对辅助结构排序后，可以不刷出中间结果，直接输出结果。输出方式和刷出的思路一致：

1. 从辅助结构中取当前需要输出的Row的下标。
2. 通过下标从排序列结构中获得Row的指针。
3. 通过指针从紧密存储结构中读出整个Row。
4. 根据给定的column的个数，生成展开格式的元组作为本次调用的输出。

#### 外排序的输出

当需要外排序时，说明整个数据被分成了多路，每路单独排序刷到了外存（其实最后一路可以不刷到外层，直接从内存中取，这取决于实现），外存文件结构见5.7.2.3。输出时只要对有序的多路进行合并即可。

Sort的合并采用堆的算法，从每路取一条数据建一个小顶堆。输出时，1. 只需取出堆顶的数据返回即可。2. 同时，从被返回的数据属于的那路取下一条数据插入到堆，如果本路输入已到结尾，就不插入。

## Scalar Aggregate

ScalarAggregate运算符用于实现没有groupby条件的聚集函数，即无需分组条件计算聚集函数。聚集函数包括count, sum, avg, min, max等。例如select sum(c1), count(c2) from t1 where c1>0。

本操作符的主要参数是聚集函数。

本操作的输出结果只有一行数据，实现非常直观。从子运算符迭代出所有行，对每一行执行聚集操作，最终产生输出。对于不同聚集函数，聚集操作也不同：

* count对应的操作为局部变量count\_加1
* sum对应的操作为局部变量sum\_+=value
* avg对应的操作为count\_++; sum\_+= value
* min对应的操作为if (min\_ > value) min\_ = value;
* max对应的操作为if (max\_ < value) max\_ = value;

## GroupBy

GroupBy运算符把输入数据按照指定的若干列进行聚集，对聚集后的每组数据可以执行count, sum, avg, min, max等聚集操作。

本操作的主要参数有分组列，聚集函数等。

### MergeGroupBy

如果子运算符的输入行已经在分组列上排好序，groupby操作可以简单的读入每行数据，如果当前行和前一行在分组列上相等，则对聚集列执行聚集函数相应的聚集操作；否则，输出聚集好后的行。参见MergeGroupBy的伪代码。

### HashGroupBy

#### 算法

HashGroupBy使用hash分桶结合桶内排序的方法来实现聚集步骤：

1. 对于输入的每一行，用聚集列计算hash值，得到hash桶编号k，把这行输出到第k个桶中
2. 对于每一个hash桶
   1. 排序（内排或外排）
   2. 对于排好序聚集到一起，在聚集列上具有相同值的每一组数据，计算聚集函数值，输出行；**如果需要在聚集列上消重后汇总[[1]](#footnote-1)，还需要做distinct消重操作，如果本行数据在聚集列上等于上一行数据，则忽略本行数据不做聚集计算。**

#### Hash桶的数据结构

HashGroupBy执行过程中使用的内存总大小需要控制。hash桶的内存结构分为以下三块（见图表 1）：

1. 桶链表头存储区，每个桶占有一个指针，指向链表节点存储区中分配的本桶第一个链表节点。
2. 链表节点存储区，每个节点存储若干个行数据所在地址的指针，以及一个指向下一个链表节点的指针。链表节点中行数据指针的个数大小可以大于1，这样可以减少链表结构中为了组织链表结构而消耗的指针比例；另一方面，节点数据结构的大小可以参考CPU cache line的大小来确定，提高链表遍历访问的性能。
3. 数据行存储区，用于存储所有压缩格式的行数据。

图表 1

当链表节点存储区或者数据行存储区大小超过指定阈值时，需要对每个桶进行排序，然后把所有桶的数据转储到磁盘上。

#### 外存文件结构

当内存结构超过阈值时，需要把所有桶的数据转储到外存文件中。一个运算符的操作过程中，可能需要转储多次，为了减少打开的文件数，多次转储的数据都存入同一个文件中。

对于每一个桶，内存中桶链表头存储区中记录最近一次该桶数据转储到文件中的偏移量。每次转储一个桶的数据时，在数据块的头部记录前一次转储本桶数据的偏移量（-1表示这是第一次转储）。这样，每个桶的数据由一个“倒着”的链表串起来，如下图所示：

做归并时，对于每个桶，先读取最近一次转储的数据块头部，顺着指针找到并读取前一次转储的数据头，以此类推就读到了全部“顺串”的第一块数据，然后用最小值堆进行归并即可。

## Distinct

物理运算符Distinct用来消除输入中重复的行。

### MergeDistinct

MergeDistinct要求输入数据在所有列上排好序，通过比较相邻行是否相同来消除多余的重复行。本操作符的子运算符的输出要保证在所有列上进行了排序，所以本运算符的子运算符往往是Sort运算符。

为了避免增加一层MergeDistinct运算符而增加的开销（为了比较一行数据和下一行是否相同，MergeDistinct需要拷贝每一行从子运算符处迭代而来的数据），MergeDistinct运算符的功能直接由Sort运算符实现，而不需要实现为独立的运算符。

