HOW 2026
Hello Open-source World

以开源之道·见致远之志
开源生态大会暨PostgreSQL高峰论坛

PG内核培训串串烧

建立内核全局认知 – 理解架构与开发核心逻辑

Cary Huang

目录

CONTENTS

关于 Cary

关于Cary

- 2012年畢業於　列顛哥倫比亞　學（UBC）　獲電機工程學位
- 畢業後直接進入智能电表和能　產業工作
- 2019 年　入Hornetlabs Technology　開啟我　PostgreSQL之旅
- 2021 年在北京　學擔任研究　導師　致力於推廣PostgreSQL開　態　統
- 參與過 PostgreSQL　多個　向　研究　包括分片增強　分布式数据　可　性　共享儲存　無服务器架構　安全　内核　发等等
- 2025 年被列　PostgreSQL Contributor

HOW 2026
Hello Open-source World

以开源之道·见致远之志
开源生态大会暨PostgreSQL高峰论坛

PostgreSQL 培训教材设计

该从哪里开始培训？

- 我们在 2024 年　某国内企业数据库团队设计　培训教材
- 总共有 6 个章节　每个章节　概 4 小时培训时间
- PostgreSQL 涉及　主　很多　我们也　清楚客户　　程和 PostgreSQL 水平如何

- 所以我们和客户讨　了几个比较经典　　发主　　把　们简化成适合课程　　小
  - **分布式数据库**（重　关注全局事务管理 + 死锁检测 + 两段式提交技巧）
  - **内存数据库存储引擎**（Table Access Method 接口使　）
  - **共享存储架构的数据快传输协议**（基　libpq COPY 协议)

- 围绕着这些主　去学习到 PostgreSQL 内核　发　从基础到　　知识分享
- 这次　‘内核培训串串烧’是　个精华版　培训 + 基　SynchDB 插件　代码解说

最终的章节设计

HOW 2026
Hello Open-source World

分布式数据库改造：

- 新增 GUC 参数
- 孤儿事务后台检测进程
- 回归测试
- 新增 SQL 函数查看
- GTM 全局事务管理机制
- 两段式提交 FDW 接口
- 新增全局死锁检测逻辑

第一章：系统基础
- GUC 架构，代码架构
- PostgreSQL 架构解析
- 回归测试架构
- 系统表和视图改造

第二章：功能扩展
- SQL 生命周期5阶段
- 后台进程架构，共享内存
- 插件架构
- 新增自定义 SQL 函数

第三章：事务管理和可见性判断
- 事务概念
- 可见性判断逻辑
- 事务锁
- 外部数据包装器 FDW 接口

第四章：数据存储和访问管理
- Buffer Manager 模块解析
- Storage Manager 模块解析
- Table Access Method 接口解析
- Index Access Method 介绍

第五章：WAL结构和流复制
- WAL 的架构解析
- WAL REDO 功能解析
- 流复制功能解析
- Libpq 复制协议解析

第六章：逻辑复制和备份恢复
- 逻辑和物理备份概念解析
- 逻辑复制功能解析
- pg_rman 备份工具解析
- 逻辑复制解码插件接口

内存数据库引擎：

- 基于 Table Access Method 接口
- 实现一个基于内存的数据库存储引擎

共享存储数据快传输

- 基于 libpq 协议接口
- 实现一个基于 COPY 的数据交换协议
- 主备节点的高度同步

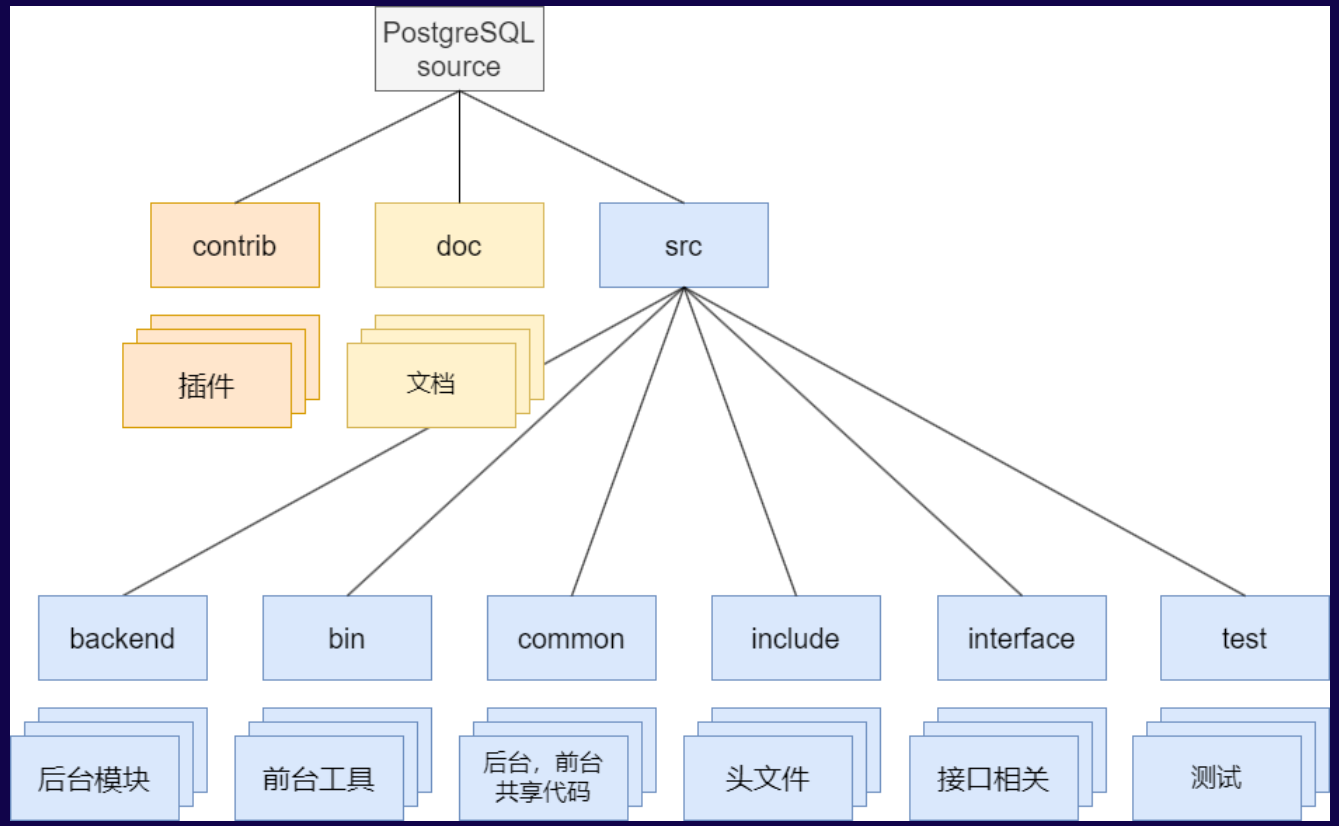
自定义逻辑复制解码

- 实现一个Mongodb逻辑复制解码插件

第一章：系统基础

PostgreSQL 代码结构（简化版）

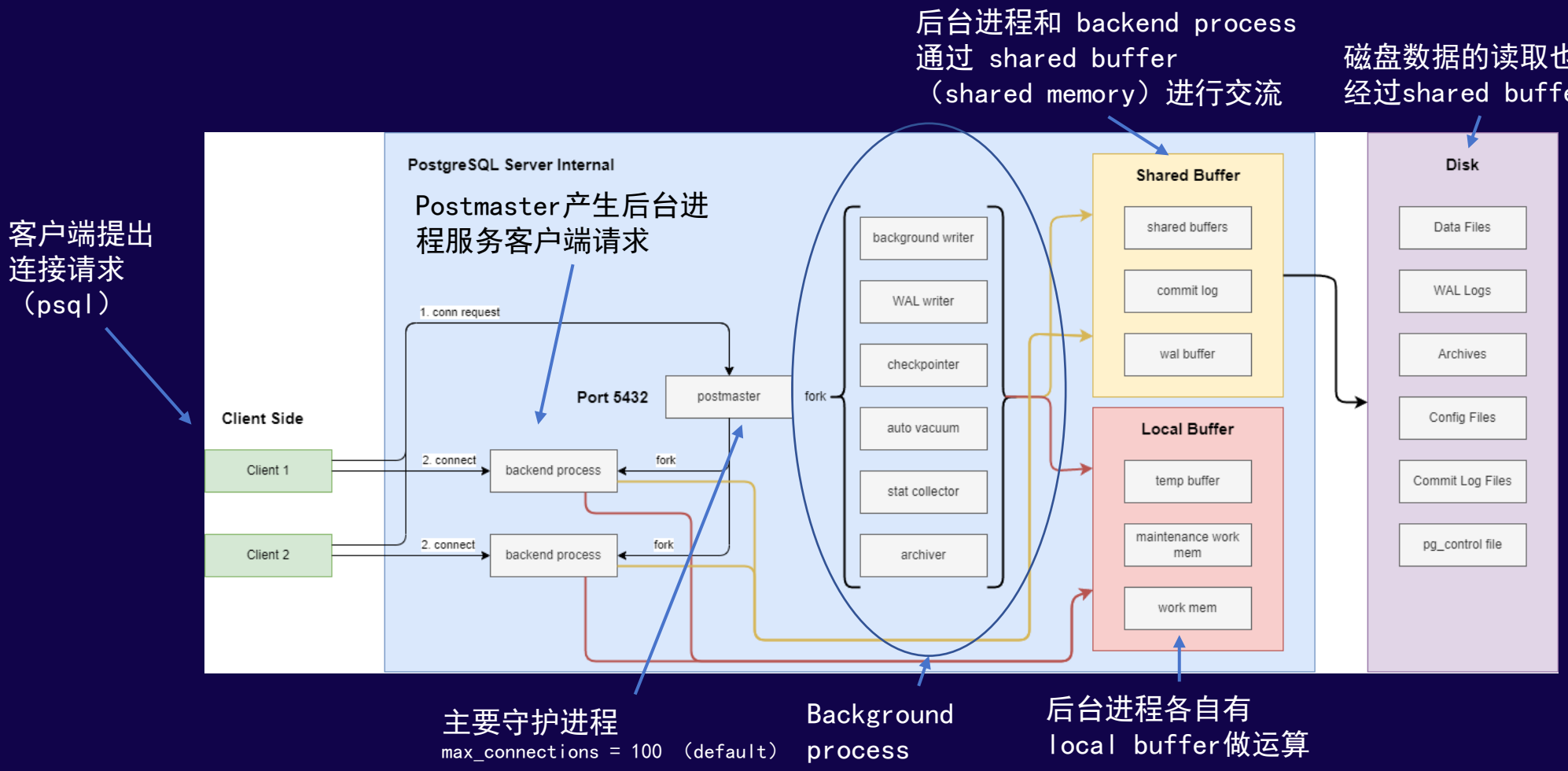
- PostgreSQL 的核心模块基本上都在 src 目录下。也是默认编译的目录
- contrib（插件）和 doc（文档）数据附加的功能，不在默认编译列表内，需要单独编译



PostgreSQL 前端和后端

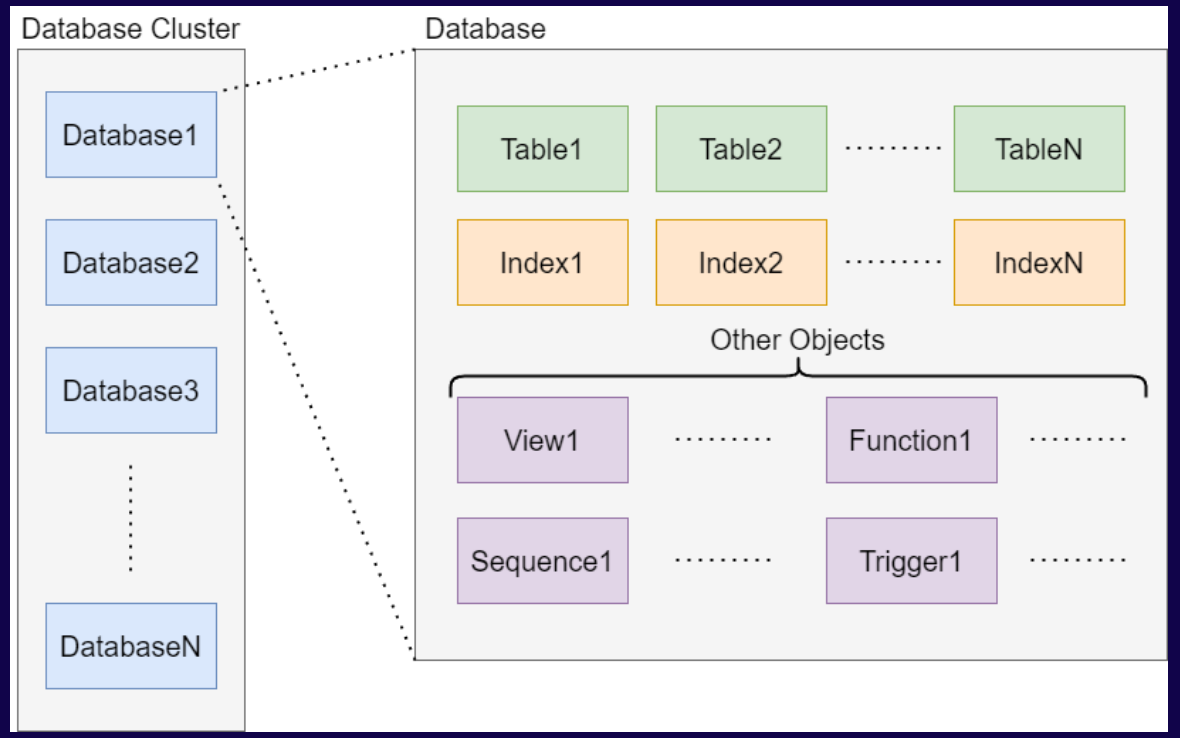
- 假设我们想要加新功能到 PostgreSQL，我们首先要考虑这个功能是属于前端，还是后端，还是两者都有。
- 能不能以插件的形式实现？如果可以，那我们尽量做成插件，这样不会改变 PostgreSQL 内核，以后合并到新版本会容易许多
- 这决定了你的新功能是放在哪一个子目录

PostgreSQL 系统架构



数据库集群的逻辑结构

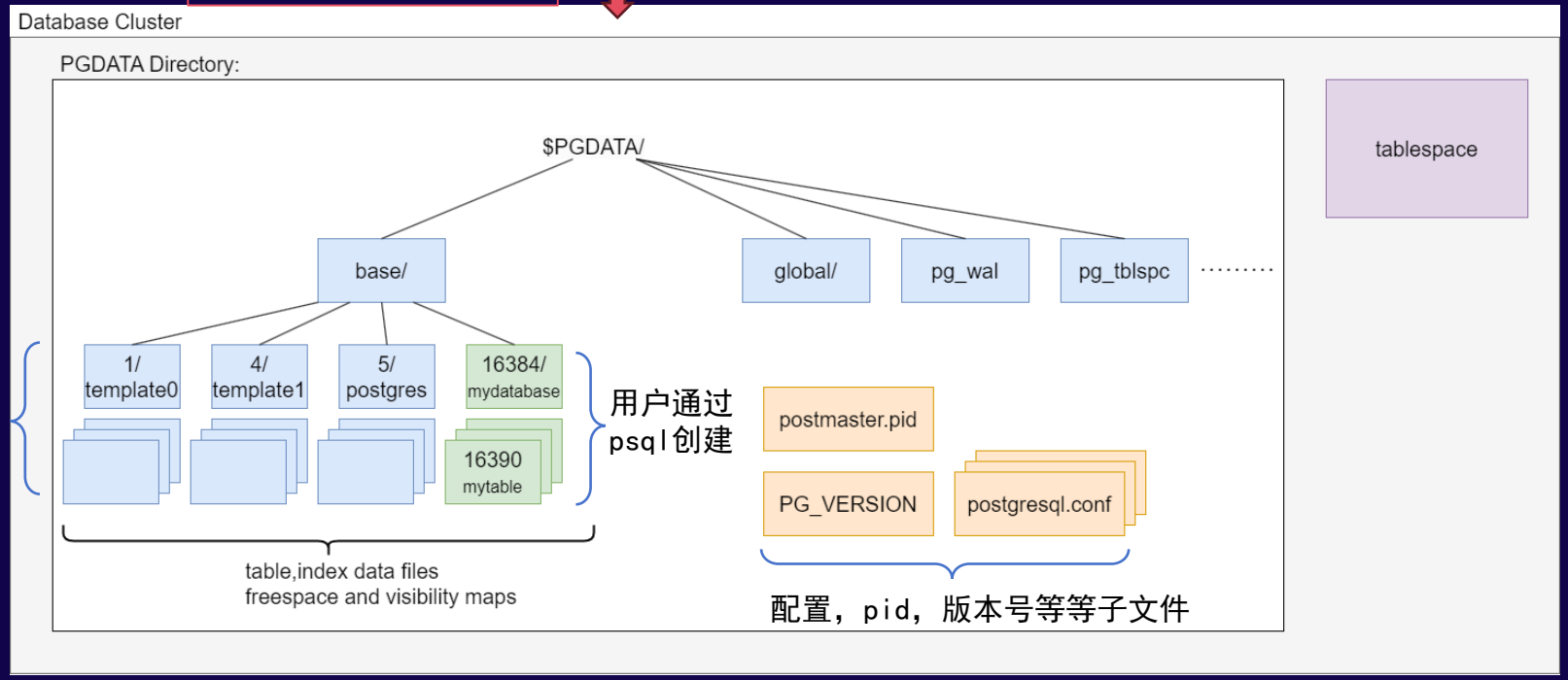
- 数据库集群（database cluster）是单个 PostgreSQL 服务器管理的数据库的集合。
- 在 PostgreSQL 中的术语“数据库集群”并不意味着“一组数据库服务器”。
- PostgreSQL 服务器在单个主机上运行并管理单个数据库集群。
- 一个“集群”包含多个 “数据库”，一个数据库又可包含多个“表，视图，函数等等”



