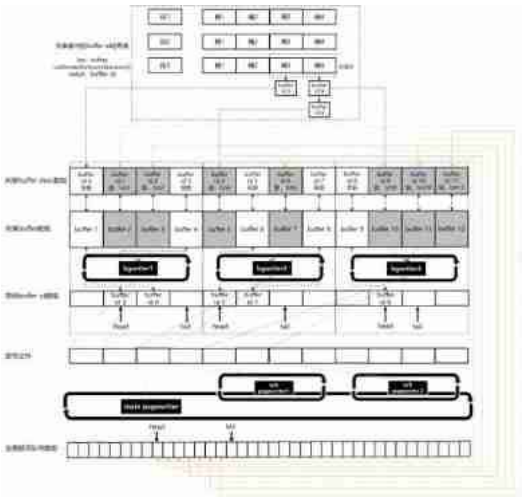
openGauss数据库源码解析系列文章——存储引擎源码解析（三）

2021-12-22 15:52:26 【openGauss】

上篇图文openGauss数据库源码解析系列文章——存储引擎源码解析（二）中，从astore和行存储索引机制两点对磁盘引擎进行了分享，本篇接着从磁盘引擎的行存储缓存机制、cstore、日志系统三方面展开介绍。

（五）行存储缓存机制

行存储缓存加载和淘汰机制如图1所示。



|图1 行存储缓存和淘汰机制示意图|

行存储堆表和索引表页面的缓存和淘汰机制主要包含以下几个部分：

1. 共享缓冲区内存页面数组下标哈希表

用于将远大于内存容量的物理页面与内存中有限个数的内存页面建立映射关系。该映射关系通过一个分段、分区的全局共享哈希表结构实现。哈希表的键值为buftag（页面标签）结构体。该结构体由“rnode”、“forkNum”、“blockNum”三个成员组成。其中“rnode”对应行存储表物理文件名的主体命名；“forkNum”对应主体命名之后的后缀命名，通过主体命名和后缀命名，可以找到唯一的物理文件；而“blockNum”对应该物理文件中的页面号。因此，该三元组可以唯一确定任意一个行存储表物理文件中的物理页面位置。哈希表的内容值为与该物理页面对应的内存页面的“buffer id”（共享内存页面数组的下标）。

因为该哈希表是所有数据页面查询的入口，所以当存在并发查询时在该哈希表上的查询和修改操作会非常频繁。为了降低读写冲突，把该哈希表进行了分区，分区个数等于NUM_BUFFER_PARTITIONS宏。在对该哈希表进行查询或修改操作之前首先需要获取相应分区的共享锁或排他锁。考虑到当对该哈希表进行插入操作时待插入的三元组键值对应的物理页面大概率不在当前的共享缓冲区中，因此该哈希表的容量等于“g_instance.attr.attr_storage.NBuffers + NUM_BUFFER_PARTITIONS”。该表具体的定义代码如下：

```
typedef struct buftag { RelFileNode rnode; /* 表的物理文件位置结构体 */ ForkNumber forkNum; /* 表的物理
```

2. 共享buffer desc数组

该数组有“g_instance.attr.attr_storage.NBuffers”个成员，与实际存储页面内容的共享buffer数组成员一一对应，用来存储相同“buffer id”（即这两个全局数组的下标）的数据页面的属性信息。该数组成员为BufferDesc结构体，具体定义代码如下：

```
typedef struct BufferDesc { BufferTag tag; /* Buffer页面标签 */ pg_atomic_uint32 state; /* 状态位、引用
```

（1）tag成员是该页面的（relfilenode，forknum，blocknum）三元组。

边栏推荐
1 第八章-支持向量
2 第五章-不确定性
3 第一章-绪论
4 与Jeff Dean聊M
5 亲手把360奇安信
6 物联网云平台管理
7 重启iptables为ll
8 Hit software co
9 Sqlmap source
10 Introduction to

猜你喜欢
Daily work gossip
【Unity3点】| Un
The story of 100 peo bottles of water
Transacti level
2021-11-15 software rehabilitation guide
2021-11-11 when we the quagmire of busi
Untitled notes
五一假期还是在家
Introduction to golar
À propos de l'acquis données par le fourr

随机推荐
What is oui in ou
DNS protocol

（2）state成员是该内存状态的标志位，主要包含BM_LOCKED（该buffer desc结构体内容的排他锁标志）、BM_DIRTY（脏页标志）、BM_VALID（有效页面标志）、BM_TAG_VALID（有效tag标志）、BM_IO_IN_PROGRESS（页面I/O状态标志）等。

（3）buf_id成员，是该成员在数组中的下标。

（4）wait_backend_pid成员，是等待页面unpin（取消引用）的线程的线程号。

（5）io_in_progress_lock成员，是用于管理页面并发I/O操作（从磁盘加载和写入磁盘）的轻量级锁。

（6）content_lock成员，是用于管理页面内容并发读写操作的轻量级锁。

（7）rec_lsn成员，是上次写入磁盘之后该页面第一次修改操作的日志lsn值。

（8）dirty_queue_loc成员，是该页面在全局脏页队列数组中的（取模）下标。

3. 共享buffer数组

该数组有“g_instance.attr.attr_storage.NBuffers”个成员，每个数组成员即为保存在内存中的行存储表页面内容。需要注意的是，每个buffer在代码中以一个整型变量来标识，该值从1开始递增，数值上等于“buffer id + 1”，即“数组下标加1”。

