|  |
| --- |
| 北京科技大学计算机科学与技术系 |
| 阅读PostgreSQL数据库源代码指导 |
| PostgreSQL数据库存储引擎 |

|  |
| --- |
| 《数据库系统原理与实现》课程教研组 曾庆峰  2024-3-1 |

目录

[1 快速走进PostgreSQL数据库世界 7](#_Toc164249903)

[1.1 基本概念 7](#_Toc164249904)

[1.1.1 数据、信息、元数据 7](#_Toc164249905)

[1.1.2 数据库集簇、数据库、模式与模式对象 8](#_Toc164249906)

[1.1.3 数据库管理系统 9](#_Toc164249907)

[1.1.4 数据库系统 10](#_Toc164249908)

[1.2 编译安装PostgreSQL数据库服务器 10](#_Toc164249909)

[1.2.1 安装前的准备工作 10](#_Toc164249910)

[1.2.2 配置编译PostgreSQL源代码 16](#_Toc164249911)

[1.2.3 编译PostgreSQL数据库源代码 17](#_Toc164249912)

[1.2.4 安装PostgreSQL数据库管理系统的二进制程序 17](#_Toc164249913)

[1.2.5 打包PostgreSQL服务器二进制程序 18](#_Toc164249914)

[1.3 PostgreSQL源代码布局结构 19](#_Toc164249915)

[1.3.1 第0级目录 19](#_Toc164249916)

[1.3.2 第1级目录：src 20](#_Toc164249917)

[1.3.3 第2级目录：src/bin 21](#_Toc164249918)

[1.3.4 第2级目录：src/backend 23](#_Toc164249919)

[1.3.5 第2级目录：src/common 25](#_Toc164249920)

[1.3.6 第2级目录：src/tools 28](#_Toc164249921)

[1.3.7 第2级目录：src/interfaces 30](#_Toc164249922)

[1.3.8 第2级目录：src/pl 30](#_Toc164249923)

[1.3.9 第2级目录：src/include 31](#_Toc164249924)

[1.3.10 第3级目录：src/backend/main 33](#_Toc164249925)

[1.3.11 第3级目录：src/backend/postmaster 33](#_Toc164249926)

[1.3.12 第3级目录：src/backend/bootstrap 34](#_Toc164249927)

[1.3.13 第3级目录：src/backend/catalog 35](#_Toc164249928)

[1.3.14 第3级目录：src/backend/libpq 38](#_Toc164249929)

[1.3.15 第3级目录：src/backend/storage 39](#_Toc164249930)

[1.3.16 第3级目录：src/backend/parser 40](#_Toc164249931)

[1.3.17 第3级目录：src/backend/rewrite 42](#_Toc164249932)

[1.3.18 第3级目录：src/backend/optimizer 43](#_Toc164249933)

[1.3.19 第3级目录：src/backend/executor 44](#_Toc164249934)

[1.3.20 第3级目录：src/backend/nodes 47](#_Toc164249935)

[1.3.21 第3级目录：src/backend/tcop 48](#_Toc164249936)

[1.3.22 第3级目录：src/backend/statistics 49](#_Toc164249937)

[1.3.23 第3级目录：src/backend/commands 50](#_Toc164249938)

[1.3.24 第3级目录：src/backend/access 52](#_Toc164249939)

[1.3.25 第3级目录：src/backend/partitioning 54](#_Toc164249940)

[1.3.26 第3级目录：src/backend/replication 54](#_Toc164249941)

[1.3.27 第3级目录：src/backend/regex 56](#_Toc164249942)

[1.3.28 第3级目录：src/backend/jit 57](#_Toc164249943)

[1.3.29 第3级目录：src/backend/foreign 58](#_Toc164249944)

[1.3.30 第3级目录：src/backend/tsearch 59](#_Toc164249945)

[1.3.31 第3级目录：src/backend/snowball 60](#_Toc164249946)

[1.3.32 第3级目录：src/backend/utils 61](#_Toc164249947)

[1.4 PostgreSQL数据库管理系统的二进制程序布局结构 63](#_Toc164249948)

[1.4.1 第0级目录 63](#_Toc164249949)

[1.4.2 第1级目录bin 64](#_Toc164249950)

[1.4.3 第1级目录include 66](#_Toc164249951)

[1.4.4 第1级目录lib 67](#_Toc164249952)

[1.4.5 第1级目录share 69](#_Toc164249953)

[1.4.6 第2级目录lib/pkgconfig 69](#_Toc164249954)

[1.4.7 第2级目录lib/postgresql 71](#_Toc164249955)

[1.4.8 第2级目录share/postgresql 73](#_Toc164249956)

[1.4.9 第3级目录share/postgresql/extension 75](#_Toc164249957)

[1.4.10 第3级目录share/postgresql/tsearch\_data 81](#_Toc164249958)

[1.5 编译安装PostgreSQL客户端 82](#_Toc164249959)

[1.5.1 准备一台Linux客户端机器 82](#_Toc164249960)

[1.5.2 安装前的准备工作 82](#_Toc164249961)

[1.5.3 配置编译PostgreSQL源代码 83](#_Toc164249962)

[1.5.4 编译PostgreSQL数据库源代码 83](#_Toc164249963)

[1.5.5 安装PostgreSQL客户端二进制程序 83](#_Toc164249964)

[1.5.6 打包PostgreSQL客户端二进制程序 84](#_Toc164249965)

[1.6 初始化PostgreSQL数据库集簇（initdb） 84](#_Toc164249966)

[1.7 首次启动PostgreSQL数据库 88](#_Toc164249967)

[1.7.1 PostgreSQL数据库配置文件 88](#_Toc164249968)

[1.7.2 配置PostgreSQL数据库的系统参数文件 88](#_Toc164249969)

[1.7.3 配置PostgreSQL数据库的访问策略 89](#_Toc164249970)

[1.7.4 启动PostgreSQL数据库 89](#_Toc164249971)

[1.7.5 查看PostgreSQL系统配置文件 90](#_Toc164249972)

[1.7.6 查看PostgreSQL数据库的运行状态 90](#_Toc164249973)

[1.7.7 关闭PostgreSQL数据库 90](#_Toc164249974)

[1.7.8 以“Standalone Backend”方式启动PostgreSQL数据库 91](#_Toc164249975)

[1.8 配置PostgreSQL数据库运行在归档日志模式 91](#_Toc164249976)

[1.9 导入大学应用数据库universitydb 93](#_Toc164249977)

[1.10 启动PostgreSQL数据库后的进程情况 93](#_Toc164249978)

[1.11 客户端连接访问PostgreSQL数据库 95](#_Toc164249979)

[1.11.1 使用IP协议远程连接PostgreSQL服务器 96](#_Toc164249980)

[1.11.2 使用Socket本地连接PostgreSQL服务器 97](#_Toc164249981)

[1.12 PostgreSQL数据库的进程 99](#_Toc164249982)

[1.12.1 PostgreSQL数据库进程类别 99](#_Toc164249983)

[1.12.2 PostgreSQL数据库实例进程 101](#_Toc164249984)

[1.13 PostgreSQL数据库对象 105](#_Toc164249985)

[1.13.1 存储数据库对象元数据信息的系统表 105](#_Toc164249986)

[1.13.2 数据库对象标识（OID） 108](#_Toc164249987)

[1.14 数据库 110](#_Toc164249988)

[1.14.1 PostgreSQL数据库集簇有哪些数据库 110](#_Toc164249989)

[1.14.2 使用SQL语句CREATE TABLESPACE创建数据库 111](#_Toc164249990)

[1.14.3 使用命令createdb创建数据库 112](#_Toc164249991)

[1.14.4 模板数据库 112](#_Toc164249992)

[1.14.5 删除数据库 113](#_Toc164249993)

[1.15 模式和模式对象 114](#_Toc164249994)

[1.15.1 存储模式信息的系统表 114](#_Toc164249995)

[1.15.2 在一个数据库中创建模式 116](#_Toc164249996)

[1.15.3 模式是命名空间 116](#_Toc164249997)

[1.15.4 模式搜索路径SEARCH\_PATH 118](#_Toc164249998)

[1.15.5 PostgreSQL数据库集簇的逻辑结构 119](#_Toc164249999)

[1.16 表空间 120](#_Toc164250000)

[1.16.1 表空间是一个操作系统目录 121](#_Toc164250001)

[1.16.2 表空间和数据库的关系 124](#_Toc164250002)

[1.16.3 观察表空间中表的物理存储 128](#_Toc164250003)

[1.16.4 PostgreSQL数据库集簇的物理结构 131](#_Toc164250004)

[1.16.5 观察删除一个表 132](#_Toc164250005)

[1.16.6 删除表空间 133](#_Toc164250006)

[1.17 PostgreSQL数据库的数据字典 134](#_Toc164250007)

[1.17.1 系统表 134](#_Toc164250008)

[1.17.2 系统视图 135](#_Toc164250009)

[1.17.3 动态视图 137](#_Toc164250010)

[1.17.4 数据字典查询示例 137](#_Toc164250011)

[1.18 PostgreSQL数据库实例的内存结构 141](#_Toc164250012)

[1.18.1 系统全局区（SGA） 142](#_Toc164250013)

[1.18.2 程序全局区（PGA） 143](#_Toc164250014)

[1.19 PostgreSQL服务器的体系结构图 145](#_Toc164250015)

[1.20 PostgreSQL数据库实例 147](#_Toc164250016)

[1.21 SQL语句的执行过程 147](#_Toc164250017)

[1.22 表的存储结构 149](#_Toc164250018)

[1.23 数据库权限 149](#_Toc164250019)

[1.23.1 系统权限 149](#_Toc164250020)

[1.23.2 对象权限 149](#_Toc164250021)

[1.24 数据库用户/角色 149](#_Toc164250022)

[附录A 将PostgreSQL数据库集簇恢复到初始状态 151](#_Toc164250023)

[附录B 将PostgreSQL数据库集簇恢复到有大学应用数据集的状态 152](#_Toc164250024)

[参考 153](#_Toc164250025)

[2 PostgreSQL服务器的体系结构 154](#_Toc164250026)

[2.1 PostgreSQL数据库术语 155](#_Toc164250027)

[2.1.1 PostgreSQL数据库术语：PostgreSQL服务器 155](#_Toc164250028)

[2.1.2 PostgreSQL数据库术语：PostgreSQL数据库集簇及其数据库 155](#_Toc164250029)

[2.1.3 PostgreSQL数据库术语：PostgreSQL数据库的模式 156](#_Toc164250030)

[2.1.4 PostgreSQL数据库术语：PostgreSQL数据库对象 157](#_Toc164250031)

[2.1.5 PostgreSQL数据库术语：PostgreSQL数据库表空间 159](#_Toc164250032)

[2.2 PostgreSQL服务器的体系结构 162](#_Toc164250033)

[2.3 PostgreSQL数据库实例 165](#_Toc164250034)

[2.3.1 PostgreSQL数据库实例的进程结构 165](#_Toc164250035)

[2.3.2 PostgreSQL数据库实例的内存结构 171](#_Toc164250036)

[2.4 PostgreSQL数据库集簇 173](#_Toc164250037)

[2.4.1 PostgreSQL数据库集簇的逻辑结构 174](#_Toc164250038)

[2.4.2 PostgreSQL数据库集簇的物理结构 181](#_Toc164250039)

[2.5 PostgreSQL数据库的数据字典 198](#_Toc164250040)

[2.5.1 pg\_catalog模式 198](#_Toc164250041)

[2.5.2 information\_schema模式 200](#_Toc164250042)

[2.5.3 查询PostgreSQL数据库的信息（数据字典的使用） 201](#_Toc164250043)

[2.6 客户端连接PostgreSQL数据库服务器 201](#_Toc164250044)

[2.7 SQL语句的执行过程 203](#_Toc164250045)

[2.8 系统配置文件 204](#_Toc164250046)

[3 参考 205](#_Toc164250047)

[3.1 解决dnf或yum安装慢的问题 205](#_Toc164250048)

[3.2 The Internals of PostgreSQL 207](#_Toc164250049)

[3.3 the different memory components of a PostgreSQL instance 207](#_Toc164250050)

[3.4 PostgreSQL进程架构与内存管理 207](#_Toc164250051)

[3.5 PostgreSQL存储结构浅析 208](#_Toc164250052)

[3.6 PostgreSQL系统概述 208](#_Toc164250053)

[3.7 管理用户/角色和权限 210](#_Toc164250054)