数据库集群的物理结构变化


```
postgres=# CREATE DATABASE mydatabase;
postgres=# \c mydatabase
mydatabase1=# CREATE TABLE mytable(a INT, b TEXT);
```

绿色方块表示上述语句
对集群目录造成的变化



通过initdb
创建

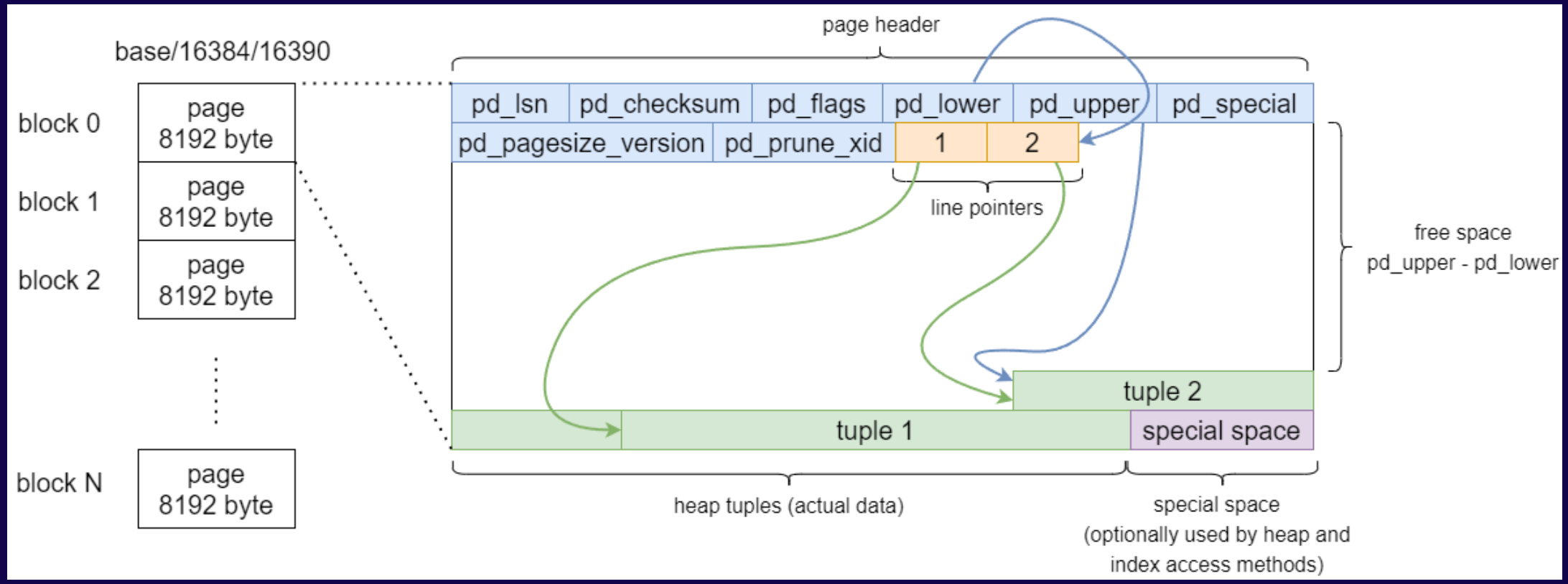
用户通过
psql创建

OID


```
mydatabase=# select datname, oid from pg_database where
datname = 'mydatabase';
  datname   |  oid
------------+-------
 mydatabase | 16384
(1 row)

mydatabase=# select relname, oid from pg_class where
relname = 'mytable';
 relname |  oid
---------+-------
 mytable | 16385
(1 row)
```

- PostgreSQL OID（对象标识符）是分配给数据库中的每一种系统对象的唯一标识符，提供了在 PostgreSQL 内部引用和识别对象的方式。
- Table，index，database，view 等等，都有相对应的 OID，由 PostgreSQL 自主生成管理
- OID 是无符号的 4 字节整数。

数据文件的内部布局

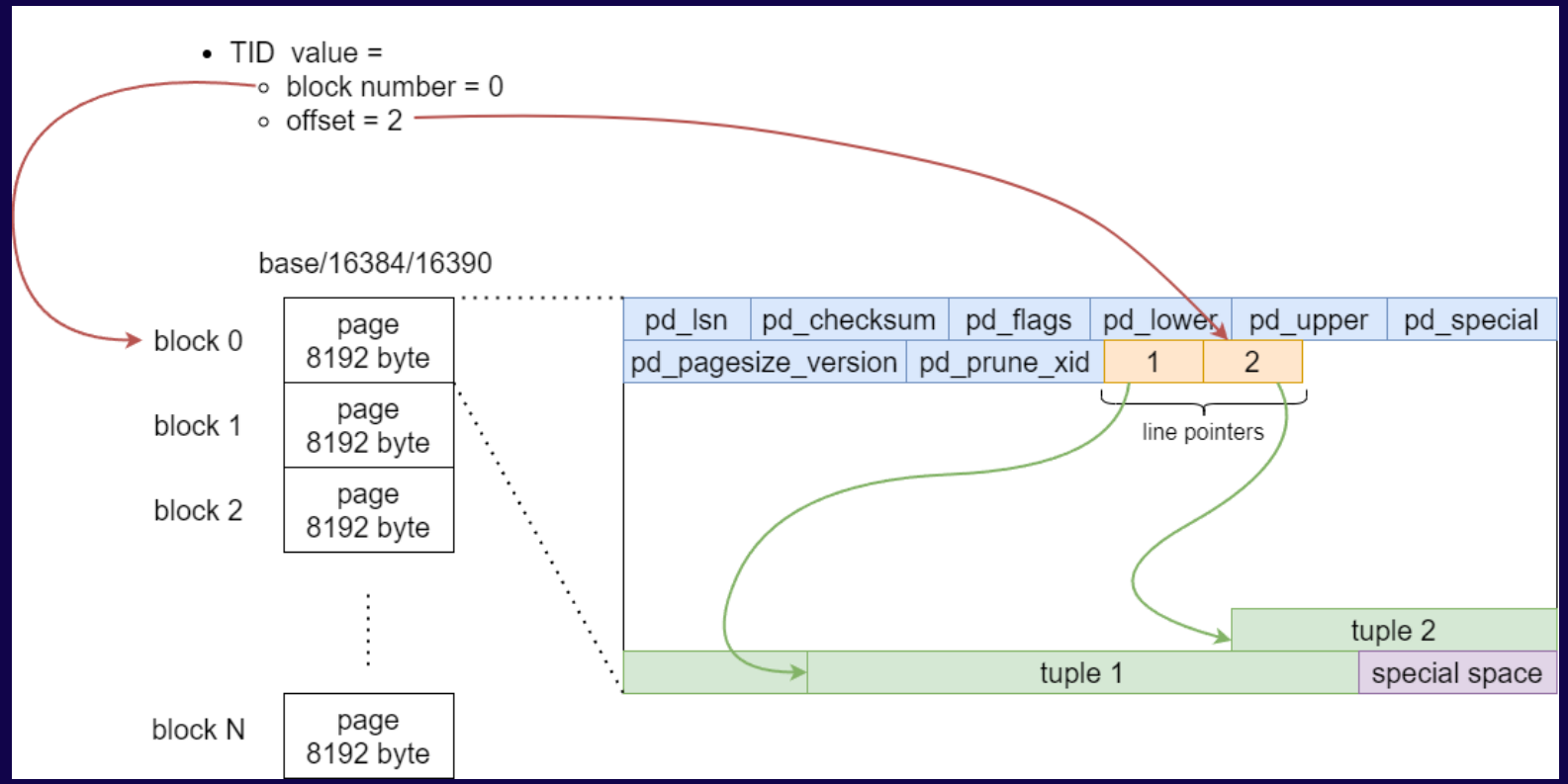

base/16384/16390
block 0
block 1
block 2
block N
page
8192 byte
page header
pd_lsn
pd_checksum
pd_flags
pd_lower
pd_upper
pd_special
pd_pagesize_version
pd_prune_xid
1
2
line pointers
free space
pd_upper - pd_lower
tuple 2
tuple 1
special space
heap tuples (actual data)
special space
(optionally used by heap and
index access methods)

数据文件的内部布局 – TID 定位

- Heap tuple - 指的是行数据，从 page底部开始按顺序堆叠
- Line pointer array（也叫item pointer） - 4 byte 指针数组，每个指针指向一个底部的 heap tuple
- Tuple Identifier （TID） - 包含 block number 和 offset number。通常在 PostgreSQL 内部被用来快速定位到一个 tuple。索引就是靠 TID 快速定位到一个 tuple。

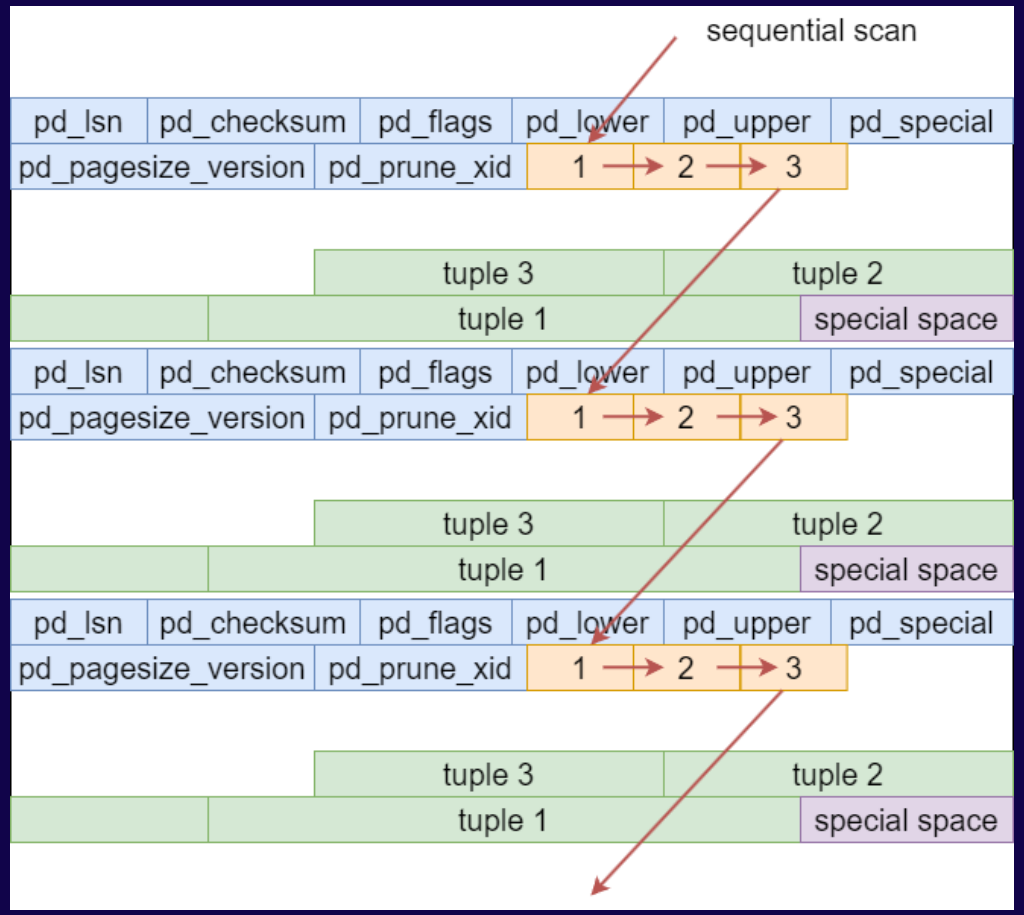
数据文件的内部布局 – TID 定位

TID 告诉我们 heap 数据具体是在哪一个 page（block）里的哪一个line pointer （offset）上



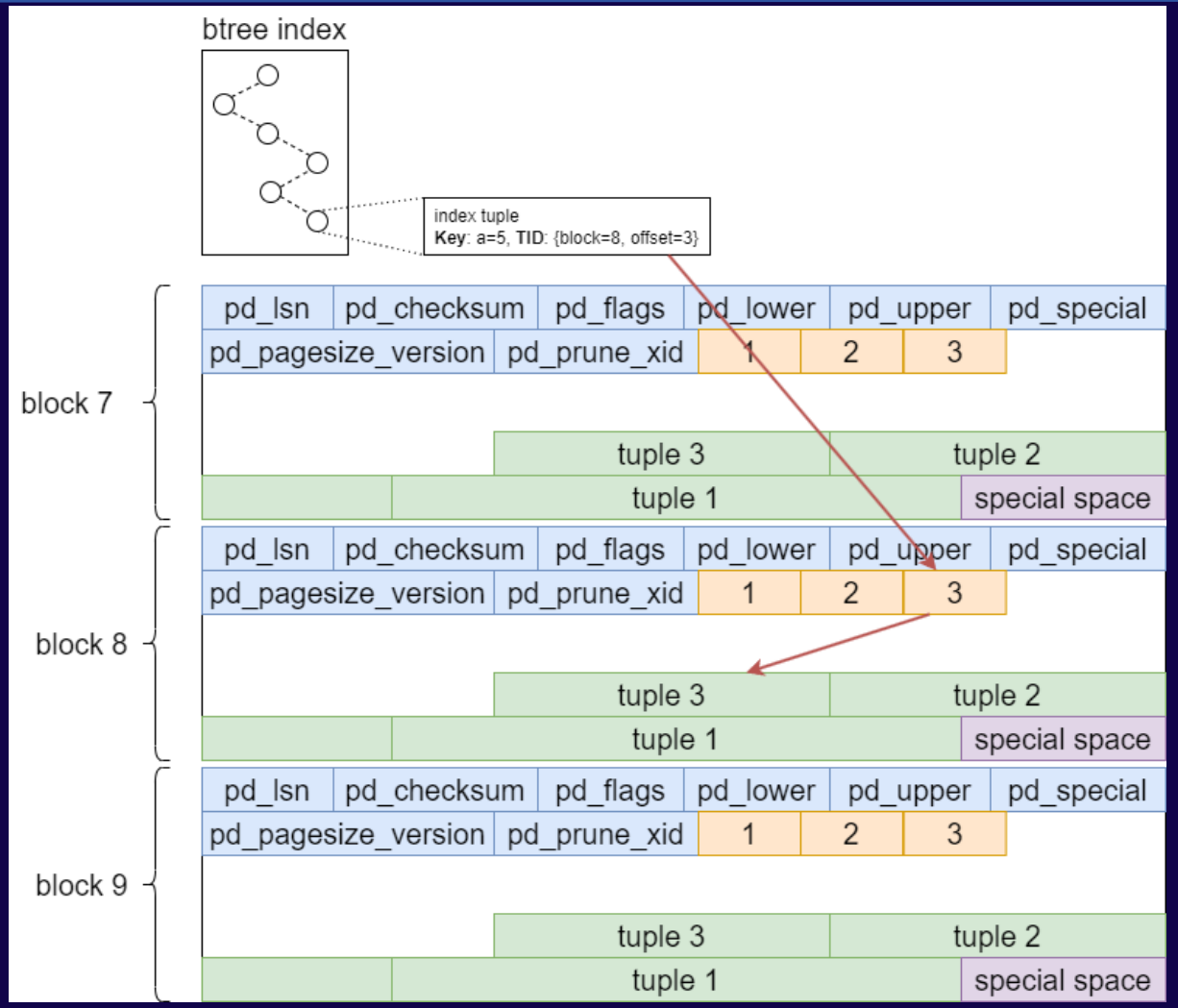
读取 Heap Tuple：顺序扫描（Seq Scan）

- 我们常说的 sequential scan 指的就是把所有 page 里的 line pointers 都扫描一遍
- 例如：SELECT * FROM mytable;



读取 Heap Tuple：索引扫描和获取

- 以 btree index 为例
- 例如：SELECT * FROM mytable WHERE a=5;
- 每个索引由索引键（key）和目标 heap tuple 的 TID 组成
- 先扫描 index btree，找到相对应的 key，得到其TID,再通过 TID 的block number 和 offset直接读出目标 tuple



