CHINA POSTGRESQL ASSOCIATION

应急管理大学

PostgreSQL 哈希索引原理浅析

应急管理大学
文一

写在前面

索引，作为数据库领域的一个重要方面，为提升数据访问效率提供了有力的支撑。
而本次报告中，我们将聚焦 PostgreSQL 哈希索引的实现原理展开研究，
力争帮助各位建立一个综合而全面的理解。

本次报告分为如下部分：

- **知识准备**
  涵盖 PostgreSQL 的设计原理、系统表等同哈希索引有所关联的部分

- **哈希算法原理以及工程实现解析**
  阐述哈希索引所依据的算法思想，并从哈希索引表的组建与元组记录插入
  分析哈希索引的原理

- **拓展延伸**
  一些额外的知识以及有用的经验分享

知识准备：先从一个宏观视角理解 PostgreSQL

CHINA POSTGRESQL ASSOCIATION

Abstract

This paper presents the preliminary design of a new database management system, called POSTGRES, that is the successor to the INGRES relational database system The main design goals of the new system are to

1) provide better support for complex objects,
2) provide user extendibility for data types, operators and access methods,
3) provide facilities for active databases (i e , alerters and triggers) and inferencing including forward- and backward-chaining,
4) simplify the DBMS code for crash recovery,
5) produce a design that can take advantage of optical disks, workstations composed of multiple tightly-coupled processors, and custom designed VLSI chips, and
6) make as few changes as possible (preferably none) to the relational model

THE DESIGN OF POSTGRES

Michael Stonebraker and Lawrence A Rowe

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

PostgreSQL 本质上是一种改进后的关系型模型的工程实现

对于关系型数据模型本身修改甚少，这就使得 PostgreSQL 各个模块（存储引擎等），都是按照关系型数据的方式来思考和组织的。

因此，所有的对于 PostgreSQL 的二次开发，本质上就是使得关系型数据模型能够存储更为复杂的数据，而不是让 PostgreSQL 成为一种使用 “全新数据模型” 的数据库。

要点：

- PostgreSQL 高度重视可定制化，但是其根本依旧是一种关系型的数据库
- 但是我们可以依靠关系模型的灵活性，在一定程度上模拟其它的存储
- 业界案例： Apache OpenDAL 的 PostgreSQL Service 部分

APACHE OpenDAL™

知识准备：先从一个宏观视角理解 PostgreSQL

CHINA POSTGRESQL ASSOCIATION

```rust
let op = Operator::new(builder)?.finish();
/* 将 `Hello PostgreSQL` 写入到一个名为 `postgres.txt` 的文件夹中 */
op.write("postgres.txt", "Hello PostgreSQL").await?;
/* 构建 Reader */
let content = op
    .reader_with("postgres.txt")
    .await?
    ;
/* 经由 Reader 获取到一个 Buffer */
let bs = content.read(0..16).await?;
```

要点：

- OpenDAL 的 PostgreSQL 服务部分，本质上就是用数据表的存储“模拟文件系统”
- 这种做法对于通用性而言非常聪明，但是涉及不了根本性（**这也是为什么不能单纯依靠 PG 索引可定制化实现真正的多数据模型**）
- 本实验复现参考**《使用 OpenDAL 连接 PostgreSQL》**一文

```
postgres=# create table example (

    /* 用于存储 OpenDAL 文件名 */
    filename text,
    /* 用于存储 OpenDAL 文件数据 */
    filedata bytea,
    /* OpenDAL 要求 filename 具有 UNIQUE 约束 */
    unique(filename)
);
CREATE TABLE
```

```
postgres=# select * from example;
   filename   |                filedata
--------------+----------------------------------------
 postgres.txt | \x48656c6c6f20506f737467726553514c
(1 row)
```

```
wenyi@fedora:~/project/opendal
b"Hello PostgreSQL"
```

知识准备：先从一个宏观视角理解 PostgreSQL

CHINA POSTGRESQL ASSOCIATION

57. Writing a Procedural Language Handler
58. Writing a Foreign Data Wrapper
59. Writing a Table Sampling Method
60. Writing a Custom Scan Provider
61. Genetic Query Optimizer
62. Table Access Method Interface Definition
63. Index Access Method Interface Definition
64. Write Ahead Logging for Extensions
65. Built-in Index Access Methods
66. Database Physical Storage
67. Transaction Processing
68. System Catalog Declarations and Initial Contents

Relation
--> Tuple
--> Index

Chapter 12. Writing a Custom Storage Engine

12.1. Introduction

With MySQL 5.1, MySQL AB has introduced a pluggable storage engine architecture that makes it possible to create new storage engines and add them to a running MySQL server without recompiling the server itself.

This architecture makes it easier to develop new storage engines for MySQL and deploy them.

This chapter is intended as a guide to assist you in developing a storage engine for the new pluggable storage engine architecture.

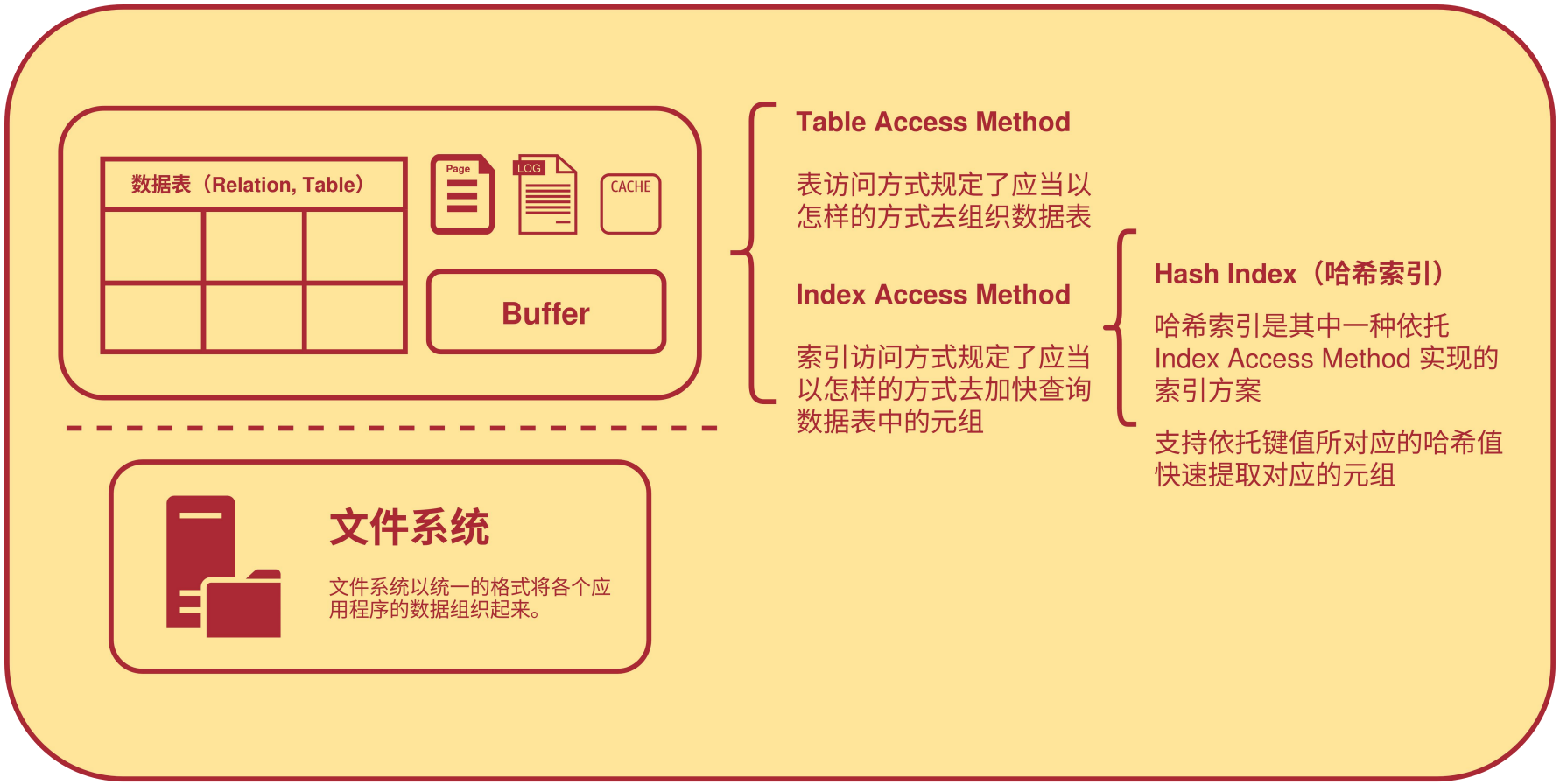
Custom storage engines can be built in a progressive manner: Developers can start with a read-only storage engine and later add support for `INSERT`, `UPDATE`, and `DELETE` operations, and even later add support for indexing, transactions, and other advanced operations.

