Postgresql存储管理中的哈希

马文韬

January 21, 2014

1 概述

1.1 引言

众所周知，哈希在计算机程序设计及各种软件系统实现中作用巨大，值得深入研究和探讨。

这篇总结主要讲述作者在对Postgresql数据库中存储管理部分的哈希桶和哈希表的使用进行分析研究时候的心得体会，尽量阐述清楚这一部分的相关内容，如有纰漏欢迎指出。

1.2 存储管理中的哈希

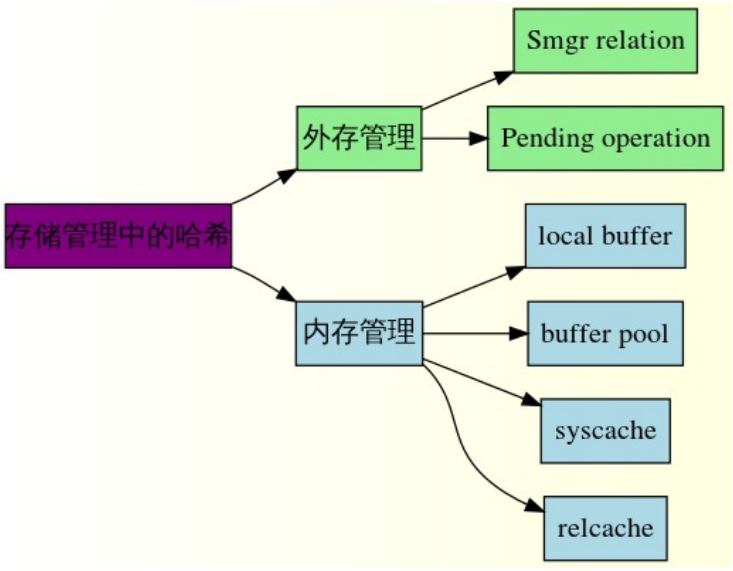

Figure 1: 存储管理中的哈希图

如Figure-1,存储管理分为内存（包括缓存）和外存，内存外存各部分又包括几个子模块。在相关的模块中，比如SysCache,SMGR等，哈希表或者哈希桶

都得到了广泛的应用，而且作用非常重要。本文就对上图所示的几个相关部分进行研究和阐述。

2 Hash table基本结构及其操作

2.1 哈希表的基本概念

哈希表（Hash table，也叫紧凑表），是根据关键字（Key value）而直接查询在存存储位置的资料结构。也就是说，它通过把键值通过一个函数的计算，映射到表中一个位置查询记录，这加快了查找速度。这个映射函数称作哈希函数，存放记录的数组称作哈希表。

一个通俗的例子是，为了查找电话簿中某人的号码，可以创建一个按照人名首字母顺序排列的表（即建立人名x到首字母F(x)的一函数关系），在首字母为W的表中查找「王」姓的电话号码，显然比直接查找就要快得多。这里使用人名作为关键字，「取首字母」是这个例子中哈希函数的函数法则F()，存放首字母的表对应哈希表。关键字和函数法则理论上可以任意确定。