第二章：PG功能扩展

SQL 语句生命周期

HOW 2026
Hello Open-source World

语法字面分析
- 检查字面上语法错误。
- 生成语法树。
- 使用 flex 和 bison 生成语法格式。
- 入口在 parser/parser.c。

查询重写
- 负责查询语句的重写来实现视图(view)和规则(rule)。
- 所以需要的时候，在这个阶段会对查询语句进行重写。
- 入口在 rewrite/rewriteHandler.c。

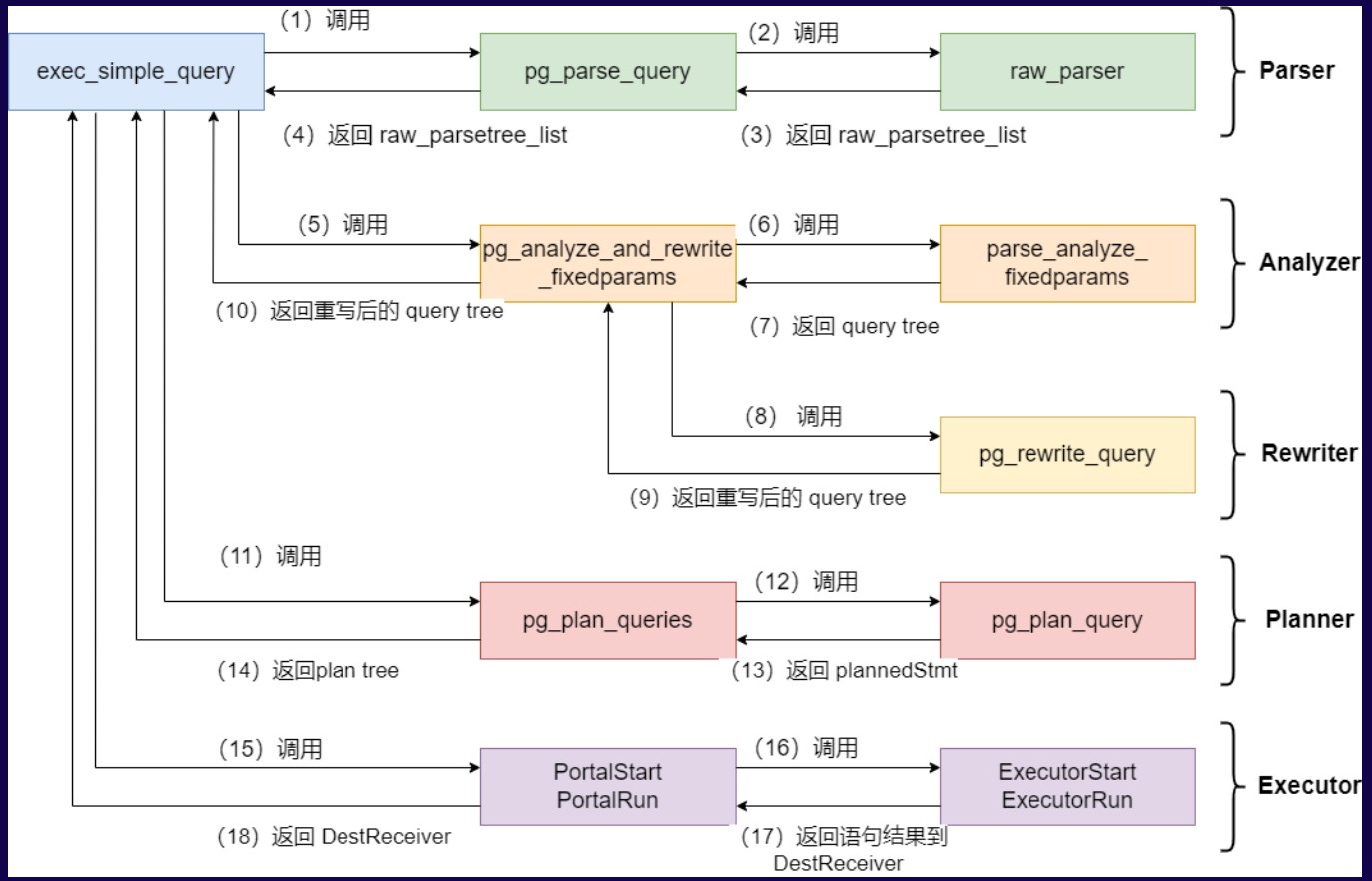
执行查询
- 执行执行计划里的步骤。
- 入口在 executor/execMain.c。

语法深层分析
- 访问数据库。
- 检查表是否存在。
- 检查数据格式。
- 把表名转换成内部OID。
- 生成查询树。
- 入口在parser/analyze.c。

生成执行计划
- 选择可以在最短时间内完成查询的方法。
- 入口在 optimizer/plan/planner.c。

SQL 语句生命周期 – 从代码角度看

- 所有语句通常都是通过 PostgresMain 主进程函数开始处理 （src/backend/tcop/postgres.c）。
- 大部分情况下都是从 exec_simple_query ()函数开始。
- 还有另一种透过 extended query protocol来执行语句
  - 基本上把这 5 个环节让用户分开来执行。
  - 这次就不细说 extended query protocol。



PostgreSQL 查询优化器（Planner）在做什么？

HOW 2026
Hello Open-source World

- 同一个查询，可以有多种执行方式。
- Planner 的任务 = 枚举所有可能 → 算成本 → 选最优
- 核心流程 = Query → Paths → Cost → Best Path → Plan

- Paths（路径）
  - 所有可能的执行方式 （seq scan, index scan, join…etc)。
- Cost（代价）
  - 路径的成本估算 （IO, CPU…etc)。
- 选择最优路径
  - 代价最低的路径。
- 生成 Plan
  - 转化成 Executor 可以执行的计划。

JOIN 为什么复杂？ （A JOIN B JOIN C)

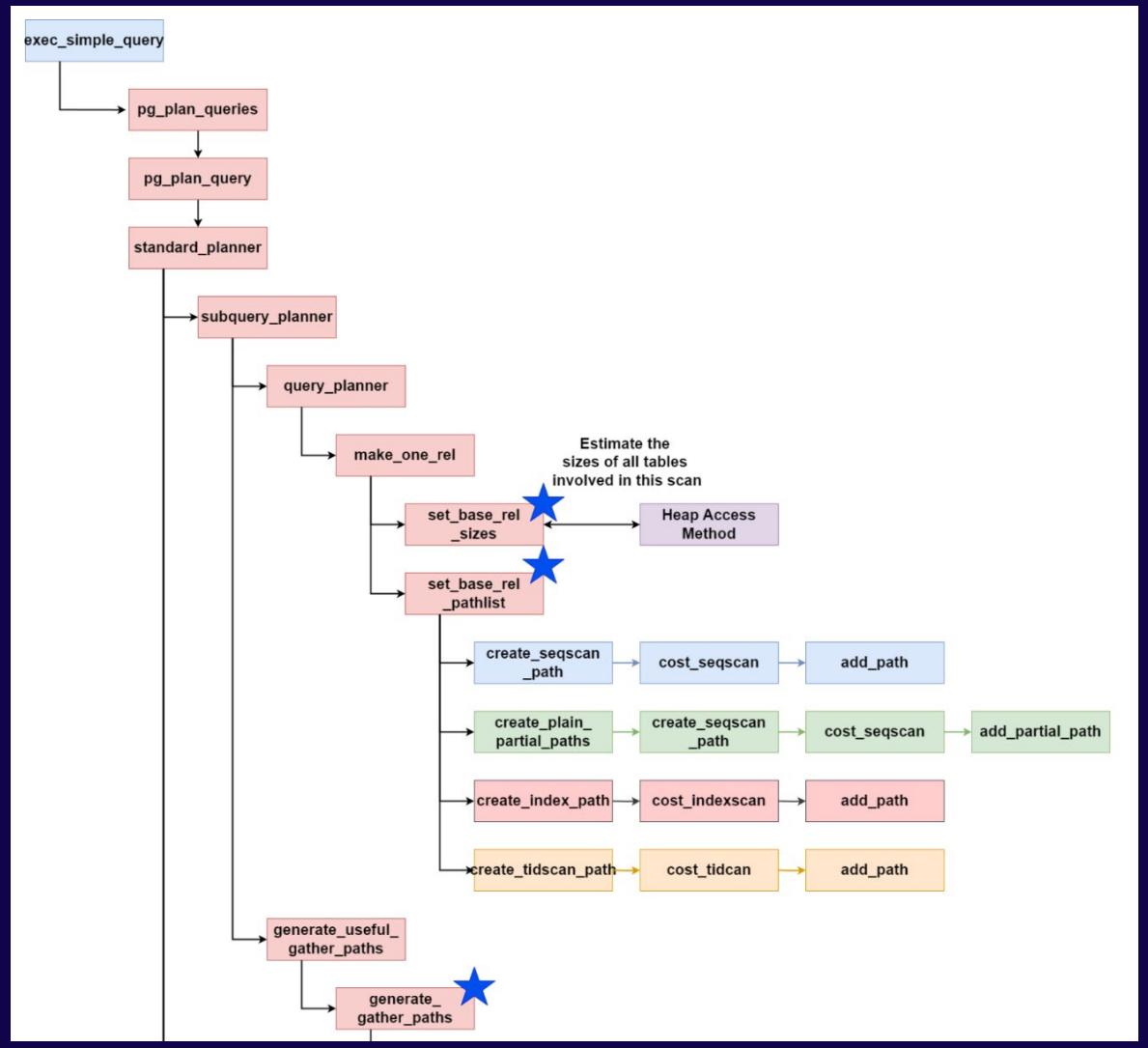
需要决定

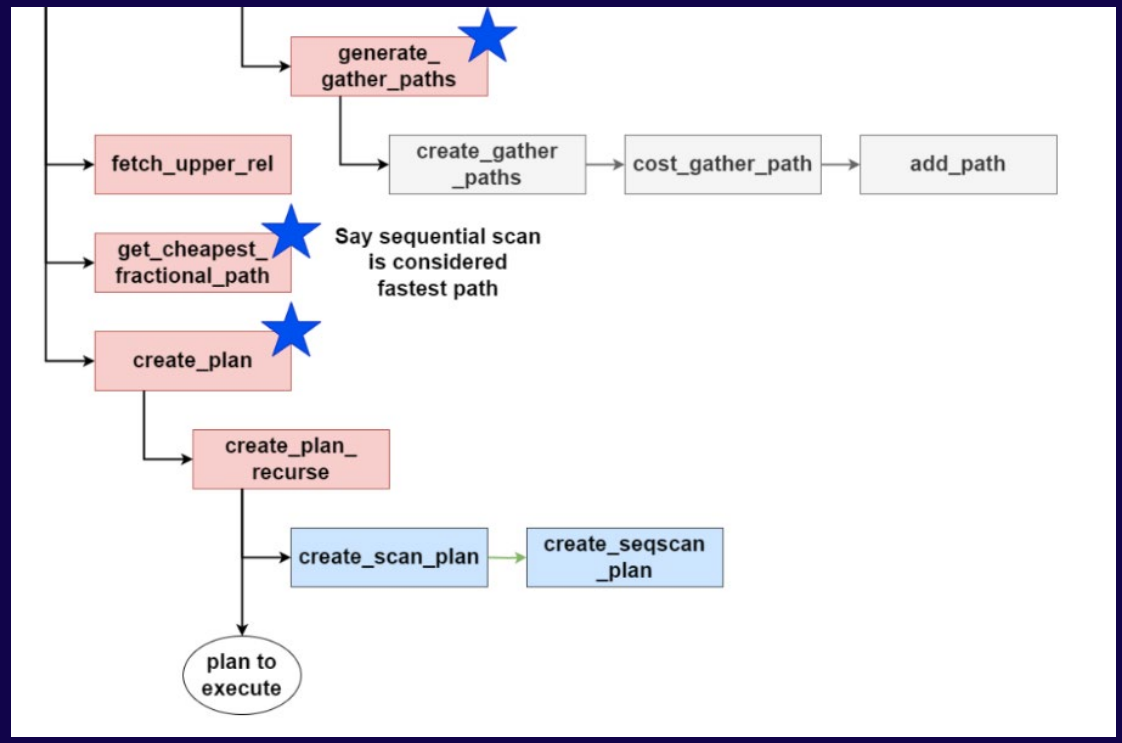
- Join 顺序
- Join 方法， （nested loop / hash / merge）
- 组合数量爆炸！

怎么解决？

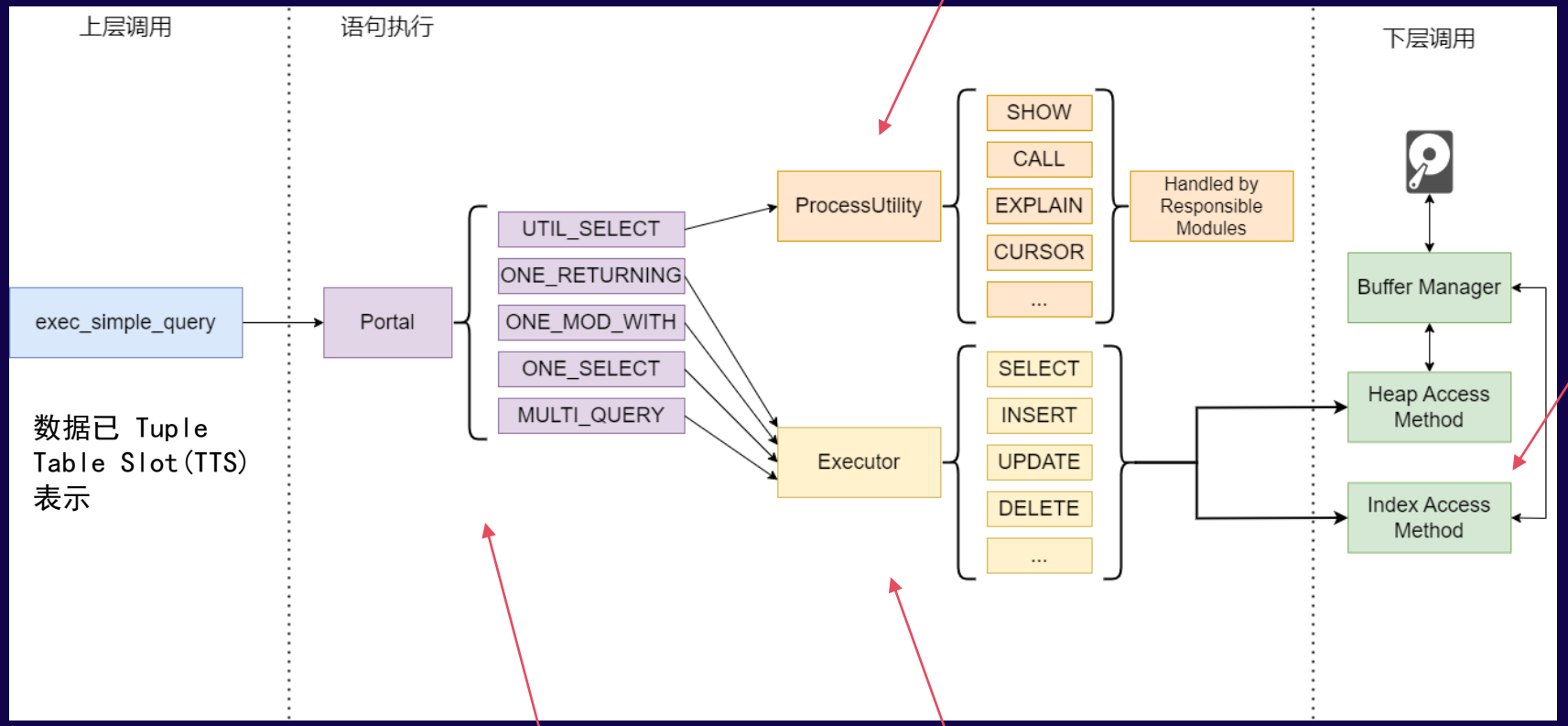
- 使用动态规则
- 保留每个阶段的最优路径
- 丢弃明显更差方案，只留 ‘有希望的’