MySQL 把 “Table” 与 “Index” 两种 Access Methon 直接统一为 “存储引擎” 并开放设计

这种更大的灵活度，客观上让 MySQL 的存储引擎生态优于 PostgreSQL

- **这种设计思想，体现到内核 API 上面**
  就是所有的内核开放设计 API
  均带着浓浓的 “数据表” 味儿
- **索引的核心作用：**
  一、加快用户查找 Tuple 的速度
  二、可独立提供数据（Index-Only Scan）
- 同时指出，**对比 MySQL，PostgreSQL 在这方面的可定制化 “并没有那么高”**

知识准备：从一个宏观视角理解 PG 索引



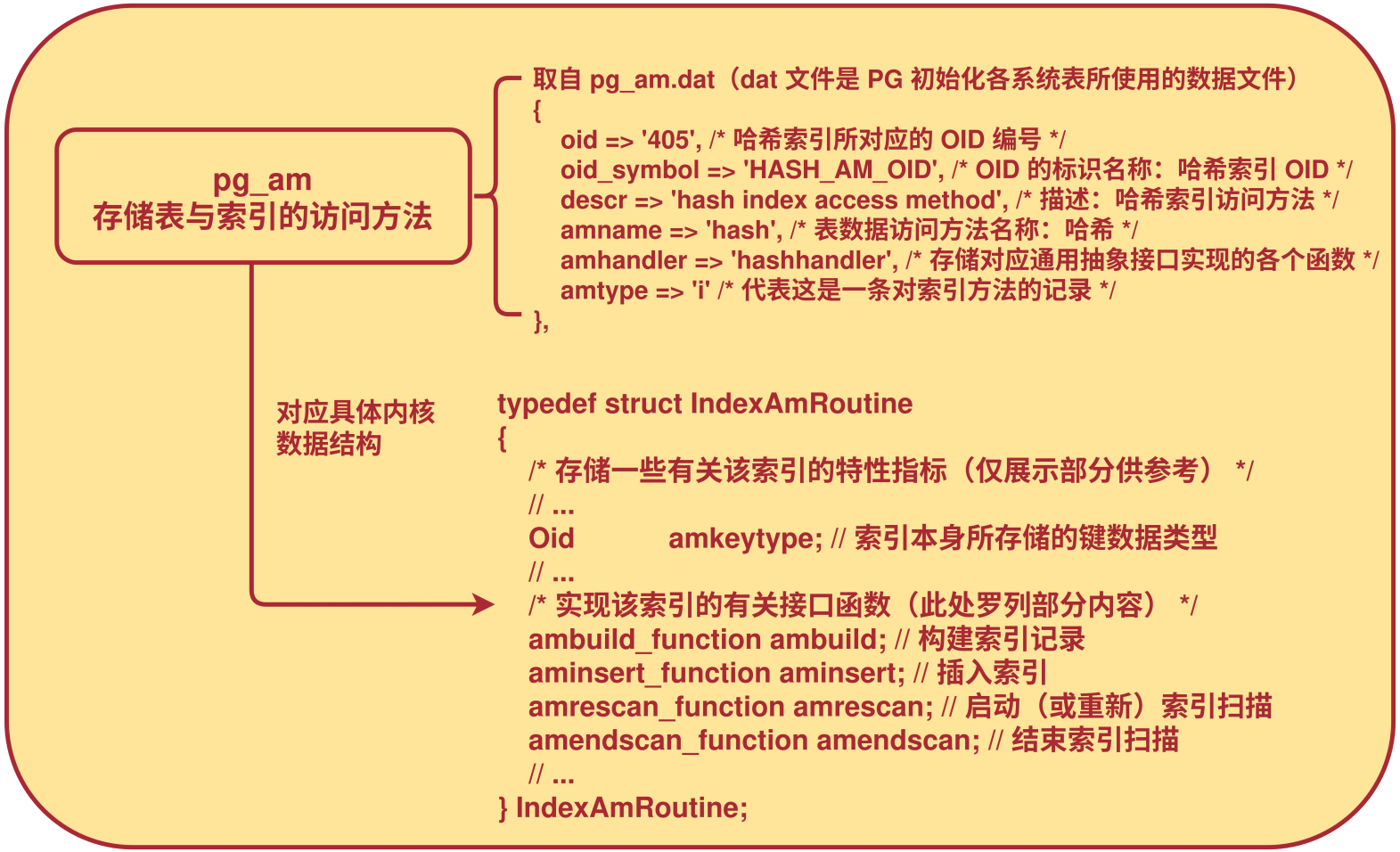
现在，我们继续把目光放在内核设施本身上面。

可以发现，工程的实现实际上就是理论的延伸。

我们只需要抓住 **Index-Access-Method** 这条“主轴”，既可以顺利地理解好 Hash Index 的实现。

这就需要我们找到一个切入点，参考下文。

知识准备：pg_am 系统表 与 IndexAmRoutine 接口



要点：

- pg_am 存储着所有的 Index-Access-Method 的注册信息（可通过 **pg_am.dat** 直接得到）
- 通过 **amhandler** 所指向的内核C语言函数，即可以找到有关索引的各个实现接口
- **IndexAmRoutine** 结构描述着一个索引所需要实现与可以实现的接口函数，并指导 PostgreSQL 将元祖的键值进行传递

知识准备：系统表 与 BKI 文件

```
postgres=# select * from pg_am;
 oid  | amname |      amhandler         | amtype
------+--------+------------------------+--------
    2 | heap   | heap_tableam_handler | t
  403 | btree  | bthandler              | i
  405 | hash   | hashhandler            | i
  783 | gist   | gisthandler            | i
 2742 | gin    | ginhandler             | i
 4000 | spgist | spghandler             | i
 3580 | brin   | brinhandler            | i
(7 rows)
```

要点:

- 通过 select ... from; 的上层接口方式，可以用一个更为直观的方式，将有关信息查询出来，事半功倍
- 理解 BKI 文件、pg_xxx.h 文件起到的支撑作用，对于理解如何自己定制索引非常重要

知识准备：系统表 与 BKI 文件

CHINA POSTGRESQL ASSOCIATION

postgresql - pg_am.h

```
/* ----------------
 *      pg_am definition.  cpp turns this into
 *      typedef struct FormData_pg_am
 * ----------------
 */
CATALOG(pg_am,2601,AccessMethodRelationId)
{
    Oid         oid;            /* oid */

    /* access method name */
    NameData    amname;

    /* handler function */
    regproc     amhandler BKI_LOOKUP(pg_proc);

    /* see AMTYPE_xxx constants below */
    char        amtype;
} FormData_pg_am;
```

src/backend/catalog/pg_xxx.h

指导 PostgreSQL 构建对应数据表

src/backend/catalog/pg_xxx.dat

指导 PostgreSQL 填充数据到表中

postgresql - pg_am.h

```
[

{ oid => '2', oid_symbol => 'HEAP_TABLE_AM_OID',
  descr => 'heap table access method',
  amname => 'heap', amhandler => 'heap_tableam_handler', amtype => 't' },
{ oid => '403', oid_symbol => 'BTREE_AM_OID',
```

这些流程于 InitDB 期间完成

要点：

- PostgreSQL 把 “关系型数据” 的思想同样运用到系统信息的管理上
- catalog 目录的 pg_xxx.h 是诸多系统表的 “原始形态”（Perl 脚本将会负责改写他们为正式的 pg_xxx_d.h 形态）
- pg_xxx.dat 将会指导向系统表里面写入初始数据

广东管理大学

知识准备：pg_am 系统表 与 IndexAmRoutine 接口

CHINA POSTGRESQL ASSOCIATION

postgresql - hash.c

```c
/*
 * Hash handler function: return IndexAmRoutine with access method parameters
 * and callbacks.
 */
Datum
hashhandler(PG_FUNCTION_ARGS)
{
    IndexAmRoutine *amroutine = makeNode(IndexAmRoutine);
    amroutine->amstrategies = HTMaxStrategyNumber;
    /* ... */
    PG_RETURN_POINTER(amroutine);
}
```

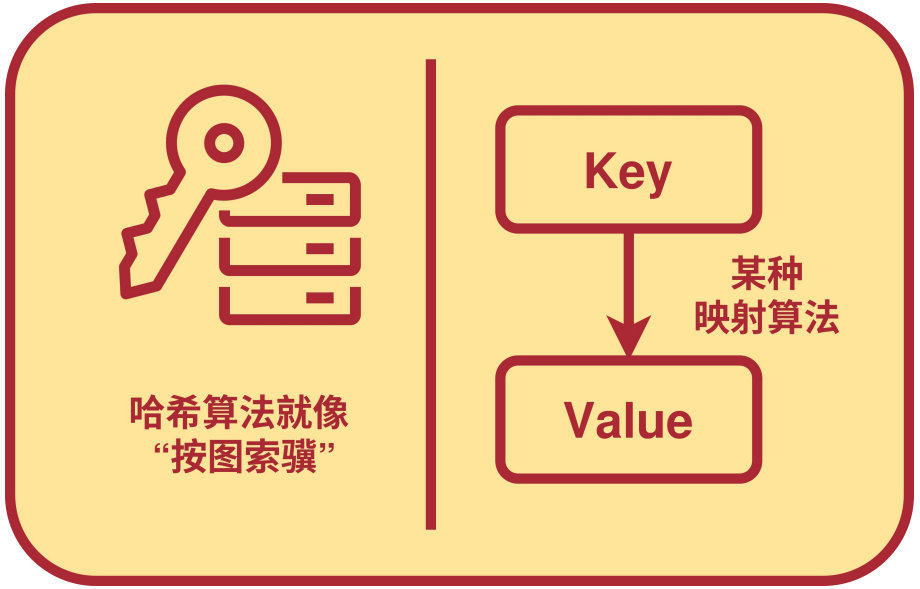
postgresql - pg_am.dat

```
{ oid => '405', oid_symbol => 'HASH_AM_OID',
  descr => 'hash index access method',
  amname => 'hash', amhandler => 'hashhandler', amtype => 'i' },
```

hashhandler 函数：

- 负责在 Relation 中注册处理各个哈希索引处理函数
- 核心逻辑就是分配一块内存，存储 IndexAmRoutine 结构，并将其反馈于 PostgreSQL