2.2 pg中的哈希表基本结构和相关函数

2.2.1 hash table

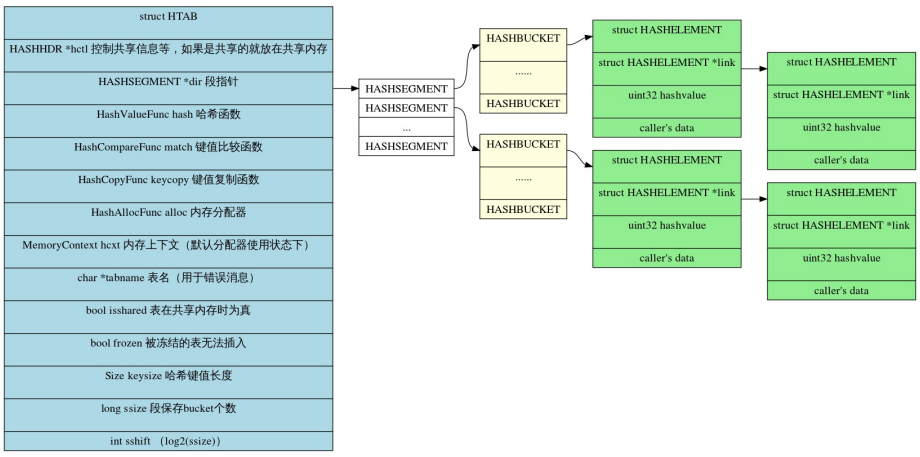

Figure 2: hash table 结构图

如图Figure-2,hash table的结构图，由图中可以窥探到一个哈希表的基本结构:段指针指向段空间，其中包含多个段，每个段空间内部有多个哈希桶，每个桶指向一个哈希entry链表。对于每一个哈希entry，私有部分为指向下一个entry的指针，还有就是32位的hashvalue域，具体存储的数据由HASHHDR结构体的相关字段来指定大小，再根据偏移量获取和存储数据。

2.2.2 HASHHDR结构体

Figure 3: HASHHDR结构体

HASHHDR结构体用于“contains all changeable info”，在哈希表的初始化阶段和改变的时候都要用到这个结构体，通过对相关参数进行设置修改来决定哈希表的大小等参数。　　如果哈希表是共享内存的哈希表，那么HASHHDR结构体需要保存在共享内存中。对于非共享内存的哈希表而言，HASHHDR和HTAB在功能上是相同的。

2.2.3 哈希结构体中的函数指针

Table 1: 哈希结构体中函数指针的默认函数及其功能

函数指针	相关函数
HashValueFunc hash	string_hash 计算string哈希值 tag_hash 计算tag哈希值 oid_hash 计算oid的哈希值 bitmap_hash计算bitmap哈希
HashCompareFunc match	string_compare，如果不自定义，默认 bitmap_match 和bitmap_hash一起用
HashCopyFunc keycopy	strlcpy 可以自定义，默认
HashAllocFunc alloc	DynaHashAlloc 调用上下文 中定义的内存分配方法

由Table 1可以看到相关指针函数，这些函数在对哈希表进行创建、查找、添加、删除等操作的时候会被提前设置，如果没有设置的话会调用系统或者上下文默认的函数来执行。

2.2.4 重要函数

在pg中哈希表的实现里，hash_起着至关重要的作用，完成了包括插入、删除、查找等一系列动作。在整个pg存储部分中这个函数也是使用最频繁的。 hash_search的函数定义如下:

void *hash_search(HTAB *hashp, void *keyPtr, HASHACTION action, bool *foundPtr);

相关参数：

- HTAB *hashp 哈希表的指针，指向要操作的哈希表
- void *keyPtr 键指针，指向键值
- HASHACTION action 这个函数的action
- bool foundPtr 用引用来指明函数是否找到相关entry，找到了返回true，否则为false，也可以设置为NULL。

action类型：

Table 2: hash_search相关操作action标记

HASH_FIND	根据key 查找哈希表
HASH_ENTER	查找哈希表， 如果entry没出现， 那么创建一个
HASH_ENTER_NULL	查找哈希表， 如果超出内存， 返回NULL
HASH_REMOVE	移除有特定key的entry

根据上表，action基本类型和作用都在里面，通过指定相关类型和参数来完成各种哈希操作。

2.3 本章总结

本章讲解了pg中哈希表的基本构成和相关的操作函数、重要的结构体等内容，整个存储管理的哈希表都是这个结构。这里明确一下这个结构和相关功能，方便下面具体介绍各个部分的哈希表的使用。

本章主要涉及的c语言源文件backend/utils/hash目录下的文件及其头文件。

3 Smgr relation中的Hash

3.1 SMGR功能简介

外存管理负责处理数据库与外存介质（在PostgreSQL中只实现了磁盘管理的操作）的交互过程。在pg中，外存管理由SMGR（代码在smgr.c)提供对外操作的统一接口。SMGR负责统管各种介质的管理器，根据上层的请求选择一个具体的介质管理器进行操作。