4. bgwriter线程组

该数组有“g_instance.attr.attr_storage.bgwriter_thread_num”个线程。每个“bgwriter”线程负责一定范围内（目前为均分）的共享内存页面的写入磁盘操作，如图4-11中所示。如果全局共享buffer数组的长度为12，一共有3个“bgwriter”线程，那么第1个“bgwriter”线程负责“buffer id 0 - buffer id 3”的内存页面的维护和写入磁盘；第2个“bgwriter”线程负责“buffer id 4 - buffer id 7”的内存页面的维护和写入磁盘；第3个“bgwriter”线程负责buffer id 8 - buffer id 11的内存页面的维护和写入磁盘。每个“bgwriter”进程在后台循环扫描自己负责的那些共享内存页面和它们的buffer desc状态，将被业务修改过的脏页收集起来，批量写入双写文件，然后写入表文件系统。对于刷完的内存页，将其状态变为非脏，并追加到空闲buffer id队列的尾部，用于后续业务加载其他当前不在共享缓冲区的物理页面。每个“bgwriter”线程的信息记录在BgWriterProc结构体中，该结构体的定义代码如下：

```
typedef struct BgWriterProc { PGPROC *proc; CkptSortItem *dirty_buf_list; uint32 dirty_list_size; int *
```

其中比较关键的几个成员含义是：

（1）dirty_buf_list为存储每批收集到的脏页面buffer id的数组。dirty_list_size为该数组的长度。

（2）cand_buf_list为存储写入磁盘之后非脏页面buffer id的队列数组（空闲buffer id数组）。cand_list_size为该数组的长度。

（3）buf_id_start为该bgwriter负责的共享内存区域的起始buffer id，该区域长度通过“g_instance.attr.attr_storage.NBuffers / g_instance.attr.attr_storage.bgwriter_thread_num”得到。

（4）head为当前空闲buffer id队列的队头数组下标，tail为当前空闲buffer id队列的队尾数组下标。

（5）next_scan_loc为上次bgwriter循环扫描时停止处的buffer id，下次收集脏页从该位置开始。

5. pagewriter线程组

“pagewriter”线程组由多个“pagewriter”线程组成，线程数量等于GUC参数（g_instance.ckpt_cxt_ctl->page_writer_procs.num）的值。“pagewriter”线程组分为主“pagewriter”线程和子“pagewriter”线程组。主“pagewriter”线程只有一个，负责从全局脏页队列数组中批量获取脏页面、将这些脏页批量写入双写文件、推进整个数据库的检查点（故障恢复点）、分发脏页给各个pagewriter线程，以及将分发给自己的那些脏页写入文件系统。子“pagewriter”线程组包括多个子“pagewriter”线程，负责将主“pagewriter”线程分发给自己的那些脏页写入文件系统。

每个“pagewriter”线程的信息保存在PageWriterProc结构体中，该结构体的定义代码如下：

```
typedef struct PageWriterProc { PGPROC* proc; volatile uint32 start_loc; volatile uint32 end_loc; volat
```

其中：

（1）proc成员为“pagewriter”线程属性信息。

（2）start_loc为分配给本线程待写入磁盘的脏页在全量脏页队列中的起始位置。

（3）end_loc为分配给本线程待写入磁盘的脏页在全量脏页队列中的结尾位置。

（4）need_flush为是否有脏页被分配给本“pagewriter”的标志。

（5）actual_flush_num为本批实际写入磁盘的脏页个数（有些脏页在分配给本“pagewriter”线程之后，可能被“bgwriter”线程写入磁盘，或者被DROP（删除）类操作失效）。

“pagewriter”线程与“bgwriter”线程的差别：“bgwriter”线程主要负责将脏页写入磁盘，以便留出非脏的buffer页面用于加载新的物理数据页；“pagewriter”线程主要的任务是推进全局脏页队列数组的进度，从而推进整个数据库的检查点和故障恢复点。数据库的检查点是数据库（故障）重启时需要回放的日志的起始位置LSN。在检查点之前的那些日志涉及的数据页面修改，需要保证在检查点推进时刻已经写入磁盘。通过推进检查点的LSN，可以减少数据库宕机重启之后需要回放的日志量，从而降低整个系统的恢复时间目标（recovery time objective，RTO）。关于“pagewriter”的具体工作原理，将在4.2.8小节进行更详细的描述。

6. 双写文件

一般磁盘的最小I/O单位为1个扇区（512字节），大部分文件系统的I/O单位为8个扇区。数据库最小的I/O单位为一个页面（16个扇区），因此如果在写入磁盘过程中发生宕机，可能出现一个页面只有部分数据写入磁盘的情况，会影响当前日志恢复的一致性。为了解决上述问题，openGauss引入了双写文件。所有页面在写入文件系统之前，首先要写入双写文件，并且双写文件以“O_SYNC |

Man in the middl
Playing back pacl
Wireshark using
Try the SSL attack
dos
Wrote a key to sv
registry tool
Using nmap to di
surviving devices
WiFi
What can scapy c
Reverie caused b
capture
Redémarrer iptab
les paramètres du
ils incorrects?
Extract notes Duc
Application scena
multicast
Use of dirsearch
Utilization of Clic
injection
SQL injection me
Give you an orde
point. Can you ta
whole database?
Environmental pr
encountered in m
testing and some
Personal develop
open related offic
Wechat has rejec
thousands of tim
wechat like my fi
jpa Query native
em.createNative
回的无实体对象集
<String,Object
map的key值为无
安装 DSM 系统简
Granafa 配置主机
Small program Q
recognition protc
2021-06-17 conti
the applet
Convert postman
Granafa 配置主機
Granafa configur
surveillance de l'
Installer un tutori
le système DSM
JPA Quei ive
em.create nativeQ
Liste de collectio
physiques retour
& lt; Chaîne, obje
valeur clé de la ca
est désordonnée
Youdao cloud no