# 普通表的存储结构

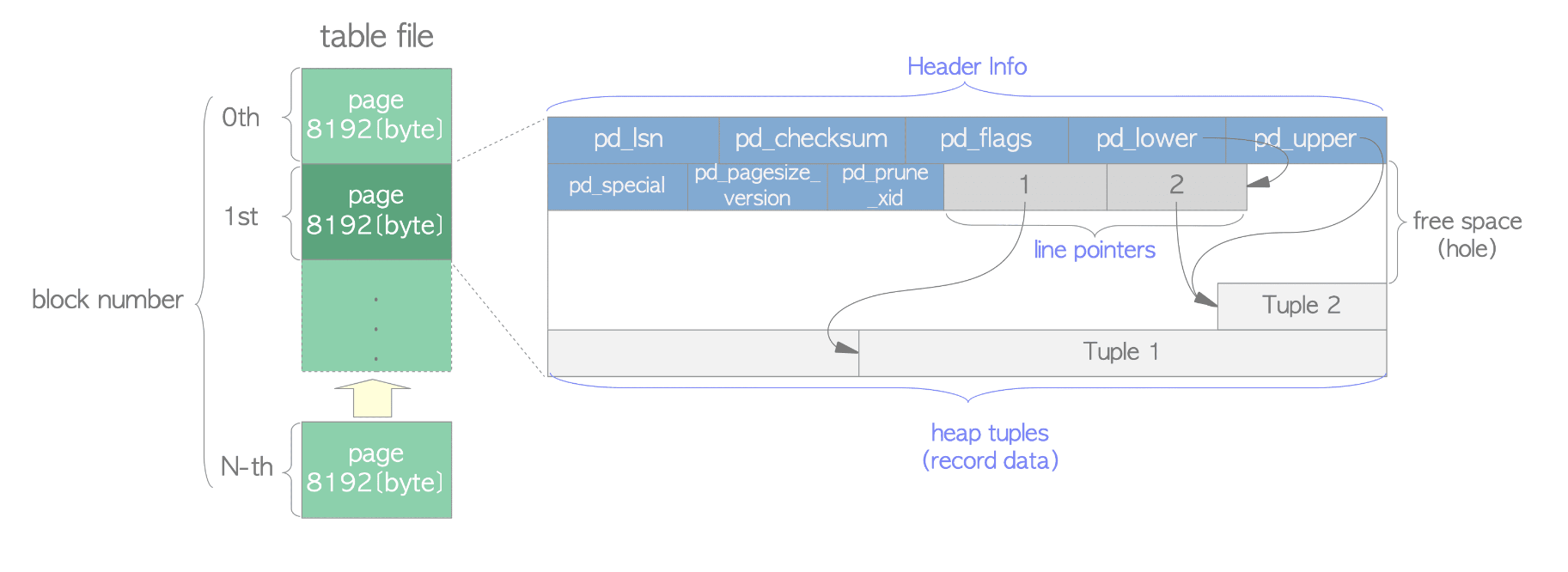
在PostgreSQL中，磁盘存储和内存中的最小管理单位都是块，保存在磁盘中的数据块称为页（Page），内存中的数据块称为缓冲区（**Buffer）**，表和索引称为关系（**Relation）**，行称为元组（**Tuple）**。

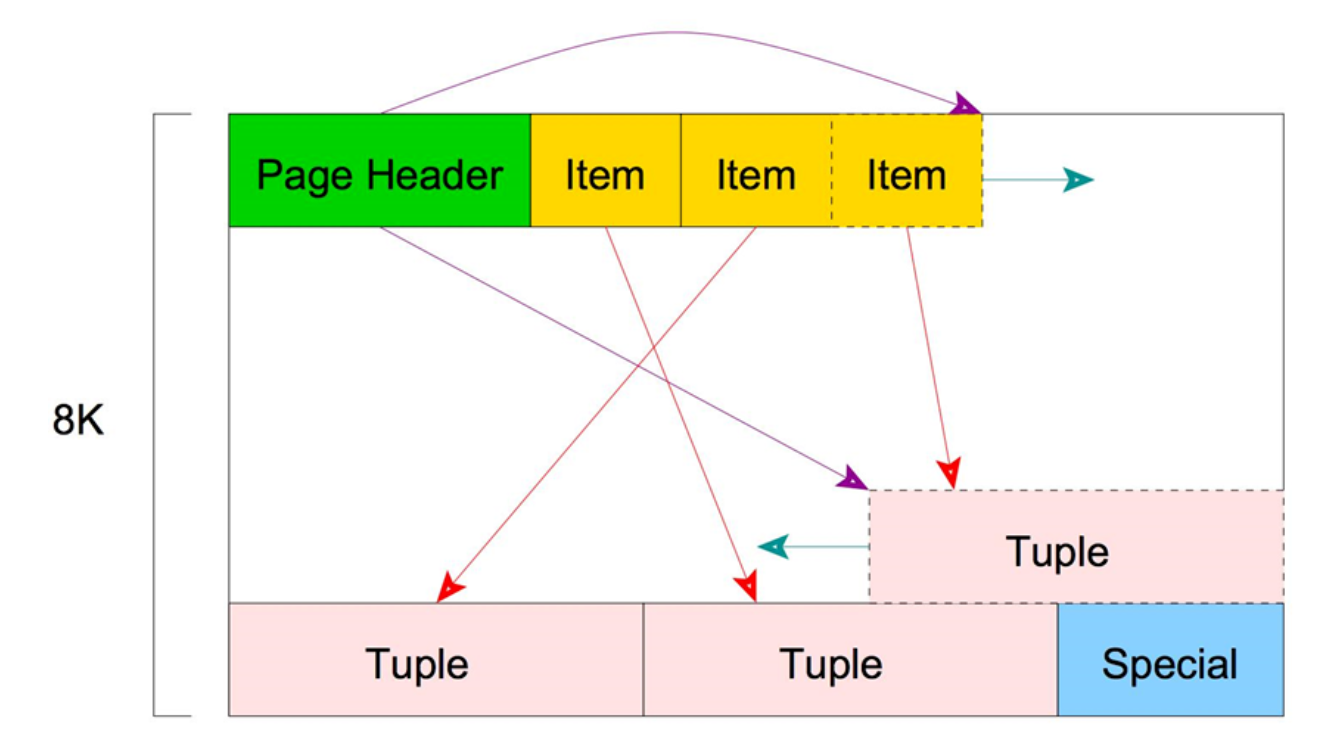
数据的读写是以页为最小单位，每个页默认大小为8kB。在源码编译时可通过./configure --with-blocksize=BLOCKSIZE设置其他大小，此后都不可更改。每个表文件由多个BLCKSZ字节大小的页组成，每个页包含若干个元组。

内存中的共享缓冲池缓存了block块（默认1000个），若缓冲池中的block块为脏，需要刷回磁盘。

### 页结构

页的结构如图所示。

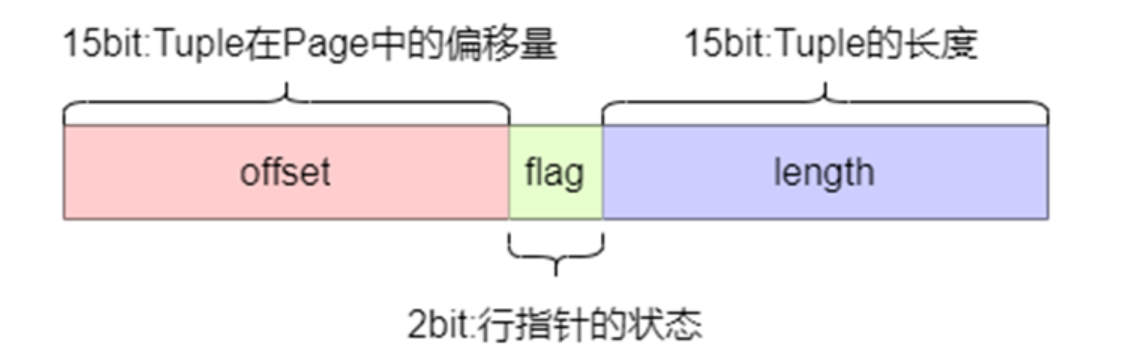




**堆表文件的页布局**

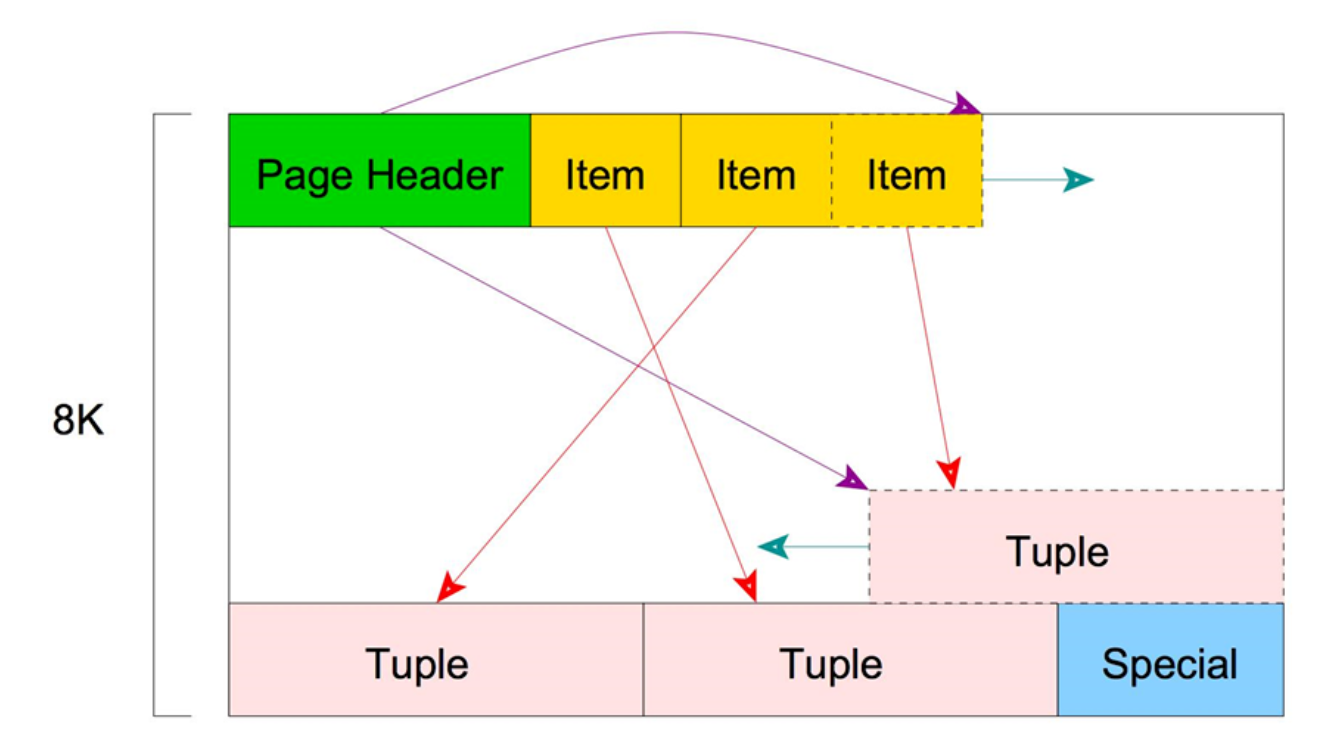
可以看到，从页的尾部往回看，有如下五个区域：

* **特殊空间（Special space）：**索引访问模式相关的数据。不同的索引访问方式存放不同的数据。在普通表中为空。
* **堆元组（Heap tuple）**：堆元组记录数据本身。堆元组按顺序从页面底部堆叠起来。
* **页的未分配的空间（Free space，空闲空间）：**行尾指针结束和最新元组开始之间的空白区域。
* **行指针（Line pointer）**：行指针长度为4字节，用于持有指向每个堆元组的指针。它也被称为**项目指针（Item Pointer）**。



行指针形成一个简单的数组，充当对元组的索引。每个索引从1开始顺序编号，称为偏移号。当页面上添加新的元组时，也会在数组中推入一个新的行指针以指向新元组。

行指针（Item）从这个区域的开头开始分配，新的行指针（Item）从其结尾开始分配。



* **头部数据（Header data）：**由结构PageHeaderData定义的头部数据分配在页面的开始处。它的长度为24字节，包含有关页面的一般信息：

typedef struct PageHeaderData @src/include/storage/bufpage.h

{

/\* XXX LSN is member of \*any\* block, not only page-organized ones \*/

PageXLogRecPtr pd\_lsn; /\* LSN: next byte after last byte of xlog

\* record for last change to this page \*/

uint16 pd\_checksum; /\* checksum \*/

uint16 pd\_flags; /\* flag bits, see below \*/

LocationIndex pd\_lower; /\* offset to start of free space \*/

LocationIndex pd\_upper; /\* offset to end of free space \*/

LocationIndex pd\_special; /\* offset to start of special space \*/

uint16 pd\_pagesize\_version;

TransactionId pd\_prune\_xid;/\* oldest prunable XID, or zero if none \*/

ItemIdData pd\_linp[1]; /\* beginning of line pointer array \*/

} PageHeaderData;

typedef PageHeaderData \*PageHeader;

typedef uint64 XLogRecPtr;

**其中：**

* pd\_lsn：此变量存储了此页面最后一次更改所写入的XLOG记录的LSN（日志序列号）。它是一个8字节的无符号整数，与WAL（预写日志）机制相关。
* pd\_checksum：存储页面的校验和值。（注意，这个变量在9.3及以后的版本中支持；在早期版本中，这部分存储的是页面的时间线ID。）
* pd\_flags：标志位
* pd\_lower：pd\_lower指向行指针的末端（空闲空间开头的偏移量）。
* pd\_upper：pd\_upper指向最新堆元组的开始（到空闲空间结尾的偏移量）。
* pd\_special：此变量用于索引。在表中的页面里，它指向页面的末端。（在索引中的页面里，它指向特殊空间的开始，这是只有索引持有的数据区域，包含根据索引类型（如B树、GiST、GiN等）的特定数据。
* pd\_pagesize\_version：页面大小和布局版本号信息
* pd\_prune\_xid：页面上最早删除元组的事务ID（XMAX），如果没有则为0。

