**MergeServer 0.3 III并发**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 | 无施 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# 需求分析

当merge/join的功能移到cs之后，ms的任务主要包括：请求解析与转换、任务拆分、子任务的分发与调度、结果合并、并发结构。

请求解析与转换主要是ObScanParam和ObGetParam的解析，将name转换为id。根据请求类型修改参数中的信息，比如top k近似。

任务拆分主要工作是把跨越tablet的请求按照tablet拆分为多个子任务，多个子任务之间可以在各个cs上并发进行。

子任务分发与调度，由于每个tablet有多个副本，这部分需要考虑cs的负载均衡；同时还需要考虑备份任务。

结果合并，根据请求情况合并从多个cs得到的结果。如果不存在groupby，那么就是一个多路归并的操作；如果存在groupby，那么就是将各个cs返回的结果再进行一次groupby。

并发结构：由于要支持并发处理，所以代码结构与现有的串行的代码结构肯定存在区别。并发结构的最主要功能点是同一个请求，各个cs的处理必须能够并发。在实现该主要目的的前提下，尽量控制代码复杂度。

# 设计思路

## 并发结构

并发是目的，异步只是一种实现方式。考虑异步代码的复杂性，仅实现cs处理的并发，使用tbnet现有框架。结合串行的工作线程与并发的网络线程。工作线程与现有ob中各个服务器模型相同，tbnet在接收到请求的时候在工作线程中调用请求处理函数。请求处理函数对请求进行解析和转换，通过非阻塞的方式向各个cs发起子请求。然后阻塞的等待所有子请求返回后对结果进行合并。当然，最好可以做到边接收请求边进行计算，这样可以减少工作线程阻塞的时间。

这样server框架部分与现有保持代码结构保持不变，可以复用现有ms的代码。需要修改的是请求处理函数、ObClientManger。

在这样的并发结构设计下，必须处理过期的子任务，为每个请求分配唯一的64位id，每个子任务都记录了这个id，通过request\_id来区分过期的子任务。

## 负载均衡

每个tablet都有多个副本，也就意味着每个子请求有三个cs可供选择。前期先采用简单的轮询调度算法。对于每个tablet的多个副本，使用tablet range的startkey+ms的ip地址，求crc，通过crc来确定该tablet第一次访问哪一个副本，计算公式如下： crc(tablet\_range.start\_key\_+ms.ip)/num\_of\_replicas)。负载均衡相关的实现逻辑必须封装在单独的类中实现，便于后续对策略进行调整。

## 备份任务

一个请求可能会存在多个子任务，为了防止少数慢任务导致整个请求的处理时间很长，可以为这种慢任务启动备份任务，选择备份任务中最先完成的任务作为子任务的结果。备份任务的难点在于选择何时启动备份任务，过早的启动备份任务会导致占用过多的带宽和计算能力；过晚启动备份任务会增加请求的latency。

前期可以使用简单的策略或者不支持备份任务。但是必须在程序结构上支持复杂的策略。

一个简单的策略就是记录所有跨越整个tablet的子任务的请求到平均处理时间，如果一个子任务超过这个平均时间还没有返回，就启动该子任务的备份任务。

## top k近似

top k近似是指当请求的数据范围太大的情况下，为了保证请求的处理时间，没有办法进行全局的排序，对于量子统计而言，可以每个tablet返回比top k更大的一个top范围，然后ms对每个cs返回的数据进行局部排序，最终得到近似的top k。当前接口，用户可以设置一个[0,1]的double类型，表示希望获得的近似度（precision），该值越大表示得到的结果越精确。

先实现一种简单的算法，近似度表示从每个tablet获取的记录数量与k的关系，全局top k等于每个tablet求取top k\*PRECISION\_FACTOR\*precision。其中PRECISION\_FACTOR是一个整数，可以在配置文件中进行配置，默认值为100。