3.2 SMGR中哈希表相关的数据结构

SMGR中哈希表相关的查询、插入、删除等操作都是以RelFileNode结构体作为键来执行的。

RelFileNode结构体字段：

- Oid spcNode:表空间
- Oid dbNode :数据库
- Oid relNode:关系

RelFileNode结构体保存了表空间、数据库和相关关系的Oid字段。

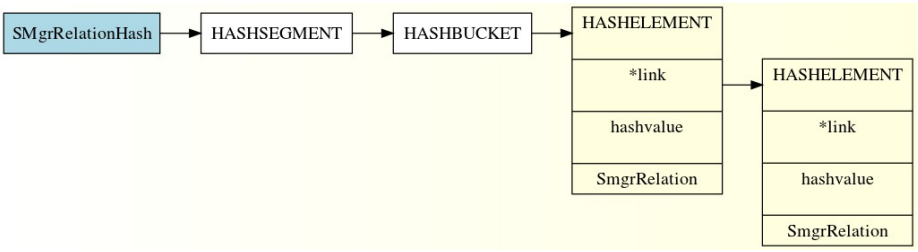

Figure 4: smgr中的哈希

由上图不难看出，SMGR hash table基本结构和之前讲到的hash table是一样的，在这里表的名称是SMgrRelationHash，每一个entry中的数据除了私有部分，还保存着SMgrRelation,也就是SMGR关系。SMGR部分的哈希操作主要就是对SMgrRelation关系的查询、插入等等。

3.3 SMGR中哈希函数的相关调用

在SMGR中有几处都使用了hash_search函数，如下表所示：

Table 3: smgr relation中的哈希函数

函数名	参数	调用函数	使用场景
hash_search	SMgrRelationHash, (void *)&rnode, HASH_ENTER, &found	SMgrRelation smgropen (RelFileNode rnode)	RelFileNode作为键 查找并返回SMgrRelation对象， 找不到就创建一个
hash_search	SMgrRelationHash, &(reln->smgr_rnode), HASH_REMOVE, NULL	void smgrclose (SMgrRelation reln)	smgr关系的RelFileNode 作主键查找并删除 SMgrRelation对象
hash_search	SMgrRelationHash, (void *)&rnode HASH_FIND, NULL	void smgrclosenode (RelFileNode rnode)	RelFileNode作为键查找返回 相应SMgrRelation对象, 由smgrclose完成删除 避免创建无用关系

在表中的函数中，hash_search指定的action为HASH_ENTER、HASH_REMOVE以及HASH_FIND,这些函数将以RelFileNode作为键对SMgrRelationHash这个哈希表进行插入、删除和查找SMgrRelation的操作。

3.4 本章小结

本章主要对SMGR管理部分的哈希函数及哈希表的使用进行简单的介绍。相关代码主要在backend/storage/smgr/smgr.c中。

4 Pending operation中的Hash

4.1 Pending operation哈希表功能简介

Pending operation哈希表的主要用途是记录同步磁盘的操作，添加或者删除相关的同步操作。

4.2 Pending operation哈希表及相关结构

PendingOperationTag结构体是对哈希表查询的键，其字段如下:

- RelFileNode rnode;
- ForkNumber forkNum;
- BlockNumber segno;

Pending operation哈希表的基本结构如下图：

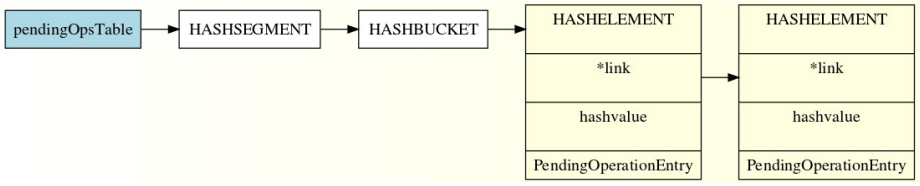

Figure 5: pending operation中的哈希

pendingOpsTable是哈希表的名称，entry存储的数据是PendingOperationEntry结构体，其字段如下:

- PendingOperationTag:作为key。
- bool canceled：标志位，表示相关操作是否已经被取消。
- CycleCtr cycle_ctr;