开始理解哈希索引：哈希算法简介



Definition at line 58 of file hash.c.

```
  59 {
  60     IndexAmRoutine *amroutine = makeNode(IndexAmRoutine);
  61
  62     amroutine->amstrategies = HTMaxStrategyNumber;
  63     amroutine->amsupport = HASHNProcs;
  64     amroutine->amoptsprocnum = HASHOPTIONS_PROC;
  65     amroutine->amcanorder = false;
```

```
uint16      amoptsprocnum;
/* does AM support ORDER BY indexed column's value? */
bool        amcanorder;
```

哈希算法的思想：

通过某种映射算法，**将一个数据映射到唯一对应的序号**，再将其存储起来。

因为“一一对应”，所以**哈希算法大幅度提升了查找数据的效率**，在不存在碰撞的情况下，时间远远短于顺序查找。

缺点在于，**因为映射算法的结果是无序的**，因此，哈希表很难支持按照某种范围提取数据，以及排序等工作，**这也是 PostgreSQL 哈希索引不支持 ORDEY BY 的原因**。

开始理解哈希索引：动态哈希表思想

RESEARCH CONTRIBUTIONS

Algorithms and Data Structures

Daniel Sleator Editor

Dynamic Hash Tables

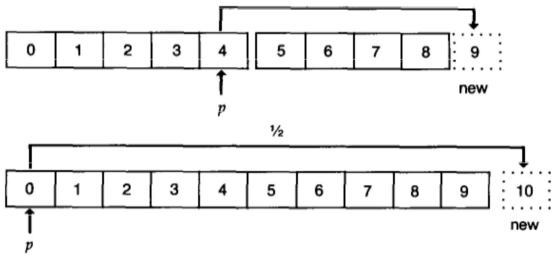
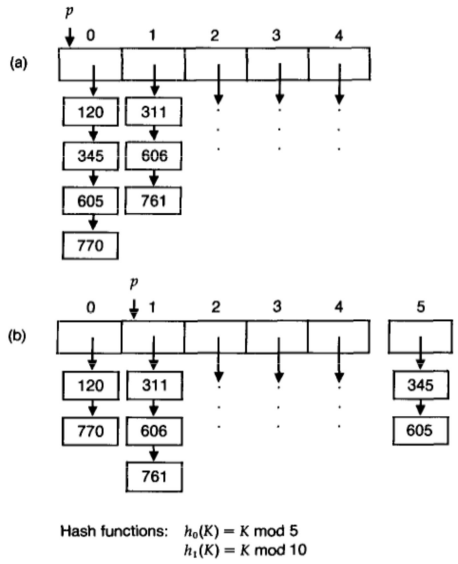


FIGURE 1. Illustration of the Expansion Process of Linear Hashing

普通哈希表

动态哈希表



Hash functions: $h_0(K) = K \bmod 5$
$h_1(K) = K \bmod 10$

FIGURE 2. An Example of Splitting a Bucket

动态哈希表算法的思想：

将普通的线性一维排列的哈希表，转变为**先划分到某个“段”，再将桶分配到段下面的小哈希表**的一种策略。

这就**降低了数据膨胀时扩充哈希表的难度**。

启发了后续很多软件的设计。

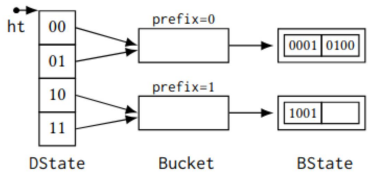
PostgreSQL 的系统缓存按照这个思想进行组织。

开始理解哈希索引：可拓展哈希表思想

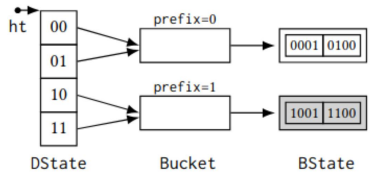
An Efficient Wait-free Resizable Hash Table

ta Fatourou Nikolaos D. Kallimanis Thomas I

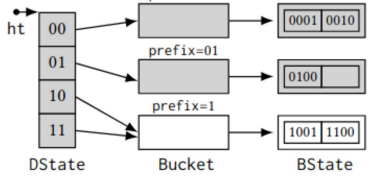
To implement a hash table that complies with these design rules, we propose an algorithm based on *extendible hashing*, a dynamic hashing technique that considers keys as bit strings [5]. An extendible hash table can be seen as an array (the *directory*) of pointer to fixed-size buckets. In its sequential implementation, every resizing operation is local, e.g., one bucket can be split into two without modifying the other buckets.



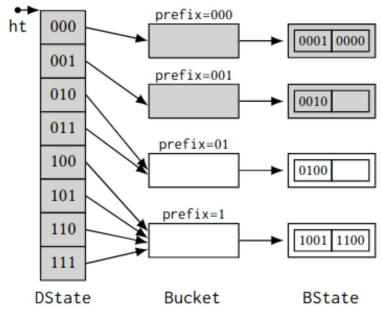
(a) Initial state



(b) After inserting item 1100



(c) After bucket splitting (inserting item 0010)



(d) After directory resizing (inserting item 0000)

可拓展哈希表可以看作为一组指向可变尺寸桶的指针数组

可拓展哈希表总是和位运算深度融合，前缀运算成为区分各个哈希桶的关键

PostgreSQL HighMask 存储新的扩容后的映射信息，而 LowMask 则是旧的映射信息

数据会先转换为 HashKey 再构建对应的 （Key, Value）键值对

The Tiny Encryption Algorithm (TEA)

One of the most secure cipher algorithms ever devised ...
... and certainly the simplest!

```
void code(long* v, long* k)  {
unsigned long y=v[0],z=v[1], sum=0,   /* set up */
 delta=0x9e3779b9,   /* a key schedule constant */
 n=32 ;
while (n-->0) {            /* basic cycle start */
  sum += delta ;
    y += ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]) ;
    z += ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]) ;
              }    /* end cycle */
v[0]=y ; v[1]=z ; }
```

要点：`amroutine->amkeytype = INT4OID;`

- PostgreSQL 先会通过 **TEA 算法**把数据转换为 **HashKey** 再同 TID 构造 （Key, Value） 键值对（这就是为什么 amkeytype 是 int）
- **扩容不需要重分布**（但是 PG 的实现并非无锁算法）
- 涉及存储层（Page）与日志模块（Wal），因此代码复杂程度上升，但基本逻辑不变

开始理解哈希索引：PostgreSQL 物理存储（Page 简介）

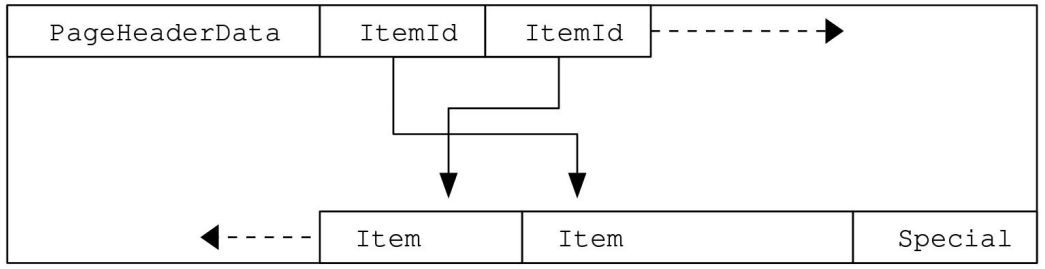
CHINA POSTGRESQL ASSOCIATION

要点：

- 'item' 用于指代存储于页面中的一条单独数据（或者是数据表中的一行，或者是一条索引记录）
- **所有的数据最终均被组织在页面之中**，页面中记录了 wal 日志序列号，事务信息的有关内容
- 每一条元组/记录都以 **(offset,length)** 的形式存储于 **ItemIdData** 部分中

A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.
（TID 与物理存储的对应关系）

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of WAL record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none



Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of item identifiers pointing to the actual items. Each entry is an (offset,length) pair. 4 bytes per item.
Free space	The unallocated space. New item identifiers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

开始理解哈希索引：PostgreSQL 物理存储（Page 简介）

要点：

- 'item' 用于指代存储于页面中的一条单独数据（或者是数据表中的一行，或者是一条索引记录）
- **所有的数据最终均被组织在页面之中**，页面中记录了 wal 日志序列号，事务信息的有关内容
- 每一条元组/记录都以 **(offset,length)** 的形式存储于 **ItemIdData** 部分中

Access Method 的通用信息均存储于 PageHeaderData 结构体中

```
* AM-generic per-page information is kept in PageHeaderData.
*
* AM-specific per-page data (if any) is kept in the area marked "special
* space"; each AM has an "opaque" structure defined somewhere that is
```

所有的数据 **Access Methon** 的自定义数据要存储于页面的 **"Special Space"(特殊存储区域)**

```
opaque = HashPageGetOpaque(page);
```