O_DIRECT”模式打开，保证同步写入磁盘。因为双写文件是顺序追加的，所以即使采用同步写入磁盘，也不会带来太明显的性能损耗。在数据库恢复时，首先从双写文件中将可能存在的部分写入磁盘的页面进行修复，然后再回放日志进行日志恢复。

此外也可以采用FPW（full page write，全页写）技术解决部分数据写入磁盘问题：在每次检查点之后，对于某个页面首次修改的日志中记录完整的页面数据。但是为了保证I/O性能的稳定性，目前openGauss默认使用增量检查点机制（关于增量检查点机制，参见“4.2.8 持久化及故障恢复机制”节），而该机制与FPW技术无法兼容，所以在openGauss中目前采用双写技术来解决部分数据写入磁盘问题。

结合图1，缓冲区页面查找的流程如下。

（1）计算“buffer tag”对应的hash值和分区值。

（2）对“buffer id”哈希表加分区共享锁，并查找“buffer tag”键值是否存在。

（3）如果“buffer tag”键值存在，确认对应的磁盘页面是否已经加载上来。如果是，则直接返回对应的“buffer id + 1”；如果不是，则尝试加载到该“buffer id”对应的buffer内存中，然后返回“buffer id + 1”。

（4）如果“buffer tag”键值不存在，则寻找一个“buffer id”来进行替换。首先尝试从各个“bgwriter”线程的空闲“buffer id”队列中获取可以用来替换的“buffer id”；如果所有“bgwriter” 线程的空闲buffer id队列都为空队列，那么采用clock-sweep算法，对整个buffer缓冲区进行遍历，并且每次遍历过程中将各个buffer的使用计数减一，直到找到一个使用计数为0的非脏页面，就将其作为用来替换的buffer。

（5）找到替换的“buffer id”之后，按照分区号从小到大的顺序，对两个“buffer tag”对应的分区同时加上排他锁，插入新“buffer tag”对应的元素，删除原来“buffer tag”对应的元素。然后再按照分区号从小到大的顺序释放上述两个分区排他锁。

（6）最后确认对应的磁盘页面是否已经加载上来。如果是，则直接返回上述被替换的“buffer id + 1”；如果不是，则尝试加载到该“buffer id”对应的buffer内存中，然后返回“buffer id + 1”。

行存储共享缓冲区访问的主要接口和含义如表1所示。

表1 行存储共享缓冲区访问的主要接口

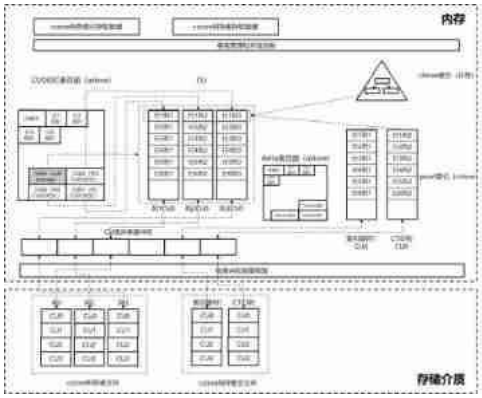
函数名	操作含义
ReadBufferExtended	读、写业务线程从共享缓冲区获取页面用于读、写查询
ReadBufferWithoutRelcache	恢复线程从共享缓冲区获取页面用于回放日志
ReadBufferForRemote	备机页面修复线程从共享缓冲区获取页面用于修复主机损坏页面

（六）cstore

列存储格式是OLAP类数据库系统最常用的数据格式，适合复杂查询、范围统计类查询的在线分析型处理系统。本节主要介绍openGauss数据库内核中cstore列存储格式的实现方式。

1. cstore整体框架

cstore列存储格式整体框架如图2所示。其主要模块代码分布参见“ 存储引擎整体架构及代码概览”小 节。与行存储格式不同，cstore列 存储的主体数据文件以CU为I/O单元，只支持追加写操作，因此cstore只有读共享缓冲区。CU间和CU内的可见性由对应的CUDESE表（astore表）决定，因此其可见性和 并发控制原理与行存储astore基本相同。



|图2 cstore列存储格式整体框架示意图|

2. cstore存储单元结构

About the basic c
of rip protocol V1
About configurin
static routes for r
Install Anaconda
别吵吵内卷了，看
连续直播70天都没
小心，元宇宙有色
ERP管理系统中的
术，包含这3部分
行业报告：2021-2
取油市场现状及未
行业报告：中国清
现状及未来发展趋
小心，元宇宙有色
支付寶内測商家版

|图3 CU结构示意图|

如图3所述，cstore的存储单元是CU，分别包括：

（1） CU的CRC值，为CU结构中除CRC成员之外，其他所有字节计算出的32位CRC值。

（2） CU的magic值，为插入CU的事务号。

（3） CU的属性值，为16位标志值，包括CU是否包含NULL行、CU使用的压缩算法等CU粒度属性信息。

（4） 压缩后NULL值位图长度，如果属性值中标识该CU包含NULL行，则本CU在实际数据内容开始处包含NULL值位图，此处储存该位图的字节长度，如果该CU不包含NULL行，则无该成员。