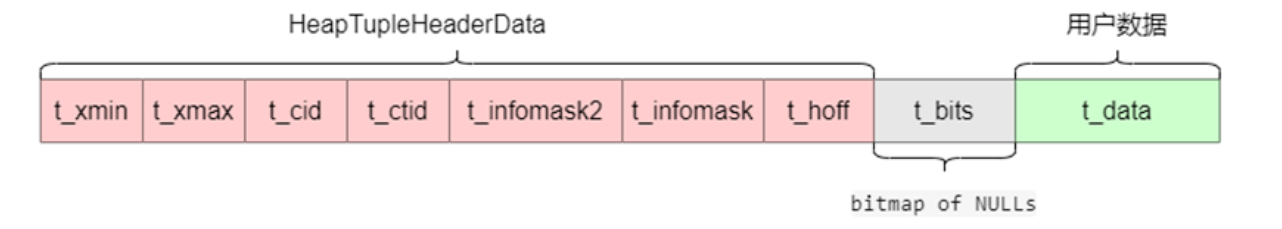
### 普通数据元组（Tuple）

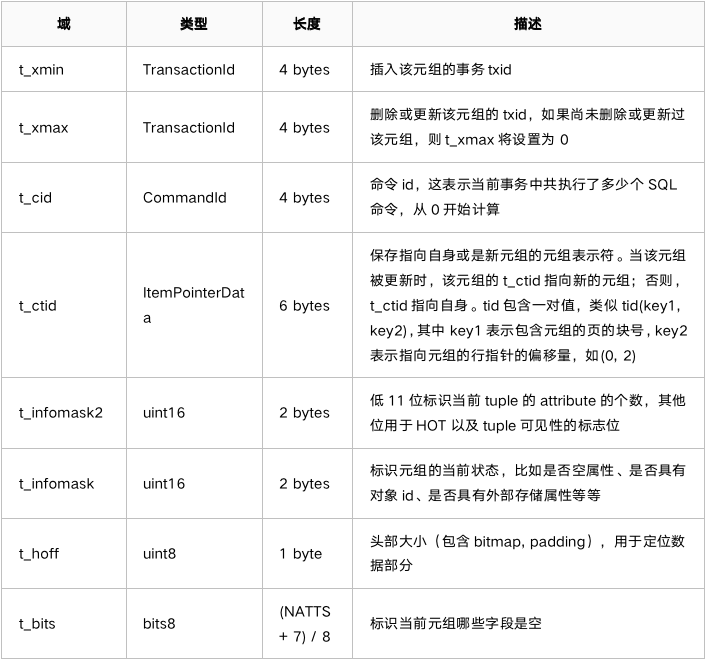
页中的元组可细分为 “普通数据元组和TOAST元组”。TOAST (The Oversized-Attribute Storage Technique，超大属性存储技术)元组主要用于存储变长数据，当要插入的元组数据大小大于约为2KB(即页的1/4)时候，会自动启动TOAST技术来存储该元组。

#### 普通数据元组

元组内部可以分为三部分，分别是：

* 堆元组头部（23字节）
* NULL值位图
* 用户存储的数据





#### 观察数据库页

**pageinspect**扩展模块提供的函数可以从低层次观察数据库页面的内容。执行下面的命令和SQL语句，安装**pageinspect**扩展。

[postgres@dbsvr ~]$ psql -d postgres -U postgres

psql (14.11)

Type "help" for help.

postgres=# CREATE EXTENSION pageinspect;

CREATE EXTENSION

postgres=#

创建测试表

postgres=# drop table if exists t1;

NOTICE: table "t1" does not exist, skipping

DROP TABLE

postgres=# create table t1 (id int, name varchar(20));

CREATE TABLE

postgres=# insert into t1 values(1,'aaa');

INSERT 0 1

postgres=#

查看 page header&item

postgres=# SELECT \* FROM page\_header(get\_raw\_page('t1', 0));

lsn | checksum | flags | lower | upper | special | pagesize | version | prune\_xid

-----------+----------+-------+-------+-------+---------+----------+---------+-----------

0/4061740 | 0 | 0 | 28 | 8160 | 8192 | 8192 | 4 | 0

(1 row)

postgres=#

postgres=# select \* from heap\_page\_items(get\_raw\_page('t1',0));

lp | lp\_off | lp\_flags | lp\_len | t\_xmin | t\_xmax | t\_field3 | t\_ctid | t\_infomask2 | t\_infomask | t\_hoff | t\_bits | t\_oid | t\_data

----+--------+----------+--------+--------+--------+----------+--------+-------------+------------+--------+--------+-------+--------------------

1 | 8160 | 1 | 32 | 852 | 0 | 0 | (0,1) | 2 | 2050 | 24 | | | \x0100000009616161

(1 row)

postgres=#

更新一行数据

postgres=# SELECT lp, lp\_off, lp\_flags, lp\_len,

postgres-# t\_xmin, t\_xmax, t\_ctid, t\_infomask2, t\_infomask

postgres-# FROM heap\_page\_items(get\_raw\_page('t1', 0));

lp | lp\_off | lp\_flags | lp\_len | t\_xmin | t\_xmax | t\_ctid | t\_infomask2 | t\_infomask

----+--------+----------+--------+--------+--------+--------+-------------+------------

1 | 8160 | 1 | 32 | 856 | 0 | (0,1) | 2 | 2050

(1 row)

postgres=#

postgres=# update t1 set name='bbb' where id=1;

UPDATE 1

postgres=#

postgres=# SELECT lp, lp\_off, lp\_flags, lp\_len,

postgres-# t\_xmin, t\_xmax, t\_ctid, t\_infomask2, t\_infomask

postgres-# FROM heap\_page\_items(get\_raw\_page('t1', 0));

lp | lp\_off | lp\_flags | lp\_len | t\_xmin | t\_xmax | t\_ctid | t\_infomask2 | t\_infomask

----+--------+----------+--------+--------+--------+--------+-------------+------------

1 | 8160 | 1 | 32 | 856 | 857 | (0,2) | 16386 | 258 （aaa）

2 | 8128 | 1 | 32 | 857 | 0 | (0,2) | 32770 | 10242 （bbb）

(2 rows)

postgres=#

查看 heap\_page\_items()，发现多了一条item，原因是PostgreSQL更新数据时并不修改原tuple，而是插入一条新的tuple，并标记原tuple.t\_xmax=新tuple的事务id。

# TOAST表

[PostgreSQL数据库大数据存储方式——TOAST机制上手\_postgresql toast-CSDN博客](https://feishujun.blog.csdn.net/article/details/120734511?spm=1001.2101.3001.6650.9&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=14)

<https://feishujun.blog.csdn.net/article/details/120734511?spm=1001.2101.3001.6650.9&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=14>

在PostgreSQL数据库中，总共有两种方式用于存储大(文本)数据对象，分别是：TOAST机制和大对象机制。

TOAST机制适用于那些可变长的数据类型，比如VARCHAR、TEXT、JSON和BYTEA等；大对象存储机制，适用于BLOB、CBLOG等数据类型。

[lightdb/postgresql toast解析&页面物理组织 - zhjh256 - 博客园 (cnblogs.com)](https://www.cnblogs.com/lightdb/p/15256306.html)

<https://www.cnblogs.com/lightdb/p/15256306.html>

[PostgreSQL中的Toast Pointer - 墨天轮 (modb.pro)](https://www.modb.pro/db/394974)

<https://www.modb.pro/db/394974>

[postgresql\_internals-14 学习笔记（四）TOAST 超尺寸字段存储技术\_postgresql的大字段存在哪了-CSDN博客](https://blog.csdn.net/Hehuyi_In/article/details/128357146?spm=1001.2101.3001.6650.8&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-128357146-blog-135987698.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-128357146-blog-135987698.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=15)

<https://blog.csdn.net/Hehuyi_In/article/details/128357146?spm=1001.2101.3001.6650.8&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-128357146-blog-135987698.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-128357146-blog-135987698.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=15>

[PostgreSQL数据库大数据存储方式——TOAST机制上手\_postgresql toast-CSDN博客](https://feishujun.blog.csdn.net/article/details/120734511?spm=1001.2101.3001.6650.9&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=14)

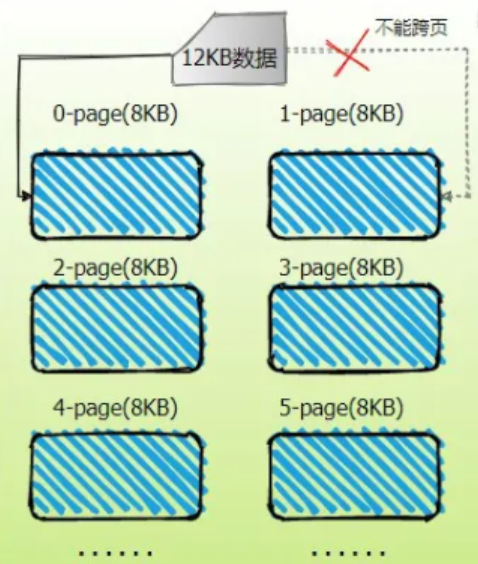
<https://feishujun.blog.csdn.net/article/details/120734511?spm=1001.2101.3001.6650.9&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-9-120734511-blog-120950944.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=14>

好好好！

### TOAST技术概述

TOAST是The Oversized-Attribute Storage Technique的缩写，直译为超大属性存储技术，不过翻译为行外存储技术更能体现其目的。

在PostgreSQL中，**表的一行（元组）不能跨页（page，一般大小为8KB）存储**。



如果要存储大字段的值，就只能使用TOAST存储。它和原表的数据是分开存储的。TOAST存储的表并不能单独创建，只有当普通表包含了main，extended或external存储格式的字段时，系统会自动创建一个和普通表关联的TOAST表。当一行记录（tuple）存储的（包括压缩后的大小）大小超过TOAST\_TUPLE\_THRESHOLD（默认2K）时，会存储到TOAST表。

### 检查是否需要使用TOAST技术存储表

源码相关：

检查是否需要TOAST存储的函数在src/backend/catalog/toasting.c

/\*

\* Check to see whether the table needs a TOAST table. It does only if

\* (1) there are any toastable attributes, and (2) the maximum length

\* of a tuple could exceed TOAST\_TUPLE\_THRESHOLD. (We don't want to

\* create a toast table for something like "f1 varchar(20)".)

\* No need to create a TOAST table for partitioned tables.

\*/

static bool

needs\_toast\_table(Relation rel)

{

int32 data\_length = 0;

bool maxlength\_unknown = false;

bool has\_toastable\_attrs = false;

TupleDesc tupdesc;

int32 tuple\_length;

int i;

声明了多个局部变量：

* data\_length用于累加属性的大小
* maxlength\_unknown是一个标志，表示是否有任何属性长度是未知的
* has\_toastable\_attrs用于检查是否有任何属性可能需要存储在TOAST表中
* 其他用于控制和描述。

分区表检查

if (rel->rd\_rel->relkind == RELKIND\_PARTITIONED\_TABLE)

return false;

检查关系是否为分区表。如果是，因为分区表不直接需要TOAST表，所以返回false。

元组描述符初始化

tupdesc = rel->rd\_att;

访问关系的元组描述符tupdesc，其中包含了表的每个属性（列）的元数据。

遍历属性

for (i = 0; i < tupdesc->natts; i++)

{

Form\_pg\_attribute att = TupleDescAttr(tupdesc, i);

循环遍历表中的所有属性。TupleDescAttr可能是一个宏或函数，用于在索引i处从元组描述符中检索属性元数据。

跳过已删除的属性

if (att->attisdropped)

continue;

如果属性标记为已删除（不再使用），则跳过此次循环迭代。

属性对齐

data\_length = att\_align\_nominal(data\_length, att->attalign);

根据属性的对齐要求对当前数据长度进行对齐。

处理属性长度

if (att->attlen > 0)

{

/\* Fixed-length types are never toastable \*/

data\_length += att->attlen;

}

else

{

int32 maxlen = type\_maximum\_size(att->atttypid,

att->atttypmod);

检查属性长度是否为固定（attlen > 0）。如果是，直接加到data\_length上。

如果不是，计算属性的最大大小。

最大长度确定

if (maxlen < 0)

maxlength\_unknown = true;

else

data\_length += maxlen;

if (att->attstorage != 'p')

has\_toastable\_attrs = true;

}

}

如果maxlen为负值，则将maxlength\_unknown设置为true，表示长度不受限制。

如果已知，将最大长度加到data\_length上。

如果属性的存储类型不是普通（'p'），则标记存在可TOAST属性。

循环后检查

if (!has\_toastable\_attrs)

return false; /\* nothing to toast? \*/

if (maxlength\_unknown)

return true; /\* any unlimited-length attrs? \*/

循环结束后，检查是否没有可TOAST的属性或者是否存在无限长度的属性。

计算元组长度和最终检查

tuple\_length = MAXALIGN(SizeofHeapTupleHeader +

BITMAPLEN(tupdesc->natts)) +

MAXALIGN(data\_length);

return (tuple\_length > TOAST\_TUPLE\_THRESHOLD);

}

计算总元组长度，考虑到头部大小和位图长度，然后检查这个长度是否超过了预定义的阈值（TOAST\_TUPLE\_THRESHOLD），这将决定是否需要一个TOAST表。

该函数仔细评估每个属性的特性，以决定是否需要TOAST表的开销，优化了PostgreSQL数据库的存储和性能。

### 定义TOAST\_TUPLE\_THRESHOLD的值

TOAST\_TUPLE\_THRESHOLD定义在src/include/access/tuptoaster.h

#define TOAST\_TUPLES\_PER\_PAGE 4

#define TOAST\_TUPLE\_THRESHOLD MaximumBytesPerTuple(TOAST\_TUPLES\_PER\_PAGE)

