**MergeServer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 | 全文 | 新建 | 2010-09-06 | 无施、解伦 |
| **2** | 0.2 | 2.4,2.6 | 需要为CS提供可扩展的代码级支持 | 2010-09-15 | 无施、谢伦 |
| **3** | 0.3 | 全文 | 根据实现情况进行修订 | 2010-10-22 | 无施、解伦 |
|  |  |  |  |  |  |

# 系统设计

## 系统综述

MergeServer模块是OceanBase系统的客户端与OceanBase系统间的中间层，它是客户端的代理。主要负责代理客户端查询数据，并对数据进行处理。MergeServer可以对数据进行的处理包括：按列排序、跨表join、结果分页、属性过滤等。

## 名词解释

**OB**：全称OceanBase，核心系统研发部的海量存储系统，一期解决淘宝的收藏夹应用；

**RS**：全程Root Server，OB系统的meta模块，负责管理所有表的元信息；

**UPS**：全称Update Server，OB系统更新子系统，通过主备保证可靠性；

**CS**：全称Chunk Server，OB静态表系统，存储每天合并UPS更新数据生成的静态表；

**API**：全称client API，客户端通过API访问OB系统，UPS也可通过API访问CS静态表数据；

**数据存储模块**：OB数据存放模块，包括CS和UPS

**MS**：全称MergeServer在API与数据存储模块之间的中间层，负责对结果进行中间处理，是API的代理模块；

## 实现的功能

静态数据与动态数据合并：OB中，最近的更新数据存放在UPS中，静态数据存放在CS中。每次处理client的查询的时候，MS需要读取CS的静态数据和UPS的动态数据进行合并，将合并的结果返回client。功能必须在一期完成。

跨表join：OB支持多表的join，而不同的表格的数据可能分布在不同的CS以及UPS上，MS需要实现跨表join的功能。以收藏夹为例，每次查询得到的宽表结果都需要和UPS中collect\_item表进行join从而得到最终的结果。OB中，暂时假设所有的join都是由宽表产生的，暂时不考虑数据库语义的join。另外，跨表join需要考虑原子性，同一次查询请求可能需要多次查询UPS进行join，为了保证一致性，多次的join查询必须看到UPS的一个一致的状态，一致性的保证可以通过btree的快照实现，一期暂不实现。

结果排序：seq中的order by功能。支持按照查询结果中的任意多列进行全局的排序。[ORDER BY <columnName1> [ASC | DESC] [, <columnName2> [ASC | DESC] ...]]。如果两行的columnName1列值相同则按照columnName2排序，直到区分出两行的顺序或者所有指定的排序列都比较完成。该功能在一期完成。

结果分页：当单次查询结果较大的时候，client往往会分页展示，只需要全局结果中从某个偏移开始的给定数量的结果。该功能在一期完成。

RS信息缓存与管理：MS端需要缓存的meta信息包括tablet分布信息、schema信息。tablet分布信息用于定位数据；schema信息用于指导静态数据与动态数据的合并以及多表join操作。该功能在一期完成。

简单的统计功能：seq中的group by功能。可以提供：COUNT，MIN，MAX，AVG等简单的统计函数。一期暂不实现，后续计划视需求而定。

属性过滤：按照给定的过滤条件对结果进行过滤，仅输出符合条件的结果。属性过滤功能在二期完成。

综上所述：MS在一期需要完成功能包括静态数据与动态数据的合并、跨表join、结果排序、结果分页、RS信息缓存与管理；二期需要完成的功能包括跨表join的一致性、group by、属性过滤。

## 性能指标

MS模块只负责处理读请求，它的性能取决于数据存储模块的性能。单次请求的latency为该请求需要访问的所有数据存储模块的最长的latency。模块的qps为：模块的线程数量\*1/处理请求的平均latency。在压力为1000qps的情况下，4个9的MS对结果的处理时间小于3ms。

多机通信性能优化：MS在处理请求的时候需要同多个后端进行通信，如果采用多线程模型，每个线程都会阻塞在等待网络调用上，整个模块能够同时并发处理的请求数量等于模块开启的线程数量。在一期实现中，模块使用tbnet，单个请求导致的对多个存储模块的网络访问是并发的，但是MS本身处理请求是阻塞的，也就是说MS在收到请求后会同时向多个存储模块发出查询请求，然后阻塞直至所有存储模块返回。在一期实现中，通过对存储模块的并发网络访问解决latency的问题。二期实现，解决并发度和线程切换导致cpu资源消耗的问题。