### HashDistinct

HashDistinct采用与HashGroupBy类似的算法，可以避免MergeDistinct要求的全局排序：

1. 对于输入的每一行，对所有列计算hash值，得到hash桶编号k，把这行输出到第k个桶中
2. 对于每一个hash桶
   1. 排序（内排或外排）
   2. 对于排好序聚集到一起，在所有列上具有相同值的每一组数据，输出第一行。

## Join

根据连接语义，Join操作有inner join, left outer join, right outer join, full outer join, left semi join, right semi-join, left anti-semi join, right anti-semi join等类型。根据连接条件，Join操作有等值连接和非等值连接之分。被Join两个关系还有多对多，一对多的区别。参见Join操作接口伪代码。

本操作的主要参数有Join类型，Join条件等。

下面用一个简单的例子来说明不同join类型的语义。假设有两个表，Users=<user\_id, user\_name>，Sales=<user\_id, item\_name>。目前两个表的内容为Users={<1, Alice>, <2, Bob>, <2, Bob>, <3, Eve>, <4, Tom>}，Sales={<2, Apple>, <2, Pear>, <3, Orange> <5, Orange>}，则这两个表不同类型join的结果（条件为Users.user\_id=Sales.user\_id）为：

* inner join: {<2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <3, Eve, 3, Orange>}
* left outer join: {<1, Alice, NULL, NULL>, <2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <3, Eve, 3, Orange>}
* right outer join:{ <2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <3, Eve, 3, Orange>, <NULL, NULL, 5, Orange>}
* full outer join: {<1, Alice, NULL, NULL>, <2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <2, Bob, 2, Apple>, <2, Bob, 2, Pear>, <3, Eve, 3, Orange>, <NULL, NULL, 5, Orange>}
* left semi join: {<2, Bob>, <2, Bob>, <3, Bob>}
* right semi join: {<2, Apple>, <2, Pear>, <3, Orange>}
* left anti-semi join: {<1, Alice>, <4, Tom>}
* right anti-semi join: {<5, Orange>}

### MergeJoin

MergeJoin算法要求两个输入表数据都分别在join列上排好序。假设两个输入表的行数分别为M、N，则MergeJoin算法复杂度为O(M+N)。MergeJoin算法可以支持有等值连接条件的所有连接类型。

#### Inner Join及Outer Join

算法如下

// 本算法支持INNER\_JOIN, LEFT\_OUTER\_JOIN, RIGHT\_OUTER\_JOIN, FULL\_OUTER\_JOIN，为了性能考虑可以拆开

**int** ObMergeJoin::normal\_get\_next\_row(**const** ObRow \*&row)

{

// 省略边界条件

**const** ObRow \*left\_row = NULL;

**const** ObRow \*right\_row = NULL;

**if** (last\_left\_row\_有效)

{

left\_row = last\_left\_row\_;

last\_left\_row\_ = NULL;

}

**else**

{

left\_op\_->get\_next\_row(left\_row);

}

**while**(两个子运算符都还有数据没有迭代完)

{

**if** (left\_row在等值条件上等于last\_join\_left\_row\_

&& right\_cache\_is\_valid\_)

{

从right\_cache\_中取下一条数据到right\_row;

**if** (right\_cache\_全部取完了一遍)

{

重置right\_cache\_中的迭代器到第一行;

left\_op\_->get\_next\_row(left\_row);

}

**else**

{

join left\_row和right\_row两行数据，输出到curr\_row\_;

row = &curr\_row\_;

**break**;

}

}

**else**

{

**if** (last\_right\_row\_有效)

{

right\_row\_ = last\_right\_row\_;

last\_right\_row\_ = NULL;

}

**else**

{

right\_op\_->get\_next\_row(right\_row);

}

**if** (left\_row和right\_row在等值join条件上相等)

{

join两行数据，输出到curr\_row\_;

**if** (left\_row在等值条件上不等于last\_join\_left\_row\_)

{

清空right\_cache\_和last\_join\_left\_row\_;

right\_cache\_is\_valid\_ = **false**; // 初始时为false

拷贝left\_row到last\_join\_left\_row\_;

}

right\_row保存添加到right\_cache\_中;

last\_left\_row\_ = left\_row;

row = &curr\_row\_;

**break**;

}

**else** **if**(left\_row在等值join条件上 < right\_row)

{

**if** (LEFT\_OUTER\_JOIN == join\_type\_ || FULL\_OUTER\_JOIN == join\_type\_)

{

用left\_row产生curr\_row\_，不足的值用NULL补齐;

last\_right\_row\_ = right\_row;

row = &curr\_row\_;

**break**;

}

**else**

{

// INNER\_JOIN or RIGHT\_OUTER\_JOIN

right\_cache\_is\_valid\_ = **true**;

last\_right\_row\_ = right\_row;

left\_op\_->get\_next\_row(left\_row);