PostgreSQL 查询优化器 callstack





Portal – 策略选择模块



Utility 命令交给 ProcessUtility 处理

数据已 Tuple Table Slot(TTS) 表示

处理 DDL 和 DML 时最终都会交给相对应的 access method 处理（数据已 Heap Tuple 表示）

Portal 模块决定如何处理该语句，在代码里叫做 Portal Strategy

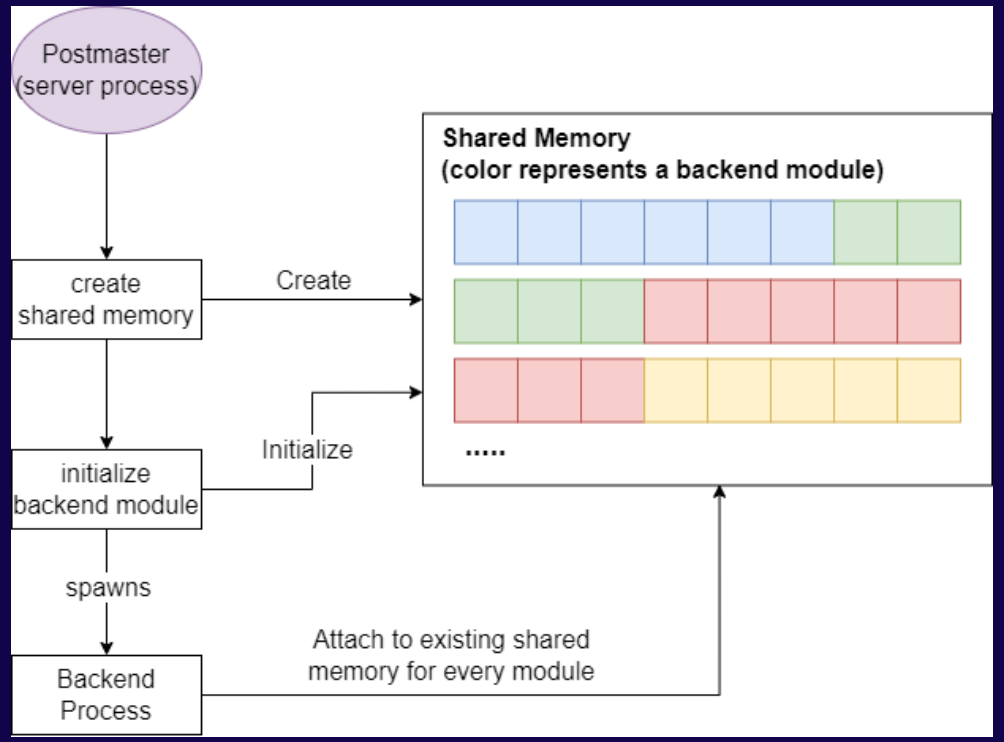
DML, DDL 交给 Executor 处理

后台模块初始化 – 流程

- Postmaster 主进程负责：
  - 创建共享内存。
  - 后台模块的初始化 （buffer manager，lock manager 等等）。
  - 启动后台进程 （background writer，checkpointer 等等）。
  - 后台进程和模块的概念是不一样的。
  - 主要入口函数 PostmasterMain（）- src/backend/postmaster/postmaster.c

- 后台模块的初始化：
  - 通常指的是共享内存的初始化 + 模块自身初始化。
  - 通常在 postmaster 启动的时候做一次。

- 后台进程的启动：
  - 此时，后台模块应当已经被初始化。
  - 进程启动后直接 attach 上已存在的共享内存即可。



数据库集群初始化 – bootstrap process

- 数据库集群初始化只在 initdb 时做一次。
- 后台模块初始化是在启动数据库 （pg_ctl -D $PGDATA start）时做一次。每次重启都做一次。两者是不同的。
- 集群的初始化在代码里被叫做 bootstrap process：
  - 入口函数： BootStrapXLOG() – src/backend/access/transam/xlog.c。
  - PostgreSQL 在这个函数里初始化 control 文件，WAL 文件，commit log， 等等。
  - 任何集群级别的东西都可以在这里做初始化。

- 假设我们想要增加一个加密 WAL 文件的功能，这个会影响到整个集群，所以在 bootstrap 的时候，我们可以把用户提供的加密密钥，在 bootstrap WAL 的时候做加密，并创建新目录储存密钥等等。

```c
/*
 * This func must be called ONCE on system install.  It creates pg_control
 * and the initial XLOG segment.
 */
void
BootStrapXLOG(void)
{
    CheckPoint  checkPoint;
    char       *buffer;
    XLogPageHeader page;
    XLogLongPageHeader longpage;
    XLogRecord *record;
    char       *recptr;
    uint64      sysidentifier;
    struct timeval tv;
    pg_crc32c   crc;

    /* allow ordinary WAL segment creation, like StartupXLOG() would */
    LWLockAcquire(ControlFileLock, LW_EXCLUSIVE);
    XLogCtl->InstallXLogFileSegmentActive = true;
    LWLockRelease(ControlFileLock);
```

```c
    /* Now create pg_control */
    InitControlFile(sysidentifier);
    ControlFile->time = checkPoint.time;
    ControlFile->checkPoint = checkPoint.redo;
    ControlFile->checkPointCopy = checkPoint;

    /* some additional ControlFile fields are set in WriteControlFile() */
    WriteControlFile();

    /* Bootstrap the commit log, too */
    BootStrapCLOG();
    BootStrapCommitTs();
    BootStrapSUBTRANS();
    BootStrapMultiXact();

    pfree(buffer);

    /*
     * Force control file to be read - in contrast to normal processing we'd
     * otherwise never run the checks and GUC related initializations therein.
     */
    ReadControlFile();
```

注册自定义 PostgreSQL 后台进程

定义进程注册函数

- 声明 BackgroundWorker 结构。
- 配置 BackgroundWorker 结构。
- 指定进程入口函数名。
- 定义进程 ps 显示名。
- 注册进程。

```c
/* Register a background worker running the foreign transaction launcher */
void
FdwXactLauncherRegister(void)
{
    BackgroundWorker bgw;
    memset(&bgw, 0, sizeof(bgw));
    bgw.bgw_flags = BGWORKER_SHMEM_ACCESS |
        BGWORKER_BACKEND_DATABASE_CONNECTION;
    bgw.bgw_start_time = BgWorkerStart_RecoveryFinished;
    snprintf(bgw.bgw_library_name, BGW_MAXLEN, "postgres");
    snprintf(bgw.bgw_function_name, BGW_MAXLEN, "FdwXactLauncherMain");
    snprintf(bgw.bgw_name, BGW_MAXLEN,
             "foreign transaction resolver");
    snprintf(bgw.bgw_type, BGW_MAXLEN,
             "foreign transaction resolver");
    bgw.bgw_restart_time = 5;
    bgw.bgw_notify_pid = 0;
    bgw.bgw_main_arg = (Datum) 0;

    RegisterBackgroundWorker(&bgw);
}
```

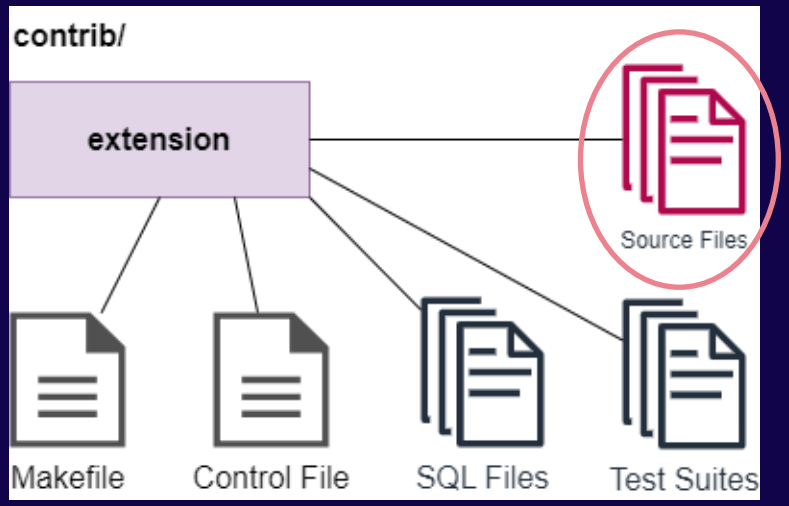
构建后台进程入口函数逻辑

定义入口函数：主逻辑循环

- 定义 shutdown 清理函数。
- 注册 signal handler - SIGHUP, SIGUSR2。
- 初始化 postgres。
- 设置主循环 WaitLatch 等待时间。
- 处理 postgres 内部 waitEvent。

```
void
FdwXactLauncherMain(Datum main_arg)
{
    TimestampTz last_start_time = 0;

    ereport(DEBUG2,
            (errmsg("fdwxact resolver started")));

    before_shmem_exit(fdwxact_launcher_onexit, (Datum) 0);

    pqsignal(SIGHUP, fdwxact_launcher_sighup);
    pqsignal(SIGUSR2, fdwxact_launcher_sigusr2);
    pqsignal(SIGTERM, die);
    BackgroundWorkerUnblockSignals();
    BackgroundWorkerInitializeConnection(NULL, NULL, 0);

    /* Enter main loop */
    for (;;)
    {
        int         rc;
        TimestampTz now = 0;
        long        wait_time = DEFAULT_NAPTIME_PER_CYCLE;


        CHECK_FOR_INTERRUPTS();
        ResetLatch(MyLatch);

        now = GetCurrentTimestamp();
        if (TimestampDifferenceExceeds(last_start_time, now, (fdw_xact_check_interval * 1000)))
        {
            last_start_time = now;
            ereport(WARNING,
                    (errmsg("%s: checking orphaned prepared transactions", __FUNCTION__)));
            TimeToCheckFdwXactOnGTM();
        }

        wait_time = (fdw_xact_check_interval * 1000);

        /* Wait for fdw_xact_check_interval timeout */
        rc = WaitLatch(MyLatch,
                       WL_LATCH_SET | WL_TIMEOUT | WL_POSTMASTER_DEATH,
                       wait_time,
                       WAIT_EVENT_FDWXACT_LAUNCHER_MAIN);

        if (rc & WL_POSTMASTER_DEATH)
            proc_exit(1);

        if (rc & WL_LATCH_SET)
        {
            ResetLatch(MyLatch);
            CHECK_FOR_INTERRUPTS();
        }

        if (got_SIGHUP)
        {
            got_SIGHUP = false;
            ProcessConfigFile(PGC_SIGHUP);
        }
```

基本插件架构

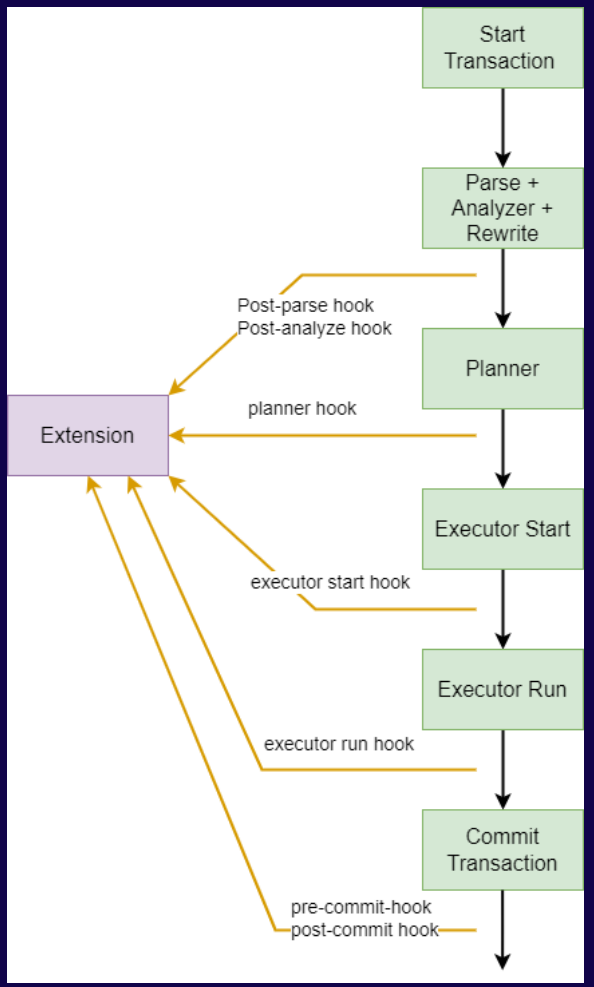



```c
#include "postgres.h"
#include "fmgr.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(get_sum);

Datum
get_sum(PG_FUNCTION_ARGS)
{
    bool isnull, isnull2;
    int a = 0, b = 0, sum = 0;

    isnull = PG_ARGISNULL(0);
    isnull2 = PG_ARGISNULL(1);
    if (isnull || isnull2)
      ereport( ERROR,
               ( errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
               errmsg("the input must be two integers")));

    a = PG_GETARG_INT32(0);
    b = PG_GETARG_INT32(1);
    sum = a + b;

    PG_RETURN_INT32(sum);
}
```

使用 hooks 参与内核语句处理流程

- 插件还可以通过 hooks （回调函数）来参与 PostgreSQL 内核的逻辑处理。
- PostgreSQL 提供了很多不同的 hook 函数原型。可以分为这几类：
  - General hooks：　一般 PostgreSQL 功能 hooks。
  - Security hooks：　安全相关 hooks（例如密码和用户创建）。
  - Function hooks：　在函数执行期间的 hooks。
  - Planner hooks：　规划器阶段的拦截更改执行计划的 hooks。
  - Executor hooks：　执行阶段的拦截 hooks。

- 以插件为基础的分布式数据库产品，像是 Citus，就是通过大量的 hooks 和 PostgreSQL 相互配合来实现分布式的功能。



使用 hooks 参与内核语句处理流程

- 如果一个插件使用了 hooks，那您必须还需要定义 _PG_init() 初始化函数。

```
void _PG_init(void)
```

- 在 _PG_init（）内，把需要的 hooks 指向到自定义的函数里。
- 我们不需要定义 _PG_fini()。Hook 的卸载 PostgreSQL 内部会自动处理。

```c
//Initialise hook variable
static ClientAuthentication_hook_type original_client_auth_hook = NULL;

//Hook method
static void auth_delay_checks(Port *port, int status)
{

    if (original_client_auth_hook)
        original_client_auth_hook(port, status);


    if (status != STATUS_OK)
    {
        pg_usleep(1000000L);
    }
}

//The custom hook method is called in the global init
void _PG_init(void)
{
    original_client_auth_hook = ClientAuthentication_hook;
    ClientAuthentication_hook = auth_delay_checks;
}
```

自定义的 auth hook

自定义的逻辑，如果校验失败，sleep 一秒

新增 _PG_init 函数

关于用户校验的 hook

把 hook 指向到自定义函数

Datum 抽象数据类型

- src/include/fmgr.h 定义了许多操作 Datum 的 MACRO。
- 根据所需的数据类型，我们都可以利用以下 MACRO 做 Datum 转换。
- 适用于 C 函数的编写。

把一个数据类型转变成 Datum 并 return

```
/* Macros for returning results of standard types */

#define PG_RETURN_DATUM(x)    return (x)
#define PG_RETURN_INT32(x)    return Int32GetDatum(x)
#define PG_RETURN_UINT32(x)   return UInt32GetDatum(x)
#define PG_RETURN_INT16(x)    return Int16GetDatum(x)
#define PG_RETURN_UINT16(x)   return UInt16GetDatum(x)
#define PG_RETURN_CHAR(x)     return CharGetDatum(x)
#define PG_RETURN_BOOL(x)     return BoolGetDatum(x)
#define PG_RETURN_OID(x)      return ObjectIdGetDatum(x)
#define PG_RETURN_POINTER(x)  return PointerGetDatum(x)
#define PG_RETURN_CSTRING(x)  return CStringGetDatum(x)
#define PG_RETURN_NAME(x)     return NameGetDatum(x)
#define PG_RETURN_TRANSACTIONID(x)  return TransactionIdGetDatum(x)
/* these macros hide the pass-by-reference-ness of the datatype: */
#define PG_RETURN_FLOAT4(x)   return Float4GetDatum(x)
#define PG_RETURN_FLOAT8(x)   return Float8GetDatum(x)
#define PG_RETURN_INT64(x)    return Int64GetDatum(x)
#define PG_RETURN_UINT64(x)   return UInt64GetDatum(x)
/* RETURN macros for other pass-by-ref types will typically look like this: */
#define PG_RETURN_BYTEA_P(x)     PG_RETURN_POINTER(x)
#define PG_RETURN_TEXT_P(x)      PG_RETURN_POINTER(x)
#define PG_RETURN_BPCHAR_P(x)    PG_RETURN_POINTER(x)
#define PG_RETURN_VARCHAR_P(x)   PG_RETURN_POINTER(x)
#define PG_RETURN_HEAPTUPLEHEADER(x)  return HeapTupleHeaderGetDatum(x)
```

。。。。

- 许多 PG 提供的内核 API 函数都是已 Datum 形式传递信息。要特别注意！

把函数 argument 的 Datum 转变成指定数据类型

```
/* Macros for fetching arguments of standard types */

#define PG_GETARG_DATUM(n)    (fcinfo->args[n].value)
#define PG_GETARG_INT32(n)    DatumGetInt32(PG_GETARG_DATUM(n))
#define PG_GETARG_UINT32(n)   DatumGetUInt32(PG_GETARG_DATUM(n))
#define PG_GETARG_INT16(n)    DatumGetInt16(PG_GETARG_DATUM(n))
#define PG_GETARG_UINT16(n)   DatumGetUInt16(PG_GETARG_DATUM(n))
#define PG_GETARG_CHAR(n)     DatumGetChar(PG_GETARG_DATUM(n))
#define PG_GETARG_BOOL(n)     DatumGetBool(PG_GETARG_DATUM(n))
#define PG_GETARG_OID(n)      DatumGetObjectId(PG_GETARG_DATUM(n))
#define PG_GETARG_POINTER(n)  DatumGetPointer(PG_GETARG_DATUM(n))
#define PG_GETARG_CSTRING(n)  DatumGetCString(PG_GETARG_DATUM(n))
#define PG_GETARG_NAME(n)     DatumGetName(PG_GETARG_DATUM(n))
#define PG_GETARG_TRANSACTIONID(n)  DatumGetTransactionId(PG_GETARG_DATUM(n))
/* these macros hide the pass-by-reference-ness of the datatype: */
#define PG_GETARG_FLOAT4(n)   DatumGetFloat4(PG_GETARG_DATUM(n))
#define PG_GETARG_FLOAT8(n)   DatumGetFloat8(PG_GETARG_DATUM(n))
#define PG_GETARG_INT64(n)    DatumGetInt64(PG_GETARG_DATUM(n))
/* use this if you want the raw, possibly-toasted input datum: */
#define PG_GETARG_RAW_VARLENA_P(n)  ((struct varlena *) PG_GETARG_POINTER(n))
/* use this if you want the input datum de-toasted: */
#define PG_GETARG_VARLENA_P(n) PG_DETOAST_DATUM(PG_GETARG_DATUM(n))
/* and this if you can handle 1-byte-header datums: */
#define PG_GETARG_VARLENA_PP(n) PG_DETOAST_DATUM_PACKED(PG_GETARG_DATUM(n))
```

。。。。

SQL 函数和 Tuple Descriptor

什么是 SQL 函数?