# 详细设计

## 并发结构

### RPC子任务

请求的子任务表示单次对cs的请求，子任务通过tbnet非阻塞发送请求，并且通过tbnet::IPacketHandler::handlePacket接收响应。接口如下：

class ObMsRequestEvent;

class ObMsRpcEvent

{

public:

int init(const uint64\_t client\_event\_id, ObMsRequestEvent \*client\_event);

int handle\_response(tbnet::Packet\* packet);

void reset();

uint64\_t get\_client\_event\_id()const;

int32\_t get\_event\_id()const;

int32\_t get\_result\_code()const;

int get\_response(char \*& response) const;

private:

int32\_t result\_code\_;

/// buffer used to store response from cs, malloced by ob\_malloc

char \*response\_;

int32\_t response\_len\_;

/// used by working thread to distinguish each sub task/event

uint64\_t event\_id\_;

/// used by working thread to distinguish delayed sub task/event

/// identify which event triggered this rpc event

uint64\_t client\_event\_id\_;

ObMsRequestEvent \*client\_event\_;

};

ObMsRpcEvent::client\_event\_被工作线程持有，用于接收rpc子任务完成的通知和对结果进行运算。它与ObMsRpcEvent的关系是：ObMsRpcEvent是client\_event\_(ObMsRequestEvent)的代理事件，代理client\_event\_进行rpc相关的处理；client\_event\_是ObMsRpcEvent的客户，client\_event\_把需要委托处理的rpc相关事件封装到ObMsRpcEvent中。

ObMsRpcEvent::handle\_response被tbnet::IPacketHandler::handlePacket调用，该函数会把响应的结果拷贝到ObMsRpcEvent::response\_中，并调用ObMsRpcEvent::agent\_event\_的接口通知工作线程rpc子任务已经完成。

### ObMultClientManager

ObMultClientManager负责处理ms与cs的通信，角色与当前ObClientManager相同，只是支持并发非阻塞的发起多个rpc调用。接口如下：

class ObMultClientManager:public tbnet::IPacketHandler

{

public:

virtual HPRetCode handlePacket(Packet \*packet, void \*args);

int send\_request(ObMsRpcEvent \*agent\_event,

const ObServer& server, const int32\_t pcode,

const int32\_t version, const int64\_t timeout,

const ObScanParam& scan\_param);

int send\_request(ObMsRpcEvent \*agent\_event,

const ObServer& server, const int32\_t pcode,

const int32\_t version, const int64\_t timeout,

const ObGetParam& get\_param);

};

ObMultClientManager::send\_request会构建(new) 一个ObPacket并调用tbnet::ConnectionManager::sendPacket非阻塞的发送请求，以agent\_event 作为tbnet::ConnectionManager::sendPacket的最后一个参数。该函数的语义就是工作线程把代理事件(agent\_event)托付给ObMultClientManager去执行。

ObMultClientManager::handlePacket会调用ObMsRpcEvent::handle\_response，通知工作线程，agent event完成。

由于是异步请求，所以存在延迟完成的agent event。工作线程必须要有相应的机制来区分延迟完成的agent event。

agent\_event由工作线程分配，也由工作线程释放。代理事件由委托者生成，托付给ObMultClientManager去执行，完成后再由委托者去释放。

### ObMsRequestEvent

该类被工作线程持有，每个请求初始化该类的一个实例。接口如下：

class ObMsRequestEvent

{

public:

/// called by ObMultClientManager::handlePacket

void handle\_rpc\_event(ObMsRpcEvent \* agent\_event);

/// called by working thread when receive a request from client

/// these two functions will trigger rpc event which will non-blocking rpc access cs

virtual int set\_request\_param(ObScanParam & scan\_param, const int64\_t timeout\_us);

virtual int set\_request\_param(ObGetParam & get\_param, const int64\_t timeout\_us);

/// called by working thread after set\_request\_param, this function will block until

/// one of the rpc event finish

virtual int reap\_agent\_events(const int64\_t timeout\_us) = 0;

private:

uint64\_t event\_id\_;

};