}

}

**else**

{

// left\_row在等值join条件上 > right\_row

**if** (RIGHT\_OUTER\_JOIN == join\_type\_ || FULL\_OUTER\_JOIN == join\_type\_)

{

用right\_row产生curr\_row\_, 不足的值用NULL补齐;

last\_left\_row\_ = left\_row;

row = &curr\_row\_;

**break**;

}

**else**

{

// INNER\_JOIN or LEFT\_OUTER\_JOIN

right\_op\_->get\_next\_row(right\_row);

}

}

}

} // end while

**if** (没有新行产生

&& (RIGHT\_OUTER\_JOIN == join\_type\_ || FULL\_OUTER\_JOIN == join\_type\_)

&& right\_row有效)

{

用right\_row产生curr\_row\_，不足的值用NULL补齐;

row = &curr\_row\_;

}

**if** (没有新行产生

&& (LEFT\_OUTER\_JOIN == join\_type\_ || FULL\_OUTER\_JOIN == join\_type\_)

&& right\_row有效)

{

用left\_row产生curr\_row\_，不足的值用NULL补齐;

row = &curr\_row\_;

}

}

为了支持多对多关系的join，引入一个cache叫做right\_cache\_。对于与左表某个行能够连接的所有右表的行，加入这个cache中。若下一行左表数据等于前一行，则从cache中迭代出所有行与左表这一行数据进行join。如果在产生查询计划的上下文中能够确保两表只有一对多（左表对右表），则可以省去把右表数据缓存入right\_cache\_的步骤，这样可以较大地提升性能。

#### Semi Join

Left semi join的算法伪代码如下，right semi join与之类似：

// LEFT\_SEMI\_JOIN

**int** ObMergeJoin::left\_semi\_get\_next\_row(**const** ObRow \*&row)

{

// 省略边界条件

**const** ObRow \*left\_row = NULL;

**const** ObRow \*right\_row = NULL;

left\_op\_->get\_next\_row(left\_row);

**if** (last\_right\_row\_有效)

{

right\_row = last\_right\_row\_;

last\_right\_row\_ = NULL;

}

**else**

{

right\_op\_->get\_next\_row(right\_row);

}

**while**(两个子运算符还有数据没有迭代完)

{

**if** (left\_row和right\_row在等值join条件上相等)

{

row = left\_row;

last\_right\_row\_ = right\_row;

**break**;

}

**else** **if**(left\_row在等值join条件上 < right\_row)

{

left\_op\_->get\_next\_row(left\_row);

}

**else**

{

// left\_row在等值join条件上 > right\_row

right\_op\_->get\_next\_row(right\_row);

}

}

}

#### Anti-semi Join

// LEFT\_ANTI\_SEMI\_JOIN

**int** ObMergeJoin::left\_anti\_semi\_get\_next\_row(**const** ObRow \*&row)

{

// 省略边界条件

**const** ObRow \*left\_row = NULL;

**const** ObRow \*right\_row = NULL;

left\_op\_->get\_next\_row(left\_row);

**if** (last\_right\_row\_有效)

{

right\_row = last\_right\_row\_;

last\_right\_row\_ = NULL;

}

**else**

{

right\_op\_->get\_next\_row(right\_row);

}

**while**(两个子运算符还有数据没有迭代完，既left\_row和right\_row都有效)

{

**if** (left\_row和right\_row在等值join条件上相等)

{

left\_op\_->get\_next\_row(left\_row);

}

**else** **if**(left\_row在等值join条件上 < right\_row)

{

row = left\_row\_;

last\_right\_row\_ = right\_row;

**break**;

}

**else**

{

// left\_row在等值join条件上 > right\_row

right\_op\_->get\_next\_row(right\_row);

}

}

**if** (没有找到行要输出

&& left\_row有效)

{

row = left\_row; // left\_op\_还有数据

}

}

## Union

Union的语义是把两部分查询的结果合并起来，最终结果的列名和类型定义与第一个查询一致。Union语句可以是Union All或者Union Distinct，默认情况下最好采用前者，即只有Union关键字时等价于Union All。下面看看Union All/Union Distinct的例子。

**表A** **表B**

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

|  |  |
| --- | --- |
| 3 | 4 |
| 3 | 4 |
| 7 | 8 |

**Union All**

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |
| 3 | 4 |
| 3 | 4 |
| 7 | 8 |

**Union Distinct**

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |
| 5 | 6 |
| 7 | 8 |

### MergeUnion

#### ALL

如果是Union All，那么MergeUnion的两个输入表没有必要是有序的，MergeUnion只需要先输出第一个表的数据，再输出第二个表的数据就可以了。

#### Distinct

如果是Union Distinct，MergeUnion算法要求两个输入表数据都有相同的排序。假设两个输入表的行数分别为M、N，则MergeUnion算法复杂度为O(M+N)。