- 用 SQL 语句定义的函数，能传递参数也能返回值：
  - 标量（scalar）
  - 单行（record）
  - 多行（setof）

```
CREATE FUNCTION get_user_name(uid int)
RETURNS text AS $$
SELECT name FROM users WHERE id = uid;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION get_all_users()
RETURNS SETOF users AS $$
  SELECT * FROM users;
$$ LANGUAGE sql;
```

什么是 Tuple Descriptor?

- 本质 = 描述一行数据的结构
- 包含：
  - 列数（natts）
  - 每列又包含：类型（type OID），名字（attname），长度（typlen），精度（typmod）。。。等
- Executor 用它来描述一个 Tuple （一行数据）
- 函数返回 record/setoff 时必须提供
- 决定返回数据长什么样子

例子： pg_visibility – 返回 setof

- SQL 函数也可以回复多行数据 （tuples）。
- 就像 SELECT 一个表，返回多行数据一样。
- 这需要使用到 Set Returning Functions （SRF） API接口 （src/include/funcapi.h）。
- pg_visibility_rel 会返回指定表的每一个数据页的可见性信息。
- SRF 接口会调用 pg_visibility_rel 直到所有的页都被处理了。

判断是不是第一次被 SRF 调用

第一次会初始化 SRF

初始化 tuple descriptor，包含了每一列的名字

user_fctx 可以用来存储任何用户的数据，可以在后续函数调用时访问

调出第一次初始化的 funcctx* 指针并取出用户数据 user_fctx

把填好的 values 和 null array 根据 tuple descriptor 造出 tuple

告诉 SRF 我们已经处理所有的 tuple了，可以返回结果给用户了

```
Datum
pg_visibility_rel(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    vbits      *info;

    if (SRF_IS_FIRSTCALL())
    {
        Oid         relid = PG_GETARG_OID(0);
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        funcctx->tuple_desc = pg_visibility_tupdesc(true, true);
        /* collect_visibility_data will verify the relkind */
        funcctx->user_fctx = collect_visibility_data(relid, true);
        MemoryContextSwitchTo(oldcontext);
    }

    funcctx = SRF_PERCALL_SETUP();
    info = (vbits *) funcctx->user_fctx;

    if (info->next < info->count)
    {
        Datum       values[4];
        bool        nulls[4] = {0};
        HeapTuple   tuple;

        values[0] = Int64GetDatum(info->next);
        values[1] = BoolGetDatum((info->bits[info->next] & (1 << 0)) != 0);
        values[2] = BoolGetDatum((info->bits[info->next] & (1 << 1)) != 0);
        values[3] = BoolGetDatum((info->bits[info->next] & (1 << 2)) != 0);
        info->next++;

        tuple = heap_form_tuple(funcctx->tuple_desc, values, nulls);
        SRF_RETURN_NEXT(funcctx, HeapTupleGetDatum(tuple));
    }

    SRF_RETURN_DONE(funcctx);
}
```

如果有空值，也要填上这个 nulls array

赋值当前行的数据到 Datum array (values)

返回 tuple

例子： pg_visibility – tuple descriptor

- pg_visibility 会根据表的大小（页的多少）回复相对应的行数。
- 上页提到的 tuple descriptor 告诉 PostgreSQL 一个 tuple 有多少栏位，什么数据类型，叫什么名字等等。
- 测试语句如下：

```
postgres=# create extension pg_visibility;
CREATE EXTENSION
postgres=# create table t3(a int, b int);
CREATE TABLE
postgres=# insert into t3 values(generate_series(1,500),0);
INSERT 0 500
postgres=# select * from pg_visibility('t3'::regclass);
 blkno | all_visible | all_frozen | pd_all_visible
-------+-------------+------------+----------------
     0 | f           | f          | f
     1 | f           | f          | f
     2 | f           | f          | f
(3 rows)
```

初始化 tuple descriptor

定义每一个栏位的属性

回复 tuple descriptor

```
static TupleDesc
pg_visibility_tupdesc(bool include_blkno, bool include_pd)
{
    TupleDesc   tupdesc;
    AttrNumber  maxattr = 2;
    AttrNumber  a = 0;

    if (include_blkno)
        ++maxattr;
    if (include_pd)
        ++maxattr;
    tupdesc = CreateTemplateTupleDesc(maxattr);
    if (include_blkno)
        TupleDescInitEntry(tupdesc, ++a, "blkno", INT8OID, -1, 0);
    TupleDescInitEntry(tupdesc, ++a, "all_visible", BOOLOID, -1, 0);
    TupleDescInitEntry(tupdesc, ++a, "all_frozen", BOOLOID, -1, 0);
    if (include_pd)
        TupleDescInitEntry(tupdesc, ++a, "pd_all_visible", BOOLOID, -1, 0);
    Assert(a == maxattr);

    return BlessTupleDesc(tupdesc);
}
```

定义 attribute number，从 1 开始，它决定了栏位排序的顺序，数字越大显示的越后

第三章：事务管理和可见性

什么是 MVCC（Multi-Version Concurrency Control）

- MVCC 也叫多版本并发控制， 也是 PostgreSQL 核心机制之一，它通过为数据维护多个版本，使读操作无需加锁即可获取一致性视图，从而实现高并发下的事务隔离与性能优化。

- 每一次写操作都会创建该数据的“新版本”，同时保留“旧版本”。
- 允许并发读和写操作，且不互相阻塞。
- PostgreSQL 使用的是 MVCC 的变体，也叫 Snapshot Isolation （快照隔离）
- 当事务要做读操作时，PostgreSQL 系统会选择其中一个版本来确保事务的隔离性。
  - 这个动作也叫“可见性判断”
  - 利用不同时期获取的“快照”，加上其它指标推算出某一数据的版本究竟可不可见。

事务号 – Transaction ID （ txid ）

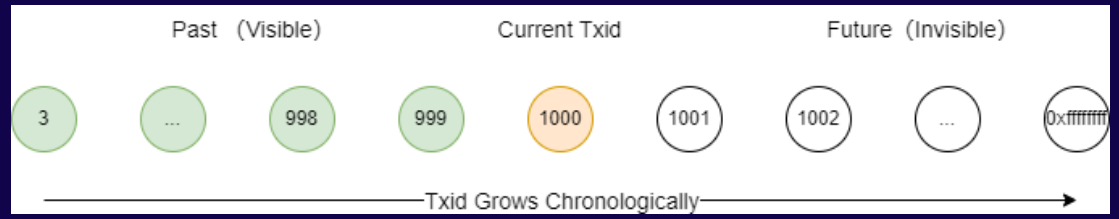
- 是一个唯一且环绕的 32 位无符号整数类型。
- 并不是所有事务，或者语句都会产生新的事务号。
- 在事务 BEGIN 的时候，只有虚拟事务号被产生，目的是用来对只读语句来上锁的。
- 当一个事务开始做写操作时，真正的事务号才会被产生。
- 事务号在可见性判断和死锁检测扮演者非常重要的角色。
- PostgreSQL 社区有关于 64 位元的 Transaction ID 的讨论，和为什么实现起来很困难。可以参考这里。

```
/* ----------------
 *      Special transaction ID values
 *
 * BootstrapTransactionId is the XID for "bootstrap" operations, and
 * FrozenTransactionId is used for very old tuples.  Both should
 * always be considered valid.
 *
 * FirstNormalTransactionId is the first "normal" transaction id.
 * Note: if you need to change it, you must change pg_class.h as well.
 * ----------------
 */
#define InvalidTransactionId        ((TransactionId) 0)
#define BootstrapTransactionId      ((TransactionId) 1)
#define FrozenTransactionId         ((TransactionId) 2)
#define FirstNormalTransactionId    ((TransactionId) 3)
#define MaxTransactionId            ((TransactionId) 0xFFFFFFFF)
```

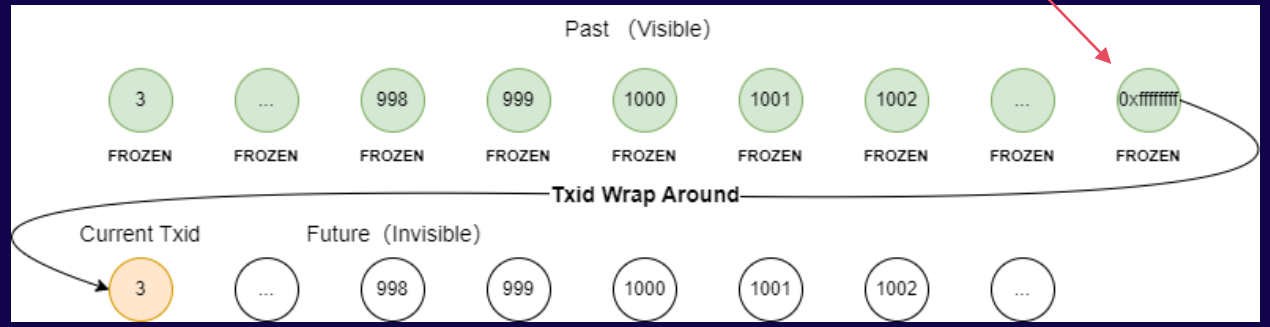
无效 txid

Bootstrap （initdb）时使用的 txid

冻结的 txid - txid 环绕时对旧数据的特殊标识



txid 环绕时，VACUUM 进程会把旧数据都标记成 FROZEN。FROZEN 的数据应当可见



事务号分配

- 当一个事务开始做写操作时，真正的事务号才会被产生。
- Heap Access Method 是主要触发事务号分配的模块：
  - src/backend/access/heap/heapam.c。
  - heap_insert()， heap_delete()。
  - heap_update()，… 等等。

```c
void
heap_insert(Relation relation, HeapTuple tup, CommandId cid,
            int options, BulkInsertState bistate)
{
    TransactionId xid = GetCurrentTransactionId();
    HeapTuple   heaptup;
    Buffer      buffer;
    Buffer      vmbuffer = InvalidBuffer;
    bool        all_visible_cleared = false;

    /* Cheap, simplistic check that the tuple matches the rel's rowtype. */
    Assert(HeapTupleHeaderGetNatts(tup->t_data) <=
           RelationGetNumberOfAttributes(relation));

    /*
     * Fill in tuple header fields and toast the tuple if necessary.
     *
     * Note: below this point, heaptup is the data we actually intend to store
     * into the relation; tup is the caller's original untoasted data.
     */
    heaptup = heap_prepare_insert(relation, tup, xid, cid, options);
```

获取新事务号

```c
/*
 *  GetCurrentTransactionId
 *
 * This will return the XID of the current transaction (main or sub
 * transaction), assigning one if it's not yet set.  Be careful to call this
 * only inside a valid xact.
 */
TransactionId
GetCurrentTransactionId(void)
{
    TransactionState s = CurrentTransactionState;

    if (!FullTransactionIdIsValid(s->fullTransactionId))
        AssignTransactionId(s);
    return XidFromFullTransactionId(s->fullTransactionId);
}
```

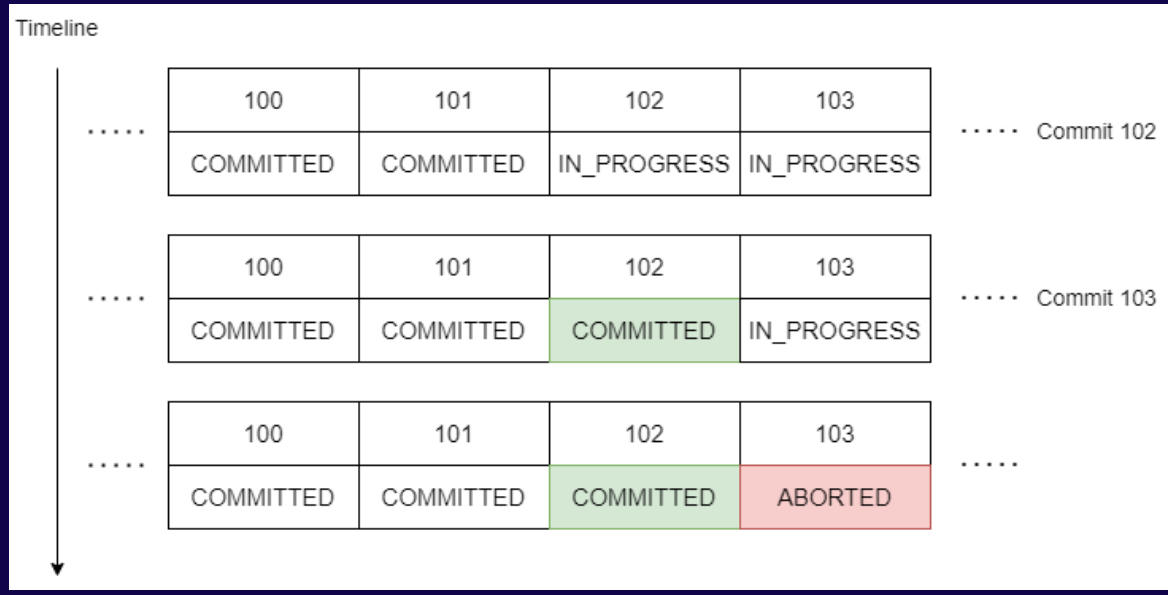
分配事务号

```c
/*
 * Subroutine for heap_insert(). Prepares a tuple for insertion. This sets the
 * tuple header fields and toasts the tuple if necessary.  Returns a toasted
 * version of the tuple if it was toasted, or the original tuple if not. Note
 * that in any case, the header fields are also set in the original tuple.
 */
static HeapTuple
heap_prepare_insert(Relation relation, HeapTuple tup, TransactionId xid,
                    CommandId cid, int options)
{
    /*
     * To allow parallel inserts, we need to ensure that they are safe to be
     * performed in workers. We have the infrastructure to allow parallel
     * inserts in general except for the cases where inserts generate a new
     * CommandId (eg. inserts into a table having a foreign key column).
     */
    if (IsParallelWorker())
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TRANSACTION_STATE),
                 errmsg("cannot insert tuples in a parallel worker")));

    tup->t_data->t_infomask &= ~(HEAP_XACT_MASK);
    tup->t_data->t_infomask2 &= ~(HEAP2_XACT_MASK);
    tup->t_data->t_infomask |= HEAP_XMAX_INVALID;
    HeapTupleHeaderSetXmin(tup->t_data, xid);
    if (options & HEAP_INSERT_FROZEN)
        HeapTupleHeaderSetXminFrozen(tup->t_data);

    HeapTupleHeaderSetCmin(tup->t_data, cid);
    HeapTupleHeaderSetXmax(tup->t_data, 0); /* for cleanliness */
    tup->t_tableOid = RelationGetRelid(relation);
```

事务号以 xmin 的形式存放在 Heap Tuple Header 里面。其它还有 xmax，cmin，cmax。这些会在后续的可见性章节解说

Commit Log (CLOG) 介绍

- CLOG 是存放在共享内存，以 8KB 页为存储单位的数组数据结构。
- 目的是快速调出一个事务号的提交状态：
  - IN_PROGRESS, COMMITTED, ABORTED, SUB_COMMITTED。
- 数组的索引就是事务号。
- 如果 8KB 页装满了，系统会再分配新的 8KB 页。
- 系统 shutdown 或是 checkpoint 进程在执行的时候，共享内存内的 CLOG 数据会被存放到 $PGDATA/pg_xact 目录下。下次系统启动会从这里再读进内存里。
- 并不是所有 CLOG 都需要被系统保留，Heap 数据上还有一个额外“hintbit”的标识，表征此数据的提交/回滚信息。
- VACUUM 负责把不需要的 CLOG 删除。