#define TOAST\_TUPLE\_TARGET TOAST\_TUPLE\_THRESHOLD

#define

MaximumBytesPerTuple(tuplesPerPage) \

MAXALIGN\_DOWN((BLCKSZ - \

MAXALIGN(SizeOfPageHeaderData + (tuplesPerPage) \* sizeof(ItemIdData))) \

/ (tuplesPerPage))

#当BLCKSZ为8KB

MAXALIGN\_DOWN((8192 - MAXALIGN(192 + (4) \* 32)) / (4)) ~=2KB

#当数据库修改BLCKSZ为32KB

MAXALIGN\_DOWN((32768 - MAXALIGN(192 + (4) \* 32)) / (4)) ~= 8KB

### 存储类型（策略）

对应存储类型的缩写定义在src/backend/commands/tablecmds.c

/\*

\* storage\_name

\* returns the name corresponding to a typstorage/attstorage enum value

\*/

static const char \*

storage\_name(char c)

{

switch (c)

{

case 'p':

return "PLAIN";

case 'm':

return "MAIN";

case 'x':

return "EXTENDED";

case 'e':

return "EXTERNAL";

default:

return "???";

}

}

--可以通过 select typname,typstorage from pg\_type查看

存储策略如下：

* PLAIN：不允许压缩和线外存储。对于那些不能TOAST机制的数据类型，默认都是该策略，比如整数类型(INT，SMALLINT，BIGINT)、字符类型(CHAR)、布尔类型(BOOLEAN)等等。
* EXTENDED：允许线外存储和压缩。这是大多数可以使用TOAST机制的数据类型默认存储策略。它首先尝试压缩，如果这仍不能将数据放入页中，则使用线外存储。
* EXTERNAL：允许线外存储，但不允许压缩。该存储策略将使TEXT、BYTEA数据类型的列中字符串操作更快，其代价是牺牲存储空间。
* MAIN：允许压缩，但不能线外存储。然而在无法使列足够小以存入页时，它还是执行线外存储。所以它仍然是可以行外存储的，和EXTERNAL具有相似性。

存储策略PLAIN、EXTENDED、EXTERNAL和MAIN在PostgreSQL源码中分别被简写为p、x、e和m。通过storage\_name函数，分别获取个存储策略所对应的名字。

### TOAST表示例

create table tbl\_test\_toast (id int primary key,name varchar(50),info text,age int);

postgres=# create table tbl\_test\_toast(id int primary key,name varchar(50),

postgres=# info text,age int);

CREATE TABLE

postgres=# \d+ tbl\_test\_toast

Table "public.tbl\_test\_toast"

Column | Type | Collation | Nullable | Default | Storage | Compression | Stats target | Description

--------+-----------------------+-----------+----------+---------+----------+-------------+--------------+-------------

id | integer | | not null | | plain | | |

name | character varying(50) | | | | extended | | |

info | text | | | | extended | | |

age | integer | | | | plain | | |

Indexes:

"tbl\_test\_toast\_pkey" PRIMARY KEY, btree (id)

Access method: heap

postgres=#

#修改字段类型的存储策略

alter table tablename alter column {$column\_name} set storage { PLAIN | MAIN | EXTERNAL | EXTENDED } ;

插入一行数据，查看是否使用TOAST存储

insert into tbl\_test\_toast values (1,'hank','51.2. Index Access Method FunctionsThe index construction and maintenance functions that an index access method must provide are:IndexBuildResult \*ambuild (Relation heapRelation, Relation indexRelation, IndexInfo \*indexInfo);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 IndexBuildHeapScan() to scan the table for existing tuples and compute the keys that need to be inserted into the index. The function must return a pallocd struct containing statistics about the new index.boolaminsert (Relation indexRelation, Datum \*values, bool \*isnull, ItemPointer heap\_tid, Relation heapRelation, IndexUniqueCheck checkUnique);Insert a new tuple into an existing index. The values and isnull arrays give the key values to be indexed, and heap\_tid is the TID to be indexed. If the access method supports unique indexes (its pg\_am.amcanunique flag is true) then checkUnique indicates the type of uniqueness check to perform. This varies depending on whether the unique constraint is deferrable; see Section 51.5 for details. Normally the access method only needs the heapRelation parameter when performing uniqueness checking (since then it will have to look into the heap to verify tuple liveness).The functions Boolean result value is significant only when checkUnique is UNIQUE\_CHECK\_PARTIAL. In this case a TRUE result means the new entry is known unique, whereas FALSE means it might be non-unique (and a deferred uniqueness check must be scheduled). For other cases a constant FALSE result is recommended.Some indexes might not index all tuples. If the tuple is not to be indexed, aminsert should just return without doing anything.IndexBulkDeleteResult \*ambulkdelete (IndexVacuumInfo \*info, IndexBulkDeleteResult \*stats, IndexBulkDeleteCallback callback, void \*callback\_state);Delete tuple(s) from the index. This is a "bulk delete" operation that is intended to be implemented by scanning the whole index and checking each entry to see if it should be deleted. The passed-in callback function must be called, in the style callback(TID, callback\_state) returns bool, to determine whether any particular index entry, as identified by its referenced TID, is to be deleted. Must return either NULL or a pallocd struct containing statistics about the effects of the deletion operation. It is OK to return NULL if no information needs to be passed on to amvacuumcleanup.Because of limited maintenance\_work\_mem, ambulkdelete might need to be called more than once when many tuples are to be deleted. The stats argument is the result of the previous call for this index (it is NULL for the first call within a VACUUM operation). This allows the AM to accumulate statistics across the whole operation. Typically, ambulkdelete will modify and return the same struct if the passed stats is not null.IndexBulkDeleteResult \*amvacuumcleanup (IndexVacuumInfo \*info, IndexBulkDeleteResult \*stats);Clean up after a VACUUM operation (zero or more ambulkdelete calls). This does not have to do anything beyond returning index statistics, but it might perform bulk cleanup such as reclaiming empty index pages. stats is whatever the last ambulkdelete call returned, or NULL if ambulkdelete was not called because no tuples needed to be deleted. If the result is not NULL it must be a pallocd struct. The statistics it contains will be used to update pg\_class, and will be reported by VACUUM if VERBOSE is given. It is OK to return NULL if the index was not changed at all during the VACUUM operation, but otherwise correct stats should be returned.As of PostgreSQL 8.4, amvacuumcleanup will also be called at completion of an ANALYZE operation. In this case stats is always NULL and any return value will be ignored. This case can be distinguished by checking info->analyze\_only. It is recommended that the access method do nothing except post-insert cleanup in such a call, and that only in an autovacuum worker process.voidamcostestimate (PlannerInfo \*root, IndexOptInfo \*index, List \*indexQuals, RelOptInfo \*outer\_rel, Cost \*indexStartupCost, Cost \*indexTotalCost, Selectivity \*indexSelectivity, double \*indexCorrelation);Estimate the costs of an index scan. This function is described fully in Section 51.6, below.bytea \*amoptions (ArrayType \*reloptions, bool validate);',30);

postgres=# SELECT relname FROM pg\_class WHERE oid = (SELECT reltoastrelid FROM pg\_class WHERE relname = 'tbl\_test\_toast');

relname

----------------

pg\_toast\_16698

(1 row)

postgres=#

postgres=# select oid,relname,reltoastrelid from pg\_class where relname='tbl\_test\_toast';

oid | relname | reltoastrelid

-------+----------------+---------------

16698 | tbl\_test\_toast | 16701

(1 row)

postgres=#

postgres=# select pg\_column\_size(info) from tbl\_test\_toast;

pg\_column\_size

----------------

2485

(1 row)

postgres=#

#注意这里是存储策略是EXTENDED，所以会压缩，压缩后，如果大小还是超过2KB，才会存储到toast,所以做实验的时候可以修改存储策略为EXTERNAL，这样可以更容易的达到实验目的，如修改为EXTERNAL，直接插入repeat('a', 2100)大小的值，就会启用TOAST，因为2104已经大于TOAST\_TUPLE\_THRESHOLD(2KB)。

postgres=# select pg\_column\_size(repeat('a', 2100));

pg\_column\_size

----------------

2104

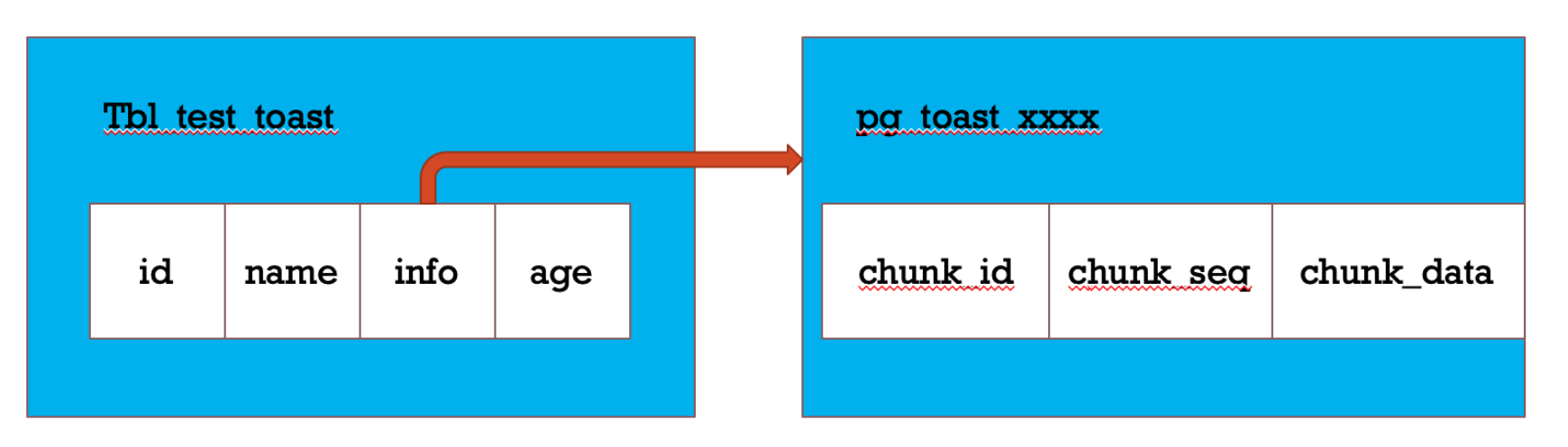
(1 row)

postgres=#

TOAST表的命名规则

对于TOAST表的命名，规则是：pg\_toast\_$(oid)。其中oid是该TOAST表所属表的oid值，比如数据表tbl\_test\_toast在系统表pg\_class中的oid是16698，所以与之关联的TOAST的表名字是pg\_toast\_16698。

如下图示：info字段使用TOAST存储



postgres=# \d pg\_toast.pg\_toast\_16698

TOAST table "pg\_toast.pg\_toast\_16698"

Column | Type

------------+---------

chunk\_id | oid

chunk\_seq | integer

chunk\_data | bytea

Owning table: "public.tbl\_test\_toast"

Indexes:

"pg\_toast\_16698\_index" PRIMARY KEY, btree (chunk\_id, chunk\_seq)

postgres=#

chunk\_id: 该toast表所分配的oid

chunk\_seq: chunk的序列号

chunk\_data: chunk中的数据

postgres=# SELECT chunk\_id, COUNT(\*) as chunks, pg\_size\_pretty(sum(octet\_length(chunk\_data)::bigint))FROM pg\_toast.pg\_toast\_16698 GROUP BY 1 ORDER BY 1;

chunk\_id | chunks | pg\_size\_pretty

----------+--------+----------------

16705 | 2 | 2485 bytes

(1 row)

postgres=#

#因为这个值太大了，不能放在一行中，所以PostgreSQL把它分成了2个块存放。

**使用TOAST的好处如下：**

* UPDATE一个普通表时，当该表的TOAST表存储的列没有修改时，TOAST表不需要更新，这样更新效率得到提升
* 由于TOAST在物理存储上和普通表分开，所以当SELECT时没有查询被TOAST的列数据时，不需要把这些TOAST的PAGE加载到内存，从而加快了检索速度并且节约了使用空间。
* 在排序时，由于TOAST和普通表存储分开，当针对非TOAST字段排序时大大提高了排序速度。

**使用TOAST存储格式注意事项：**

* 当变长字段上需要使用索引时，权衡CPU和存储的开销，考虑是否需要压缩或非压缩存储。（压缩节约磁盘空间，但是带来CPU的开销）
* 对于经常要查询或UPDATE的变长字段，如果字段长度不是太大，可以考虑使用MAIN存储。
* 在超长字段，或者将来会插入超长值的字段上建索引的话需要注意，因为索引最大不能超过三分之一的PAGE，所以超长字段上可能建索引不成功，或者有索引的情况下，超长字段插入值将不成功。解决办法一般可以使用MD5值来建，当然看你的需求了。

### toast表使用示例

执行下面的命令，恢复有大学应用测试数据集的PostgreSQL数据库集簇：

[postgres@dbsvr ~]$ cd /opt/db

[postgres@dbsvr db]$ pg\_ctl stop

[postgres@dbsvr db]$ rm -rf userdb

[postgres@dbsvr db]$ tar xf userdbWithData.tar

[postgres@dbsvr db]$ pg\_ctl start

以Linux用户postgres的身份，执行下面的命令，安装了psycopg2：

pip install psycopg2

pip install psycopg2-binary

[postgres@dbsvr db]$ psql -U postgres -d universitydb

universitydb=# CREATE TABLE songs (

universitydb(# id SERIAL PRIMARY KEY,

universitydb(# file\_name VARCHAR(255),

universitydb(# song\_title VARCHAR(255),

universitydb(# artist VARCHAR(255),

universitydb(# song\_file BYTEA

universitydb(# );