一个自页面中提取数据的例子

```
/* page type (flags) */
pagetype = opaque->hasho_flag & LH_PAGE_TYPE;
if (pagetype == LH_META_PAGE)
    type = "metapage";
```

```
#define PageGetSpecialPointer(page) \
( \
    PageValidateSpecialPointer(page), \
    ((page) + ((PageHeader) (page))->pd_special) \
)
```

一个**HashPageGetOpaque** 本质上是提取 **pg_special** 区域

开始理解哈希索引：PostgreSQL 物理存储（Smgr 简介）

CHINA POSTGRESQL ASSOCIATION

smgr.c src/backend/storage/smgr/smgr.c/NSmgr

```
typedef struct f_smgr
    void        (*smgr_registersync)
    (SMgrRelation reln, ForkNumber forknum);
} f_smgr;

static const f_smgr smgrsw[] = {
    /* magnetic disk */
    {
        .smgr_init = mdinit,
        .smgr_shutdown = NULL,
```

要点：

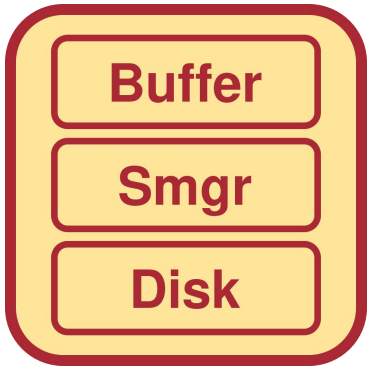
- **Smgr（Storage Managers），是衔接物理数据（Page）与内存缓冲区（Buffer）的桥梁**
- **一般哈希索引乃至于其它索引更多是与 Buffer 层打交道，提取数据**

postgresql - hashpage.c

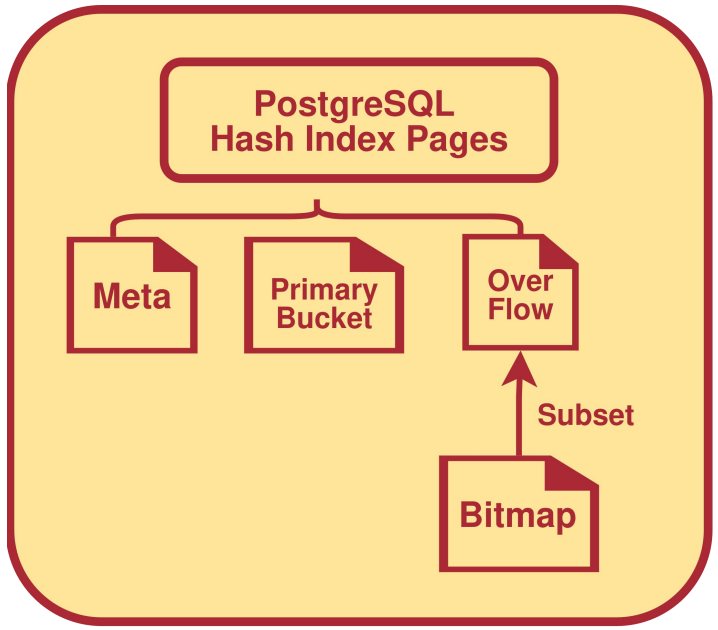
```
page = BufferGetPage(buf);
opaque = HashPageGetOpaque(page);
```

bufmgr.c 2 gr.c/ExtendBufferedRel(BufferManagerRelation, ForkNumber, BufferAccessStrategy, uint32)

```
Buffer
ReadBufferWithoutRelcache(RelFileLocator rlocator, ForkNumber forkNum,
                          BlockNumber blockNum, ReadBufferMode mode,
                          BufferAccessStrategy strategy, bool permanent)
{
    SMgrRelation smgr = smgropen(rlocator, INVALID_PROC_NUMBER);
```



开始理解哈希索引：PostgreSQL 物理存储（Buffer 简介）



要点：

- **Buffer 层**是 PostgreSQL 存储于内存中的数据，**采用页面的方式来组织与管理数据**
- **PostgreSQL 的索引均为二级索引**，因此索引本身所涉及的自定义数据与物理数据都是分离存储的
- **WAL 日志同样以页面为逻辑单位展开工作**，日志是实现故障后恢复以及流复制的关键组件

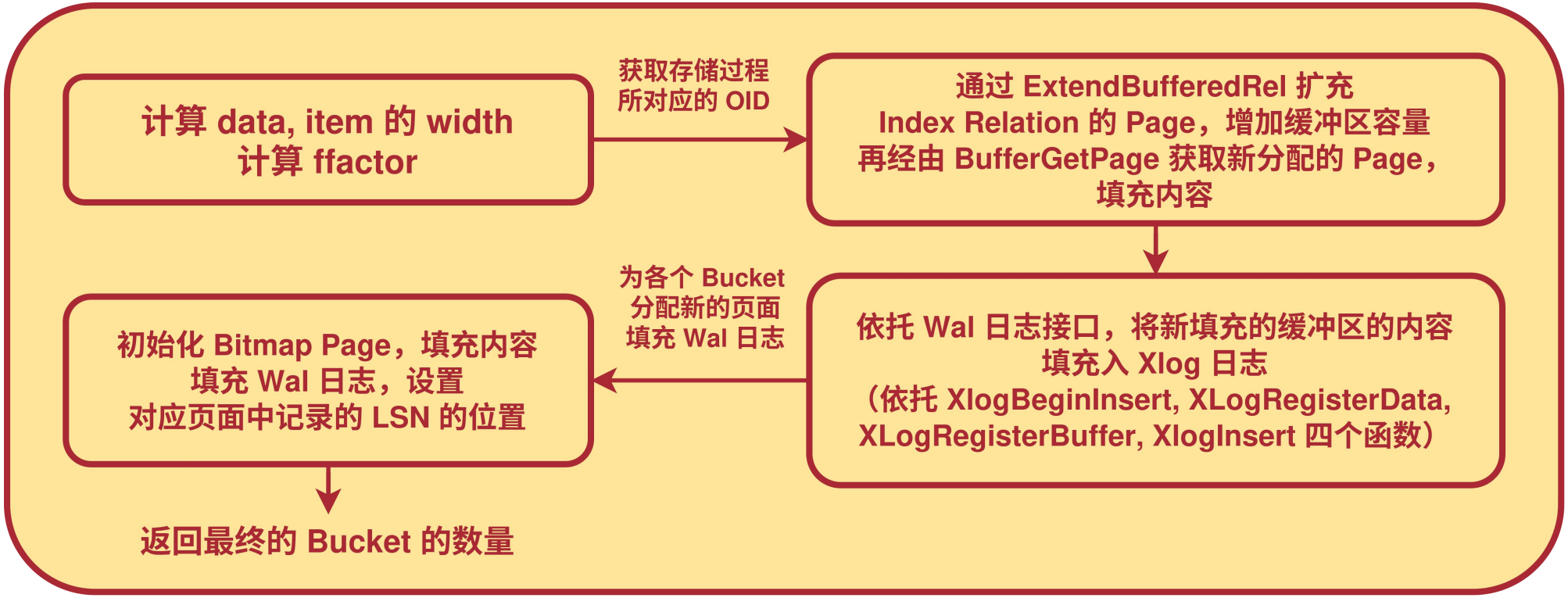
postgresql - hashpage.c

```
buf = ReadBuffer(rel, blkno);

if (access != HASH_NOLOCK)
    LockBuffer(buf, access);

/* ref count and lock type are correct */

_hash_checkpage(rel, buf, flags);
```

postgresql - hashpage.c

```
XLogBeginInsert();
XLogRegisterData((char *) &xlrec, SizeOfHashInitBitmapPage);
XLogRegisterBuffer(0, bitmapbuf, REGBUF_WILL_INIT);
```

深入理解哈希索引：哈希索引表的组建



要点：

- 每一个 Bucket 对应着一个 Block Number
- 每一个 Bucket 里面储存着数条 TID，即这条哈希记录所关联着的元组
- Wal 日志保证了哈希索引的持久化存储

postgresql - hash.c

```
amroutine->ambuildempty = hashbuildempty;
```

```
hashbuildempty(Relation index)
{
    _hash_init(index, 0, INIT_FORKNUM);
}
```

知识补充：PostgreSQL Index Relation 创建的时间

postgresql - index.c

```
Oid
index_create(Relation heapRelation,
             const char *indexRelationName,
             Oid indexRelationId,
             Oid parentIndexRelid,
             Oid parentConstraintId,
             RelFileNumber relFileNumber,
             IndexInfo *indexInfo,
             const List *indexColNames,
             Oid accessMethodId,
```

postgresql - index.c

```
indexRelation = heap_create(indexRelationName
```

要点：

- **Index Relation** 用于存储索引私有的数据，这体现了 PostgreSQL 索引数据与物理数据相互独立的特点
- **accessMethodId** 区分了不同种类的索引，PostgreSQL 将会通过函数指针的方法，，让这些索引各自独立地完成索引构建的工作

深入理解哈希索引：哈希索引表的组建

CHINA POSTGRESQL ASSOCIATION

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

既有数据表的数据

索引数据表的数据

额外的索引数据

存储索引数据的 数据表已经创建，但此时没有内容

Build a new index. The index relation has been physically created, but is empty. It must be filled in with whatever fixed data the access method requires, plus entries for all tuples already existing in the table. Ordinarily the `ambuild` function will call `table_index_build_scan()` to scan the table for existing tuples and compute the keys that need to be inserted into the index.