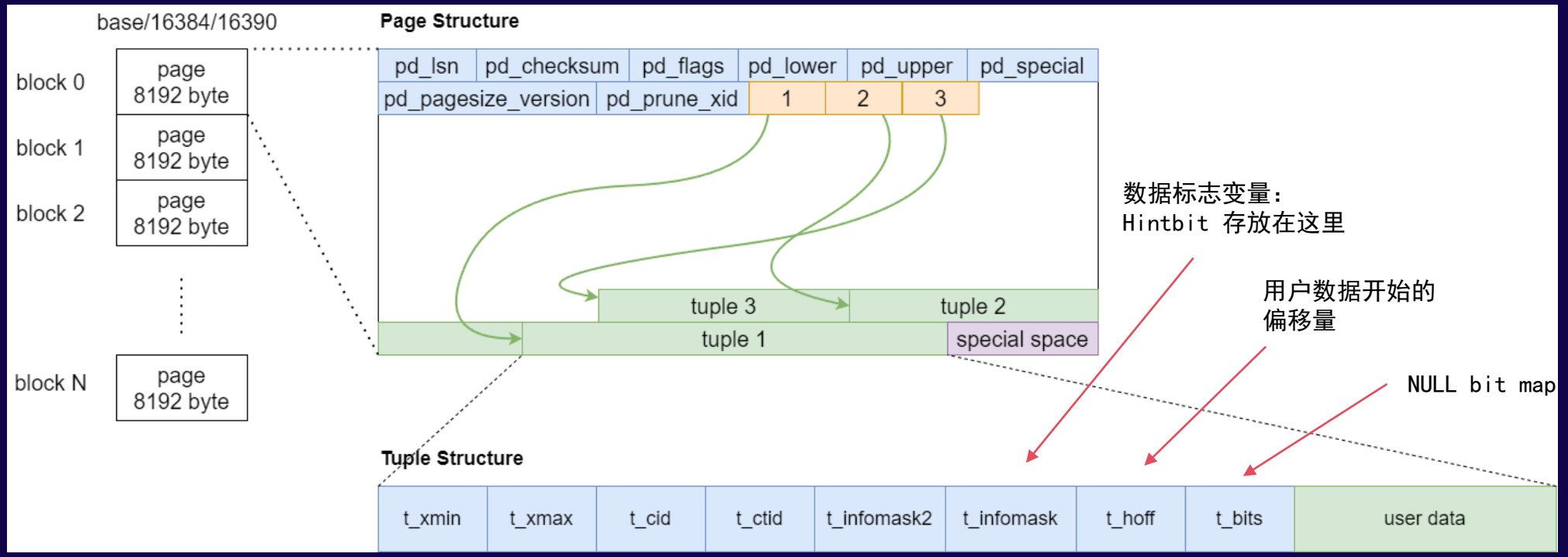
Hintbit 是什么？

- Hintbit 是储存在 Heap Tuple 的 t_infomask 的一个 flag 值。
- 可见性判断的过程可能会访问 CLOG 来查看相关的事务（xmin 和 xmax）的提交/回滚状态。
- 对每一条数据频繁的访问 CLOG 会有明显的性能问题。
- 访问 CLOG 的结果会以 hintbit 的形式储存在 t_infomask 里来达到缓存目的。
- 以后的可见性判断就不需要一直访问 CLOG，直接从 hintbit 里去读取。
- 可能的 hintbit 值：

```
#define HEAP_XMIN_COMMITTED     0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID       0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMAX_COMMITTED     0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID       0x0800 /* t_xmax invalid/aborted */
```

Heap Tuple 存储结构

- Heap Tuple 指的是实际存储在磁盘上的存储结构， 也是 Heap Access Method 在处理数据时主要使用的类型。



Transaction Snapshot（事务快照）是什么？

HOW 2026
Hello Open-source World

- Transaction Snapshot （或 Snapshot）是一个数据集，它储存在某个时间点处于活动状态的事务信息（Active Transaction）。
- PostgreSQL 内部以文本格式表征一个事务快照：

```
【xmin, xmax, xip_list】
```

  - xmin：仍处于活动状态的最小 Transaction ID - 小于这个值 = 事务已提交或回滚。
  - xmax：已提交或回滚的最大 Transaction ID + 1 - 大于或等于这个值 = 事务还未提交或回滚。
  - xip_list：正在进行中的事务号列表 - 应当大于 xmin 且小于 xmax。
- 例如：

```
postgres=# select pg_current_snapshot();
 pg_current_snapshot
---------------------
 747:750:747,748
(1 row)
```

- READ COMMITTED 模式的 Transaction Block，每一个 SELECT 语句都会请求新的 Snapshot
  - 所以可能读到其它事务提交的数据
- REPEATABLE READ 模式的 Transaction Block，在 BEGIN 的时候请求一次 Snapshot，不更新
  - 所以读不到其它事务提交的数据

Transaction Snapshot – 例子

Backend A

1）启动事务 747 – 尚未提交

```
postgres=# BEGIN;
Postgres*# INSERT INTO mytable VALUES('A');
Postgres*# SELECT txid_current();
 txid_current
--------------
          747
(1 row)
```

5）查看当前 Snapshot

```
Postgres*# select pg_current_snapshot();
 pg_current_snapshot
---------------------
 747:750:748
(1 row)
```

Backend B

2）启动事务 748 – 尚未提交

```
postgres=# BEGIN;
postgres=# INSERT INTO mytable VALUES('B');
postgres=# SELECT txid_current();
 txid_current
--------------
          748
(1 row)
```

6）查看当前 Snapshot

```
postgres=*# select pg_current_snapshot();
 pg_current_snapshot
---------------------
 747:750:747
(1 row)
```

Backend C

3）启动事务 749

```
postgres=# BEGIN;
Postgres*# INSERT INTO mytable VALUES('C');
Postgres*# SELECT txid_current();
 txid_current
--------------
          749
(1 row)
```

4）提交事务 749

```
Postgres*# COMMIT;
```

7）查看当前 Snapshot

```
postgres=# select pg_current_snapshot();
 pg_current_snapshot
---------------------
 747:750:747,748
(1 row)
```

Backend A

- txid 747 插入的数据可见 – 当前事务内 （747 < 750）
- txid 748 插入的数据还不可见 – 未提交
- txid 749 插入的数据可见 – 已提交

Backend B

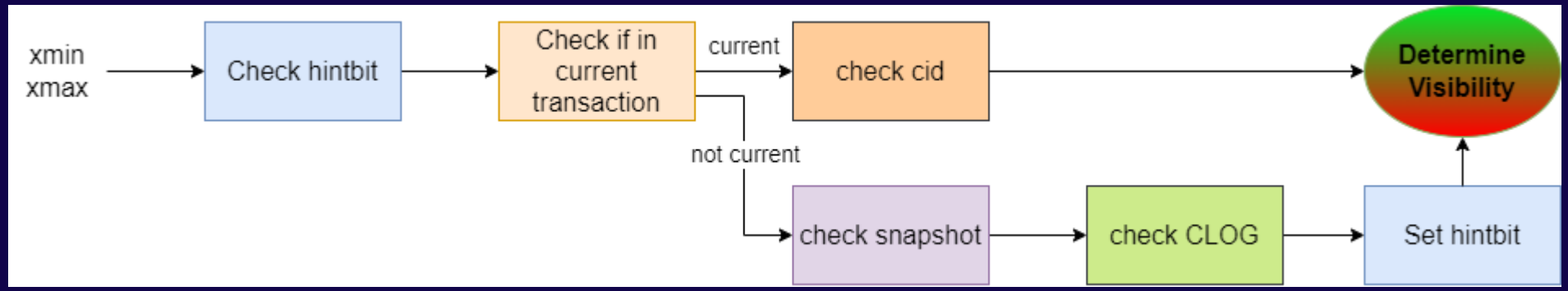
- txid 747 插入的数据不可见 – 未提交
- txid 748 插入的数据可见 – 当前事务内 （748 < 750）
- txid 749 插入的数据可见 – 已提交

Backend C

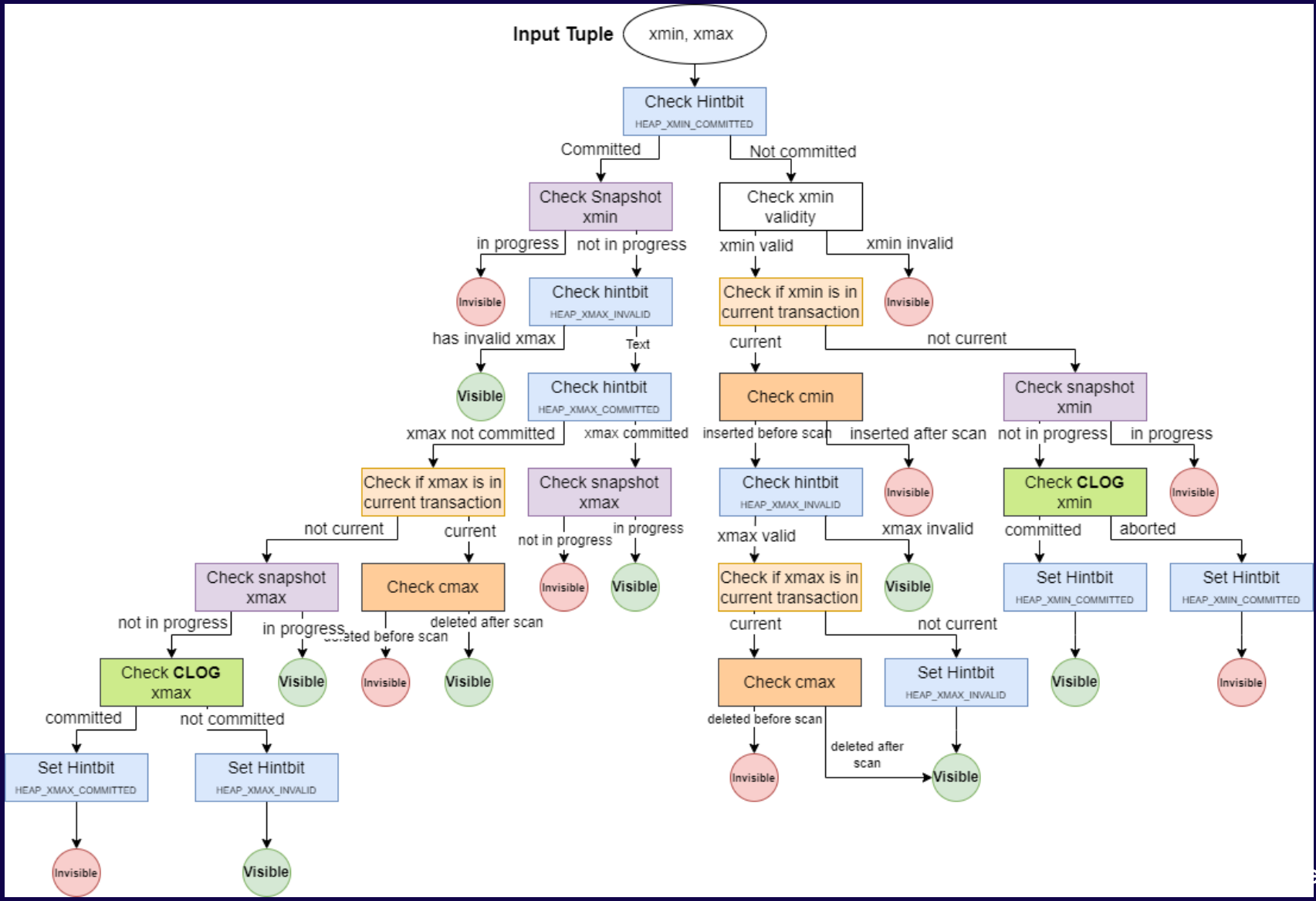
- txid 747 和 748 插入的数据都不可见 – 未提交
- txid 749 插入的数据可见 – 已提交

可见性判断 – 基本流程

- 影响可见性判断的主要指标：
  - Transaction ID, xmin, xmax, cid, snapshot, CLOG, hintbit。
- Heap Tuple 数据里带着 xmin, xmax, cid, 和 hintbit 的值。
- Snapshot 和当前 Transaction ID 由 Transaction Manager 提供。
- CLOG 信息存在共享内存里。
- 基本上会对 Heap Tuple 内的 xmin 和 xmax 做以下的可见性判断。



可见性判断 – 完整流程

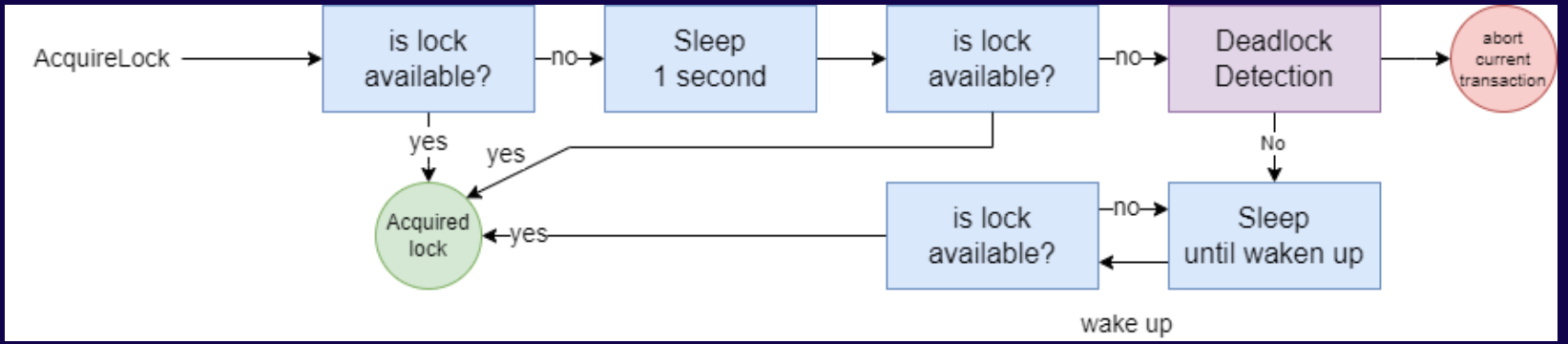

Input Tuple
xmin, xmax
Check Hintbit
HEAP_XMIN_COMMITTED
Committed
Not committed
Check Snapshot xmin
Check xmin validity
in progress
not in progress
xmin valid
xmin invalid
Invisible
Check hintbit
HEAP_XMAX_INVALID
Check if xmin is in current transaction
Invisible
has invalid xmax
Text
current
not current
Visible
Check hintbit
HEAP_XMAX_COMMITTED
Check cmin
Check snapshot xmin
xmax not committed
xmax committed
inserted before scan
inserted after scan
not in progress
in progress
Check if xmax is in current transaction
Check snapshot xmax
Check hintbit
HEAP_XMAX_INVALID
Check CLOG xmin
Invisible
not current
current
not in progress
in progress
xmax valid
xmax invalid
committed
aborted
Check snapshot xmax
Check cmax
Visible
Check if xmax is in current transaction
Set Hintbit
HEAP_XMIN_COMMITTED
not in progress
in progress
deleted before scan
deleted after scan
current
not current
Check CLOG xmax
Check cmax
Set Hintbit
HEAP_XMAX_INVALID
committed
not committed
deleted before scan
deleted after scan
Set Hintbit
HEAP_XMAX_COMMITTED
Set Hintbit
HEAP_XMAX_INVALID

PostgreSQL 死锁检测

- PostgreSQL 允许用户以任何顺序申请锁，所以死锁还是会发生。
- 对于死锁的设计思想是：当没有死锁时，希望锁的获取和释放尽可能快，以避免死锁检测 + 处理带来的额外开销。
- 采用一个“乐观等待”的方法，如果没有马上获取锁，设定一个定时器（DeadlockTimeout 默认1秒）然后进入睡眠。
- 此时系统会做一次‘死锁检查’

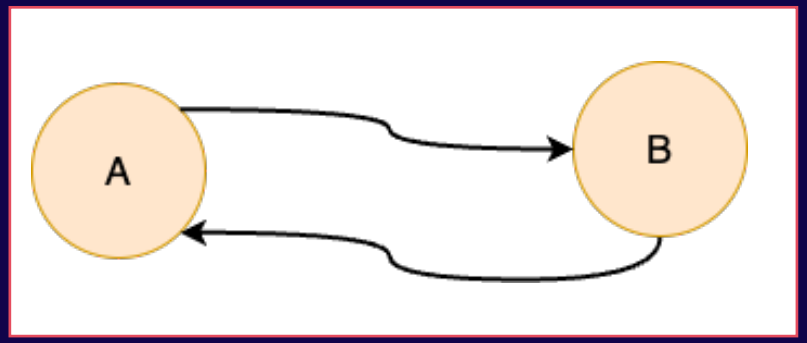
那问题来了：

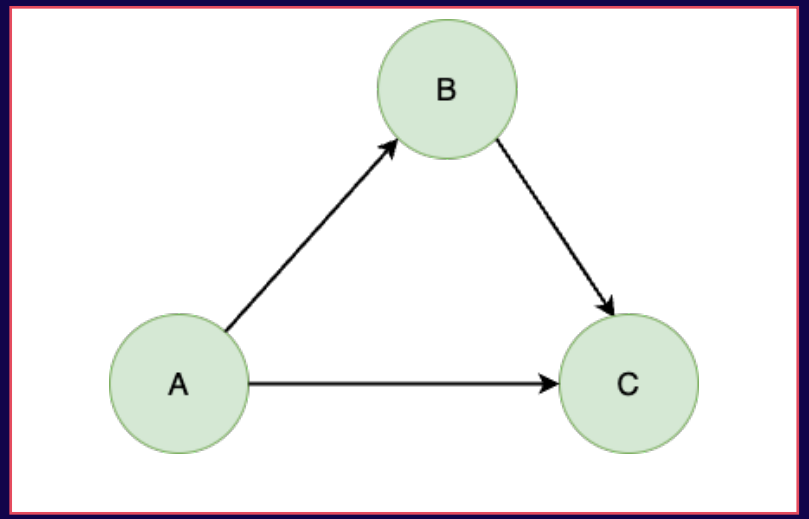
- PG 的死锁检测机制只能在同一个数据库服务下工作。
- 在一个分布式数据库情况下，一个跨界点的语句产生的死锁目前无法被检测到。