4.3 Pending operation的哈希函数调用

Pending operation中哈希函数的相关调用如下表：

Table 4: pending operation中的哈希函数

函数名	参数	调用函数	使用场景
hash_search	pendingOpsTable, &entry->tag, HASH_REMOVE, NULL	void mdsync(void)	写操作同步到磁盘时删除已经无效的操作,PendingOperationTag作为键查找相应操作并移除
hash_search	pendingOpsTable, &key,HASH_ENTER, &found	void RememberFsyncRequest (RelFileNode rnode, ForkNumber forknum, BlockNumber segno)	PendingOperationTag作为主键将fsync的请求添加到哈希表中

5 SysCache中的Hash

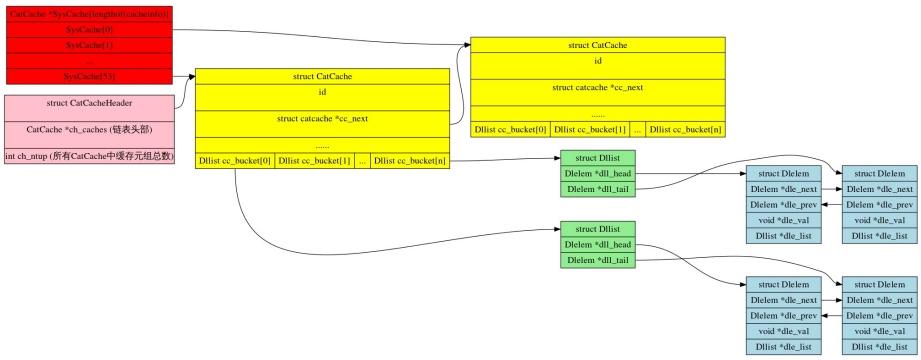

Figure 6: hash桶结构图

Table 5: 精确匹配基本流程及相关函数

初始化关键字信息	根据四个关键字初始化cur_skey
计算哈希值和索引	CatalogCacheComputeHashValue 由关键字计算hashValue HASH_INDEX宏利用hashValue 和cc_buckets计算索引
在索引对应的桶中查找	HeapKeyTest检查key是否匹配 将匹配的元组移动到链表头

Table 6: 部分匹配基本流程及相关函数

初始化关键字信息	根据部分关键字初始化cur_skey
计算哈希值和索引	CatalogCacheComputeHashValue 根据关键字个数计算lhashValue
在CatCache的cc_lists 指向的CatCList链表中查找	HeapKey Test检查key是否匹配 将匹配的CatClist放到cc_lists 链表的头部 不存在CatCList，扫描物理表并构建

6 RelCache中的Hash

Relation 的Oid作为key进行查询。

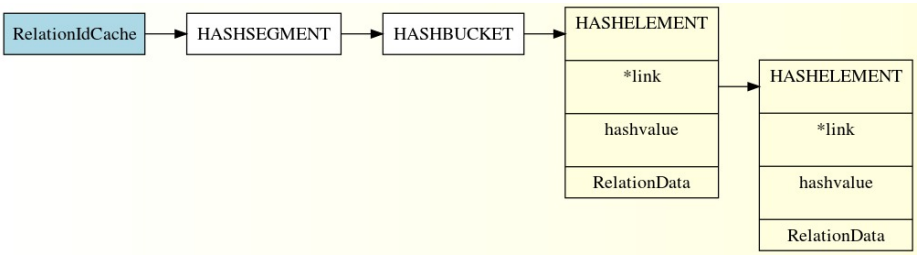

Figure 7: relcache中的哈希

Table 7: RelCache中的hash

函数名	参数	调用宏	使用场景
hash_search	RelationIdCache, &(RELATION->rd_id), HASH_ENTER ，&found	RelationCacheInsert (RELATION)	以关系的Oid作为主键将新的关系插入到relcache哈希表中
hash_search	RelationIdCache, &(ID),HASH_FIND, NULL	RelationIdCacheLookup (ID,RELATION)	用关系Oid作为主键在relcache中查找相应对象
hash_search	RelationIdCache, &(RELATION->rd_id), HASH_REMOVE， NULL	RelationCacheDelete (RELATION)	用关系Oid作为主键查找并删除relcache相应对象

7 Buffer pool中的Hash

- rnode(表空间OID，数据库OID和表OID组成);
- forkNum 枚举类型，标记缓冲区中文件块类型;
- blockNum 块号;