（5） 压缩前数据长度，即CU数据内容在压缩前的字节长度，用于读取CU时进行内存申请和校验。

（6） 压缩后数据长度，即CU数据内容在压缩后的字节长度，用于插入CU时进行内存申请和校验。

（7） 压缩后NULL值位图内容，如果属性值中标识该CU包含NULL行，则该成员即为每行的NULL值位图，否则无该成员。

（8） 压缩后数据内容，即实际写入磁盘的CU主体数据内容。

每个CU最多保存对应字段的MAX_BATCH_ROWS行（默认60000行）数据。相邻CU之间按8KB对齐。

CU模块提供的主要CU操作接口如表2所示。

表2 CU操作接口

函数名称	接口含义
AppendCuData	向组装的CU中增加一行（仅对应字段）
Compress	压缩（若需）和组装CU
FillCompressBufHeader	填充CU头部
CompressNullBitmapIfNeed	压缩NULL值位图
CompressData	压缩CU数据
CUDataEncrypt	加密CU数据
ToVector	将CU数据解构为向量数组结构
UnCompress	解压（若需）和解析CU
UnCompressHeader	解析CU头部内容
UnCompressNullBitmapIfNeed	解压NULL值位图
UnCompressData	解压CU数据
CUDataDecrypt	解密CU数据

3. cstore多版本机制

cstore支持完整事务语义的DML查询，原理如下：

（1）CU间的可见性：每个CU对应CUDESC表（astore行存储表）中的一行记录（一对一），该CU的可见性完全取决于该行记录的可见性。

（2）同一个CU内不同行的可见性：每个CU的内部可见性对应CUDESC表中的一行（多对一），该行的bitmap字段为最长MAX_BATCH_ROWS个bit的删除位图（bit 1表示删除，bit 0表示未删除），通过该位图记录的可见性和多版本，来支持CU内不同行的可见性。同时由于DML操作都是行粒度操作的，因此对于行号范围相同的、不同字段的多个CU均对应同一行位图记录。

（3）CU文件读写并发控制：CU文件自身为APPEND-ONLY，只在追加时对文件大小扩展进行加锁互斥，无须其他并发控制机制。

（4）同一个字段的不同CU，对应严格单调递增的cu_id编号，存储在对应的CUDESC表记录中，该cu_id的获取也通过上述文件扩展锁来进行并发控制。

（5）对于cstore表的单条插入以及更新操作，同时也提供与每个cstore表对应的delta表（astore行存储表），来接受这些新插入的或更新后的元组，以降低CU文件的碎片化。

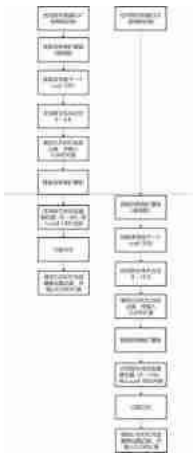
可见，cstore表的可见性依赖于对应CUDESC表中记录的可见性。一个CUDESC表的结构如表3所示，其与CU的对应关系如图4所示。

表3 CUDESC表的结构

字段名	类型	含义
col_id	integer	字段序号，即该cstore列存储表的第几个字段；特殊的，对于CU位图记录，该字段恒为-10
cu_id	oid	CU序号，即该列的第几个CU
min	text	该CU中该字段的最小值
max	text	该CU中该字段的最大值
row_count	integer	该CU中的行数
cu_mode	integer	CU模式
size	bigint	该CU大小
cu_pointer	text	该CU偏移（8K对齐）；特殊的，对于CU位图记录，该字段为删除位图的二进制内容
magic	integer	该CU magic号，与CU头部的magic相同，校验用
extra	text	预留字段

|图4 CUDESC表和CU对应关系示意图|

如图5、图6所示，下面结合并发插入和并发插入查询2种具体场景，介绍openGauss中cstore多版本的具体实现方法。



|图5 cstore表并发插入示意图|

|图6 cstore表并发插入和查询示意图|

1） 并发插入操作：

对于并发的插入操作，session 1和session 2首先分别在各自的局部内存中完成待插入CU的拼接。然后假设session 1先获取到cstore表的扩展锁，那么session 2会阻塞在该锁上。在持锁阶段，session 1申请到该字段下一个cuid 1001，预占了该cu文件0 - 6 K的内容（即cuid 1001的内容大小），将cuid的大小、偏移以及cuid 1001头部部分信息填充到CUDESC记录中，并完成CUDESC记录的插入。接着，session 1放锁，并将cuid 1001的内容写入到CU对应偏移处，记录日志，再将删除位图记录插入CUDESC表中。当session1释放cstore表的扩展锁之后，session 2就可以获取到该锁，然后，类似session 1的后续操作，完成cuid 1002的插入操作。

2） 并发插入和查询操作：

假设在上述session 2的插入事务（事务号101）执行过程中，有并发的查询操作执行。对于查询操作，首先基于col_id和cuid这两个索引键对CUDESC表做索引扫描。由于事务号101在查询的快照中，因此cuid 1002的所有记录对于查询事务不可见，查询事务只能看到cuid 1001（事务号100）的那些记录。然后，查询事务根据CUDESC记录中对应的CU文件偏移和CU大小，将cuid 1001的数据从磁盘文件或缓存中加载到局部内存中，并拼接成向量数组的形式返回。