# 模块设计

## 概述

MS处理单个请求的粗略流程如下：

根据请求定位chunkserver

向chunkserver发起查询请求

向updateserver发起查询请求

执行merge

执行join

执行排序等处理

响应客户端

按照请求处理流程，子模块划分如下：

tablet位置信息缓存模块：负责管理缓存的tablet位置信息

schema管理模块：负责管理schema，在进行merge和join的时候需要schema信息。由于schema存在全局的版本号，schema是MS中唯一存在状态的数据。

中间结果处理模块：负责merge、join、排序、分页、属性过滤等结果处理

RPC模块：主要负责与系统其他模块的交互

请求处理流程模块：负责请求处理逻辑，调用其他子模块的接口完成请求处理

模块框架：负责处理启动流程、网络框架

### 子模块调用关系图



#### 子模块数据流图



## tablet位置信息缓存子模块

### 模块描述

子模块负责管理最近两天（当天和前一天）的schema信息。schema会在两种情况下被更新：MS启动的时候需要从rootserver获取当前有效的schema；如果CS返回的schema的timestamp比MS管理的最新的schema更新，MS从rootserver获取最新的schema并进行更新。如果在发现CS返回的schema的timestamp比MS管理所有schema都新的时候才向rootserver获取最新的schema，会导致本次请求的latency加大，作为优化，MS可以向rootserver轮询最新的schema，减少由于schema timestamp陈旧导致请求latency增大的概率，优化方案可以考虑在模块的第二期实现。

### 关键数据结构与算法

### 模块对外接口

class ObMergerSchemaManager

{

public:

// max version

static const uint64\_t MAX\_VERSION\_COUNT = 4;

// init the newest schema

int init(const common::ObSchemaManager & schema);

// add a new schema, del the oldest-version and not used scheam if fullfilled

int add\_schema(const common::ObSchemaManager & schema,

const common::ObSchemaManager \*\* manager = NULL);

// get schema of pointed version if not exist return null

const common::ObSchemaManager \* get\_schema(const int64\_t version);

// release the schema not used already

int release\_schema(const int64\_t version);

// get the latest schema version

inline int64\_t get\_latest\_version(void) const;

// get the oldest scheam version

int64\_t get\_oldest\_version(void) const;

// get last update timestamp

inline int64\_t get\_update\_timestamp(void) const;

// dump all schema info

int print\_info(void) const;

};

## tablet位置信息缓存子模块

### 模块描述

模块用于缓存tablet的位置信息，必须支持按时间淘汰。仅在CS返回tablet不存在的情况下，MS需要查询rootserver获取最新的位置信息。由于OB中，tablet的位置信息更新并不频繁，暂时可以使用有序数组的方式组织缓存数据。

### 关键数据结构与算法

### 模块对外接口

class ObMergerTabletLocationCache

{

public:

/// init the cache item count

int init(const uint64\_t mem\_size, const uint64\_t count, const int64\_t timeout);

/// get table\_id.rowkey location server list

int get(const uint64\_t table\_id, const common::ObString & rowkey,

common::ObTabletLocationList & location);

/// set the new range location, if range not valid, then del the old info

int set(const common::ObRange & range, common::ObTabletLocationList & location);

/// delete the cache item according to table\_id.rowkey

int del(const uint64\_t table\_id, const common::ObString & rowkey);

/// cache item count

uint64\_t size(void);

};

## 中间结果处理子模块

### 模块描述

中间结果处理模块负责：静态数据与动态数据的合并（merge）、跨表join、结果排序、属性过滤、结果分页。

### 关键数据结构与算法

使用cell的二维数组来保存中间结果，并在二维数组的接触上进行数据处理。对于get请求，二维数组退化为一维数组，即使get请求查询多行，我们也将所有的结果按照一个大行进行处理；对于scan，scan结果中的一行对应了二维数组中的一行，对于结果中不存在的列，在二维数组中以NULL表示。get请求按照一个大行进行处理，而不是将get请求中的一行与二维数组中的一个行对应，因为get请求中的多行可能获取不同的列，甚至获取不同的表，这就意味着如果把get中的行与二维数组中的行对应的话，这个二维数组不规整，并且很难处理。