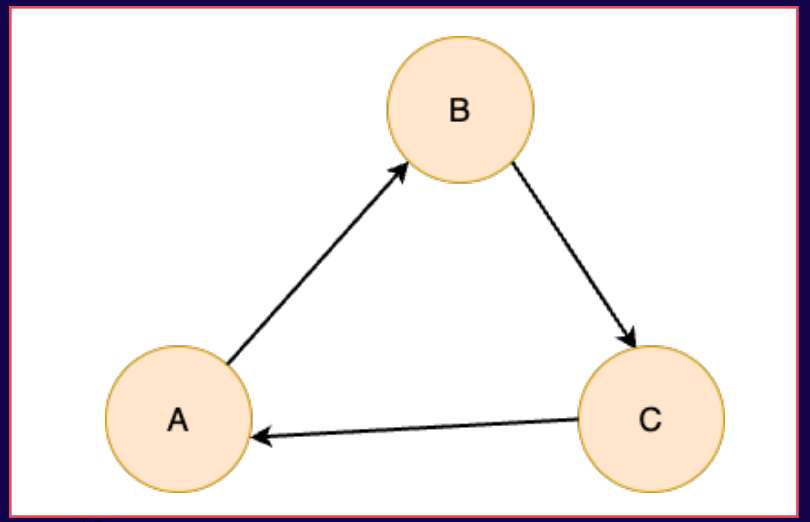


PostgreSQL 死锁检测

- 多个进程在执行过程中，如出现相互等待对方释放所持有的锁资源时，称之为死锁。
- 在没有其它进程的帮助下，这种等待会一直持续下去。
- 例子：
  - 节点 A，B，C，表示独立进程。
  - 箭头 A 到 B 表示 A 在等待 B 释放某个锁资源。
  - 有环路，表示死锁存在。







二段式提交 – Two Phase Commit（2PC)

- 使用 BEGIN 关键字定义一个“事务块”（transaction block）
- 输入需要执行的语句
- 使用 PREPARE TRANSACTION <name> 来结束事务块。
- 事务块的信息会被 PostgreSQL 储存在磁盘上，就算系统重启也没关系

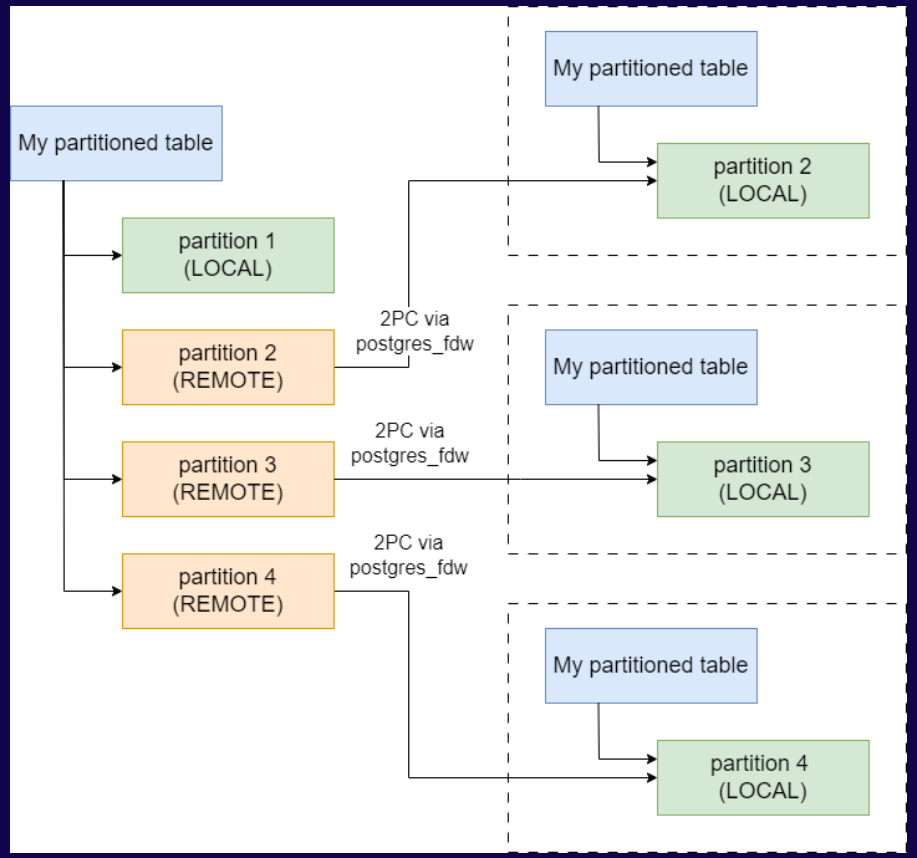
```
BEGIN;
CREATE TABLE test2 (a INT, b INT);
INSERT INTO test2 VALUES (123, 456);
INSERT INTO test2 VALUES (123, 456);
INSERT INTO test2 VALUES (123, 456);
PREPARE TRANSACTION 'mytransaction'
```

- 一段时间后，如果需要提交 ‘mytransaction’：

```
COMMIT PREPARED 'mytransaction';
```

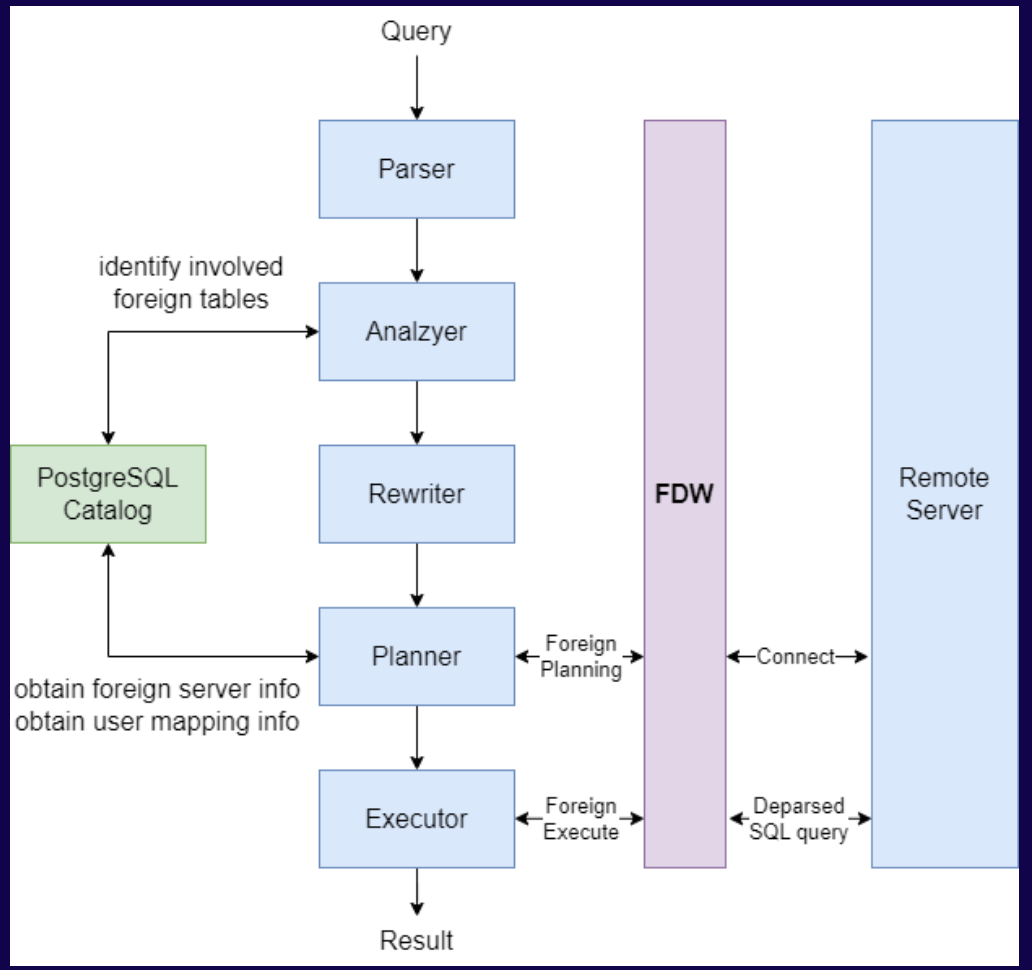
二段式提交和分布式数据库

- 分布式数据库指的是一个表的数据分布在一个或多个远端节点上（也叫 sharding）。
- 通常还配合 PostgreSQL 的分区表功能，把数据分成多个分区，有些分区数据存在本地，有些透过 postgres_fdw 插件存在远端。
- 基于 PostgreSQL 做分布式的一个典型问题就是没有办法保证数据在远端节点也提交成功。
  - 如果一个远端节点提交失败，那么其它远端节点都必须 Rollback，本地节点也要 Rollback。
  - 为了实现这个功能，我们必须改造 postgres_fdw 插件，让它支持使用 2PC 来协调所有远端节点。
  - 也必须要改造 Transaction Manager，让它在适当的时候对远端节点做 2PC。
  - 跨界点的 “死锁检测” 也是通过改造过的 Transaction Manager 来协调



FDW 基本架构

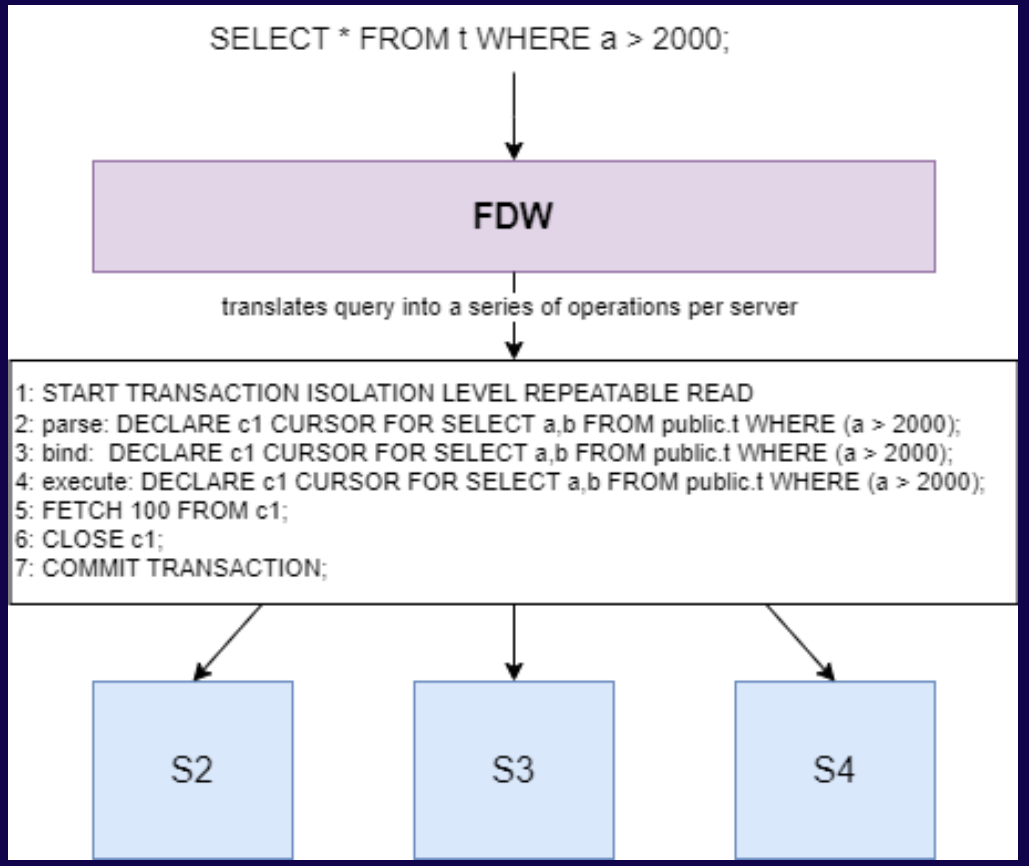
- Planner 和 Executor 是访问 FDW 最主要的模块
- 当表的类型为 RELKIND_FOREIGN_TABLE 时会触发 FDW 逻辑。
- **Planner：**
  - 通过 FDW 获取远程表的大小，远程表的执行计划。
  - 估算远程执行每一个计划的成本。
  - 最终选定最佳执行计划
- **Executor ：**
  - 通过 FDW 把原生语句翻译（Deparse）成远程表能处理的语句。
  - 获取远程表回复的结果。
  - 聚合最终结果。（透过合并其它远程表和本地表的结果）
- **FDW:**
  - 负责连接到所有参与的远程节点，并维持这些链接
  - 根据原生语句，翻译成相对应的远程语句



FDW 架构解析

- 发送语句并等待结果
  - 为确保数据的一致性，Deparse 后的语句并不会直接被发送到远程服务器上。
  - 它会被拆解成一系列的操作。
  - 这个操作运用到了 Extended Query Protocol，Cursor，Fetch 和 REPEATABLE READ 隔离级别

- 详细步骤：
  - 1：使用 REPEATABLE READ 隔离级别启动新事务块
  - 2，3，4：使用 Extended Query Protocol把语句用 CURSOR 包装然后分解成单独的步骤。
  - 5：FETCH 远端返回的数据直到没有数据了为止。
  - 6：释放 CURSOR。
  - 7：提交远程事务。



FDW 回调函数开发

- PostgreSQL 以插件的形式自带了两种 FDW：
  - postgres_fdw
  - file_fdw
- 构建一个新的 FDW，需要实现一个处理程序函数（Handler Function），以及一个验证函数（Validator Function）（可有可无）。
  - Handler Function - 返回一个回调函数指针结构
  - Validator Function - 用来检查传进 FDW 的参数（Options）是否正确。如果不提供此函数则不校验任何参数。
- 大部分的开发工作量都是在实现 Handler Function 里面回调函数。

postgres_fdw--1.0.sql

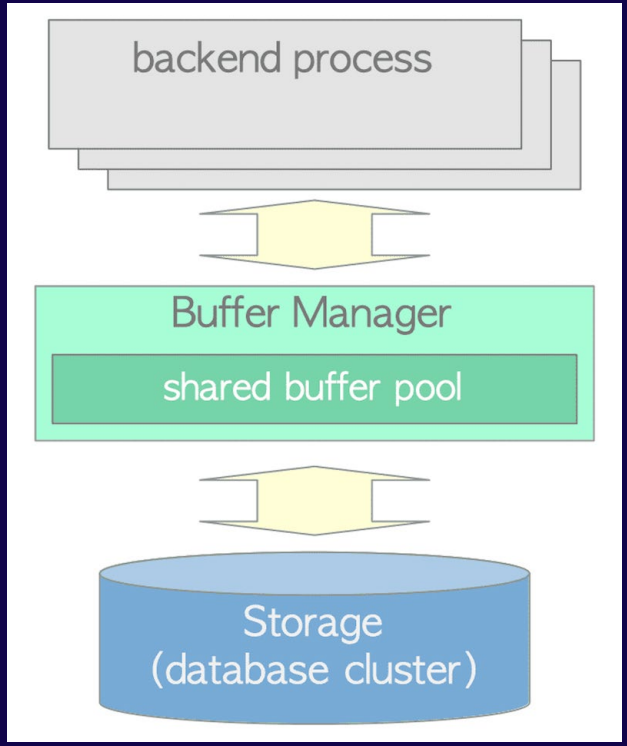
```
/* contrib/postgres_fdw/postgres_fdw--1.0.sql */

-- complain if script is sourced in psql, rather than via CREATE EXTENSION
\echo Use "CREATE EXTENSION postgres_fdw" to load this file. \quit

CREATE FUNCTION postgres_fdw_handler()
RETURNS fdw_handler
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FUNCTION postgres_fdw_validator(text[], oid)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FOREIGN DATA WRAPPER postgres_fdw
  HANDLER postgres_fdw_handler
  VALIDATOR postgres_fdw_validator;