需要依托既有数据构建索引数据并插入到索引数据表中

根据传递参数，估计数据规模

```
1  /* Estimate the number of rows currently present in the table */
2  estimate_rel_size(heap, NULL, &relpages, &reltuples, &allvisfrac);
3  /* Initialize the hash index metadata page and initial buckets */
4  num_buckets = _hash_init(index, reltuples, MAIN_FORKNUM);
```

先初始化 meta 数据页并初始化哈希表

```
/* do the heap scan */
reltuples = table_index_build_scan(heap, index, indexInfo, true,
                                   true, hashbuildCallback, (void *) &buildstate, NULL);
```

逐个扫描数据表的元组将其插入到哈希索引数据表之中

PostgreSQL 很喜欢把某个连续处理过程中需要长期储存而非短期引用的数据叫做 “xxxstate”

```
postgresql - hash.c

amroutine->ambuild = hashbuild;
```

要点：

- **hashbuildempty** 与 **hashbuild** 的主要区别在于，一个是完全新建哈希表，一个是完全新建哈希表以后需再填充数据
- **hashbuild** 涉及到数据统计、依托排序接口做元组排序等额外的情况，因此更加复杂

深入理解哈希索引：哈希索引表的组建

CHINA POSTGRESQL ASSOCIATION

postgresql - hash.c

```c
/* do the heap scan */
reltuples = table_index_build_scan(heap, index, indexInfo, true,
            true, hashbuildCallback, (void *) &buildstate, NULL);
```

postgresql - hash.c

```c
/*
 * Per-tuple callback for table_index_build_scan
 */
static void
hashbuildCallback(Relation index,
                  ItemPointer tid,
                  Datum *values,
                  bool *isnull,
                  bool tupleIsAlive,
                  void *state)
{
    HashBuildState *buildstate = (HashBuildState *) state;
    Datum       index_values[1];
    bool        index_isnull[1];
    IndexTuple  itup;

    /* convert data to a hash key; on failure, do not insert anything */
    if (!_hash_convert_tuple(index,
                             values, isnull,
                             index_values, index_isnull))
        return;
```

缓冲区

```c
typedef struct
{
    HSpool     *spool;      /* NULL if not using spooling */
    double      indtuples;  /* # tuples accepted into index */
    Relation    heapRel;    /* heap relation descriptor */
} HashBuildState;
```

已经插入索引表的元组数量

数据表的描述符（非 Index Relation）

```c
    hashkey = _hash_datum2hashkey(index, user_values[0]);
    index_values[0] = UInt32GetDatum(hashkey);
    index_isnull[0] = false;
```

原始元组数据

使用 TEA 算法把元组数据转换为 HashKey

索引数据

```c
/* Either spool the tuple for sorting, or just put it into the ind
if (buildstate->spool)
    _h_spool(buildstate->spool, tid, index_values, index_isnull);
else
{
    /* form an index tuple and point it at the heap tuple */
    itup = index_form_tuple(RelationGetDescr(index),
                            index_values, index_isnull);
    itup->t_tid = *tid;
    _hash_doinsert(index, itup, buildstate->heapRel, false);
    pfree(itup);
}

buildstate->indtuples += 1;
```

调用 heap_fill_tuple 等接口

在物理存储上为新的索引记录开辟空间

所有的 IndexTuple 都应当对应一条 Tuple ID

执行插入操作

更新统计数据

要点：

- 索引记录诸条插入（插入单条元组的过程实现相当复杂，因此不在这里阐述）
- PostgreSQL 内部存在着通用的插入索引记录的接口，元组排序也是

深入理解哈希索引：插入新哈希索引


```
if (!_hash_convert_tuple(rel,
  使用 TEA 算法把数据转换为 HashKey    values, isnull,
                                index_values, index_isnull))
    return false;
/* form an index tuple and point it at the heap tuple */
itup = index_form_tuple(RelationGetDescr(rel), index_values, index_isnull);
itup->t_tid = *ht_ctid;  申办新的 Index Tuple，并把 tid 设置为对应的 tid
_hash_doinsert(rel, itup, heapRel, false);
pfree(itup);    执行插入操作
```

```
amroutine->aminsert = hashinsert;
```

要点：

- hashinsert 函数可以理解为 hashbuild 函数中“逐个插入元组”的另外一种实现
  - 只是这一次**只插入了一条元组**
  - 数据的来源**直接源自于用户**，而不是数据表既有数据（**因此传入参数复杂很多**）
  - **研究透 _hash_doinsert 是理解哈希索引的关键**

深入理解哈希索引：掩码运算的作用


```
Bucket
_hash_hashkey2bucket(uint32 hashkey, uint32 maxbucket,
                     uint32 highmask, uint32 lowmask)
{
    Bucket      bucket;

    bucket = hashkey & highmask;
    if (bucket > maxbucket)
        bucket = bucket & lowmask;

    return bucket;
}
/* Starting a new doubling */
metap->hashm_lowmask = metap->hashm_highmask;
metap->hashm_highmask = new_bucket | metap->hashm_lowmask;
```

按位或运算符|常用于将某些二进制位置为 1，例如：

```
x = x | SET_ON;
```

位向量一个很有用的应用就是表示有限集合。我们可以用位向量$[a_{w-1}, \cdots, a_1, a_0]$编码任何子集$A \subseteq \{0, 1, \cdots, w-1\}$，其中$a_i = 1$当且仅当$i \in A$。例如(记住我们是把$a_{w-1}$写在左边，而将$a_0$写在右边)，位向量$a \doteq [01101001]$表示集合$A = \{0, 3, 5, 6\}$，而$b \doteq [01010101]$表示集合$B = \{0, 2, 4, 6\}$。使用这种编码集合的方法，布尔运算|和&分别对应于集合的并和交，而~对应于于集合的补。还是用前面那个例子，运算$a \& b$得到位向量[01000001]，而$A \cap B = \{0, 6\}$。

按位与运算符&经常用于屏蔽某些二进制位，例如：

```
n = n & 0177;
```

该语句将 n 中除 7 个低二进制位外的其它各位均置为 0

要点：

- 掩码运算是承载 HashKey 与 Bucket 编号的桥梁（& 屏蔽了 HashKey 与 HighMask 不同的部分）
- 扩容时，则把新的 Bucket 编号整合到 HighMask 中，而旧的 LowMask 则存旧的 HighMask 值

深入理解哈希索引：关于 splitpoint

```
When a new bucket is to be added to the index,
exactly one existing bucket will need to be "split", with some of its
tuples being transferred to the new bucket according to the updated
key-to-bucket-number mapping.  This is essentially the same hash table
management technique embodied in src/backend/utils/hash/dynahash.c for
in-memory hash tables.
```

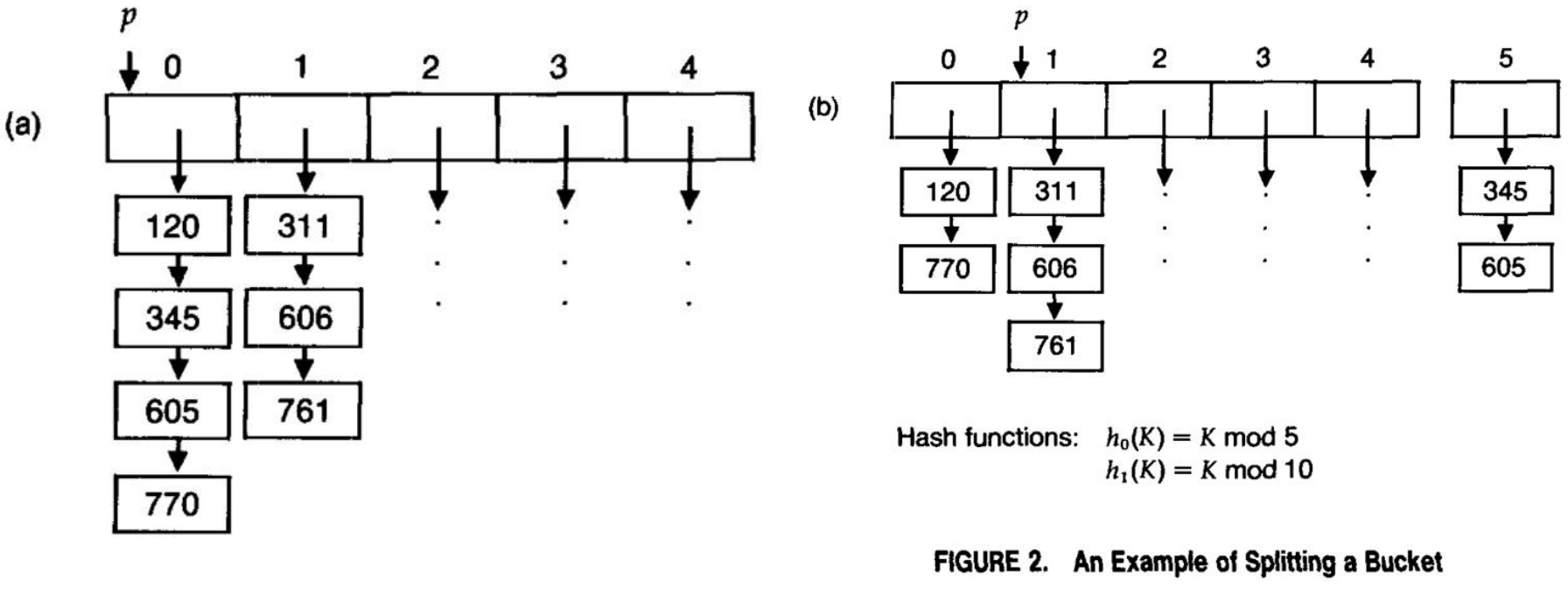


FIGURE 2. An Example of Splitting a Bucket

PostgreSQL 的 SplitPoint 本质上是为“哈希表扩容”而准备的，其想法可以参考《Dynamic Hash Table》（动态哈希表）一文。

这种做法的好处就是可以把 Hash 表扩容时涉及的重分布操作的影响给尽量缩小，进而提升效率。

深入理解哈希索引：关于 splitpoint

```
/* account for buckets before splitpoint_group */
total_buckets = (1 << (splitpoint_group - 1));

/* account for buckets within splitpoint_group */
phases_within_splitpoint_group =
    (((splitpoint_phase - HASH_SPLITPOINT_GROUPS_WITH_ONE_PHASE) &
      HASH_SPLITPOINT_PHASE_MASK) + 1); /* from 0-based to 1-based */
total_buckets +=
    (((1 << (splitpoint_group - 1)) >> HASH_SPLITPOINT_PHASE_BITS) *
     phases_within_splitpoint_group);
```