二维数组通过协助类ObMerger完成静态数据与动态数据的merge操作。ObMerge的输入是从CS和UPS得到的ObScanner，输出是按照给定顺序的cell流。

在算法描述中，按照下面的方式描述数据：table.row.column。比如，1.2.5表示的是table1中，rowkey为2的行的columnid为5的cell的数据。

#### merge

了解merge的处理过程，首先要了解UPS和CS处理查询的返回结果。

这里对get请求中出现的一种情况做一个定义：ObGetParam或者ObGetParam的响应中，一个或者多个连续的cell如果属于同一个table的同一行，即它们拥有相同的tableid和rowkey，我们称这些连续出现的cell为一个subrow。

|  |  |  |
| --- | --- | --- |
|  | scan请求 | get请求 |
| chunkserver | 1. 结果按照行有序 2. 每行数据包含该行的所有cell，不需要对cell进行过滤 3. 行内cell无序 | 1. 响应subrow的顺序与请求subrow的顺序相同，任意请求subrow都必须被响应（响应中至少包含一个cell） 2. subrow内的cell可以乱序 3. subrow必须作为一个整体进行对待，在响应缓冲区满的情况下，响应必须以subrow进行边界对齐，对ObGetParam中出现的任意subrow，要么响应subrow中的所有cell，要么不响应subrow的任何cell 4. 如果请求中的某个subrow对应的rowkey或者table在server上不存在，server必须返回该subrow的至少一个cell，该cell设置OB\_ROW\_NOT\_EXIS或者OB\_TABLE\_NOT\_EXIST标志 5. 任意subrow的响应中已经至少出现了一个cell，请求subrow中的其他cell在满足以下任意一种情况，可以不被响应：a) rowkey不存在；b) cell为空（NULL）或者被删除（delete row）；c) 适用于updateserver，rowkey被删除并且后续没有关于所请求的cell的更新（包括insert）操作 |
| updateserver | 1. 结果按照rowkey有序 2. 每行数据包含该行的所有cell，不需要对cell进行过滤 3. 行内cell无序 4. 行内同一cell的多次update可以可以分布在行内的任意位置 5. 如果请求范围中的某个rowkey已经被删除，响应中必须包含该rowkey，且rowkey对应的第一个cell必须标志OB\_ROW\_DELETED；如果在删除之后有insert或者update操作，这些操作在可以追加在第一个cell的后面 | 1. 与chunkserver的get请求的所有要求相同 2. 如果同一个cell在一个subrow中出现多次，那么对该cell的任意一次由客户端产生的update能且只能在与请求subrow对应的响应subrow中出现一次 3. 对同一个cell的客户端产生的多次update可以分布在响应subrow的任意位置 4. 如果某个请求subrow对应的rowkey已经被删除，该subrow必须被响应，响应中必须有且只有一个cell标志为OB\_ROW\_DELETED，并且这个cell必须出现在响应subrow的第一个位置（意味着updateserver必须对多次的delete row操作进行合并）；如果在删除之后有insert或者update操作，这些操作在可以追加在第一个cell的后面 |

merge的处理流程如下：

定位目标结果中多个属于同一table同一行的所有cell，用subrow来描述这些cell

/// 读取cs结果中的下一行数据

while !cs\_result.end() and ! cs\_result.row\_end() // 遍历cs结果中同一table同一行的所有cell

for each target\_cell in subrow // 将cs结果中的当前cell应用到subrow的所有匹配的cell

if cs\_result.current\_cell match target\_cell //同一行同一列

target\_cell.apply ( cs\_result.current\_cell)

endif

endfor

endwhile

/// 读取ups结果中的下一行数据

while !ups\_result.end() and ! ups\_result.row\_end() // 遍历ups结果中同一table同一行的所有cell

for each target\_cell in subrow // 将ups结果中的当前cell应用到subrow的所有匹配的cell

if ups\_result.current\_cell match target\_cell //同一行同一列

target\_cell.apply ( ups\_result.current\_cell)