4. cstore访存接口和索引机制

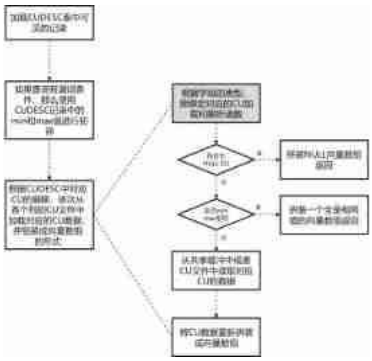
cstore访存接口如表4所示，主要包括扫描、插入、删除和查询操作。

表4 cstore访存接口

接口名称	接口含义
CStoreBeginScan	开启cstore扫描
CStore::RunScan	执行cstore扫描，根据执行计划，内层执行cstore顺序扫描或者cstore min-max过滤扫描
CStoreGetNextBatch	继续扫描，返回下一批向量数组
CStoreEndScan	结束cstore扫描
CStore::CStoreScan	cstore顺序扫描
CStore::CStoreMinMaxScan	cstore min-max过滤扫描
CStoreInsert::BatchInsert(VectorBatch)	将输入的向量数组批量插入cstore表中
CStoreInsert::BatchInsert(bulkload_rows)	将输入的多行数组插入cstore表中
CStoreInsert::BatchInsertCommon	将一批多行数组（最多MAX_BATCH_ROWS行）插入cstore表各个列的CU文件中、插入对应CUDESC表记录、插入索引
CStoreInsert::InsertDeltaTable	将一批多行数组插入cstore表对应的delta表中
InsertIdxTableIfNeed	将一批多行数组插入cstore表的索引表中

CStoreDelete::PutDeleteBatch	将一批待删除的向量数组暂存到局部数据结构中，如果达到局部内存上限，则触发一下删除操作
CStoreDelete::PutDeleteBatchForTable	CStoreDelete::PutDeleteBatch对于普通cstore表的内层实现
CStoreDelete::PutDeleteBatchForPartition	CStoreDelete::PutDeleteBatch对于分区cstore表的内层实现
CStoreDelete::PutDeleteBatchForUpdate	CStoreDelete::PutDeleteBatch对于更新cstore表操作的内层实现（更新操作由删除操作和插入操作组合而成）
CStoreDelete::ExecDelete	执行cstore表删除，内层调用普通cstore表删除或分区cstore表删除
CStoreDelete::ExecDeleteForTable	执行普通cstore表删除
CStoreDelete::ExecDeleteForPartition	执行分区cstore表删除
CStoreDelete::ExecDelete(rowid)	删除cstore表中特定一行的接口
CStoreUpdate::ExecUpdate	执行cstore表更新

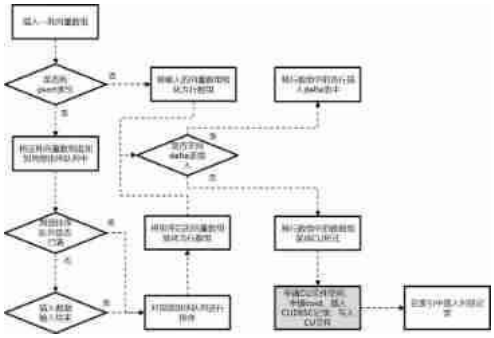
cstore表查询执行流程，可以参考图7中所示。 其中，灰色部分实际上是在初始化cstore扫描阶段执行的，根据每个字段的具体类型，绑定不同的CU扫描和解析函数，主要有FillVector、FillVectorByTids、FillVectorLateRead3类CU扫描解析接口。



|图7 cstore表查询流程示意图|

cstore表插入执行流程，可以参考图8所示。其中灰色部分内的具体流程可以参考图5、图6中所示。当满足以下3个条件时，可以支持delta表插入：

（1） 打开enable_delta_store GUC参数。
（2） 该批向量数组为本次导入的最后一批向量数组。
（3） 该批向量数组的行数小于delta表插入的阈值。



|图8 cstore表插入流程示意图|

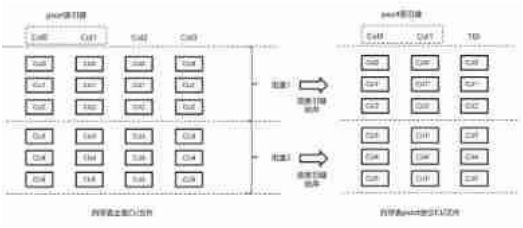
cstore表的删除流程主要分为两步。

（1） 如果存在delta表，那么先从delta表中删除满足谓词条件的记录。
（2） 在CUDESC表中更新待删除行所在CU的删除位图记录。

cstore表的更新操作由删除操作和插入操作组合而成，流程不再赘述。

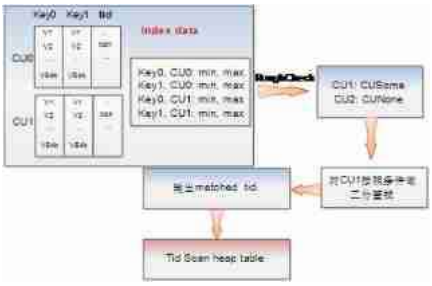
openGauss的cstore表支持psort和cbtree两种索引。

psort索引是一种局部排序聚簇索引。psort索引表的组织形式也是cstore表，该cstore表的字段包括索引键中的各个字段，再加上对应的行号（TID）字段。如图9所示，将一定数量的记录按索引键经过排序聚簇之后，与TID字段共同拼装成向量数组之后，插入psort索引cstore表中，插入流程和上面cstore表插入流程相同。



|图9 psort索引插入原理图|

查询时如果使用psort索引扫描，会首先扫描psort索引cstore表（扫描方式和上面cstore表扫描流程相同）。在一个psort索引CU的内部，由于做了局部聚簇索引，因此可以使用基于索引键的二分查找方式，快速找到符合索引条件的记录在该psort索引中的行号，该行的TID字段值即为该条记录在cstore主表中的行号。上述流程如图10所示。值得一提的是由于做了局部聚簇索引，因此在索引cstore表扫描过程中，在真正加载索引表CU文件之前，可以通过CUDESC中的min max做到非常高效的初筛过滤。