要点：

- splitpoint 代表着哈希桶扩容的次数（所以可以通过它计算出哈希桶的总容量）
- 根源思想可以阅读《**Extendible Hashing-A Fast Access Method for Dynamic Files**》这篇论文
- PostgreSQL 涉及到物理存储地址计算，因此增加了难度

If the new local depth of P is bigger than the current directory depth, then do the following.

a. Increase the depth of the directory by one.
b. Double the size of the directory, and update the pointers in the obvious manner.
c. (Optional) Set to zero the entry giving the number of entries on the leaf pages P and P^*.

INSERT all $(K, I(K))$ pairs one at a time from the temporary area Q.

深入理解哈希索引：插入新哈希索引

CHINA POSTGRESQL ASSOCIATION

postgresql - hash.c

```c
/* 依据数据构造 hashkey */
hashkey = _hash_get_indextuple_hashkey(itup);

/* 获取 Meta 的页面信息 */
metabuf = _hash_getbuf(rel, HASH_METAPAGE, HASH_NOLOCK, LH_META_PAGE);
metapage = BufferGetPage(metabuf);

/* Lock the primary bucket page for the target bucket. */
buf = _hash_getbucketbuf_from_hashkey(rel, hashkey, HASH_WRITE,
                                      &usedmetap);
```

```c
typedef struct HashPageOpaqueData
{
    BlockNumber hasho_prevblkno;    /* see above */
    BlockNumber hasho_nextblkno;    /* see above */
    Bucket      hasho_bucket;       /* bucket number this pg belongs to */
    uint16      hasho_flag;         /* page type code + flag bits, see above */
    uint16      hasho_page_id;      /* for identification of hash indexes */
} HashPageOpaqueData;
```

存储哈希桶链表中前一页的数据块编号

下一页的数据块编号，没有则为 InvalidBlockNumber

本哈希桶所属块编号

描述运行时状态

用于区分不同索引

要点：

- 可以发现，**构造 HashKey 与获取 Meta 页面**是哈希索引经常要做的一项工作
- 各类页面按照一定的结构进行组织（难点：把一个链表结构存储于**分块存储**的结构之上，**所以我们要记住“块的地址”**）

广东管理大学

深入理解哈希索引：插入新哈希索引


```
if (H_BUCKET_BEING_SPLIT(oopaque))
#define H_NEEDS_SPLIT_CLEANUP(opaque)   (((opaque)->hasho_flag & LH_BUCKET_NEEDS_SPLIT_CLEANUP) != 0)
#define H_BUCKET_BEING_SPLIT(opaque)    (((opaque)->hasho_flag & LH_BUCKET_BEING_SPLIT) != 0)
#define H_BUCKET_BEING_POPULATED(opaque)    (((opaque)->hasho_flag & LH_BUCKET_BEING_POPULATED) != 0)
#define H_HAS_DEAD_TUPLES(opaque)       (((opaque)->hasho_flag & LH_PAGE_HAS_DEAD_TUPLES) != 0)
```

```
if (PageGetFreeSpaceForMultipleTuples(npage, nitups + 1) < (all_tups_size + itemsz))
{
    // ...
    /* chain to a new overflow page */
    nbuf = _hash_addovflpage(rel, metabuf, nbuf, (nbuf == bucket_nbuf));
    npage = BufferGetPage(nbuf);
    nopaque = HashPageGetOpaque(npage);
}
```

There is currently no provision to shrink a hash index, other than by rebuilding it with REINDEX. There is no provision for reducing the number of buckets, either.

除非使用 REINDEX 指令
否则哈希桶的数量不会削减

要点：

- PostgreSQL 哈希页面中的 flag 记录着哈希索引的状态，而 SPLIT 是哈希的扩容操作
- 扩容会调动 PostgreSQL 中的动态哈希表接口用以存储那些已经被移动过的 TID，避免被二次移动，同时 PostgreSQL 将会添加新的 OverFlow 页面用以完成扩容操作
- OvewFlow 页的添加即是扩容（为什么这么复杂？因为涉及到磁盘储存，块地址计算）

深入理解哈希索引：插入新哈希索引


```
while (PageGetFreeSpace(page) < itemsz)
{
    BlockNumber nextblkno;
    // ...
    nextblkno = pageopaque->hasho_nextblkno;

    // 有，就用现在的
    if (BlockNumberIsValid(nextblkno))
    {
        buf = _hash_getbuf(rel, nextblkno, HASH_WRITE, LH_OVERFLOW_PAGE);
        page = BufferGetPage(buf);
    }
    else // 无，就开辟新的 overflow 页面
    {
        buf = _hash_addovflpage(rel, metabuf, buf, (buf == bucket_buf));
        page = BufferGetPage(buf);
    }
    pageopaque = HashPageGetOpaque(page);
}
itup_off = _hash_pgaddtup(rel, buf, itemsz, itup, sorted);
MarkBufferDirty(buf);
PageAddItem(page, (Item) itup, itemsize, itup_off, false, false)
```

要点：

- 下面这个 for 循环的目的就在于，找到一个可以信赖的页面，存储 TID（HashKey 可以通过计算得到，所以并不需要存 HashKey）
- 标记页面为脏页意味着页面在稍后会被刷新到磁盘上
- 还需要通过 Wal 日志展开备份工作
- 添加 OverFlow 页面意味着整张哈希表也有可能扩充
- 由此便是哈希索引插入时基本所执行的操作（出于简化目的，我们没有谈论锁）

实践验证：哈希索引 在 Index-Scan 下


```
postgres=# create table test (id integer);
CREATE TABLE
postgres=# insert into test values(generate_series(1, 2560000) * random() * 100);
INSERT 0 2560000
postgres=# explain analyze select * from test where id = 256;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..25661.43 rows=1 width=4) (actual time=39.210..41.454 rows=0 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test  (cost=0.00..24661.33 rows=1 width=4) (actual time=37.380..37.382 rows=0 loops=3)
         Filter: (id = 256)
         Rows Removed by Filter: 853333
 Planning Time: 0.079 ms
 Execution Time: 41.469 ms
(8 rows)

postgres=# create index on test using hash (id);
CREATE INDEX
postgres=# explain analyze select * from test where id = 256;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Scan using test_id_idx on test  (cost=0.00..8.02 rows=1 width=4) (actual time=0.010..0.011 rows=0 loops=1)
   Index Cond: (id = 256)
 Planning Time: 0.165 ms
 Execution Time: 0.022 ms
(4 rows)

postgres=#
```

插入 256 万条记录

执行 '=' 搜索（无哈希索引）

顺序扫描耗时 41.469 ms

建立哈希索引

哈希索引扫描耗时 0.022ms

实践验证：哈希索引 在 Bitmap-Index-Scan 下

```
postgres=# create table test (id text);
CREATE TABLE
postgres=# insert into test values (generate_series(1, 1280000, 1)::text);
INSERT 0 1280000
postgres=# explain analyze select * from test where id = '256';
                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..11061.20 rows=3852 width=32) (actual time=0.189..39.131 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test  (cost=0.00..9676.00 rows=1605 width=32) (actual time=22.676..34.860 rows=0 loops=3)
         Filter: (id = '256'::text)
         Rows Removed by Filter: 426666
 Planning Time: 0.066 ms
 Execution Time: 39.143 ms
(8 rows)

postgres=# create index on test using hash (id);
CREATE INDEX
postgres=# explain analyze select * from test where id = '256';
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=177.60..6190.95 rows=6400 width=32) (actual time=0.012..0.013 rows=1 loops=1)
   Recheck Cond: (id = '256'::text)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on test_id_idx  (cost=0.00..176.00 rows=6400 width=0) (actual time=0.007..0.007 rows=1 loops=1)
         Index Cond: (id = '256'::text)
 Planning Time: 0.076 ms
 Execution Time: 0.021 ms
(7 rows)
```