endif

endfor

endwhile

#### join操作

不管是get请求还是scan请求，对于一个生成的结果，那么其中哪些cell需要做join操作都是确定的。按照需要join的cell在二维数组中出现的顺序，逐个添加到GetParam中，然后想UPS发送get请求，使用得到的结果进行join，这样结果中的cell与需要join的cell是一一对应的，遍历一边二维数组可以完成join。在每日合并的时候，join数据会放入cache中，因此对于需要join的cell需要首先判断是否命中cache。下面描述join的逻辑，该算法是建立在merge的结果之上，实际实现的过程中，某些操作可能与merge合并，减少对结果的遍历。

vector join\_get\_param

vector join\_get\_cell\_offset

for each cell in cell\_array produced by merge process

if cur\_cell 需要join

if cache中存在该cell的join数据

使用cache数据进行merge操作

else

将当前cell加入join\_get\_param中

将当前cell的offset加入join\_get\_cell\_offset

endif

endfor

使用join\_get\_param向updateserver发出查询请求

使用查询结果以及join\_get\_cell\_offset中记录的需要join的cell的位置信息进行join操作

#### 排序

对于scan请求，join后的结果按行保存在二维数组中，只需要按照order by的参数对二维数组的行进行排序就可以了。

#### 分页

分页表述为limit(offset,count)，翻译成二维数组对应的描述方式：offset对应二维数组的行下表；count表示从二维数组的第offset行开始读取count行数据。

### 模块对外接口

#### ObCellArray

该类封装了cell的随机访问数组，并支持orderby和分页。

/// @class ObCellInfo vector, can access by offset

class ObCellArray : public oceanbase::common::ObIterator

{

public:

friend class iterator;

/// @fn append a cell into the array, the whole cell whill be copied

int append(const oceanbase::common::ObCellInfo &cell,

oceanbase::common::ObCellInfo \*& cell\_out);

/// @fn expand the array

int expand(int32\_t expanding\_size);

/// @fn apply changes to a given cell

int apply(const oceanbase::common::ObCellInfo &cell, const int64\_t offset,

oceanbase::common::ObCellInfo \*& cell\_out);

/// @fn get a specific cell in this array, there will be no reference number

int get\_cell(const int64\_t offset, oceanbase::common::ObCellInfo\*& cell) const;

/// @fn get cell according to operator []

oceanbase::common::ObCellInfo & operator[](int64\_t offset);

/// @fn get number of cell in the array

int64\_t get\_cell\_size()const;

/// @fn clear all infomation and cell stored in the cell

void clear();

/// @fn get memory size used by this array

int64\_t get\_memory\_size\_used();

public:

class iterator

{

public:

iterator &operator ++();

iterator operator ++(int);

iterator operator +(int64\_t inc\_num);

oceanbase::common::ObCellInfo & operator\*();

oceanbase::common::ObCellInfo \* operator->();

bool operator !=(const ObCellArray::iterator &other);

};

iterator begin();

iterator end();

public:

/// @struct describe each cell order

struct OrderDesc

{

int32\_t cell\_idx\_;

int32\_t order\_;

};

/// @fn order all rows, number of cell in each row is identified by row\_width,

/// @note only affect output of member functions of ObIterator

int orderby(int64\_t row\_width, OrderDesc \*order\_desc, int64\_t desc\_size);

/// @fn limit the output

/// @param offset only out put row in range [offset, offset+count), offset begin with 0

/// @param count the row count, 0 means unlimited

/// @note if need order the output, one should call orderby first, and row\_width

/// must be consistent, this class's implementation will not check these rules

/// @note only affect output of member functions of ObIterator

int limit(int64\_t offset, int64\_t count, int32\_t row\_width);

public:

void reset\_iterator();

virtual int next\_cell();

virtual int get\_cell(oceanbase::common::ObCellInfo\*\* cell);

virtual int get\_cell(oceanbase::common::ObCellInfo \*\*cell, bool \* is\_row\_changed);

};

#### ObMergeJoinOperator

该类实现了真正的merge，join逻辑，并把产生的结果保存在一个ObCellArray中。

/// @class do real jobs like merge, join; but it only represent stage result

/// ObMergeJoinAgent will merge stage result into final result

class ObMergeJoinOperator : public ObCellArray

{

public:

virtual int next\_cell();

virtual int get\_cell(oceanbase::common::ObCellInfo\*\* cell);

public:

/// @fn set request parameter

/// @param max\_memory\_size if the intermediate results take memory size more than this