CREATE TABLE

universitydb=#

[postgres@dbsvr ~]$ vi toastexamp1.py

# toastexamp1.py:将mp3歌曲导入到数据库

import psycopg2

from psycopg2 import extras

def insert\_song(file\_path, file\_name, song\_title, artist):

""" 插入歌曲信息和文件到数据库 """

conn = None

cur = None

try:

# 尝试连接到数据库

conn = psycopg2.connect("dbname=universitydb user=postgres password=postgres123 host='127.0.0.1' port='5432'")

cur = conn.cursor()

# 读取文件内容

with open(file\_path, 'rb') as file:

song\_file = file.read()

# 构建插入SQL命令

sql = """

INSERT INTO songs (file\_name, song\_title, artist, song\_file)

VALUES (%s, %s, %s, %s)

"""

# 执行SQL命令

cur.execute(sql, (file\_name, song\_title, artist, psycopg2.Binary(song\_file)))

# 提交事务

conn.commit()

print("歌曲信息已成功插入数据库。")

except Exception as e:

print("发生错误：", e)

finally:

# 无论成功还是失败，关闭游标和连接

if cur is not None:

cur.close()

if conn is not None:

conn.close()

# 示例调用

file\_path = '/home/postgres/mysong1.mp3' # 确保更改为正确的文件路径

file\_name = 'mysong1.mp3'

song\_title = '笑脸'

artist = '谢东'

insert\_song(file\_path, file\_name, song\_title, artist)

[postgres@dbsvr ~]$ vi toastexamp2.py

# toastexamp2.py:从数据库导出mp3歌曲

import psycopg2

def export\_song(file\_name, output\_path):

""" 从数据库中导出歌曲到指定文件路径 """

try:

# 连接到数据库

conn = psycopg2.connect("dbname=universitydb user=postgres password=postgres123 host='127.0.0.1' port='5432'")

cur = conn.cursor()

# 查询数据库以获取歌曲文件

cur.execute("SELECT song\_file FROM songs WHERE file\_name = %s", (file\_name,))

song\_file = cur.fetchone()

if song\_file:

# 将二进制数据写入文件

with open(output\_path, 'wb') as file:

file.write(song\_file[0])

print("歌曲已成功导出到：", output\_path)

else:

print("未找到指定的歌曲。")

except Exception as e:

print("导出过程中发生错误：", e)

finally:

# 关闭游标和连接

if cur:

cur.close()

if conn:

conn.close()

# 示例调用

file\_name = 'mysong1.mp3' # 数据库中存储的文件名

output\_path = '/home/postgres/mysong2.mp3' # 输出文件的路径

export\_song(file\_name, output\_path)

#### 参考：

https://hakibenita.com/sql-medium-text-performance

https://github.com/digoal/blog/blob/master/201103/20110329\_01.md

[干货！PostgreSQL 从小白到专家 - 第24讲：TOAST技术\_Toast\_存储\_字段 (sohu.com)](https://www.sohu.com/a/706684816_100057833)

<https://www.sohu.com/a/706684816_100057833>

https://www.postgresql.org/docs/13/storage-toast.html

3.2. TOAST

73.2.1. 线下、磁盘上的TOAST存储

73.2.2. 线下、内存中的TOAST存储

本节提供了TOAST（过大属性存储技术）的概述。

PostgreSQL使用固定的页面大小（通常为8kB），并且不允许元组跨多个页面。因此，无法直接存储非常大的字段值。为了克服这一限制，大字段值会被压缩和/或分解为多个物理行。这对用户来说是透明的，对大部分后端代码的影响很小。这种技术被亲切地称为TOAST（或“自切面包以来的最佳事物”）。TOAST基础设施也用于改善内存中大数据值的处理。

只有某些数据类型支持TOAST —— 对于无法产生大字段值的数据类型，没有必要施加开销。为了支持TOAST，数据类型必须有一个可变长度（varlena）表示，在这种表示中，通常存储的任何值的第一个四字节字包含值的总长度（包括它自身）。TOAST不限制数据类型的其余表示。统称为TOASTed值的特殊表示通过修改或重新解释这个初始长度字来工作。因此，支持TOASTable数据类型的C级函数必须小心处理可能的TOASTed输入值：输入可能实际上不包括四字节长度字和内容，直到它被detoast之后。（这通常通过调用PG\_DETOAST\_DATUM来完成，在处理输入值之前，但在某些情况下，可能有更高效的方法。更多细节见第38.13.1节。）

TOAST占用了varlena长度字的两个位（在大端机器上是高阶位，在小端机器上是低阶位），因此限制了任何TOASTable数据类型的值的逻辑大小为1 GB（2^30 - 1字节）。当两个位都是零时，值是数据类型的普通未TOASTed值，长度字的其余位给出了总数据大小（包括长度字）的字节数。当最高阶或最低阶位被设定时，值只有一个单字节头，而不是正常的四字节头，那个字节的其余位给出了总数据大小（包括长度字节）的字节数。这种替代支持值小于127字节时的空间效率存储，同时仍然允许数据类型在需要时增长到1 GB。具有单字节头的值没有在任何特定边界上对齐，而具有四字节头的值至少在四字节边界上对齐；这种对齐填充的省略为与短值相比提供了额外的空间节省，这是显著的。作为特殊情况，如果单字节头的其余位都是零（对于包含自身长度的值来说是不可能的），则值是指向线下数据的指针，具体有几种可能的替代方式，如下所述。这种TOAST指针的类型和大小由数据的第二个字节中存储的代码决定。最后，当最高阶或最低阶位清除但相邻位设定时，数据内容已被压缩，使用前必须解压缩。在这种情况下，四字节长度字的其余位给出了压缩数据的总大小，而不是原始数据。需要注意的是，线下数据也可以压缩，但varlena头不会显示是否发生了压缩 —— 相反，TOAST指针的内容会告诉这一点。

对于内联或线下压缩数据使用的压缩技术可以通过在CREATE TABLE或ALTER TABLE中设置COMPRESSION列选项为每个列单独选择。没有明确设置的列的默认设置是在数据插入时查询default\_toast\_compression参数。

如前所述，有多种类型的TOAST指针数据。最早和最常见的类型是指向与包含TOAST指针数据的表分开但关联的TOAST表中存储的线下数据的指针。这些磁盘上的指针数据由TOAST管理代码（在access/common/toast\_internals.c中）创建，当要存储在磁盘上的元组太大而无法按原样存储时。更多细节见第73.2.1节。另一种选择，TOAST指针数据可以包含指向内存中其他位置的线下数据的指针。这样的数据显然是短暂的，永远不会出现在磁盘上，但它们对于避免复制和重复处理大数据值非常有用。更多细节见第73.2.2节。

73.2.1. 线下、磁盘上的TOAST存储

如果表的任何列是TOASTable的，该表将有一个关联的TOAST表，其OID存储在表的pg\_class.reltoastrelid条目中。磁盘上的TOASTed值保存在TOAST表中，如下文更详细描述。

线下值在压缩（如果使用）之后被分解为最多TOAST\_MAX\_CHUNK\_SIZE字节的块（默认情况下，这个值是选择的，以便四个块行可以适合一页，使其约为2000字节）。每个块作为属于拥有表的TOAST表中的单独一行存储。每个TOAST表都有chunk\_id（标识特定TOASTed值的OID）、chunk\_seq（块在其值中的序列号）和chunk\_data（块的实际数据）列。chunk\_id和chunk\_seq上的唯一索引提供了值的快速检索。因此，代表线下磁盘上TOASTed值的指针数据需要存储要查找的TOAST表的OID和特定值的OID（其chunk\_id）。为方便起见，指针数据还存储逻辑数据大小（原始未压缩数据长度）、物理存储大小（如果应用了压缩则不同）和使用的压缩方法（如果有的话）。允许varlena头字节，一个磁盘上TOAST指针数据的总大小因此是18字节，无论代表值的实际大小如何。

只有在表中要存储的行值宽于TOAST\_TUPLE\_THRESHOLD字节（通常是2kB）时才会触发TOAST管理代码。TOAST代码将压缩和/或将字段值移到线下，直到行值短于TOAST\_TUPLE\_TARGET字节（也通常是2kB，可调）或无法获得更多收益为止。在UPDATE操作期间，未更改字段的值通常会保留原样；因此，如果没有线下值更改，更新具有线下值的行不会产生TOAST成本。

TOAST管理代码识别四种不同的策略来在磁盘上存储TOASTable列：

PLAIN阻止压缩或线下存储。这是非TOASTable数据类型列的唯一可能策略。

EXTENDED允许压缩和线下存储。这是大多数TOASTable数据类型的默认设置。首先尝试压缩，然后如果行仍然太大，则进行线下存储。

EXTERNAL允许线下存储但不允许压缩。使用EXTERNAL将使宽文本和bytea列的子字符串操作更快（以增加存储空间为代价），因为这些操作被优化为仅在值未压缩时获取所需的线下值部分。

MAIN允许压缩但不允许线下存储。（实际上，这样的列仍会进行线下存储，但只作为在没有其他方法可以使行足够小以适合页面时的最后手段。）

每个TOASTable数据类型指定该数据类型的列的默认策略，但可以通过ALTER TABLE ... SET STORAGE更改给定表列的策略。

可以使用ALTER TABLE ... SET (toast\_tuple\_target = N)调整TOAST\_TUPLE\_TARGET以适用于每个表。

这种方案与允许行值跨页的更直接方法相比具有许多优势。假设查询通常通过与相对较小的键值进行比较来进行资格认定，大部分执行者的工作将使用主行条目来完成。只有在发送结果集给客户端时才会拉出（如果根本选择）TOASTed属性的大值。因此，主表更小，其更多行可以适合共享缓冲区缓存，这是没有任何线下存储情况下不可能的。排序集也会缩小，排序将更频繁地完全在内存中完成。一项小测试显示，一个包含典型HTML页面及其URLs的表存储的数据量大约是原始数据大小的一半，包括TOAST表，而主表只包含整个数据的大约10%（URLs和一些小HTML页面）。与未经TOAST的对比表相比，运行时间没有差异，在那个表中，所有HTML页面都被切割到7kB以适应。

73.2.2. 线下、内存中的TOAST存储

TOAST指针可以指向不在磁盘上而是在当前服务器进程的内存中其他位置的数据。这样的指针显然不能长期存在，但它们仍然很有用。当前有两个子情况：指向间接数据和指向展开数据的指针。

间接TOAST指针简单地指向存储在内存中某处的非间接varlena值。这种情况最初只是作为一个概念验证而创建的，但当前在逻辑解码期间使用，以避免可能必须创建超过1GB的物理元组（因为可能需要将所有线下字段值拉入元组）。这种情况的用途有限，因为创建指针数据的人完全负责引用的数据在指针可能存在的整个时间内存活下来，没有基础设施来帮助这一点。

展开的TOAST指针对于其磁盘表示不特别适合计算目的的复杂数据类型很有用。例如，PostgreSQL数组的标准varlena表示包括维度信息、如果有任何null元素则为nulls位图，然后按顺序是所有元素的值。当元素类型本身是可变长度时，找到第N个元素的唯一方法是扫描所有前面的元素。这种表示适合磁盘存储因为其紧凑性，但对于使用数组进行计算来说，拥有一个“展开”或“解构”的表示更好，在这种表示中，所有元素的起始位置已被识别。TOAST指针机制通过允许按引用传递的Datum指向标准varlena值（磁盘表示）或指向某处内存中的展开表示的TOAST指针来支持这种需求。这种展开表示的细节取决于数据类型，尽管它必须具有标准头部并满足src/include/utils/expandeddatum.h中给出的其他API要求。处理数据类型的C级函数可以选择处理任何一种表示。不了解展开表示的函数，但只是简单地将PG\_DETOAST\_DATUM应用于其输入的，将自动接收传统的varlena表示；因此，对展开表示的支持可以逐渐引入，一次一个函数。

展开值的TOAST指针进一步细分为读写和只读指针。指向的表示是相同的，但接收读写指针的函数被允许在原地修改引用的值，而接收只读指针的函数则不得；如果它想制作值的修改版本，则必须先创建副本。这种区分和一些相关的约定使在查询执行期间避免不必要的展开值复制成为可能。

对于所有类型的内存中TOAST指针，TOAST管理代码确保没有这样的指针数据意外地存储在磁盘上。内存中的TOAST指针在存储之前会自动展开为正常的内联varlena值 — 然后可能转换为磁盘上的TOAST指针，如果包含的元组否则将过大。

# 大对象存储机制

[PostgreSQL的大对象以及空间使用 (1) - abce - 博客园 (cnblogs.com)](https://www.cnblogs.com/abclife/p/13790262.html)

<https://www.cnblogs.com/abclife/p/13790262.html>

<https://www.cnblogs.com/abclife/p/13790588.html>

<https://www.cnblogs.com/abclife/p/13791355.html>

原文：

<https://www.ludovicocaldara.net/dba/pgsql-lo-space-usage-part-1/>

<https://www.ludovicocaldara.net/dba/pgsql-lo-space-usage-part-2/>

<https://www.ludovicocaldara.net/dba/pgsql-lo-space-usage-part-3/>

在PostgreSQL数据库中，总共有两种方式用于存储大(文本)数据对象，分别是：TOAST机制和大对象机制。大对象存储机制，适用于BLOB、CBLOG等数据类型。