|图10 psort索引查询原理图|

cstore表的cbtree索引和行存储表的B-Tree索引在结构和使用方式上几乎完全一致，相关原理可以参考行存储索引章节（“行存储索引机制”节），此处不再赘述。

openGauss cstore表索引对外提供的主要接口如表5所示。

表5 cstore表索引对外接口

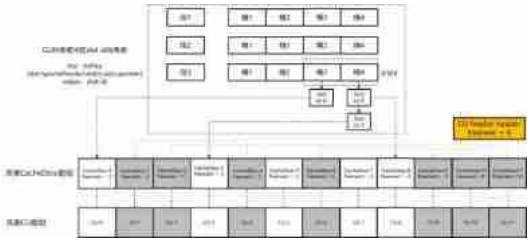
接口名称	接口含义
psortgettuple	通过psort索引，返回下一条满足索引条件的元组。伪接口，实际psort索引扫描通过CStore::RunScan实现
psortgetbitmap	通过psort索引，返回满足索引条件的元组的tid bitmap。伪接口，实际psort索引扫描通过CStore::RunScan实现
psortbuild	构建psort索引表数据。主要流程包括，从cstore主表中扫描数据、局部聚簇排序、插入到psort索引cstore表中
cbtreegettuple	通过cbtree索引，返回下一条满足索引条件的元组。内部和btgettuple都是通过调用_bt_gettuple_internal函数实现的
cbtreegetbitmap	通过cbtree索引，返回满足索引条件的元组的tid bitmap。内部和btgetbitmap都是通过调用_bt_next函数实现的
cbtreebuild	构建cbtree索引表数据。内部实现与btbuild类似，先后调用_bt_spoolinit、CStoreGetNextBatch、_bt_spool、_bt_leafbuild和_bt_spooldestroy等几个主要函数实现。与btbuild区别在于，B-Tree的构建过程中，扫描堆表是通过heapam接口实现的，而cbtree扫描的是cstore表，因此使用的是CStoreGetNextBatch

5. cstore缓存机制

考虑到cstore列存储格式主要面向只读查询居多的OLAP类业务，因此openGauss提供只读的共享CU缓冲区机制。

openGauss中CU只读共享缓冲区的结构如图11所示。和行存储页面粒度的共享缓冲区类似，最上层为共享哈希表，哈希表键值为CU的slot类型、relfilenode、colid、cuid、cupointer构成的五元组，哈希表的记录值为该CU对应的缓冲区槽位slot id（对应行存储共享缓区的buffer id）。在全局CacheDesc数组中，用CacheDesc结构体记录与slot id对应的缓存槽位的状态信息（对应行存储缓冲区的BufferDesc结构体）。在共享CU数组中，用CU结构体记录与slot id对应的缓存CU的结构体信息。

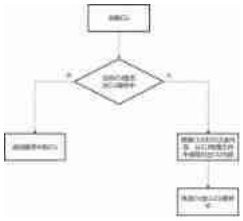
与行存储固定的页面大小不同，不同CU的大小可能是不同的（行存储页面大小都是8 K），因此上述CU槽位只记录指向实际内存中CU数据的指针。另一方面为了保证共享内存大小可控，通过另外的全局变量来记录已经申请的有效槽位中所有CU的大小总和。



|图11 CU只读共享缓存结构示意图|

CU只读共享缓冲区的工作机制如图12所示。

（1） 当从磁盘读取一个CU放如Cache Mgr时，需要从FreeSlotList里拿到一个free slot（空闲槽位）存放CU，然后插入到哈希表中。
（2） 当FreeSlotList为NULL的时，需要根据LRU算法淘汰掉一个slot（槽位），释放CU data占的内存，减小CU总大小计数，并从hash table中删除，然后存放新的CU，再插入哈希表中。
（3） Cache内存大小可以配置。如果内存超过设置的Cache大小，需要淘汰掉适量的slot，并释放CU data占用的内存。
（4） 支持缓存压缩态的CU或解压态的CU两种模式，可以通过配置文件修改，同时只能存在一种模式。



|图12 CU只读共享缓存读取示意图|

与CU只读共享缓冲区相关的关键数据结构代码如下：

```
typedef struct CUSlotTag { RelFileNodeOld m_rnode; int m_colId; int32 m_CUId; uint32 m_padding; CUPoint
```

（七）日志系统

内存是一种易失性存储介质，在断电等场景下存储在内存介质中的数据会丢失。为了保障数据的可靠性需要将共享缓冲区中的脏页写入磁盘，此即数据的持久化过程。对于最常用的持久化存储介质磁盘，由于每次读写操作都有一个“启动”代价，导致磁盘的读写操作频率有一个上限。即使是超高性能的SSD磁盘，其读写频率也只能达到10000次/秒左右。如果多个磁盘读写请求的数据在磁盘上是相邻的，就可以被合并为一次读写操作。因为合并后可以等效降低读写频率，所以磁盘顺序读写的性能通常要远优于随机读写。由于如上原因，数据库通常都采用顺序追加的预写日志（write ahead log，WAL）来记录用户事务对数据库页面的修改。对于物理表文件所对应的共享内存中的脏页会等待合适的时机再异步、批量地写入磁盘。

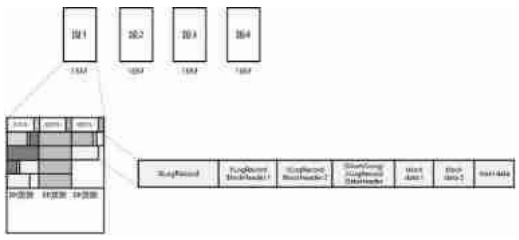
日志可以按照用户对数据库不同的操作类型分为以下几类，每种类型日志分别对应一种资源管理器，负责封装该日志的子类、具体结构以及回放逻辑等。如表6所示。