工作线程请求处理流程如下：

解析客户端请求参数

初始化ObMsRequestEvent、调用set\_request\_param

调用ObMsRequestEvent::reap\_client\_events处理cs返回的结果

返回客户端

ObMsRequestEvent::set\_request\_param会发起对cs的异步请求，生成委托事件(ObMsRpcEvent)，并把委托事件交付给ObMultClientManager去执行；ObMsRequestEvent::reap\_agent\_event会阻塞等待异步请求结束(等待ObMultClientManager对委托事件的执行结果)，对于每个cs返回的结果进行相应的处理。

ObMultClientManager::handlePacket会在收到来自cs的响应后调用ObMsRequestEvent::handle\_rpc\_event，该函数会通知阻塞在ObMsRequestEvent::reap\_client\_event上的工作线程rpc请求收到响应。

前面章节的描述中提到延迟的rpc请求，ObMsRequest必须能够判断延迟的响应。这里通过递增的ObMsRequestEvent::event\_id\_来解决这个问题，每次初始化ObMsRequestEvent的时候，该字段都递增。而ObMsRpcEvent中保存了触发自己的ObMsRequestEvent（委托方）的event\_id\_值，该值保存在ObMsRpcEvent::client\_id\_中。这样，ObMsRequestEvent::reap\_client\_events就可以通过自身的event\_id\_来判断延迟的rpc响应，对于延迟的rpc响应直接忽略。

注意，由于延迟ObMsRpcEvent的存在，因此工作线程多个请求是复用同一个ObMsRequestEvent的，这样ObMsRpcEvent::client\_event\_才是有效的指针。

ObMsRequestEvent::reap\_client\_events中，关于收到rpc响应的处理方法与具体请求相关。参见文档后面部分的描述。

## GET请求

ms在处理get请求的时候，ms仅仅是转发的作用，处理get请求采用串行的模式。接口如下：

class ObMsGetEvent : public ObMsRequestEvent

{

public:

virtual int set\_request\_param(ObGetParam & get\_param, const int64\_t timeout\_us);

virtual int reap\_agent\_events(const int64\_t timeout\_us);

private:

int get\_next\_param();

ObScanner result\_;

int32\_t result\_code\_;

const ObGetParam \*org\_get\_param\_;

ObGetParam cur\_get\_param\_;

int32\_t got\_cell\_num\_;

uint64\_t last\_agent\_event\_id\_;

};

ObMsGetEvent::get\_next\_param根据当前已经得到的cell数量got\_cell\_num\_，和原始的请求org\_get\_param\_，计算下一次rpc的请求参数cur\_get\_param。

ObMsGetEvent::set\_request\_param会计算第一次rpc使用的参数，并选择cs，发起一个异步请求。

ObMsGetEvent::reap\_client\_events在接收到rpc响应后的处理逻辑如下：

if got\_rpc\_event.event\_id\_ == last\_agent\_event\_id\_

if got\_rpc\_event 成功

将got\_rpc\_event的结果追加到result\_中

计算下一次rpc请求的参数

获得下次rpc请求的cs服务器

else

重试当前请求，获得重试的cs服务器

endif

if 请求没有结束

发起下一次rpc调用

endif

else

释放agent event资源

/// do nothing

endif

## 调度之负载均衡

每个tablet有多个副本，ms转发请求的时候需要考虑负载均衡。前期负载均衡可以做得简单一些，但是接口设计上必须保证负载均衡策略的调整不会影响代码的整体结构。

前期先采用简单的轮询调度算法。对于每个tablet的多个副本，使用tablet range的startkey+ms的ip地址，求crc，通过crc来确定该tablet第一次访问哪一个副本，计算公式如下： crc(tablet\_range.start\_key\_+ms.ip)/num\_of\_replicas)。负载均衡相关的实现逻辑必须封装在单独的类中实现，便于后续对策略进行调整。

我们把负载均衡的策略集中管理，便于策略调整。策略调整只能利用ms本地的信息。

接口如下：

struct ObChunkServer

{

/// address of the server

ObServer addr\_;

/// indicate if the current request has requested this server

int status\_;

};

class ObChunkServerTaskDispatcher

{

public:

/// replicas\_in\_out available servers for current request

/// last\_query\_idx\_in last query used which chunkserver

/// tablet\_in this request will access which tablet

/// return < 0 on error; >= 0 indicate to use which cs for current query

int select\_cs(ObChunkServer \* replicas\_in\_out, const int32\_t replica\_count\_in,

const int32\_t last\_query\_idx\_in, const ObRange & tablet\_in);

int select\_cs(ObChunkServer \* replicas\_in\_out, const int32\_t replica\_count\_in,

const int32\_t last\_query\_idx\_in, const ObCellInfo & get\_cell\_in);

/// replica\_in\_out filled with chunkserver's addres used for current subquery

/// replica\_count\_in sizeof replica\_in\_out

/// query\_range\_in [in] query range of current subquery

/// tablet\_range\_out current subquery will query which tablet, used for subquery division

/// return OB\_SUCCESS on success, OB\_ITER\_END if no other subquery needed, other for error

int get\_tablet\_locations(ObChunkServer \*replica\_in\_out, const int32\_t replica\_count\_in,

const ObRange & query\_range\_in, ObRange, ObRange & tablet\_range\_out);

int get\_tablet\_locations(ObChunkServer \*replica\_in\_out, const int32\_t replica\_count\_in,

const ObCellInfo & get\_cell\_in, ObRange &tablet\_range\_out);

};

ObChunkServerTaskDispatcher负责管理tablet location信息，同时统计对每个cs的访问状态，根据访问状态制定负载均衡策略。

ObChunkServerTaskDispatcher::get\_tablet\_location，查询tablet location cache，得到当前请求可以访问的多个replica副本。

ObChunkServerTaskDispatcher::select\_cs，根据本次请求的历史访问情况和全局cs负载信息，确定本次rpc请求访问哪个cs，并返回本次请求会访问的tablet的range。每次rpc访问过的replica历史记录必须由rpc自己保存。

## 调度之备份任务

在并发处理scan请求的时候，为了保证请求的latency，需要为“慢”rpc请求启动备份任务，因此一个子请求可能有多个备份任务。同一个子任务之间只需要第一个完成的子任务的结果。

关于在何时发起备份任务，一个简单的策略就是记录所有跨越整个tablet的子任务的请求到平均处理时间，如果一个子任务超过这个平均时间还没有返回，就启动该子任务的备份任务。备份任务的策略与负载均衡的策略相同，必须集中处理。

我们使用下面的类来管理备份任务，接口如下：

class ObMsSubScanRequest

{

public:

int init(const ObRange & query\_range, const bool scan\_full\_tablet,

const ObChunkServer cs\_replicas[], const int32\_t replica\_count);

/// create a rpc event for this sub request

ObMsRpcEvent \*create\_rpc\_event(ObScanParam &scan\_param, ObServer & cs);

/// check if agent\_event belong to this, and if agent\_event is the first finished backup task

/// if agent\_event belong to this, set belong\_to\_this to true, and increment finished\_backup\_task\_count\_

/// if agent\_event belong to this, and it is not the first finished backup task, agent\_event

/// will be directly destroyed

int agent\_event\_finish(ObMsRpcEvent \*& agent\_event, bool &belong\_to\_this);

/// called to check if this tablet has been scanned over or tablet has been divided

/// if tablet has scanned over and tablet has not been divided, return OB\_ITERATOR\_END

int get\_next\_scan\_param(ObScanParam & param);

const ObRange &get\_query\_range()const;

bool scan\_full\_tablet()const;

/// check if this sub request finished, if finished

bool finish() const;

/// get the result of current sub request

ObMsRpcEvent \* get\_result();

private:

static const int32\_t MAX\_BACKUP\_TASK\_NUM = 16;

/// does this sub request scan a full tablet

bool scan\_full\_tablet\_;

ObRange query\_range\_;

uint64\_t backup\_agent\_tasks\_[MAX\_BACKUP\_TASK\_NUM];

int32\_t triggered\_backup\_task\_count\_;

int32\_t finished\_backup\_task\_count\_;