/// stop process, and continue when user call this function next time

int set\_request\_param(const oceanbase::common::ObScanParam &scan\_param,

ObCellStream &cs\_stream,

ObCellStream &ups\_stream,

ObCellStream &ups\_join\_stream,

const oceanbase::common::ObSchemaManager &schema\_mgr,

const int64\_t max\_memory\_size);

int set\_request\_param(const oceanbase::common::ObGetParam &get\_param,

ObCellStream &cs\_stream,

ObCellStream &ups\_stream,

ObCellStream &ups\_join\_stream,

const oceanbase::common::ObSchemaManager &schema\_mgr,

const int64\_t max\_memory\_size);

/// @fn do merge part of part of or whole scan request

///

/// @param ups\_join\_stream cell stream get from update server, used for join

int do\_merge\_join();

/// @fn clear all result stored

void clear();

/// @fn reset intermediate result

void reset();

/// @fn check if result has been finised yet

bool is\_request\_finished()const ;

};

#### ObMergeJoinAgent

该类是子模块对外暴露的接口，同时也是mergeserver提供给chunkserver每日合并的代码。其他模块直接使用该接口。ObMergeJoinAgent与ObMergeJoinOperator的区别是：由于Filter的存在，可能会出现查询的全集很大，但是最终结果可能很小的情况，为了对这种情况进行优化处理，在实现中会边查询边过滤。因此一次ObMergeJoinAgent的查询可能多次调用ObMergeJoinOperator产生多次的中间结果，每次产生的中间结果在过滤以后放入最终结果中。

/// @class only encapsulate ObScanMergeJoinAgentImp and ObGetMergeJoinAgentImp

class ObMergeJoinAgent: public ObMergeJoinAgentImp

{

public:

ObMergeJoinAgent();

~ObMergeJoinAgent();

public:

virtual int get\_cell(oceanbase::common::ObCellInfo \* \*cell);

virtual int next\_cell();

public:

/// @param max\_memory\_size 只有在chunkserver每日合并的时候，

/// 将max\_memory\_size设置为大于0，表示合并过程中最多使用的内存大小，

/// 如果以后一个tablet在合并过程中可以完全放入内存，可以取消该参数

virtual int set\_request\_param(const oceanbase::common::ObScanParam &scan\_param,

ObCellStream &cs\_stream,

ObCellStream &ups\_stream,

ObCellStream &ups\_join\_stream,

const oceanbase::common::ObSchemaManager &schema\_mgr,

const int64\_t max\_memory\_size = -1);

virtual int set\_request\_param(const oceanbase::common::ObGetParam &get\_param,

ObCellStream &cs\_stream,

ObCellStream &ups\_stream,

ObCellStream &ups\_join\_stream,

const oceanbase::common::ObSchemaManager &schema\_mgr,

const int64\_t max\_memory\_size = -1);

virtual void clear();

virtual bool is\_request\_fullfilled();

};

## RPC远程调用子模块

### 模块描述

模块主要封装了对系统中其他节点的单次远程调用，这些节点包括：rts，ups，cs。

### 关键数据结构与算法

### 模块对外接口

// this class encapsulates network rpc interface as bottom layer,

// and it only take charge of "one" rpc call.

// if u need other operational work, please use rpc\_proxy for interaction

class ObMergerRpcStub

{

public:

// warning: rpc\_buff should be only used by rpc stub for reset

// param @rpc\_buff rpc send and response buff

// @rpc\_frame client manger for network interaction

int init(const common::ThreadSpecificBuffer \* rpc\_buffer,

const common::ObClientManager \* rpc\_frame);

// get tables schema info through root server rpc call

// param @timeout action timeout

// @root\_server root server addr

// @merge\_server merge server addr

// @is\_merger merge server status

int register\_server(const int64\_t timeout, const common::ObServer & root\_server,

const common::ObServer & merge\_server, const bool is\_merger) const;

// get update server addr through root server rpc call

// param @timeout action timeout

// @root\_server root server addr

// @update\_server output server addr

int find\_server(const int64\_t timeout, const common::ObServer & root\_server,

common::ObServer & update\_server) const;

// get tables schema info through root server rpc call

// param @timeout action timeout

// @root\_server root server addr

// @timestamp fetch cmd input param

// @schema fetch cmd output schema data

int fetch\_schema(const int64\_t timeout, const common::ObServer & root\_server,

const int64\_t timestamp, common::ObSchemaManager & schema) const;

public:

// get tablet location info through root server rpc call

// param @timeout action timeout

// @root\_server root server addr

// @root\_table root table name

// @table\_id look up table id

// @row\_key look up row key

// @scanner scaned tablets location result set

int fetch\_tablet\_location(const int64\_t timeout, const common::ObServer & root\_server,

const uint64\_t root\_table\_id, const uint64\_t table\_id,

const common::ObString & row\_key, common::ObScanner & scanner) const;

// get row info through rpc call

// param @timeout action timeout

// @server server addr

// @get\_param get parameter

// @scanner return result

int get(const int64\_t timeout, const common::ObServer & server,

const common::ObGetParam & get\_param, common::ObScanner & scanner) const;

// sort the locationlist asc by distance between input server ip

// retry another server in the list when the previous one failed

// waring: timeout is every rpc call time

// param @timeout action timeout

// @server input server addr

// @list unsorted location list

// @get\_param get parameter

// @scanner return result

int get(const int64\_t timeout, const common::ObServer & server,

common::ObTabletLocationList & list, const common::ObGetParam & get\_param,

common::ObScanner & scanner) const;

// scan row info through rpc call

// param @timeout action timeout

// @server server addr

// @scan\_get scan parameter

// @scanner return result

// @end\_flag is\_end flag result

int scan(const int64\_t timeout, const common::ObServer & server,

const common::ObScanParam & scan\_param, common::ObScanner & scanner) const;

// sort the locationlist asc by distance between input server ip

// retry another server in the list when the previous one failed

// waring: timeout is every rpc call time

// param @timeout action timeout

// @server input server addr

// @list unsorted location list

// @scan\_param scan parameter

// @scanner return result

int scan(const int64\_t timeout, const common::ObServer & server,

common::ObTabletLocationList & list, const common::ObScanParam & scan\_param,

common::ObScanner & scanner) const;

};

## Cell流式输出层子

### 模块描述

由于系统限制rpc包的大小为2M，客户端的一次请求可能需要发起多次rpc网络调用，因此子模块需要把对UPS和CS的多次访问封装在流式接口中：ObCellStream。

### 关键数据结构与算法

Cell的流式输出需要封装多次rpc调用，模块在对get请求进行拆分的时候，必须要保证子请求是按照原始请求中的subrow对齐的。

### 模块对外接口

#### cell的流式输出接口

class ObMSGetCellArray

{

public:

/// @fn get cell number

int64\_t get\_cell\_size()const;

/// @fn get cell according to operator []

const oceanbase::common::ObCellInfo & operator[](int64\_t offset);

};

typedef enum ServerType

{

CHUNK\_SERVER = 100,

UPDATE\_SERVER = 101,

UNKNOW\_SERVER = 102,

} server\_type;

/// @class encapsulate stream read of ups, cs or something else

class ObCellStream : public oceanbase::common::ObIterator

{

public:

/// @fn prepare to do get query, may just prepare,

/// whether doing actions was decided by implementation

/// @param timestamp, which version of schema was based by schema parsing

/// virtual int get(const server\_type type, const oceanbase::common::ObGetParam &get\_param,

/// const int64\_t timestamp) = 0;

/// @note 不使用ObGetParam是因为在join的时候上层调用的get可能包含很多cell，拆分由ObCellStream完成

/// virtual int get(const server\_type type, const oceanbase::common::ObReadParam &read\_param,

/// const ObCellArray &get\_param, const int64\_t timestamp) = 0;

virtual int get(const server\_type type, const oceanbase::common::ObReadParam &read\_param,

ObMSGetCellArray & get\_cells, const int64\_t timestamp)=0;

/// @fn prepare to do scan query, may just prepare,

/// whether doing actions was decided by implementation

virtual int scan(const server\_type type, const oceanbase::common::ObScanParam &scan\_param,

const int64\_t timestamp) = 0;

/// @fn set cache to cache result get according to rpc, if not set nothing will be cached,

/// cached content is [tableid,rowkey,columnid]->(array of ObObj)

virtual void set\_cache(oceanbase::common::ObVarCache<> &);