插入 128 万条记录，
以字符串类型填入

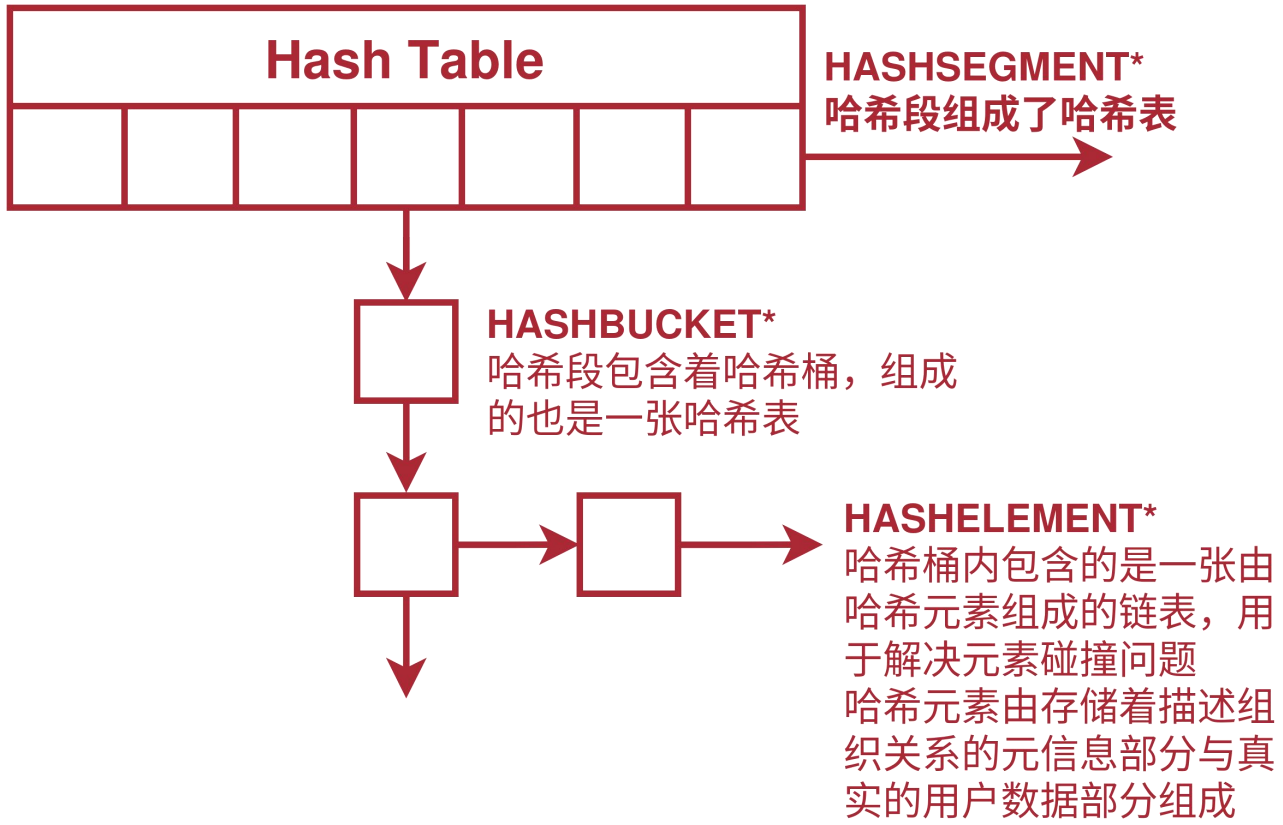
执行 '=' 搜索

并行顺序扫描耗时 39.143ms

创建哈希索引

哈希索引位图索引扫描耗时 0.021 ms

拓展：PG 系统缓存的实现原理简介



要点：

- PostgreSQL 系统缓存实际上就是一张组织起来的动态哈希表
- 在动态哈希表中不存在对应内容时，会调用前面提到的 Smgr 接口，自磁盘上提取页面
- 因为 PostgreSQL 是多用户同时使用，所以需要锁机制展开协调，这就是 FreeList 存在的原因

postgresql - relcache.c

```
static HTAB *RelationIdCache;
```

postgresql - relcache.c

```
/* Hash table control struct is an opaque
 type known only within dynahash.c */
typedef struct HTAB HTAB;
```

知识拓展：PostgreSQL 物理存储文件命名特点

CHINA POSTGRESQL ASSOCIATION

Fork

A concept describing the type of a relation's physical storage

In PostgreSQL's data directory, each relation's data is stored in a so-called **`fork`**:

main fork (no suffix)
free space map (suffix: `fsm`)
visibility map (suffix: `vm`)
initialization fork(suffix: `init`)

The main `fork`, which contains the actual data (possibly split over multiple files), is always present.

要点：

- **对于普通的关系数据**，物理文件或者索引会被命名为 **FileNode 所对应的数字**（可以在 **pg_class.relfilenode** 中看到）
- **对于临时表而言**，文件名会以 **BBB_FFF** 的形式存在（**BBB 为 PostgreSQL 进程 id**，**FFF 为 FileNode 所对应的数值**）
- 文件名称视情况会加后缀
  - 对于存储最重要信息的物理文件，不会加后缀名（即 **main fork**）
  - **_fsm** 后缀代表该文件存储着关系表中尚且空闲的数据区域
  - **_vm** 后缀存储着关系数据中有关死元组的信息（同 MVCC 机制有关系）
  - 无日志表或者索引的数据存储于 **_init** 后缀的文件中

拓展：ManGrove 项目与 PG 生态建设的一点思考

Mangrove 拓展管理器

Mangrove 是一款基于 Python 制作的 PostgreSQL 包管理器，目前已经托管于 AtomGit 平台之上，项目的目标在于降低 PostgreSQL 的内核开发门槛与拓展发布门槛，促进中国 PostgreSQL 行业整体水平的提升。


```
# python -m mangrove -i [希望安装的 PostgreSQL 拓展]
python -m mangrove -i vector
```

```
wenyi@192:~$ python -m mangrove -i vector
    Extension      Source    Repo    Description
--  -------------  --------  ------  ----------------------------------------------------
 0  vector         PGXN      PGXN    Open-source vector similarity search for Postgres
 1  vectorize      PGXN      PGXN    The simplest way to do vector search on Postgres
 2  vops           PGXN      PGXN    Vectorized Operations
 3  aggs_for_vecs  PGXN      PGXN    Aggregate functions for vectors (arrays) of numbers
Which extension to download? [0 ~ 3]
0
This extension may need higher privilege to install, make sure you give these privilege
```

（实际工作时的部分截图）

要点：

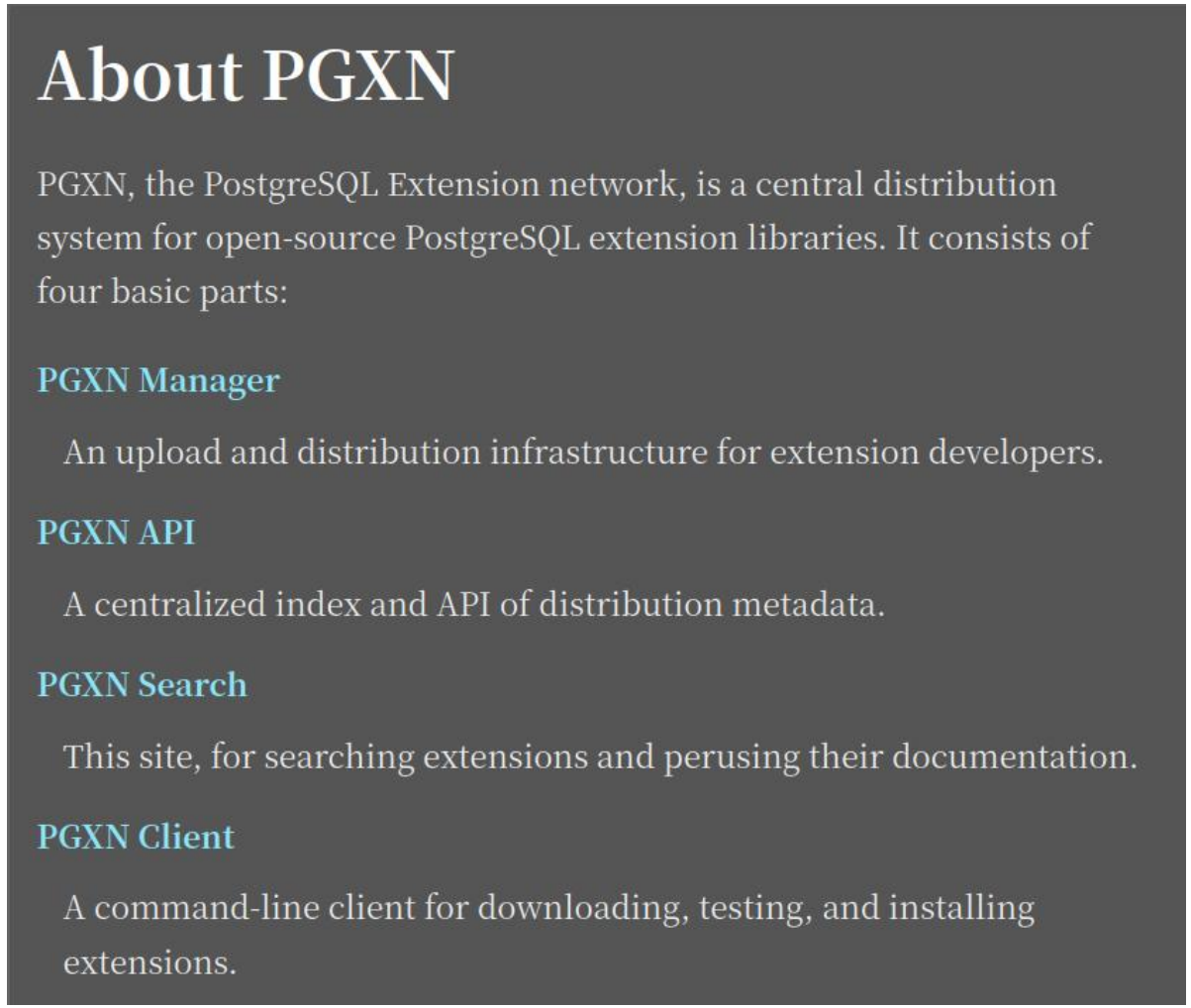
- 开源基础设施多年发展，如今**几乎全部的 PostgreSQL 拓展均使用 git 展开源代码管理**
- PGXS 使得 **PostgreSQL 拓展安装流程规范化**，几乎都是 “./configure --- make --- make install”
- pgxn.org 积极鼓励镜像站，因此**建设新的拓展管理器并非难事**

```
python -m mangrove pg_hello_world
```

```
wenyi@fedora:~$ python -m mangrove -s pg_hello_world
    Extension       Source    Repo         Description
--  --------------  --------  -----------  ---------------------------
 0  pg_hello_world  MANGROVE  magv-index   PostgreSQL Hello world 拓展程序
wenyi@fedora:~$
```