ObMsRpcEvent \*result\_;

ObChunkServer cs\_replicas\_[ObChunkServerTaskDispatcher::MAX\_REPLICA\_COUNT];

int32\_t total\_replica\_count\_;

int32\_t tried\_replica\_count\_;

int32\_t last\_tried\_replica\_idx\_;

};

我们将策略相关的内容封装在下面的类中，该类是一个单粒，接口如下：

class ObMsBackupTaskStrategyManager

{

public:

/// record agent event processing info

void rpc\_event\_finish(const ObMsRpcEvent &agent\_event, const ObMsSubScanRequest & request);

/// check if a subrequest need backup tasks

bool need\_backup\_task(const ObMsSubScanRequest & request)const;

/// check if need start to check unfinished sub requests

bool need\_backup\_task\_check(const int32\_t total\_sub\_request\_count,

const int32\_t finished\_sub\_request\_count);

private:

};

处理scan请求的类如下，这里只描述与分发、调度、备份任务相关的部分：

class ObMsScanEvent:public ObMsRequestEvent

{

public:

/// called by working thread when receive a request from client

/// these two functions will trigger rpc event which will non-blocking rpc access cs

virtual int set\_request\_param(ObScanParam &scan\_param, const int64\_t timeout\_us);

/// called by working thread after set\_request\_param, this function will block until

/// one of the rpc event finish

virtual int reap\_agent\_events(const int64\_t timeout\_us);

private:

static const int32\_t MAX\_SUBREQUEST\_NUM = 1024;

int32\_t total\_sub\_request\_count\_;

int32\_t finished\_sub\_request\_count\_;

ObMsSubScanRequest sub\_requests\_[MAX\_SUBREQUEST\_NUM];

/// other properties associated with result processing

};

ObMsScanEvent::set\_request\_param的处理流程如下：

///创建子任务，并发起异步调用

while ObChunkServerTaskDispatcher::get\_tablet\_locations != OB\_ITER\_END

创建新的子任务（ObMsSubScanRequest）并初始化

调用ObMsSubScanRequest::create\_rpc\_event创建agent\_event

调用ObMultClientManager::send\_request异步发起请求

endwhile

ObMsScanEvent::reap\_agent\_events的处理流程如下：

while 请求没有处理完成

等待rpc任务完成

if rpc任务不属于当前请求///通过event\_id\_判断

释放rpc任务

else

找到rpc任务所属的subrequest，调用ObMsSubScanRequest::agent\_event\_finish

if 该rpc任务是所属的subrequest的第一个完成的备份任务

处理该结果 /// 通过其他类描述具体实现

if ObMsSubScanRequest::get\_next\_scan\_param返回值不为OB\_ITERATOR\_END

创建新的ObMsSubScanRequest并发起异步rpc调用

endif

endif

endif

endwhile

## cs&ms职责划分

在ObScanParam中包含以下信息：

ObScanParam

* Basic Column
* Composite Column
* Filters
* GroupByParam
  + Groupby columns
  + Aggregate columns
  + Composite columns
  + Filters
* OrderBy
* Limit info

cs和ms的指针可以通过描述各自处理的参数进行描述。

### 对原始行的操作

ObScanParam::BasicColumn、ObScanParam::CompositeColumn、ObScanParam::Filters这部分计算都是在cs完成的，因此在转发给cs的子请求中必须包含这部分内容。

### GroupBy

Groupby的所有运算都需要除了Filter之外都需要同时在ms和cs执行。

cs要处理的内容包括：ObScanParam::ObGroupByParam::GroupByColumns、ObScanParam::ObGroupByParam::AggregateColumns、ObScanParam::ObGroupByParam::CompositeColumns。同时，这些结构到了ms还需要重新计算一遍。

ObScanParam::ObGroupByParam::Filters只能在ms处理，因为Filters是针对groupby之后的全局结果的。发给cs的请求中不能包含这部分信息。