具体实现如下：

1

2

1

2

5

6

5

6

7

8

7

8

表A

3

4

3

4

5

6

5

6

表A

结果

1. 如果第一个表的当前元组小于第二个表的当前元组，或者第二个表结束，那么输出第一个表的元组，且跳过相等的元组。

1

2

1

2

5

6

5

6

7

8

7

8

表A

3

4

3

4

5

6

5

6

表A

1

2

结果

1. 如果第一个表的当前元组大于第二个表的当前元组，或者第一个表结束，那么输出第二个表的元组，且跳过相等的元组。

1

2

1

2

5

6

5

6

7

8

7

8

表A

3

4

3

4

5

6

5

6

表A

1

2

结果

3

4

1. 如果第一个表的当前元组等于第二个表的当前元组，那么输出第一个表的元组，且跳过第一个和第二个表的相同元组。

1

2

1

2

5

6

5

6

7

8

7

8

表A

3

4

3

4

5

6

5

6

表A

1

2

结果

3

4

5

6

1. 如果两个表都结束，则返回结束。

## Intersect

MergeIntersect算法要求两个输入表数据都有相同的排序。假设两个输入表的行数分别为M、N，则MergeIntersect算法复杂度为O(M+N)。

具体算法如下：

1

2

1

2

5

6

5

6

5

6

7

8

表A

3

4

3

4

5

6

5

6

表A

结果

1. 如果第一个表的当前元组l\_row小于第二个表的当前元组r\_row，那么跳过第一个表中所有与l\_row相同的元组。

1

2

1

2

5

6

5

6

5

6

7

8

表A

3

4

3

4

5

6

5

6

表A

结果

1. 如果第一个表的当前元组l\_row大于第二个表的当前元组r\_row，那么跳过第二个表中所有与r\_row相同的元组。

1

2

1

2

5

6

5

6

5

6

7

8

表A

3

4

3

4

5

6

5

6

表A

结果

1. 如果第一个表的当前元组l\_row等于第二个表的当前元组r\_row，那么输出第一个表的当前元组。IF Distinct，第一个表和第二个表跳过所有与l\_row/r\_row相等的元组；else （ALL），跳过第一个表和第二个表的当前元组。

1

2

1

2

5

6

5

6

5

6

7

8

表A

3

4

3

4

5

6

5

6

表A

ALL结果

5

6

1. 如果任何一个表到结尾，则结束。

## Except

MergeExcept算法要求两个输入表数据都有相同的排序。假设两个输入表的行数分别为M、N，则MergeExcept算法复杂度为O(M+N)。

具体算法如下：

1. 如果第一个表的当前元组l\_row小于第二个表的当前元组r\_row，那么输出第一个表的当前结果。IF Distinct，第一个表跳过所有与l\_row相等的元组；else （ALL），跳过第一个表当前元组l\_row。
2. 如果第一个表的当前元组l\_row大于第二个表的当前元组r\_row，那么跳过第二个表的当前元组。
3. 如果第一个表的当前元组l\_row等于第二个表的当前元组r\_row，那么，IF Distinct，第一个表和第二个表跳过所有与l\_row/r\_row相等的元组；else （ALL），跳过第一个表和第二个表的当前元组。
4. 如果第二个表结束，则输出第一个表的当前元组。
5. 如果第一个表结束，则结束。

不再赘图，参看Union和Intersect的图示。

# 附录

## CS端的TabletScan伪代码

**namespace** oceanbase

{

**namespace** sql

{

// 用于CS从磁盘或缓冲区扫描一个tablet

**class** ObSstableScan: **public** ObPhyOperator

{

**public**:

ObSstableScan();

**virtual** ~ObSstableScan();

**int** open();

**int** close();

**int** get\_next\_row(**const** ObRow \*&row);

**int** get\_next\_rowkey(**const** ObRow \*&rowkey);

/\*\*

\* 设置要扫描的tablet range

\*/

**int** set\_range(**const** ObNewRange &tablet\_range);

**private**:

// disallow copy

ObSstableScan(**const** ObSstableScan &other);

ObSstableScan& **operator**=(**const** ObSstableScan &other);

**private**:

// data members

ObNewRange range\_;

ObRow curr\_row\_;

ObRowDesc row\_desc\_;

};

} // end namespace sql

} // end namespace oceanbase

**int** ObSstableScan::open()

{

根据扫描range\_打开文件或者初始化缓冲区;

根据元数据初始化row\_desc\_;

}

**int** ObSstableScan::close()

{

释放资源，例如关闭文件或者释放缓冲区等;

}

**int** ObSstableScan::get\_next\_row(**const** ObRow \*&row)

{

从sstable读取下一行的若干cell构造curr\_row\_;

curr\_row\_.set\_row\_desc(row\_desc\_);

row = &curr\_row\_;

}

**namespace** oceanbase

{

**namespace** sql

{

// 用于CS从UPS扫描一批动态数据

**class** ObUpsScan: **public** ObPhyOperator

{

**public**:

ObUpsScan();

**virtual** ~ObUpsScan();

**int** open();

**int** close();

**int** get\_next\_row(**const** ObRow \*&row);

**int** get\_next\_rowkey(**const** ObRow \*&rowkey);

/\*\*

\* 设置要扫描的range

\*/

**int** set\_range(**const** ObNewRange &range);

**private**:

// disallow copy

ObUpsScan(**const** ObUpsScan &other);