/// @fn get a row in the cache, 在这个函数中处理每日合并join的cache和bloomfilter

///

/// @param key 需要查询的cell

/// @param result ObScanner串行化后的结果

/// -# 如果命中cache，直接返回cache项

/// -# 如果bloomfilter发现rowkey对应的记录没有发生修改，那么返回一个特殊结果，结果中包含0个cell

virtual int get\_cache\_row(const oceanbase::common::ObCellInfo & /\*key\*/,

oceanbase::common::ObCachePair & /\*result\*/);

};

# 接口设计

## 对外接口

/// @class ObReadParam OB read parameter, API should not concern these parameters,

/// and mergeserver will directly ignore these parameters

class ObReadParam

{

public:

/// @fn get data whose timestamp is newer or as new as the given timestamp,

/// -# when reading cs, if not setted, the result is decided by the server;

/// -# when reading ups, this parameter must be setted

int set\_timestamp(const uint64\_t timestamp);

int get\_timestamp(const uint64\_t &timestamp)const;

/// @fn when reading cs, indicating whether the result (including intermediate result,

/// like sstable block readed from sstable) of this operation should be cached.

///

/// ups just ignores this parameter

int set\_is\_result\_cached(const bool cached);

int get\_is\_result\_cached(bool & cached) const;

/// @fn when reading ups, indicating whether the update server

/// should read frozen memory table only, if not setted,

/// result only being decided by timestamp.

///

/// cs just ignores this parameter

int set\_is\_read\_frozen\_only(const bool read\_frozen\_only);

int get\_is\_read\_frozen\_only(bool &read\_frozen\_only) const;

};

/// @class not implemented

class ObPropertyFilter;

/// @class ObScanParam接口变更

class ObScanParam : public ObReadParam

{

public:

enum Order

{

ASC,

DESC

};

/// set and get order by information

int add\_orderby\_column(const ObString & column\_name, Order order = ASC);

int add\_orderby\_column(const uint64\_t column\_id, Order order = ASC);

int get\_orderby\_column\_num(int64\_t &column\_num)const;

int get\_orderby\_column(ObString\* const & names, Order \* const & orders,

const int64\_t column\_array\_size)const;

int get\_orderby\_column(uint64\_t\* const & column\_ids, Order \* const & orders,

const int64\_t column\_array\_size)const;

/// set and get limit information

int set\_limit\_info(const int64\_t offset, const int64\_t count);

int get\_limit\_info(int64\_t &offset, int64\_t &count) const;

/// set and get filter

int set\_filter(const ObPropertyFilter & filter);

const ObPropertyFilter \* const get\_filter()const;

};

class ObScanner

{

public:

/// @fn indicating the timestamp of the data getted, must be setted by cs and ups

/// API should not concern timestamp information in ObScanner

/// @note implementation detail: if timestamp was not setted, the get\_timestamp

/// must return an error and print warn log, this feature can be used to tell API not

/// to concern timestamp information in ObScanner, because merge server will never

/// set this information

int set\_timestamp(const uint64\_t timestamp);

int get\_timestamp(uint64\_t &timestamp) const;

/// @fn indicating the request was fullfiled, must be setted by cs and ups

/// @param is\_fullfilled 当rpc的缓冲区不足导致scanner中只包含部分结果的时候设置为false

int set\_is\_req\_fullfilled(const bool &is\_fullfilled);

int get\_is\_req\_fullfilled(const bool &is\_fullfilled) const;

};

/// 对外接口

ObResultCode ob\_get(const ObGetParam& get\_param, ObScanner& scanner, ObOperateResult\* operation);

ObResultCode ob\_scan(const ObScanParam& scan\_param, ObScanner& scanne);

## 与chunkserver的接口

/// 内部接口, to chunkserver

ObResultCode cs\_get(const ObGetParam& get\_param, ObScanner& scanner, ObOperateResult\* operation);

ObResultCode cs\_scan(const ObScanParam& scan\_param, ObScanner& scanner);

## 与updateserver的接口

/// 内部接口, to updateserver

ObResultCode ups\_get(const ObGetParam& get\_param, ObScanner& scanner, ObOperateResult\* operation);

ObResultCode ups\_scan(const ObScanParam& scan\_param, ObScanner& scanner);

## 与Root Server接口

// 从Root Server网络获取table schema信息

ObResultCode **rts\_fetch\_schema**(const uint64\_t timestamp, ObTableSchema& schema);

## 提供给CS每日合并的接口

参见第2.4.3.3节对ObMergeJoinAgent的描述。