表6 日志类型

日志类型名字	资源管理器类型	对应操作
XLOG	RM_XLOG_ID	pg_control控制文件修改相关的日志，包括检查点推进、事务号分发、参数修改、备份结束等
Transaction	RM_XACT_ID	事务控制类日志，包括事务提交、回滚、准备、提交准备、回滚准备等
Storage	RM_SMGR_ID	底层物理文件操作类日志，包括文件的创建和截断

CLOG	RM_CLOG_ID	事务日志修改类日志，包括CLOG拓展、CLOG标记等
Database	RM_DBASE_ID	数据库DDL类日志，包括创建、删除、更改数据库等
Tablespace	RM_TBLSPC_ID	表空间DDL类日志，包括创建、删除、更新表空间等
MultiXact	RM_MULTIXACT_ID	MultiXact类日志，包括MultiXact槽位的创建、成员页面的清空、偏移页面的清空等
RelMap	RM_RELMAP_ID	表文件名字典文件修改日志
Standby	RM_STANDBY_ID	备机支持只读相关日志
Heap	RM_HEAP_ID	行存储文件修改类日志，包括插入、删除、更新、pd_base_xid修改、新页面、加锁等操作
Heap2	RM_HEAP2_ID	行存储文件修改类日志，包括空闲空间清理、元组冻结、元组可见性修改、批量插入等
Heap3	RM_HEAP3_ID	行存储文件修改类日志，目前该类日志不再使用，后续可以拓展
Btree	RM_BTREE_ID	B-Tree索引修改相关日志，包括插入、节点分裂、插入叶子节点、空闲空间清理等
hash	RM_HASH_ID	hash索引修改相关日志
Gin	RM_GIN_ID	GIN索引（generalized inverted index，通用倒排索引）修改相关日志
Gist	RM_GIST_ID	Gist索引修改相关日志
SPGist	RM_SPGIST_ID	SPGist索引相关日志
Sequence	RM_SEQ_ID	序列修改相关日志，包括序列推进、属性更新等
Slot	RM_SLOT_ID	流复制槽修改相关日志，包括流复制槽的创建、删除、推进等
MOT	RM_MOT_ID	内存引擎相关日志

openGauss日志文件、页面和日志记录的格式如图13所示。



|图13 日志文件、页面和记录格式示意图|

日志文件在逻辑意义上是一个最大长度为64位无符号整数的连续文件。在物理分布上，该逻辑文件按XLOG_SEG_SIZE大小（默认为16MB）切断，每段日志文件的命名规则为“时间线+日志id号+该id内段号”。“时间线”用于表示该日志文件属于数据库的哪个“生

命历程”，在时间点恢复功能中使用。“日志id号”从0开始，按每4G大小递增加1。“id内段号”表示该16MB大小的段文件在该4G“日志id号”内是第几段，范围为0至255。上面3个值在日志段文件名中都以16进制方式显示。

每个日志段文件都可以用XLOG_BLCKSZ（默认8KB）为单位，划分为多个页面。每个8KB页面中，起始位置为页面头，如果该页是整个段文件的第一个页面，那么页面头为一个长页头（XLogLongPageHeader），否则为一个正常页头（短页头）（XLogPageHeader）。在页头之后跟着一条或多条日志记录。每个日志记录对应一个数据库的某种操作。为了降低日志记录的大小（日志写入磁盘时延是影响事务时延的主要因素之一），每条日志内部都是紧密排列的。各条日志之间按8字节（64位系统）对齐。一条日志记录可以跨两个及以上的日志页面，其最大长度限制为1G。对于跨页的日志记录，其后续日志页面页头的标志位XLP_FIRST_IS_CONTRECORD会被置上。

长、短页头结构体的定义如下，其中存储了用于校验的magic信息、页面标志位信息、时间线信息、页面（在整个逻辑日志文件中的）偏移信息、有效长度信息、系统识别号信息、段尺寸信息、页尺寸信息等。

短页头结构体的代码如下：

```
typedef struct XLogPageHeaderData { uint16 xlp_magic; /* 日志magic校验信息 */ uint16 xlp_info; /* 标志位
```

长页头结构体的代码如下：

```
typedef struct XLogLongPageHeaderData { XLogPageHeaderData std; /* 短页头 */ uint64 xlp_sysid; /* 系统标
```

单条日志记录的结构如图13中所示，其由5个部分组成：

（1）日志记录头，对应XLogRecord结构体，存储了记录长度、主备任期号、事务号、上一条日志记录起始偏移、标志位、所属的资源管理器、crc校验值等信息。

（2）1 - 33个相关页面的元信息，对应XLogRecordBlockHeader结构体，存储了页面下标（0 - 32）、页面对应的物理文件的后缀、标志位、页面数据长度等信息；如果该日志没有对应的页面信息，则无该部分。

（3）日志数据主体的元信息，对应（长/短）XLogRecordDataHeader结构体，记录了特殊的页面下标，用于和第二部分区分，以及主体数据的长度。

（4）1 - 33个相关页面的数据；如果该日志没有对应的页面信息，则无该部分。

（5）日志数据主体。

这5部分对应的结构体代码如下。如上所述，在记录日志内容时，每个部分之间是紧密挨着的，无补空字符。如果一个日志记录没有对应的相关页面信息，那么第2和第4部分将被跳过。

```
typedef struct XLogRecord { uint32 xl_tot_len; /* 记录总长度 */ uint32 xl_term; TransactionId xl_xid; /
```