ObUpsScan& **operator**=(**const** ObUpsScan &other);

**private**:

// data members

ObNewRange range\_;

};

} // end namespace sql

} // end namespace oceanbase

**namespace** oceanbase

{

**namespace** sql

{

// 用于CS从合并sstable中的静态数据和UPS中的增量数据

**class** ObTabletFuse: **public** ObPhyOperator

{

**public**:

ObTabletFuse();

**virtual** ~ObTabletFuse();

**int** add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator);

**int** open();

**int** close();

**int** get\_next\_row(**const** ObRow \*&row);

**private**:

// disallow copy

ObTabletFuse(**const** ObTabletFuse &other);

ObTabletFuse& **operator**=(**const** ObTabletFuse &other);

**private**:

// data members

ObPhyOperator \*sstable\_scan\_; // 从sstable读取的静态数据

ObPhyOperator \*incremental\_scan\_; // 从UPS读取的增量数据

ObRow curr\_row\_;

ObRow \*last\_incr\_row\_;

};

} // end namespace sql

} // end namespace oceanbase

**int** ObTabletFuse::add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator)

{

**int** ret = OB\_SUCCESS;

**switch**(child\_idx)

{

**case** 0:

sstable\_scan\_ = &child\_operator;

**break**;

**case** 1:

incremental\_scan\_ = &child\_operator;

**break**;

**default**:

ret = OB\_ERR\_UNEXPECTED;

**break**;

}

**return** ret;

}

**int** ObTabletFuse::open()

{

sstable\_scan\_->open();

incremental\_scan\_->open();

}

**int** ObTabletFuse::close()

{

sstable\_scan\_->close();

incremental\_scan\_->close();

}

// 分别从sstable\_scan\_和incremental\_scan\_读取数据，按照rowkey归并排序输出

**int** ObTabletFuse::get\_next\_row(**const** ObRow \*&row)

{

// 省略流返回OB\_ITER\_END的情况

ObRow \*sstable\_row = NULL;

**bool** is\_deleted = **false**;

**do**

{

**if** (!last\_incr\_row\_is\_valid\_)

{

incremental\_scan\_->get\_next\_row(last\_incr\_row\_);

last\_incr\_row\_is\_valid\_ = **true**;

}

sstable\_scan\_->get\_next\_row(sstable\_row);

**if** (get\_rowkey(last\_incr\_row\_) < get\_rowkey(sstable\_row))

{

row = &last\_inc\_row\_;

last\_inc\_row\_is\_valid\_ = **false**;

}

**else** **if** (get\_rowkey(last\_incr\_row\_) == get\_rowkey(\*sstable\_row))

{

fuse\_row(last\_incr\_row\_, sstable\_row, curr\_row\_, is\_deleted);

**if** (!is\_deleted)

{

row = &curr\_row\_;

}

}

**else**

{

row = sstable\_row;

}

} **while**(is\_delete);

}

**namespace** oceanbase

{

**namespace** sql

{

// 用于CS从UPS获取多行数据

**class** ObUpsMultiGet: **public** ObPhyOperator

{

**public**:

ObUpsMultiGet();

**virtual** ~ObUpsMultiGet();

**int** open();

**int** close();

**int** get\_next\_row(**const** ObRow \*&row);

/\*\*

\* 设置MultiGet的参数

\*/

**int** set\_get\_param(**const** ObGetParam &range);

**private**:

// disallow copy

ObUpsMultiGet(**const** ObUpsMultiGet &other);

ObUpsMultiGet& **operator**=(**const** ObUpsMultiGet &other);

**private**:

// data members

};

} // end namespace sql

} // end namespace oceanbase

**namespace** oceanbase

{

**namespace** sql

{

**class** ObTabletJoin: **public** ObPhyOperator

{

**public**:

ObTabletJoin();

**virtual** ~ObTabletJoin();

**int** add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator);

**int** open();

**int** close();

**int** get\_next\_row(**const** ObRow \*&row);

**void** set\_batch\_count(**const** int64\_t batch\_count);

**int** set\_join\_info(**const** ObJoinInfo &join\_info);

**private**:

// disallow copy

ObTabletJoin(**const** ObTabletJoin &other);

ObTabletJoin& **operator**=(**const** ObTabletJoin &other);

**private**:

// data members

JoinInfo join\_info\_;

int64\_t batch\_count\_;

ObPhyOperator \*fused\_scan\_;

HashTable<ObRowkey, ObRow> join\_cache\_;

Array<ObRow> fused\_cache\_;

ObRow curr\_row\_;

};

} // end namespace sql

} // end namespace oceanbase

**int** ObTabletJoin::add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator)

{

**int** ret = OB\_SUCCESS;

**switch**(child\_idx)

{

**case** 0:

fused\_scan\_ = &child\_operator;

**break**;

**default**:

ret = OB\_ERR\_UNEXPECTED;

**break**;

}

**return** ret;

}

**int** ObTabletJoin::open()

{

fused\_scan\_->open();