#### AggregateColumn之sum、count、max、min

这些运算都是满足交换律的，可以不需要做任何修改就发送给cs。ms在结果的基础上继续做聚集运算。

#### AggregateColumn之avg

对于客户端发过来的avg请求，需要转换为三个列：sum、count、sum/count。cs处理这3各列，ms在sum和count的结果上进行聚集运算，在ms端重新计算sum/count。这样的转换在解析客户端请求的时候进行，通过修改is\_return保证协议兼容性，也就是说在解码客户端请求的时候碰到avg，分解为sum，count，sum/count，只是前面两列设置为不返回，后面一列设置为返回。

#### AggregateColumn之distinct、count(distinct)

这两个功能比较坑爹。暂时没有这个需求，所以在0.3中先不支持。

#### Composite Column

ms必须计算composite column，因为composite column是针对全局的结果。cs是否计算composite column取决与cs是否执行top k近似，因为composite column在cs不能用于属性过滤，也不能作为全局的结果，ms更加不能利用各个cs的composite column自己进行聚合运算。所以cs上计算composite column只能够用于top k近似。因此cs是否进行composite column的计算取决于原始请求中是否有top k近似。

#### 参数变化

现在我们只处理max、min、sum、count这几个聚集函数，这些函数满足交换律和结合律。ms处理的是cs返回的结果，因此是已经groupby之后的结果。因此在存在groupby的情况下cs和ms的参数是不同的。因为ms处理的是中间结果。遵守先定义后使用的协议标准，假设原始请求如下：

|  |  |  |  |
| --- | --- | --- | --- |
| ObScanParam | 原始列 | I |  |
|  |  | A |  |
|  |  | B |  |
|  |  | C |  |
|  | 针对原始列的计算 | I+C as D |  |
|  | GroupBy | Group划分列 | A |
|  |  |  | B |
|  |  | Aggregate 列 | count(C) as COUNT\_C |
|  |  |  | sum(D) as SUM\_D |
|  |  | 复合列 | SUM\_D/COUNTC |

该请求对应的sql为select count(C) as COUNT\_C, sum(C+I) as SUM\_D, SUM\_D/COUNT\_C from table group by A,B

ms内部计算的时候会为每个临时表中第一次出现的列分配一个index，表示该列在中间表当前行的偏移。在当前实现中ms只会处理两层结果，意味着会有两个中间表：basic\_result（表示原始行以及在原始行上进行的四则运算的结果），group\_result（表示对原始行上进行groupby之后的结果）。cs需要处理的请求在翻译后表示为：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ObScanParam | 原始列 | I |  | basic\_result.0 |
|  |  | A |  | basic\_result.1 |
|  |  | B |  | basic\_result.2 |
|  |  | C |  | basic\_result.3 |
|  | 针对原始列的计算 | C+I as D |  | basic\_result.4 |
|  | GroupBy | Group划分列 | basic\_result.1 | group\_result.0 |
|  |  |  | basic\_result.2 | group\_result.1 |
|  |  | Aggregate 列 | count(basic\_result.3) as COUNT\_C | group\_result.2 |
|  |  |  | sum(basic\_result.4) as SUM\_D | group\_result.3 |
|  |  |  | group\_result.3/group\_result.2 | group\_result.4 |

由于ms看到的是cs groupby之后的结果，因此ms在进行groupby运算的时候见到的中间结果就不同，导致groupby的参数不同，ms看到的groupby的参数解析如下：

|  |  |  |  |
| --- | --- | --- | --- |
| GroupBy | Group划分 | cs\_result.0 | group\_result.0 |
|  |  | cs\_result.1 | group\_result.1 |
|  | aggregate | count(cs\_result.2) | group\_result.2 |
|  |  | sum(cs\_result.3) | group\_result.3 |
|  |  | group\_result.3/group\_result.2 | group\_result.4 |

groupby在计算的时候只关心上一个临时表每一行的输出，只保存各种计算利用的cell取上一个临时表中每行的第几个元素（即只保存了），因此ms和cs的groupbyparam的参数是不同的。ms的groupbyparam中，group划分列和aggregate列都作用于自己在groupbyparam中的下标。复合列保持不变。