执行下面的命令，恢复有大学应用测试数据集的PostgreSQL数据库集簇：

[postgres@dbsvr ~]$ cd /opt/db

[postgres@dbsvr db]$ pg\_ctl stop

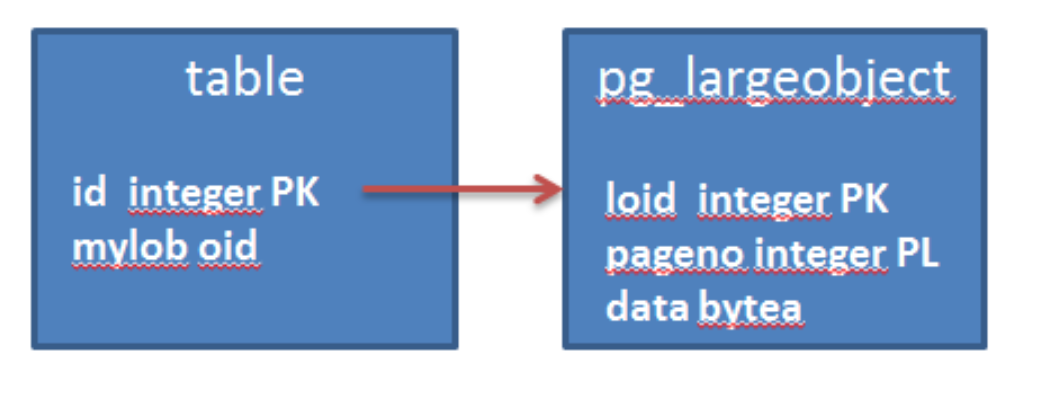
[postgres@dbsvr db]$ rm -rf userdb

[postgres@dbsvr db]$ tar xf userdbWithData.tar

[postgres@dbsvr db]$ pg\_ctl start

### PostgreSQL传统的大对象

在应用表中，大对象的列被定义为指向pg\_largeobject表内数据块(chunks)的oid。



因为大对象是独立于引用它的表列创建的，所以当你从表中删除指向大对象的行时，大对象本身不会被删除。

此外，pg\_largeobject被设计用于存储数据库中存在的所有大对象。这使得该表的管理维护对于数据库管理至关重要。

### 传统大对象的存储空间组织

大对象是如何组织空间的？（从pg\_largeobject为空的一个数据库开始）

[postgres@dbsvr db]$ psql -d postgres -U postgres

postgres=# select count(\*) from pg\_largeobject;

count

-------

0

(1 row)

postgres=# vacuum full pg\_largeobject;

VACUUM

postgres=# select pg\_total\_relation\_size('pg\_largeobject');

pg\_total\_relation\_size

------------------------

8192

(1 row)

postgres=#

可以看到，当前pg\_largeobject表只有一个块（8192字节=8KB）。

执行下面的SQL查询，查看磁盘上pg\_largeobject表对应的数据文件：

postgres=# SELECT pg\_relation\_filepath('pg\_largeobject');

pg\_relation\_filepath

----------------------

base/13052/16505

(1 row)

postgres=# \! ls -l /opt/db/userdb/u00/pgdata/base/13052/16505

-rw-------. 1 postgres dba 0 Apr 19 09:42 /opt/db/userdb/u00/pgdata/base/13052/16505

postgres=#

可以看到，此时文件是空的。这意味着在表中有一些数据之前不会物理地创建第一个block。

现在，让我们为我们的测试创建两个大小为1MB的文件，一个用零填充，另一个随机填充：

postgres=# \! dd if=/dev/zero of=/tmp/zeroes bs=1024 count=1024

1024+0 records in

1024+0 records out

1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00248128 s, 423 MB/s

postgres=# \! dd if=/dev/urandom of=/tmp/randoms bs=1024 count=1024

1024+0 records in

1024+0 records out

1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00513781 s, 204 MB/s

postgres=# \! ls -l /tmp/zeroes /tmp/randoms

-rw-r--r--. 1 postgres dba 1048576 Apr 19 09:49 /tmp/randoms

-rw-r--r--. 1 postgres dba 1048576 Apr 19 09:49 /tmp/zeroes

postgres=#

执行下面的操作，导入用0填充的文件：

postgres=# \lo\_import '/tmp/zeroes';

lo\_import 16509

postgres=# select count(\*) from pg\_largeobject\_metadata;

count

-------

1

(1 row)

postgres=# select count(\*) from pg\_largeobject;

count

-------

512

(1 row)

postgres=#

可以看到，大对象被切分成大小为每个2048bytes的chunk，因此一共有512个（总大小/2KB=1MB/2KB=1024KB/2KB=512）。

执行下面的命令，查看pg\_largeobject表的物理大小：

postgres=# \! ls -lh /opt/db/userdb/u00/pgdata/base/13052/16505

-rw-------. 1 postgres dba 40K Apr 19 11:13 /opt/db/userdb/u00/pgdata/base/13052/16505

postgres=#

可以看到，物理大小只有40K。这就意味着chunk被压缩了。PostgreSQL使用了pglz\_compress函数，其算法在源代码src/common/pg\_lzcompress.c中。

下面，我们来测试导入随机填充的文件的情况：

postgres=# \lo\_import '/tmp/randoms';

lo\_import 16510

postgres=# select count(\*) from pg\_largeobject where loid=16510;

count

-------

512

(1 row)

postgres=# select pg\_relation\_size('pg\_largeobject');

pg\_relation\_size

------------------

1441792

(1 row)

postgres=# \! ls -l /opt/db/userdb/u00/pgdata/base/13052/16505

-rw-------. 1 postgres dba 1441792 Apr 19 09:59 /opt/db/userdb/u00/pgdata/base/13052/16505

postgres=#

可以看到，段增加了超过1MB！准确地说，1441792-40960 = 1400832字节。为什么?

这个大对象被再次拆分为512个chunk，每个chunk都有2048个字节，PostgreSQL再次尝试压缩它们。但是，因为一个随机字符串不能被压缩，所以段仍然（平均）是1MB字节。

一个数据库块（block，或者称为页page）的大小是8192字节，如果减去block header的大小，就没有足够的空间容纳4个2048字节的chunk！每个数据库块（block）将只包含3个未压缩的chunk（注意区分block和chunk）。因此，512个chunk将分布在171个block上(CEIL(512/3.0))，得到：

postgres=# select ceil(1024\*1024/2048/3.0)\*8192;

?column?

----------

1400832

(1 row)

postgres=#

pg\_largeobject表当前有2个大对象，现在我们继续在为该表添加一个随机填充的大对象：

postgres=# select count(\*) from pg\_largeobject\_metadata;

count

-------

2

(1 row)

postgres=# select pg\_relation\_size('pg\_largeobject');

pg\_relation\_size

------------------

1441792

(1 row)

postgres=# \lo\_import '/tmp/randoms';

lo\_import 16511

postgres=# select pg\_relation\_size('pg\_largeobject');

pg\_relation\_size

------------------

2842624

(1 row)

postgres=# select count(\*) from pg\_largeobject\_metadata;

count

-------

3

(1 row)

postgres=# select oid, \* from pg\_largeobject\_metadata;

oid | oid | lomowner | lomacl

-------+-------+----------+--------

16509 | 16509 | 10 |

16510 | 16510 | 10 |

16511 | 16511 | 10 |

(3 rows)

postgres=#

由于新增的是不能压缩的随机字符序列，因此pg\_largeobject表又增加了171个block。

执行下面的SQL，安装扩展pageinspect：

postgres=# CREATE EXTENSION pageinspect;

CREATE EXTENSION

postgres=#

执行下面的SQL，检查pg\_largeobjects页面的内容：

postgres=# select \* from page\_header(get\_raw\_page('pg\_largeobject',0));

lsn | checksum | flags | lower | upper | special | pagesize | version | prune\_xid

-----------+----------+-------+-------+-------+---------+----------+---------+-----------

0/4004790 | 0 | 4 | 452 | 488 | 8192 | 8192 | 4 | 0

(1 row)

postgres=#

postgres=# select \* from page\_header(get\_raw\_page('pg\_largeobject',4));

lsn | checksum | flags | lower | upper | special | pagesize | version | prune\_xid

-----------+----------+-------+-------+-------+---------+----------+---------+-----------

0/4015548 | 0 | 4 | 360 | 2144 | 8192 | 8192 | 4 | 0

(1 row)

postgres=#

postgres=# select \* from page\_header(get\_raw\_page('pg\_largeobject',5));

lsn | checksum | flags | lower | upper | special | pagesize | version | prune\_xid

-----------+----------+-------+-------+-------+---------+----------+---------+-----------

0/401E390 | 0 | 4 | 36 | 1928 | 8192 | 8192 | 4 | 0

(1 row)

postgres=#

我们知道：page header占用24个字节，并且行指针为每个元组使用4个字节。

前4个页的偏移量较低，为452个字节，这意味着我们有（452-24）/ 4 = 107个元组。

第5页（page number是4）的lower为360：（360-24）/ 4 = 84元组。

第6页（page number是5）的lower为36：（36-24）/ 4 = 3个元组。

让我们检查是否正确：

postgres=# select generate\_series as page,

postgres-# ( select count(\*)

postgres-# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres-# ) as tuples

postgres-# from generate\_series(0,346);

page | tuples

------+--------

0 | 107

1 | 107

2 | 107

3 | 107

4 | 84

5 | 3

6 | 3

省略了许多输出

346 | 1

(347 rows)

postgres=# select generate\_series as page,

postgres-# ( select count(\*)

postgres-# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres-# ) as tuples

postgres-# from generate\_series(0,347);

ERROR: block number 347 is out of range for relation "pg\_largeobject"

postgres=#

（总共有347个页面（数据库块））

现在，让我们删除1Mb的随机文件（第2次增加的），并再次检查空间:

postgres=# select pg\_relation\_size('pg\_largeobject');

pg\_relation\_size

------------------

2842624

(1 row)

postgres=# \lo\_unlink 16510

lo\_unlink 16510

postgres=# select pg\_relation\_size('pg\_largeobject');

pg\_relation\_size

------------------

2842624

(1 row)

postgres=# select oid, \* from pg\_largeobject\_metadata;

oid | oid | lomowner | lomacl

-------+-------+----------+--------

16509 | 16509 | 10 |

16511 | 16511 | 10 |

(2 rows)

postgres=#

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres-# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres-# )

postgres-# from generate\_series(0,346);

pageno | count

--------+-------

0 | 107

1 | 107

2 | 107

3 | 107

4 | 84

5 | 1

6 | 1

7 | 1

8 | 1

9 | 1

10 | 1

11 | 1

12 | 1

(省略了许多输出)

345 | 3

346 | 1

(347 rows)

postgres=#

可以看到，该空间仍在使用，并且元组仍然在那里。不过我们可以通过检查t\_xmax的有效性来检查不再使用的元组。实际上，根据文档，如果XMAX无效，则该行是最新版本：

[…] a tupleis the latest version of its row iff XMAX is invalid or t\_ctid points to itself (in which case, if XMAX is valid, the tuple is either locked or deleted). […]

(源代码文件htup\_details.h lines 87-89).

对第12位(2048，或0x0800)检查信息进行检查：

#define HEAP\_XMAX\_INVALID 0x0800 /\* t\_xmax invalid/aborted \*/

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres-# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres(# where t\_infomask::bit(16) & x'0800'::bit(16) = x'0800'::bit(16)

postgres(# )

postgres(# from generate\_series(0,346);

pageno | count

--------+-------

0 | 107

1 | 107

2 | 107

3 | 107

4 | 84

5 | 0

6 | 0

7 | 0

8 | 0

（省略了许多输出）

(347 rows)

postgres=#

可以看到，大对象被分割成压缩的chunk，其内部行为与常规行相同：删除行，其先前占用的空间不会被自动回收后者再次使用！

仍然有347个页面

执行full vacuum，空闲就被释放了:

postgres=# select pg\_relation\_size('pg\_largeobject');

pg\_relation\_size

------------------

2842624

(1 row)

postgres=# vacuum full pg\_largeobject;

VACUUM

postgres=# select pg\_relation\_size('pg\_largeobject');

pg\_relation\_size

------------------

1441792

(1 row)

postgres=#

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres-# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres-# )

postgres-# from generate\_series(0,175);

173 | 3

174 | 3

175 | 2

(176 rows)

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres-# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres-# )

postgres-# from generate\_series(0,176);

ERROR: block number 176 is out of range for relation "pg\_largeobject"

postgres=#

回收了171个页面，还有176个页面

我们已经看到大对象被分成每个元组，每个元组包含2048个字节，每个chunk的行为与常规元组非常相似。

注意：在PostgreSQL中，借助TOAST技术，可以将大量数据与表一起存储。

**传统大对象表示例**

大对象没有插入应用表中，但是以不同的方式被处理。使用大对象的应用常具有一个包含OID类型的列的表。当应用程序创建新的大对象时，将为其分配一个新的OID号，并将此号插入到应用表中。

现在，对于来自其他RDBMS（例如Oracle）的人来说，这是一个常见错误，认为当删除该行的引用，大对象会自动取消链接。事实并非如此，我们需要将其与应用明确断开链接(unlink)。让我们从一个简单的示例开始，从一个空的pg\_largeobject表开始：