}

**int** ObTabletJoin::close()

{

fused\_scan\_->open();

}

**int** ObTabletJoin::get\_next\_row(**const** ObRow \*&row)

{

**if** (本批fused\_cache\_中数据使用完了)

{

从fused\_scan\_中读取下一批batch\_count\_行数据到fused\_cache\_中;

**if** (fused\_scan\_中至少有一行不在join\_cache\_中)

{

// 从UPS multiget数据用于做join，ObUpsMultiGet也可以返回CompactRow，待定

ObUpsMultiGet multi\_get;

设置不在join\_cache\_中行的multi\_get参数;

multi\_get.open();

ObRow \*multi\_get\_row = NULL;

**for**(multi\_get.get\_next\_row(multi\_get\_row))

{

把multi\_get\_row加入join\_cache\_中;

}

multi\_get.close();

}

}

**if** (本批fused\_cache\_没有使用完)

{

从fused\_cache\_中读取下一行;

从join\_cache\_中读取相应行数据;

join两行数据到curr\_row\_;

row = &curr\_row\_;

}

}

**namespace** oceanbase

{

**namespace** sql

{

// 用于CS从磁盘扫描一个tablet，合并、join动态数据，并执行计算过滤等

// @code

// int cs\_handle\_scan(…)

// {

// ObTabletScan tablet\_scan\_op;

// ScanResult results;

// 设置tablet\_scan的参数;

// tablet\_scan\_op.open();

// const ObRow \*row = NULL;

// for(tablet\_scan\_op.get\_next\_row(row))

// {

// results.output(row);

// }

// tablet\_scan\_op.close();

// send\_response(results);

// }

**class** ObTabletScan: **public** ObPhyOperator

{

**public**:

ObTabletScan();

**virtual** ~ObTabletScan();

**int** open();

**int** close();

**int** get\_next\_row(**const** ObRow \*&row);

/// 设置要扫描的tablet

**int** set\_range(**const** ObNewRange &tablet\_range);

/// 设置批量做join的个数，需要做join时才生效

**int** set\_join\_batch\_count();

/\*\*

\* 添加一个需输出的column

\*

\* @note 只有通过复合列结算新生成的列才需要new\_column\_id

\* @param expr [in] 需输出的列（这个列可能是个复合列的结果）

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** add\_output\_column(Expression& expr, **const** uint64\_t new\_column\_id);

/\*\*

\* 添加一个filter

\*

\* @param expr [in] 过滤表达式

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** add\_filter(Expression& expr);

/\*\*

\* 指定limit/offset

\*

\* @param limit [in]

\* @param offset [in]

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** set\_limit(**const** int64\_t limit, **const** int64\_t offset);

**private**:

// disallow copy

ObTabletScan(**const** ObTabletScan &other);

ObTabletScan& **operator**=(**const** ObTabletScan &other);

**private**:

// data members

ObNewRange scan\_range\_;

ObArray<Expression> filters\_;

ObArray<Expression> out\_columns\_;

int64\_t limit\_offset\_;

int64\_t limit\_count\_;

int64\_t join\_batch\_count\_;

int64\_t curr\_row\_idx\_; // 已经输出了几行数据

ObPhyOperator \*op\_root\_;

ObSstableScan \*op\_sstable\_scan\_;

ObUpsScan \*op\_ups\_scan\_;

ObTabletFuse \*op\_tablet\_fuse\_;

ObTabletJoin \*op\_tablet\_join\_;

};

} // end namespace sql

} // end namespace oceanbase

**int** ObTabletScan::open()

{

op\_sstable\_scan\_ = **new** ObSstableScan();

op\_ups\_scan\_ = **new** ObUpsScan();

op\_sstable\_scan\_->set\_range(scan\_range\_);

op\_ups\_scan\_->set\_range(scan\_range\_);

op\_tablet\_fuse\_ = **new** ObTabletFuse();

op\_tablet\_fuse\_->add\_child(0, \*op\_sstable\_scan\_);

op\_tablet\_fuse\_->add\_child(1, \*op\_ups\_scan\_);

**if** (本表需要做join)

{

op\_tablet\_join\_ = **new** ObTabletJoin();

op\_tablet\_join\_->set\_join\_info();

op\_tablet\_join\_->set\_batch\_count(join\_batch\_count\_);

op\_tablet\_join\_->add\_child(0, \*ob\_fused\_join\_);

op\_root\_ = ob\_tablet\_join\_;

}

**else**

{

op\_root\_ = ob\_fused\_join\_;

}

op\_root\_->open();

}

**int** ObTabletScan::close()

{

op\_root\_->close();

释放内存等;

}

**int** ObTabletScan::get\_next\_row(**const** ObRow \*&row)

{

**int** ret = OB\_SUCCESS;

**if** (0 == limit\_count\_)

{

ret = OB\_ITER\_END;

}

**else** **if** (curr\_row\_idx\_ >= limit\_offset\_ + limit\_count\_)

{

ret = OB\_ITER\_END;