### LimitInfo

ms肯定会执行limitinfo。如果没有近似，那么cs不需要执行limitinfo；如果存在近似，cs按照近似算法得到的limitinfo执行limit。

先实现一种简单的算法，近似度表示从每个tablet获取的记录数量与k的关系，全局top k等于每个tablet求取top k\*PRECISION\_FACTOR\*precision。其中PRECISION\_FACTOR是一个整数，可以在配置文件中进行配置，默认值为100。

### OrderBy

ms肯定需要执行orderby。cs是否执行orderby取决于是否有top k近似，只有原始请求中包含top k近似的时候cs需要执行orderby。

### 参数解析接口

int ob\_ms\_decode\_scan\_param(const ObScanParam &org\_scan\_param,

ObScanParam & ms\_scan\_param,

ObScanParam & cs\_scan\_param,

ObRange &scan\_range);

该接口根据上面描述的规则将原始参数转换为两个请求参数。一个用于向cs发送的请求，一个被ms使用。

## MS计算逻辑

### cs结果分类

cs返回的结果可以分为3类：成功且当前tablet已经计算结束、成功且当前tablet没有结束、失败。

对于失败的请求，如果是网络原因失败则重试下一个replica；如果请求本身必定会失败，比如由于当前ob所有计算都在内存如果内存不够导致的失败，那么整个请求失败，不再继续执行。

成功的两种情况都是请求的range在cs端没有处理完成，对于这种情况ms的处理逻辑是新建ObMsSubScanRequest，修改参数并发起新的rpc异步调用。

### cs结果的顺序

如果原始请求中不包含groupby，那么多cs返回的结果必须按照顺序进行处理，这种情况下ms会等待所有的cs否返回结果之后再进行处理。将所有ObMsSubScanRequest按照tablet range+结果中的last\_row\_key进行排序，并按排序输出结果。对于这种情况没有什么计算，就是按照特定的顺序返回cs的结果就ok了。这里存在的一个优化是，对于不包含orderby，当然也不包含groupby的请求，在满足limitinfo或者结果包的大小超过2M的时候就结束请求的处理。

如果原始请求中包含groupby，由于groupby的计算满足结合律和交换律，每次有cs返回结果后就立即进行计算。Groupby的代码已经在0.2的ms中设计并实现，这里不再赘述。

在原始请求包含groupby的情况下，存在一种情况，就是同一个group不可能出现在多个tablet中，这样其实在ms端是没有必要进行运算的。考虑到首先ms没有简单的办法预知这样的情况，同时ob采用hash的方式进行groupby，这样的计算代价并不大，暂时不对这种情况进行优化。

groupby、orderby、limit的计算重用现有代码的oceanbase::mergeserver::ObGroupByOperator。

### 内存问题

ms在进行计算的时候都是把中间结果存放在内存中。这样存在两个问题：如何限制内存使用；如果内存不够如何处理。

限制内存使用有两种方法，一种方案是限制每个请求使用的内存；另外一种方案是限制进程内存使用，而不限制每个请求的内存使用。第一种方案可能导致出现的情况是即使进程的内存还有剩余，但是确无法处理较大的请求；第二种方案可能导致某个进程持有太多的内存导致其他线程都无法工作。ms采用两种方案结合，每个线程使用的内存不能够超过进程可以使用的内存的某个百分比，该百分比可以配置。

如果中间结果太大，无法保存在内存中，初期先不考虑这种情况。因为这个东西不存在需要调整整个代码结构的问题，不需要过度设计。存放中间结果的类ObGroupByOperator，发现超过内存限制的情况下，把现有数据按照groupby的rowkey排序，写入磁盘。在cs的数据返回结束后，调用ObGroupByOperator::end\_inputs()，ObGroupByOperator进行多路归并排序，得到最终结果。该类的访问接口保持不变，ms的代码框架也不存在改动。本期先不考虑这个问题。