（可以发现，目前的社区仓库已经可以搜索到我们的 pg_hello_world 拓展了）

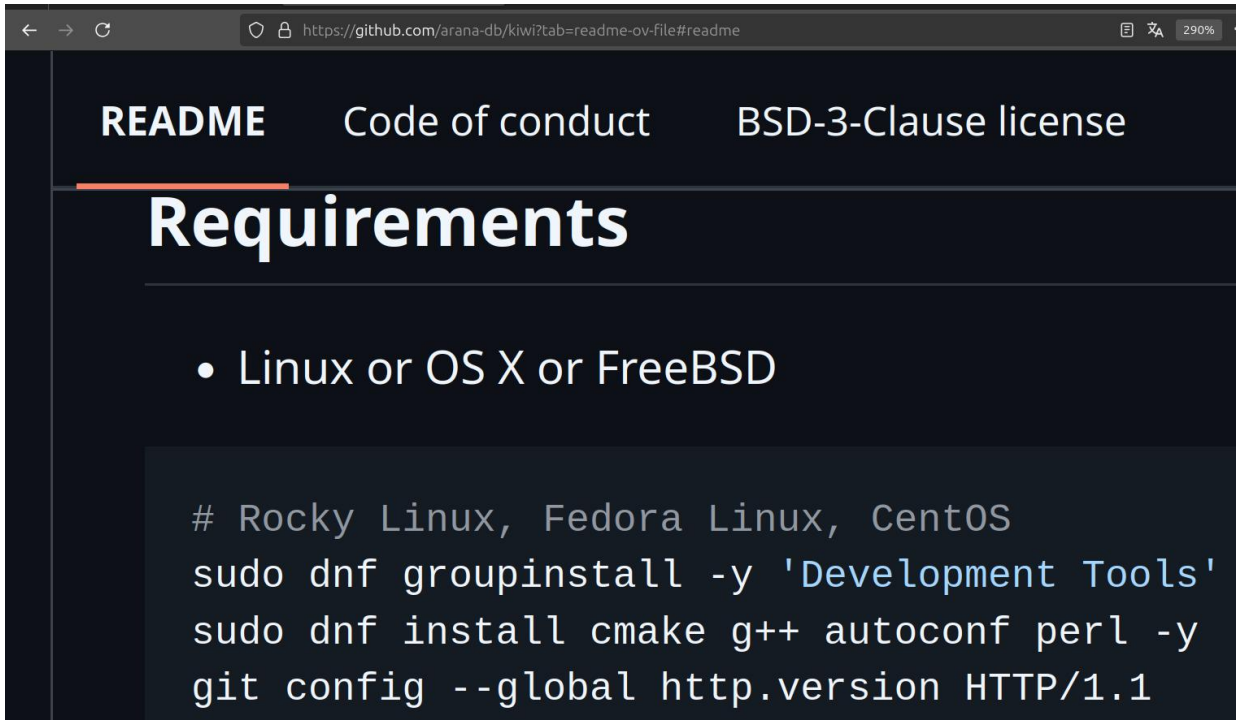
拓展：ManGrove 项目与 PG 生态建设的一点思考



要点：

- PostgreSQL 国内拓展生态建设不温不火，既是也不是一个技术原因
  - **不是一个技术原因**
    - 建设 PostgreSQL 拓展管理器并非难事，同时**技术上已证明整合 pgxn.org 生态具有可行性**
    - PostgreSQL 拓展的**英文技术材料，内核原理书籍已经相当丰富**
    - **国内厂商数目繁多**
  - **是一个技术原因**
    - 国内“开箱即用”的内核材料依旧太少
    - **很多技术材料综合性依旧偏弱，同时看不到“代码背后的东西”**（真正好的材料是论文联系代码，理论联系工程）
    - 同解决类似问题的社区（如 Apache OpenDAL，Kiwi 存储引擎，TiDB 社区）走动过少（**但是这也和技术栈不太一样有关**）
    - 因为各行其政，导致难以形成合力共同促进 数据库生态

拓展：ManGrove 项目与 PG 生态建设的一点思考



要点：

- **PostgreSQL 内核的研发工作，可以促进其它方面存储引擎、中间件、数据库的发展**（比如，Kiwi 存储引擎的部署部分，就借鉴了 IvorySQL/PostgreSQL 的智慧）
- 同样的，**其它的社区可以促进 PostgreSQL 的繁荣（参与 Kiwi 存储引擎的研发极大促进了我对于 PostgreSQL 的理解）**
- 可以通过一些中立方（如开放原子开源基金会），**以公共技术材料/公共知识的方式对不同项目间的技术栈差异进行磨合**
- **呼吁行业内社区，行业内组织与具备普遍代表性的开放原子开源基金会进行联系，推动联系高校，减少重复建设**

拓展：ManGrove 项目与 PG 生态建设的一点思考

徐浩荣

目前就读于 广州市新侨学校， ManGrove 项目作者，在计算机科学领域已经有了一定的学习与积累。

Blog: **https://haorongx.github.io**

目标：“To remove all barriers in the way of computer science”（移除所有计算机科学中的障碍）

祝福他学有所成!

简单总结

这一次我们所解读的哈希索引实现，依旧是管中窥豹，未涉及于全貌（哈希索引的扫描，虽然从数据结构而言是容易的，但是它的工程实现，会牵涉到 PostgreSQL 查询引擎的不少知识，进而导致报告内容过多，时间有限，故本次不阐述），但依旧可以帮助各位以一个较为深入的角度理解 PostgreSQL 哈希索引。

感兴趣的读者可以去阅读《PostgreSQL 内核实现分析》，PostgreSQL 文档中牵涉到哈希索引的有关部分进行探究（但是我们同时也指出，PostgreSQL 的很多材料是不会告诉你这种方法所对应的背后论文的，这就需要读者自己想办法去搜集材料）。

我们同时期待业内风气能够更加坦率开放，让业内的材料、观点、想法更加充分流动起来，因此我们也在这一次报告中分享了我们自身的经验与思考。

我们非常期待将来能够更多地**“团结国内内核研发力量，降低工业级数据库研发门槛！”**，解决外方指导内行的唯一办法，就是让内核的知识变得不那么难以理解，以及业内信任的土壤积淀地更多更深。

写在最后

首先，非常谢谢中国科学院愿意提供这样一个舞台！

其次，所有呈上台前的精彩，都离不开幕后很多同志的支持，在这里将其中的许多朋友罗列出来，奉上我诚挚的谢意。

感谢我的家里人，以及我的本科生导师，系主任**袁国铭老师**，感谢中国 PG 分会的**魏波、王其达老师**，感谢开放原子开源基金会**张凯、臧秀涛**、研发部的所有老师，以及 OpenTenBase 社区符芬菊、Mark 老师，IvorySQL 社区任娇、牛世继老师，青学会会长吴洋老师的帮助与支持。

感谢武汉华科开放原子俱乐部王振辰的友谊，以及曾经提供启发与帮助的 **@Makio**，@红发哥，@dba悠然，**@bitstring**，**@_Alex**，**@XuanWo**，@邂逅哥，@大炮，@m佬，@追光者，@陈小伟老师，@进化哥，@可可，@彭冲，@不语，@念念哥，@梦里，@雪宝，@夏夏，@尚卓燃，@奶啤，@于雨，@黄宏亮老师，@严少安，@高英凯，@孤傲小二～阿沐，@徐浩荣，@刘誉达（祝最后这位多睡觉少熬夜）。

期待将来与各位共成长，祝大家工作顺利，谢谢！

期待与你共精彩！

CHINA POSTGRESQL ASSOCIATION

文一

目前，我们正在积极建设基于开放原子开源基金会开源运营专区的“内核观察”开源社区（**https:/datapromoto.atomgit.com**）

期待与行业开发更多可靠、可信、可用的数据库内核研发材料，**降低工业级数据库研发门槛**!

内核观察

内核观察

团结国内内核研发力量，
降低工业级数据库研发门槛！