}

**else**

{

**bool** did\_output = **false**;

**do**

{

**do**

{

op\_root\_->get\_next\_row(row);

compute(row, curr\_row\_); // 根据out\_columns\_的内容，进行计算产生符合列等

filter(curr\_row\_, did\_output); // 根据filters\_的内容，过滤判断本行是否满足条件

} **while**(!did\_output);

**if** (did\_output)

{

row = &curr\_row\_;

++curr\_row\_idx\_;

}

} **while**(curr\_row\_idx\_ < limit\_offset\_); // limit条件判断

**if** (curr\_row\_idx\_ >= limit\_offset\_ + limit\_count\_)

{

ret = OB\_ITER\_END;

}

}

}

## MS上TableScan的伪代码

**namespace** oceanbase

{

**namespace** sql

{

// 用于MS进行全表扫描

**class** ObTableScan: **public** ObPhyOperator

{

**public**:

ObTableScan();

**virtual** ~ObTableScan();

**int** add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator);

**int** open();

**int** close();

**int** get\_next\_row(**const** ObRow \*&row);

/\*\*

\* 添加一个需输出的column

\*

\* @note 只有通过复合列结算新生成的列才需要new\_column\_id

\* @param expr [in] 需输出的列（这个列可能是个复合列的结果）

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** add\_output\_column(Expression& expr, **const** uint64\_t new\_column\_id);

/\*\*

\* 设置table\_id

\* @note 只有基本表被重命名的情况才会使两个不相同id，其实两者相同时base\_table\_id可以给个默认值。

\* @param table\_id [in] 输出的table\_id

\* @param base\_table\_id [in] 被访问表的id

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** set\_table(**const** uint64\_t table\_id, **const** uint64\_t base\_table\_id);

/\*\*

\* 添加一个filter

\*

\* @param expr [in] 过滤表达式

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** add\_filter(Expression& expr);

/// 获得输出列的数目

int64\_t get\_column\_num() **const**;

/\*\*

\* 指定limit/offset

\*

\* @param limit [in]

\* @param offset [in]

\*

\* @return OB\_SUCCESS或错误码

\*/

**int** set\_limit(**const** int64\_t limit, **const** int64\_t offset);

**private**:

// disallow copy

ObTableScan(**const** ObTableScan &other);

ObTableScan& **operator**=(**const** ObTableScan &other);

**private**:

// data members

// 表的id。对于需要新生成表id的情况，是新生成的表id

// 生成输出列的RowDesc时使用

uint64\_t table\_id\_;

// 表的id。对于需要新生成表id的情况，是原始的表id

// 访问基本表时使用

uint64\_t base\_table\_id\_;

// 需要对上层操作符输出的column：

// 1. 基本列，由table\_id和原始column\_id唯一确定；

// 2. 复合列，table\_id=NULL，column\_id为新生成的id，这个id在整个查询中唯一。

ObArray<std::pair<Expression, out\_cid> > out\_columns\_;

// 每个过滤条件之间是and的语义。

// 过滤表达式可以引用基本列，也可以引用其他表达式。

ObArray<Expression> filters\_;

// limit值（优化结构）

int64\_t limit\_;

// offset值（优化结构）

int64\_t offset\_;

// Scan的是中间结果表时，下层需调用ObPhyOperator。如果是基本表，则为NULL。

ObPhyOperator \*op\_;

// 被迭代的当前row，get\_next\_row()的返回值

ObRow curr\_row\_;

// 输出row的字段描述符。

ObRowDesc row\_desc\_;

// 存储CS返回的扫描结果

List<ObNewScanner\*> scan\_results\_;

};

} // end namespace sql

} // end namespace oceanbase

**int** ObTableScan::open()

{

根据参数构建row\_desc\_;

根据base\_tid\_查询tablets location;

对于每一个相关cs，发送前k个tablet的scan请求;

当第一个tablet的第一段结果返回后，发起下一个tablet请求;

}

**int** ObTableScan::close()

{

**int** ret = OB\_SUCCESS;

释放资源等;

**return** ret;

}

**int** ObTableScan::get\_next\_row(**const** ObRow \*&row)

{

**int** ret = OB\_SUCCESS;

// 忽略limit,offset相关逻辑

ObCompactRow \*compact\_row = NULL;

**bool** did\_output = **false**;

**do**

{

get\_next\_compact\_row(compact\_row); // 如果需要等待CS返回

convert(compact\_row, curr\_row\_);

compute(curr\_row\_); // 根据out\_columns\_的内容，进行计算产生符合列等

filter(curr\_row\_, did\_output); // 根据filters\_的内容，过滤判断本行是否满足条件

} **while**(!did\_output);

**if** (did\_output)

{

row = &curr\_row\_;

}

**return** ret;

}

## GroupBy接口的伪代码

**namespace** oceanbase

{

**namespace** sql

{

**class** ObGroupBy: **public** ObPhyOperator

{

**public**:

ObGroupBy();