在第1个Linux终端，执行下面的命令和SQL语句：

postgres=# \lo\_unlink 16509

lo\_unlink 16509

postgres=# \lo\_unlink 16511

lo\_unlink 16511

postgres=# vacuum full pg\_largeobject;

VACUUM

postgres=# select count(\*) from pg\_largeobject\_metadata;

count

-------

0

(1 row)

postgres=# select pg\_relation\_size('pg\_largeobject')/8192 as pages;

pages

-------

0

(1 row)

postgres=#

插入一个新的LOB并在表t中引用它：

postgres=# CREATE TABLE t (id integer, file oid);

CREATE TABLE

postgres=# \lo\_import /tmp/zeroes

lo\_import 16565

postgres=# INSERT INTO t VALUES (1, 16565);

INSERT 0 1

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres-# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres(# where t\_infomask::bit(16) & x'0800'::bit(16) = x'0800'::bit(16)

postgres(# )

postgres(# from generate\_series(0,4);

pageno | count

--------+-------

0 | 107

1 | 107

2 | 107

3 | 107

4 | 84

(5 rows)

postgres=#

postgres=# \lo\_import /tmp/zeroes

lo\_import 16566

postgres=# INSERT INTO t VALUES (2, 16566);

INSERT 0 1

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres(# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres(# where t\_infomask::bit(16) & x'0800'::bit(16) = x'0800'::bit(16)

postgres(# )

postgres-# from generate\_series(0,4);

pageno | count

--------+-------

0 | 107

1 | 107

2 | 107

3 | 107

4 | 107

(5 rows)

postgres=#

postgres=# select \* from t;

id | file

----+-------

1 | 16565

2 | 16569

(2 rows)

postgres=#

如果我们删除表t中id=1的行，它的LOB块仍然存在，且有效:

postgres=# DELETE FROM t WHERE id=1;

DELETE 1

postgres=# select \* from t;

id | file

----+-------

2 | 16569

(1 row)

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres(# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres(# where t\_infomask::bit(16) & x'0800'::bit(16) = x'0800'::bit(16)

postgres(# )

postgres-# from generate\_series(0,9);

pageno | count

--------+-------

0 | 107

1 | 107

2 | 107

3 | 107

4 | 107

5 | 107

6 | 107

7 | 107

8 | 107

9 | 107

(10 rows)

postgres=#

如果我们想清理掉LOB，我们必须解除它的链接，要么显式地解除，要么在删除应用程序表中的一条记录时使用触发器解除LOB的链接。

另一种方法是使用PostgreSQL中包含的二进制*vacuumlo*。

它扫描pg\_largeobject\_metadata并搜索具有OID列的表，查找是否有对lob的引用。未被引用的LOB被解除链接。

注意:这意味着，如果您使用其他方法引用OID列以外的lob，那么vacuumlo可能会断开仍然需要的lob的链接!

打开第2个Linux终端，执行下面的命令：

[postgres@dbsvr ~]$ vacuumlo -U postgres postgres

第1个psotgres是数据库用

第2个postgres是数据库

回到第1个Linux终端：

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres(# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres(# where t\_infomask::bit(16) & x'0800'::bit(16) = x'0800'::bit(16)

postgres(# )

postgres-# from generate\_series(0,9);

pageno | count

--------+-------

0 | 0

1 | 0

2 | 0

3 | 0

4 | 23

5 | 107

6 | 107

7 | 107

8 | 107

9 | 61

(10 rows)

postgres=#

事实上，vacuumlo已经解除了对第一个LOB的链接，但是删除的元组在执行vacuum之前不会被释放:

postgres=# vacuum pg\_largeobject;

VACUUM

postgres=#

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres(# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres(# where t\_infomask::bit(16) & x'0800'::bit(16) = x'0800'::bit(16)

postgres(# )

postgres-# from generate\_series(0,4);

pageno | count

--------+-------

0 | 107

1 | 107

2 | 107

3 | 107

4 | 84

(5 rows)

postgres=# select generate\_series as pageno,

postgres-# ( select count(\*)

postgres(# from heap\_page\_items(get\_raw\_page('pg\_largeobject',generate\_series))

postgres(# where t\_infomask::bit(16) & x'0800'::bit(16) = x'0800'::bit(16)

postgres(# )

postgres-# from generate\_series(0,5);

ERROR: block number 5 is out of range for relation "pg\_largeobject"

postgres=#

只剩下5个页面了！

**结论：vacuumlo没有对pg\_largeobject表进行任何清理。**

### 传统大对象的使用示例

psql -U postgres -d universitydb

# 导入大对象文件，获取 OID

\lo\_import '/home/postgres/mysong1.mp3'

CREATE TABLE songs (

id SERIAL PRIMARY KEY,

file\_name VARCHAR(255),

song\_title VARCHAR(255),

artist VARCHAR(255),

song\_file\_oid OID

);

INSERT INTO songs (file\_name, song\_title, artist, song\_file\_oid) VALUES ('mysong1.mp3', '笑脸', '谢东', 16577);

# 导出大对象

\lo\_export 16577 '/opt/db/mysong1.mp3'

## 大对象存储机制

[PostgreSQL的大对象以及空间使用 (1) - abce - 博客园 (cnblogs.com)](https://www.cnblogs.com/abclife/p/13790262.html)

<https://www.cnblogs.com/abclife/p/13790262.html>

<https://www.cnblogs.com/abclife/p/13790588.html>

<https://www.cnblogs.com/abclife/p/13791355.html>

原文：

<https://www.ludovicocaldara.net/dba/pgsql-lo-space-usage-part-1/>

<https://www.ludovicocaldara.net/dba/pgsql-lo-space-usage-part-2/>

<https://www.ludovicocaldara.net/dba/pgsql-lo-space-usage-part-3/>

[PostgreSQL源码剖析\_插入大对象时各存储相关模块交互流程 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/100424042)

<https://zhuanlan.zhihu.com/p/100424042>

不错！

**实战大对象**

[PostgreSQL大对象\_postgres存储音频-CSDN博客](https://blog.csdn.net/z465759695/article/details/130229981)

https://blog.csdn.net/z465759695/article/details/130229981

**PostgreSQL中的大对象**

[PostgreSQL中的大对象\_postgres 大对象-CSDN博客](https://postgresql.blog.csdn.net/article/details/83149617?spm=1001.2101.3001.6650.3&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-83149617-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-83149617-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=6)

<https://postgresql.blog.csdn.net/article/details/83149617?spm=1001.2101.3001.6650.3&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-83149617-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-3-83149617-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=6>

[postgreSQL源码分析——存储管理——外存管理（7）\_postgresql 大字段对象存放在哪里-CSDN博客](https://blog.csdn.net/weixin_45644897/article/details/120953940?spm=1001.2101.3001.6650.8&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-120953940-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-120953940-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=15)

<https://blog.csdn.net/weixin_45644897/article/details/120953940?spm=1001.2101.3001.6650.8&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-120953940-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-8-120953940-blog-130229981.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=15>

不错！

# 索引

[PostgreSQL-源码学习笔记(5)-索引 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/617710106)

# 参考

## 源码分析

[PostgerSQL\_J.Kuchiki的博客-CSDN博客](https://blog.csdn.net/qq_43899283/category_12377546.html?spm=1001.2014.3001.5482)

<https://blog.csdn.net/qq_43899283/category_12377546.html?spm=1001.2014.3001.5482>

## The Internals of PostgreSQL

<https://www.interdb.jp/pg/index.html>

## the different memory components of a PostgreSQL instance

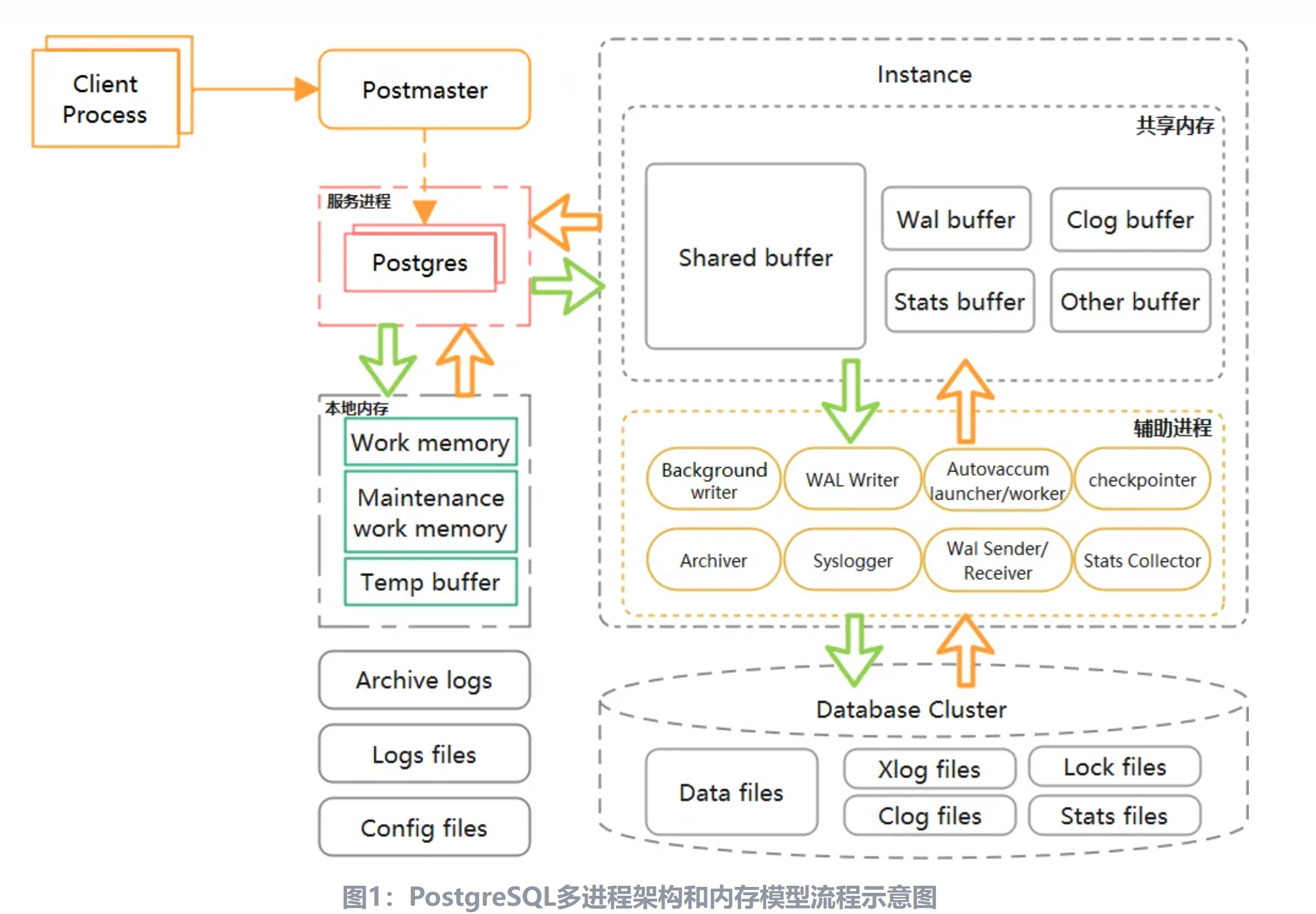
[Let's get back to basics - PostgreSQL memory components (fastware.com)](https://www.postgresql.fastware.com/blog/lets-get-back-to-basics-postgresql-memory-components)

<https://www.postgresql.fastware.com/blog/lets-get-back-to-basics-postgresql-memory-components>

## PostgreSQL进程架构与内存管理

[PostgreSQL技术内幕（十四）探索PG的进程与内存管理 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/687002997?utm_campaign=shareopn&utm_medium=social&utm_psn=1758156379016138752&utm_source=wechat_session)

<https://zhuanlan.zhihu.com/p/687002997?utm_campaign=shareopn&utm_medium=social&utm_psn=1758156379016138752&utm_source=wechat_session>