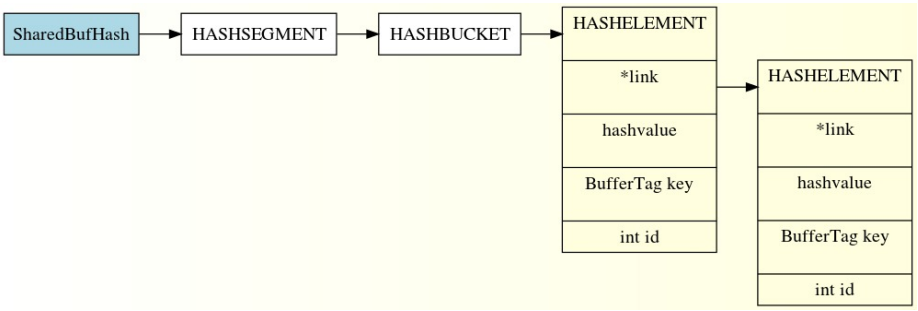

Figure 8:

Table 8: buffer pool中的hash

函数名	参数	调用函数	使用场景
hash_search_with _hash_value	SharedBufHash, tagPtr, hashcode, HASH_FIND, NULL	int BufTableLookup (BufferTag *tagPtr, uint32 hashcode)	根据BufferTag 在ShareBufHash中查询, 返回buffer ID
hash_search_with _hash_value	SharedBufHash, tagPtr, hashcode, HASH_REMOVE, NULL	void BufTableDelete (BufferTag *tagPtr,uint32 hashcode)	根据BufferTag删除 ShareBufHash中的entry
hash_search_with _hash_value	SharedBufHash, tagPtr, hashcode, HASH_ENTER, &found	int BufTableInsert(BufferTag *tagPtr, uint32 hashcode, int buf_id	根据BufferTag和 buffer ID插入entry, 如果有冲突entry, 返回冲突entry的buffer ID

8 Local buffer中的Hash

和buffer pool一样，使用BufferTag作为键进行查找。

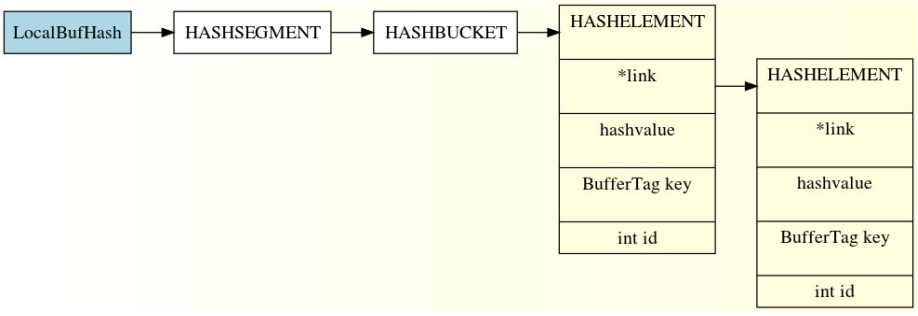

Figure 9: local buffer中的哈希

Table 9: local buffer中的哈希函数

函数名	参数	调用函数	使用场景
hash_search	LocalBufHash,&newTag, HASH_FIND,NULL	void LocalPrefetchBuffer (SMgrRelation smgr, ForkNumber forkNum, BlockNumber blockNum)	异步读取一个关系块 用smgr等参数创建一个tag, 查找LocalBufHash中相应块
hash_search	LocalBufHash,&newTag, HASH_FIND,NULL	void LocalBufferAlloc (SMgrRelation smgr, ForkNumber forkNum, BlockNumber blockNum, bool *foundPtr)	用smgr等参数创建一个tag, 为给定关系的给定页面创建 local buffer
hash_search	LocalBufHash, &bufHdr->tag, HASH_REMOVE,NULL	void LocalBufferAlloc (SMgrRelation smgr, ForkNumber forkNum, BlockNumber blockNum, bool *foundPtr)	更新LocalBufHash, 移除旧的entry
hash_search	LocalBufHash, &bufHdr->tag, HASH_ENTER,NULL	void LocalBufferAlloc (SMgrRelation smgr, ForkNumber forkNum, BlockNumber blockNum, bool *foundPtr)	更新LocalBufHash, 创建新的entry