**virtual** ~ObGroupBy();

**virtual** **int** add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator);

**virtual** **int** open();

**virtual** **int** close();

**virtual** **int** get\_next\_row(**const** ObRow \*&row);

/\*\*

\* 添加一个group column

\*

\* @param expr [in] group列（这个列可能是个复合列的结果）

\* @param new\_column\_id [in] 复合列的新id

\*

\* @return OB\_SUCCESS或错误码

\*/

**virtual** **int** add\_group\_column(Expression& expr, **const** uint64\_t new\_column\_id) = 0;

/\*\*

\* 添加一个agregate column

\*

\* @param expr [in] agg列（这个列可能是个复合列的结果）

\* @param new\_column\_id [in] 聚集列的新id

\*

\* @return OB\_SUCCESS或错误码

\*/

**virtual** **int** add\_agg\_column(Expression& expr, **const** uint64\_t new\_column\_id) = 0;

/\*\*

\* 添加一个column，这个列既不是group列，也不是聚集列

\*

\* @param expr [in] 需输出的列（这个列可能是个复合列的结果）

\* @param new\_column\_id [in] 复合列的新id

\*

\* @return OB\_SUCCESS或错误码

\*/

**virtual** **int** add\_other\_column(Expression& expr, **const** uint64\_t new\_column\_id) = 0;

**private**:

// disallow copy

ObGroupBy(**const** ObGroupBy &other);

ObGroupBy& **operator**=(**const** ObGroupBy &other);

};

} // end namespace sql

} // end namespace oceanbase

## MergeGroupBy的伪代码

**int** ObMergeGroupBy::get\_next\_row(**const** ObRow \*&row)

{

// 忽略child\_op\_迭代完返回OB\_ITER\_END

**if** (last\_input\_row\_无效)

{

child\_op\_->get\_next\_row(last\_input\_row\_);

初始化作为聚集后输出的curr\_row\_;

}

**const** ObRow \*input\_row;

**while**(child\_op\_->get\_next\_row(input\_row))

{

**if**(input\_row在聚集列上等于last\_input\_row\_)

{

根据input\_row计算聚集列，修改curr\_row\_;

}

**else**

{

last\_input\_row\_ = input\_row;

**break**;

}

}

row = &curr\_row\_;

}

## HashGroupBy的伪代码

**int** ObHashGroupBy::open()

{

构造row\_desc\_;

// 从子运算符读取所有数据，插入桶中

child\_op\_->open();

**const** ObRow \*row = NULL;

**for**(child\_op\_->get\_next\_row(row))

{

根据groupby列计算row归属的hash桶ID;

add\_to\_backet(backet\_idx, row);

}

child\_op\_->close();

**for**(每个桶)

{

根据groupby列排序这个桶的内容，如果必要使用外排。

}

}

**int** ObHashGroupBy::add\_to\_backet(**const** int64\_t backet\_idx, **const** ObRow\* row)

{

把row写入第backet\_idx个桶中; // 内存存储结构同ObSort，采用ObRow和ObCompactRow结合的方式

**if** (第backet\_idx个桶大小超过backet\_mem\_size)

{

dump这个桶到磁盘;

}

}

**int** ObHashGroupBy::close()

{

释放内存等资源;

}

**int** ObHashGroupBy::get\_next\_row(**const** ObRow \*&row)

{

**for**(当前桶curr\_backet\_idx\_的具有相同groupby列的下k个行)

{

计算聚集列，修改curr\_row\_;

}

**if** (当前桶处理完)

{

++curr\_backet\_idx\_;

}

row = &curr\_row\_;

}

## Join操作接口伪代码

**namespace** oceanbase

{

**namespace** sql

{

**class** ObJoin: **public** ObPhyOperator

{

**public**:

ObJoin();

**virtual** ~ObJoin();

**virtual** **int** add\_child(int32\_t child\_idx, ObPhyOperator &child\_operator);

**virtual** **int** open();

**virtual** **int** close();

**virtual** **int** get\_next\_row(**const** ObRow \*&row);

**enum** JoinType{

INNER\_JOIN,

LEFT\_OUTER\_JOIN,

RIGHT\_OUTER\_JOIN,

FULL\_OUTER\_JOIN,

LEFT\_SEMI\_JOIN,

RIGHT\_SEMI\_JOIN,

LEFT\_ANTI\_SEMI\_JOIN,

RIGHT\_ANTI\_SEMI\_JOIN

};

**virtual** **int** set\_join\_type(**const** JoinType join\_type);

**virtual** **int** add\_equijoin\_condition(Expression &expr);

**virtual** **int** add\_other\_join\_condition(Expression &expr);

**private**:

// disallow copy

ObJoin(**const** ObJoin &other);

ObJoin& **operator**=(**const** ObJoin &other);

**private**:

// data members

ObPhyOperator \*left\_op\_;

ObPhyOperator \*right\_op\_;

JoinType join\_type\_;

};

} // end namespace sql

} // end namespace oceanbase

1. 例如select \* from t1 group by c1 having count(DISTINCT c2) > 3; [↑](#footnote-ref-1)