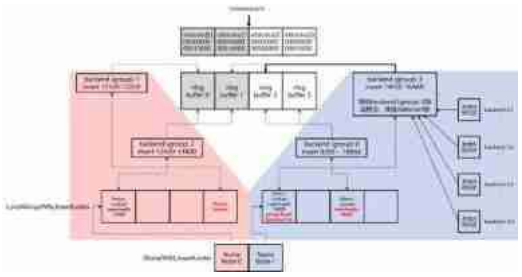
单条日志记录的操作接口主要分为插入（写）和读接口。其中，一个完整的日志插入操作一般包含以下几步接口，如表7所示。

表7 日志插入操作

步骤序号	接口名称	对应操作
1	XLogBeginInsert	初始化日志插入相关的全局变量
2	XLogRegisterData	注册该日志记录的主体数据
3	XLogRegisterBuffer/ XLogRegisterBlock	注册该日志记录相关页面的元信息
4	XLogRegisterBufData	注册该日志记录相关页面的数据
5	XLogInsert	执行真正的日志插入，包含5.1和5.2
5.1	XLogRecordAssemble	将上述注册的所有日志信息，按照图4-23中所示的紧密排列的5部分，重新组合成完整的二进制串
5.2	XLogInsertRecord	在整个逻辑日志中，预占偏移和长度，计算CRC，将完整的日志记录拷贝到日志共享缓冲区中

日志的读接口为XLogReadRecord接口。该接口从指定的日志偏移处（或上次读到的那条记录结尾位置处）开始读取和解析下一条完整的日志记录。如果当前缓存的日志段文件页面中无法读完，那么会调用ReadPageInternal接口加载下一个日志段文件页面到内存中继续读取，直到读完所有等于日志头部xl_tot_len长度的日志数据。然后，调用DecodeXLogRecord接口，将日志记录按图13中所示的5个组成部分进行解析。

日志文件读写的最小I/O粒度为一个页面。在事务执行过程中，只会进行（顺序追加）写日志操作。为了提高写日志的性能，在共享内存中，单独开辟一片特定大小的区域，作为写日志页面的共享缓冲区。对该共享缓冲区的并发操作（拷贝日志记录到单个页面中、淘汰LSN过老的页面、读取单个页面并写入磁盘）是事务执行流程中的关键瓶颈之一，对整个数据库系统的并发能力至关重要。



|图14 并发日志写入流程示意图|

如图14所示，在openGauss中对该共享缓冲区的操作采用Numa-aware的同步机制，具体步骤如下。

（1） 业务线程在本地内存中将日志记录组装成图13中所示的、5部分组成的字节流。

（2） 找到本线程所绑定的Numa Node对应的日志插入锁组，并在该锁组中随机找一个槽位对应的锁。

（3） 检查该锁的组头线程号。如果没有说明本线程是第一个请求该锁的，那么这个锁上所有的写日志请求将由本线程来执行，将锁的组头线程号设置为本线程号；否则说明已经存在这批写日志请求的组头线程，记录下当前组头线程的线程号，并将自己加入到这批的插入组队列中，等待组头线程完成日志插入。

（4） 对于组头线程，获取该日志插入锁的排他锁。

（5） 为该组所有的插入线程在逻辑日志文件中占位，即对当前该文件的插入偏移进行原子CAS（compare and swap，比较后交换）操作。

（6） 将该组所有后台线程本地内存中的日志依次拷贝到日志共享缓冲区的对应页面中。每当需要拷贝到下一个共享内存页面时，需要判断下一个页面对应的逻辑页面号是否和插入者的预期页面号一致（因为共享内存有限，因此同一个共享内存页面对应取模相同的逻辑页面）。首先，将自己预期的逻辑页面号，写入当前持有的日志插入锁的槽位中，然后进行上述判断。如果不一致，并且日志写入磁盘偏移比该共享内存页当前逻辑页面号还要小，那么需要将该页面写入磁盘以便复用。为了防止可能还有并发业务线程在拷贝日志数据到老逻辑页面号上，因此需要阻塞遍历每个日志插入锁，直到日志插入锁被释放，或者被持有的插入锁的逻辑页面号大于目标共享内存页面中现有的逻辑页面号。经过上述检查之后，就可以保证没有并发的业务还在对该共享内存页面进行拷贝写入操作，可以将其内容写入磁盘，并更新其对应的逻辑页面号到目标逻辑页面号。

（7） 重复上一步操作，直到把该组所有后台线程待插入的日志记录拷贝完。

（8） 释放日志插入锁。

（9） 唤醒本组所有后台线程。

由于内容较多，关于磁盘引擎方面的其他内容将在下篇图文中进行分享，敬请期待！

往 期 推 荐

openGauss数据库源码解析系列文章——存储引擎源码解析（一）

openGauss数据库源码解析系列文章——存储引擎源码解析（二）

本文分享自微信公众号 - openGauss（openGauss）。
如有侵权，请联系 support@oschina.cn 删除。
本文参与 “OSC源创计划”，欢迎正在阅读的你也加入，一起分享。

版权声明
本文为[openGauss]所创，转载请带上原文链接，感谢
https://my.oschina.net/u/5059795/blog/5376502

免责声明

本站以网络数据为基准，引入优质的垂直领域内容。本站内容仅代表作者观点，与本站立场无关，本站不对其真实合法性负责
如有内容侵犯了您的权益，请告知，本站将及时删除。联系邮箱：chxpostbox@gmail.com
Copyright © 2020 文章整合 All Rights Reserved.