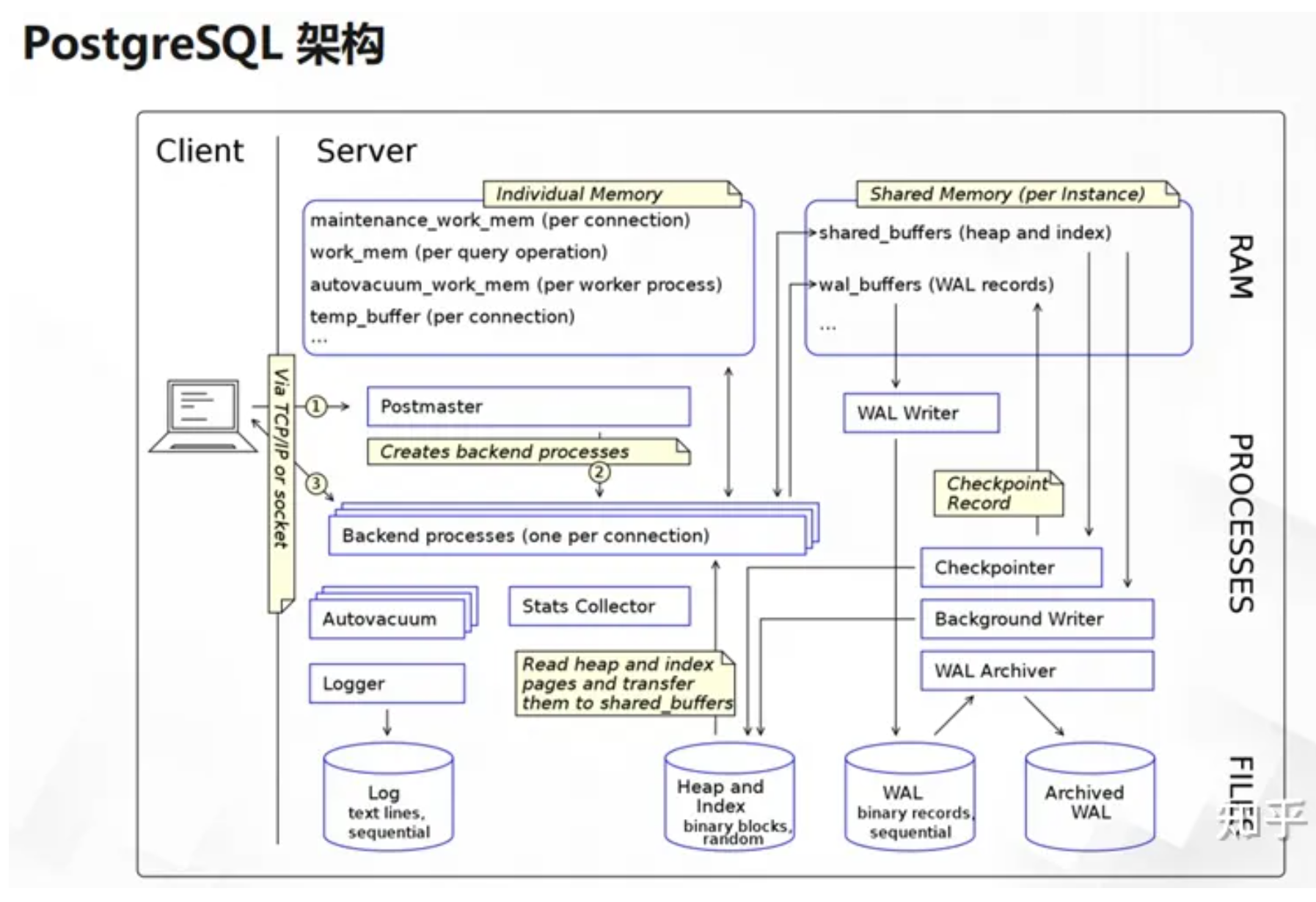
## PostgreSQL存储结构浅析

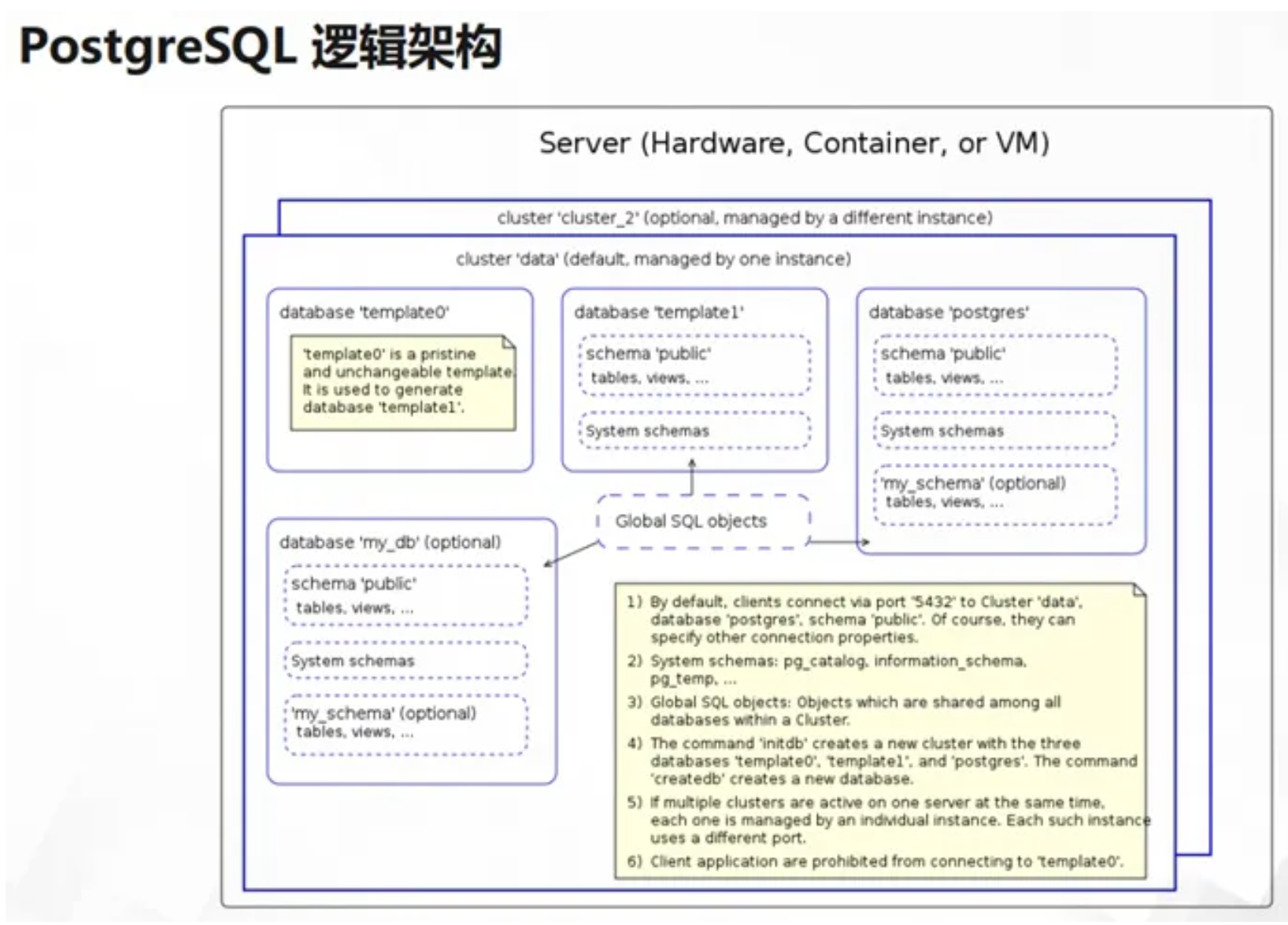
<http://geekdaxue.co/read/fcant@sql/uw0wuu>

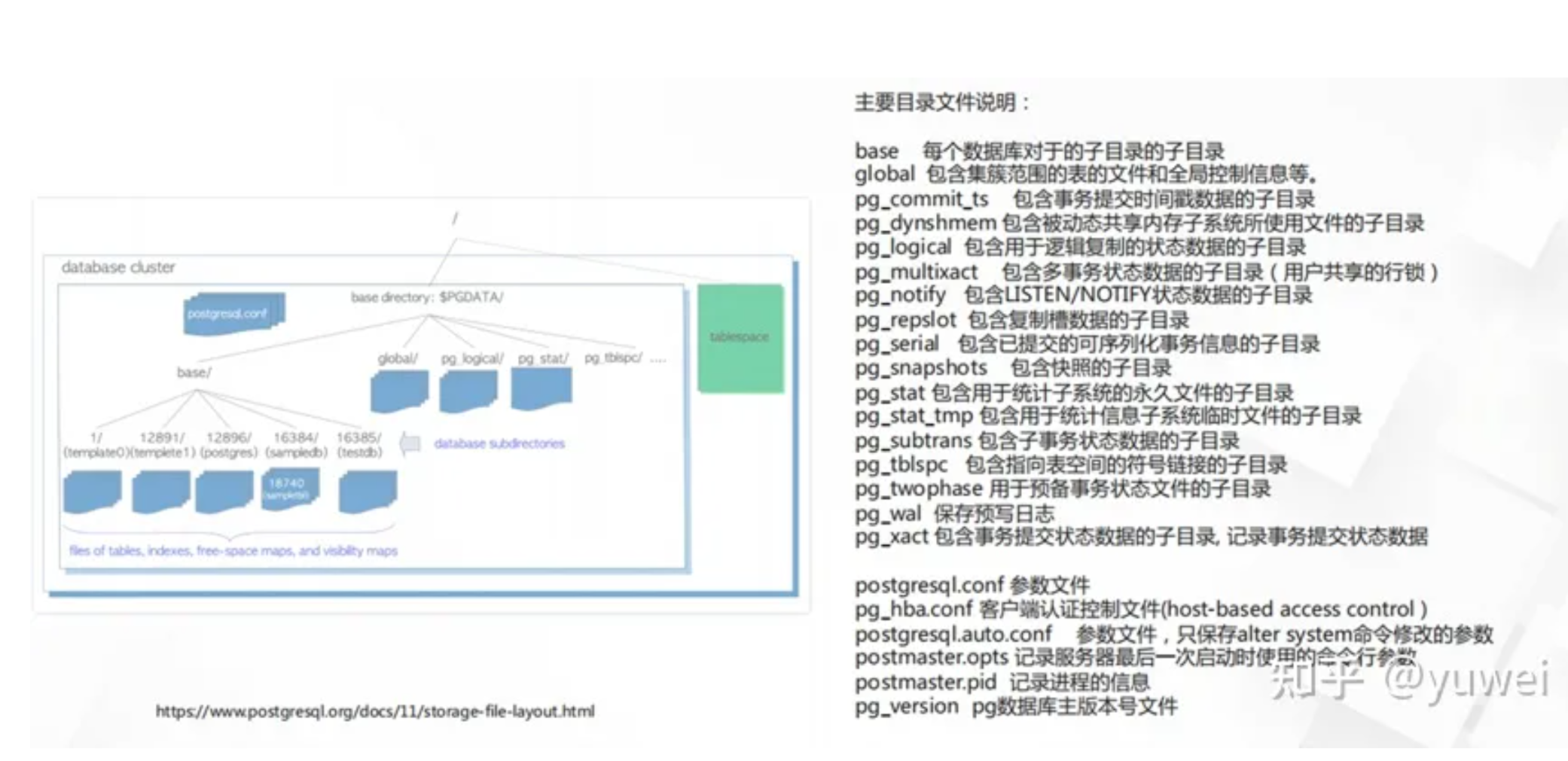
## PostgreSQL系统概述

PG内核解读-第1节PostgreSQL系统概述 - 知乎 (zhihu.com)

<https://zhuanlan.zhihu.com/p/668983232>









## 存储管理

[PostgreSQL源码分析-存储管理总结\_postgresql源代码 存储管理-CSDN博客](https://blog.csdn.net/weixin_45644897/article/details/122014195?spm=1001.2101.3001.6650.15&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-15-122014195-blog-121254012.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-15-122014195-blog-121254012.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=19)

<https://blog.csdn.net/weixin_45644897/article/details/122014195?spm=1001.2101.3001.6650.15&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-15-122014195-blog-121254012.235%5Ev43%5Epc_blog_bottom_relevance_base6&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-15-122014195-blog-121254012.235%5Ev43%5Epc_blog_bottom_relevance_base6&utm_relevant_index=19>

[【PostgreSQL内核学习（十八）—— 存储管理（存储管理的体系结构）】\_postgresql存储原理-CSDN博客](https://blog.csdn.net/qq_43899283/article/details/135088540?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-135088540-blog-122014195.235%5ev43%5epc_blog_bottom_relevance_base6&spm=1001.2101.3001.4242.1&utm_relevant_index=3)

[https://blog.csdn.net/qq\_43899283/article/details/135088540?utm\_medium=distribute.pc\_relevant.none-task-blog-2~default~baidujs\_baidulandingword~default-0-135088540-blog-122014195.235^v43^pc\_blog\_bottom\_relevance\_base6&spm=1001.2101.3001.4242.1&utm\_relevant\_index=3](https://blog.csdn.net/qq_43899283/article/details/135088540?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-135088540-blog-122014195.235%5ev43%5epc_blog_bottom_relevance_base6&spm=1001.2101.3001.4242.1&utm_relevant_index=3)

[【PostgreSQL内核学习（十九）—— 存储管理（元组操作）】-CSDN博客](https://blog.csdn.net/qq_43899283/article/details/135112278?spm=1001.2014.3001.5502)

<https://blog.csdn.net/qq_43899283/article/details/135112278?spm=1001.2014.3001.5502>

### 内存管理

[postgreSQL源码分析——存储管理——内存管理（1）\_memorycontext postgres-CSDN博客](https://blog.csdn.net/weixin_45644897/article/details/121043976)

<https://blog.csdn.net/weixin_45644897/article/details/121043976>

[postgreSQL源码分析——存储管理——内存管理（2）\_palloc-CSDN博客](https://blog.csdn.net/weixin_45644897/article/details/121087469)

<https://blog.csdn.net/weixin_45644897/article/details/121087469>

[postgreSQL源码分析——存储管理——内存管理（3）\_postgresql syscache-CSDN博客](https://blog.csdn.net/weixin_45644897/article/details/121254012#comments_29174722)

<https://blog.csdn.net/weixin_45644897/article/details/121254012#comments_29174722>

### 外存管理

[postgreSQL源码分析——存储管理——外存管理（3）\_postgresql vector源码-CSDN博客](https://blog.csdn.net/weixin_45644897/article/details/120640263)

<https://blog.csdn.net/weixin_45644897/article/details/120640263>

## 源码综述

[postgreSQL源码分析综述-CSDN博客](https://blog.csdn.net/weixin_45644897/article/details/120494879)

<https://blog.csdn.net/weixin_45644897/article/details/120494879>

## 安装psycopg2

pip install psycopg2

pip install psycopg2-binary