```

```
/*
 * Foreign-data wrapper handler function: return a struct with pointers
 * to my callback routines.
 */
Datum
postgres_fdw_handler(PG_FUNCTION_ARGS)
{
    FdwRoutine *routine = makeNode(FdwRoutine);

    /* Functions for scanning foreign tables */
    routine->GetForeignRelSize = postgresGetForeignRelSize;
    routine->GetForeignPaths = postgresGetForeignPaths;
    routine->GetForeignPlan = postgresGetForeignPlan;
    routine->BeginForeignScan = postgresBeginForeignScan;
    routine->IterateForeignScan = postgresIterateForeignScan;
    routine->ReScanForeignScan = postgresReScanForeignScan;
    routine->EndForeignScan = postgresEndForeignScan;

    /* Functions for updating foreign tables */
    routine->AddForeignUpdateTargets = postgresAddForeignUpdateTargets;
    routine->PlanForeignModify = postgresPlanForeignModify;
    routine->BeginForeignModify = postgresBeginForeignModify;
    routine->ExecForeignInsert = postgresExecForeignInsert;
    routine->ExecForeignBatchInsert = postgresExecForeignBatchInsert;
    routine->GetForeignModifyBatchSize = postgresGetForeignModifyBatchSize;
    routine->ExecForeignUpdate = postgresExecForeignUpdate;
    routine->ExecForeignDelete = postgresExecForeignDelete;
    routine->EndForeignModify = postgresEndForeignModify;
    routine->BeginForeignInsert = postgresBeginForeignInsert;
    routine->EndForeignInsert = postgresEndForeignInsert;
    routine->IsForeignRelUpdatable = postgresIsForeignRelUpdatable;
    routine->PlanDirectModify = postgresPlanDirectModify;
    routine->BeginDirectModify = postgresBeginDirectModify;
    routine->IterateDirectModify = postgresIterateDirectModify;
    routine->EndDirectModify = postgresEndDirectModify;
```

第四章：数据存储和访问

数据缓冲管理器：buffer manager

- Buffer Manager 是 PostgresSQL 的核心模块，几乎所有的其他模块都需要直接或者间接的访问 Buffer Manager。
- Buffer Manager 管理共享内存提供缓冲数据，通过 Storage Manager 与存储设备进行数据交互。
- PostgreSQL 的 Buffer Manager 为其他模块提供了高效的数据块访问设计。
- Buffer Manager 对数据库的读写性能有至关重要的影响。



http://www.interdb.jp/pg/pgsql08.html

访问 buffer pool 中已有的数据块

(1) 后台进程提供构建 buffer tag 的基本信息：spcNode，dbNode，relNode，forkNum，blockNum

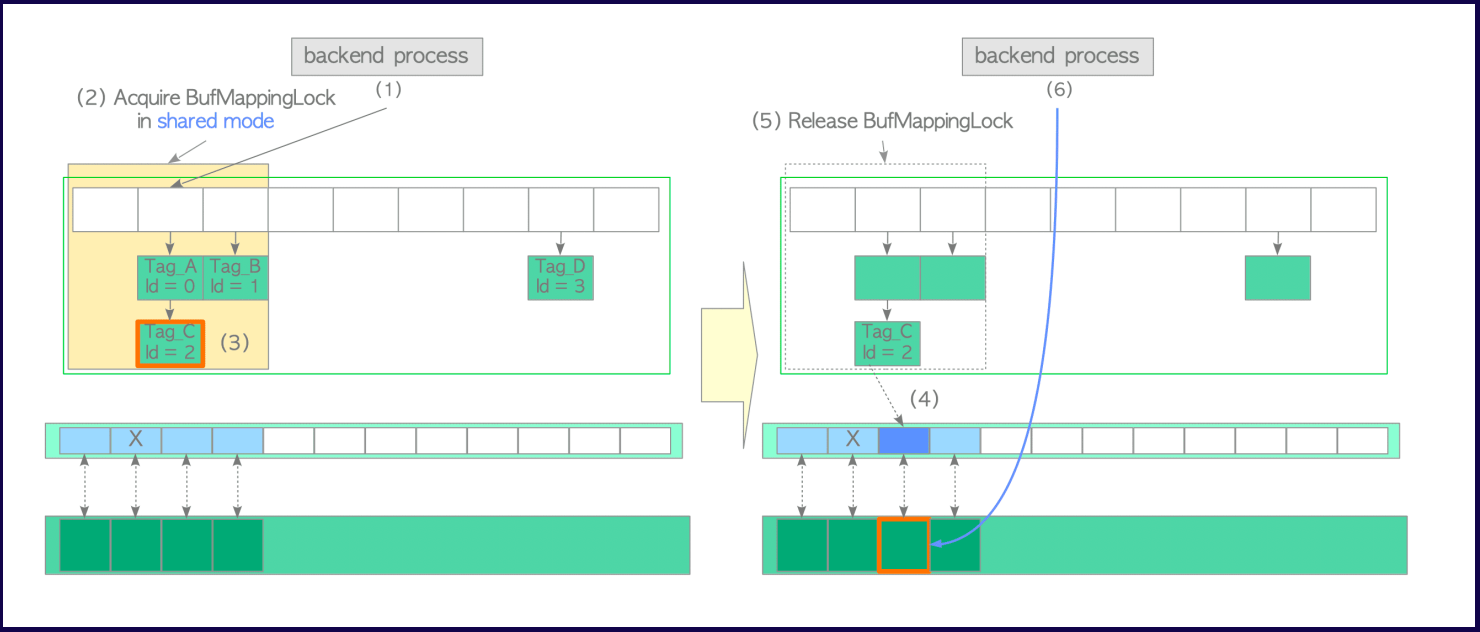
(2) 创建被访问数据块的 Buffer Tag，‘Tag_C’ - 使用哈希函数计算 'Tag_C' 对应的哈希 bucket。

(3) 以共享模式获取该哈希 bucket 对应的 BufMappingPartitionLock 分区锁。

(4) 在 Buffer Table 中查找 'Tag_C' 对应的数据，并获取 buf_id = 2，固定/锁定 buf_id 为 2 的 buffer descriptor。

(5) 释放 BufMappingPartitionLock。

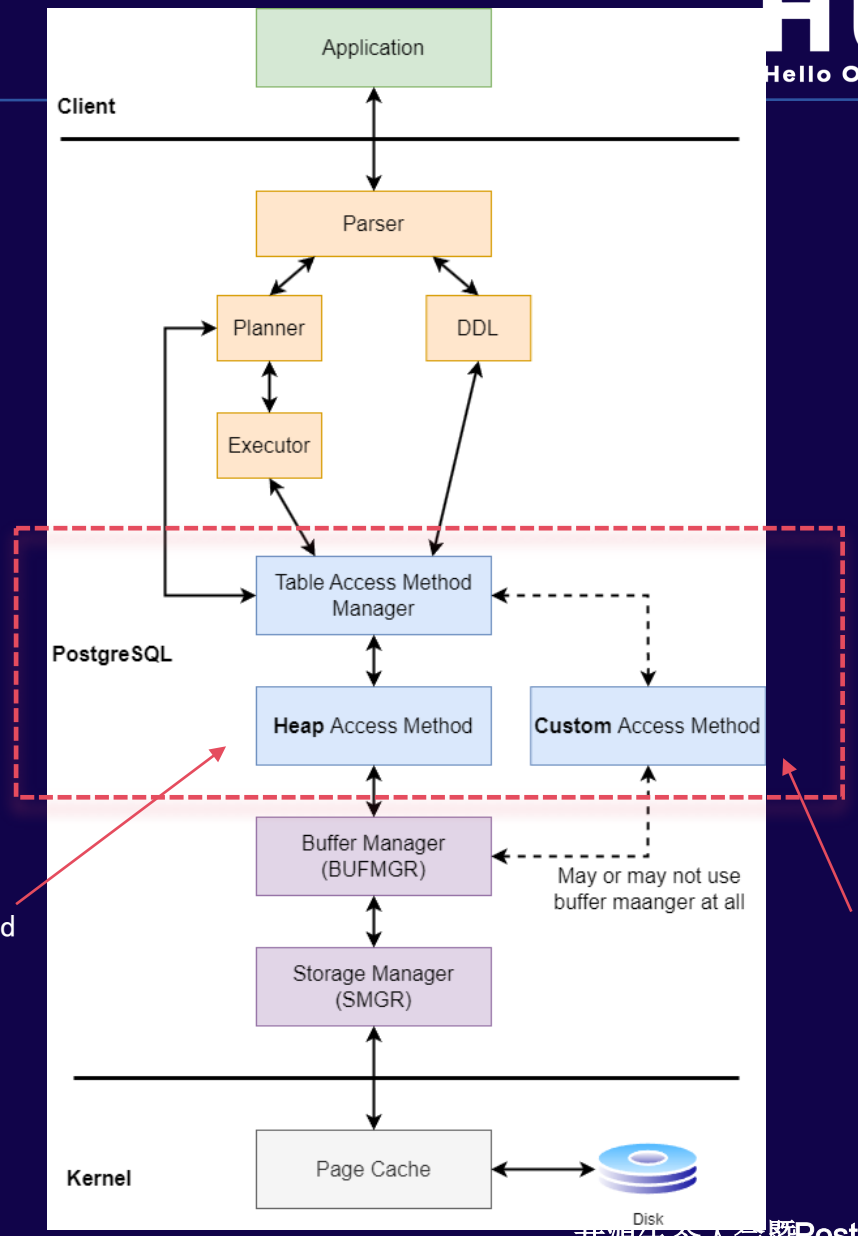
(6) 访问缓冲池中标识为 2 的缓冲区槽位。




```
1136    /* create a tag so we can lookup the buffer */
1137    INIT_BUFFERTAG(newTag, smgr->smgr_rnode.node, forkNum, blockNum);
1138
1139    /* determine its hash code and partition lock ID */
1140    newHash = BufTableHashCode(&newTag);
1141    newPartitionLock = BufMappingPartitionLock(newHash);
1142
1143    /* see if the block is in the buffer pool already */
1144    LWLockAcquire(newPartitionLock, LW_SHARED);
1145    buf_id = BufTableLookup(&newTag, newHash);
1146    if (buf_id >= 0)
1147    {
1148        /*
1149         * Found it.  Now, pin the buffer so no one can steal it from the
1150         * buffer pool, and check to see if the correct data has been loaded
1151         * into the buffer.
1152         */
1153        buf = GetBufferDescriptor(buf_id);
1154
1155        valid = PinBuffer(buf, strategy);
1156
1157        /* Can release the mapping lock as soon as we've pinned it */
1158        LWLockRelease(newPartitionLock);
1159
1160        *foundPtr = true;
```

什么是 Table Access Method

- 负责处理表数据的存储和管理等等：
  - 顺序扫描（Sequential Scan）。
  - 并行扫描（Parallel Scan）。
  - 索引获取（Index Fetch）。
  - 语句执行评估（Query Estimate）。
  - 插入（Insert），更新（Update），删除（Delete）。
  - 表创建（Create Table），真空（Vacuum，Vacuum Full）。
  - 大数据 （TOAST）
  - 等等
- 不是所有的 API 回调函数都需要被实现。



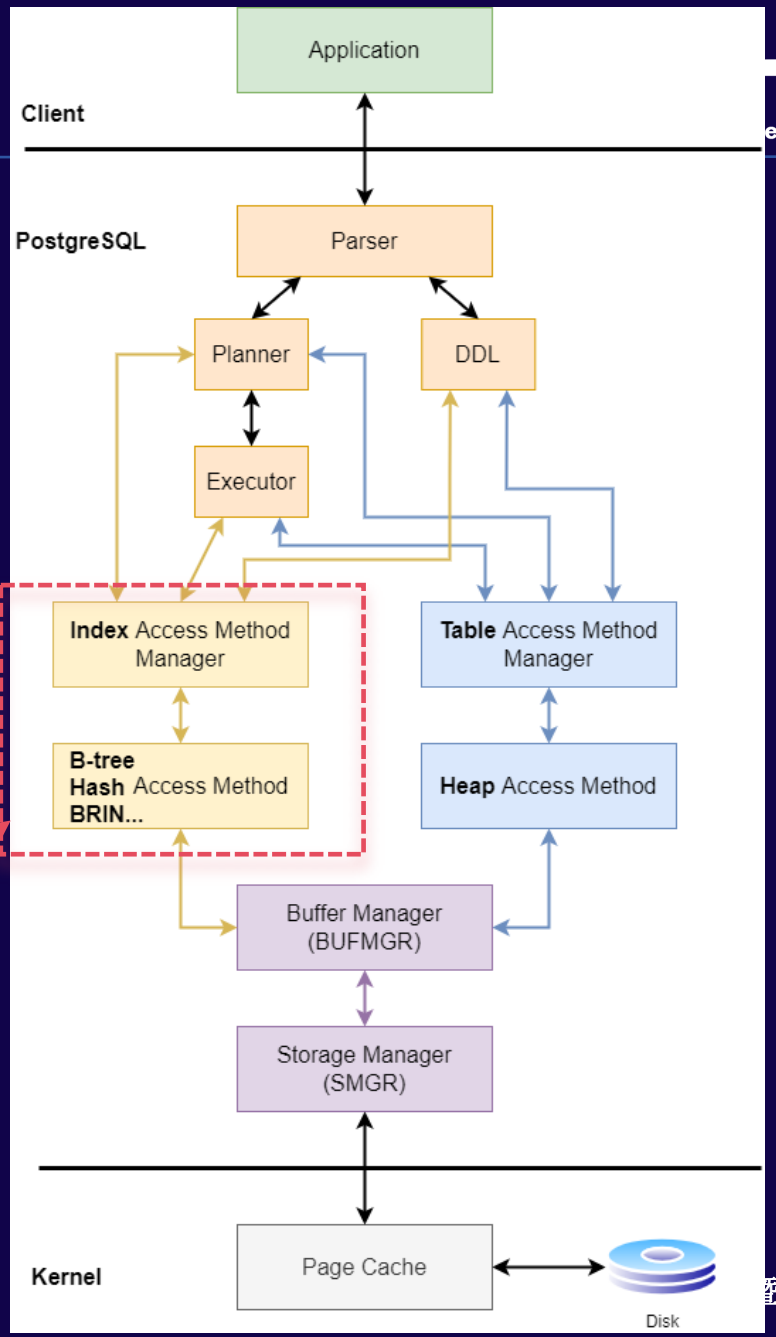
PostgreSQL 默认 Heap Access Method

自定义 Access Method（如果有）

什么是 Index Access Method

- 负责处理索引的存储和管理等等：
  - 构建 （Build）。
  - 索引扫描和并行扫描（Index Scan and Parallel Scan）。
  - 索引扫描评估（Estimate）。
  - 插入（Insert），更新（Update），删除（Delete）。
  - 唯一约束（Unidex Constraint）- 如果支持。
  - 排序（Sort）- 如果支持。
  - 索引操作符验证 （Validate）。
  - 等等
- 不是所有的 API 回调函数都需要被实现。

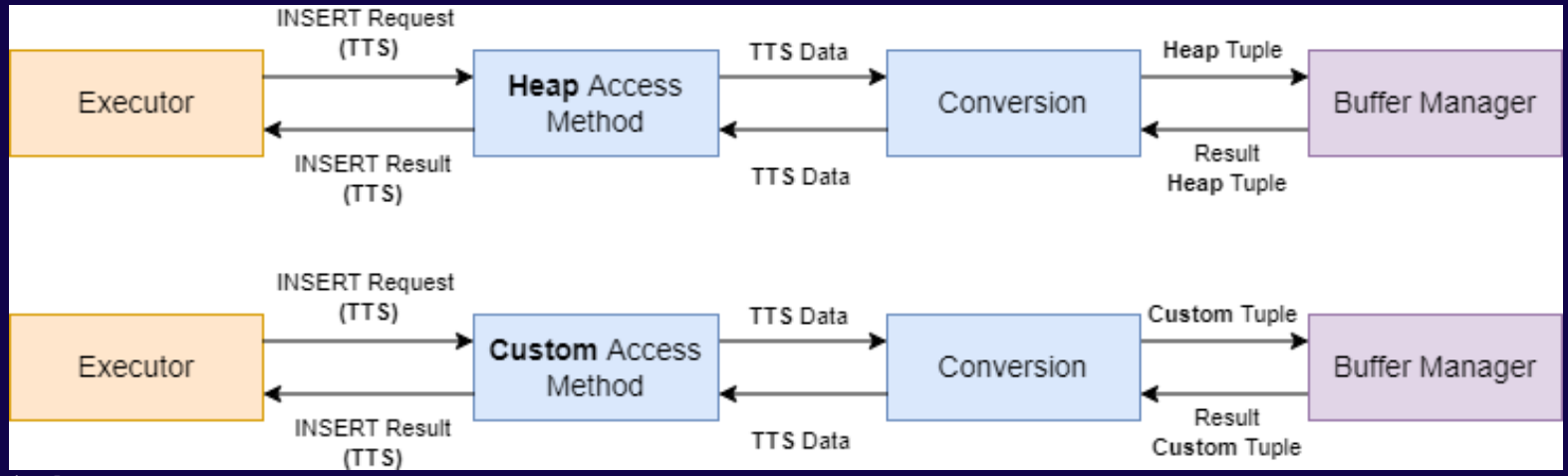
每一种 PostgreSQL 支持的索引都是通过 Index Access Method API 实现



Tuple Table Slot (TTS)

- TTS 是一种保存单行数据（包括列值）的内部数据结构。
- 它是语句处理（Query Processing）过程中的基本组件。
- 用来储存查询返回的行，也可用来储存要插入或更新的行。
- 主要由 Executor 模块和 Access Method 对接的时候使用。
- 他们的生命周期也是跟着语句处理走的。
- TTS 通常还跟着一个 TTS Operation 回调函数来告诉 PostgreSQL 怎么转换 TTS 到 Heap Tuple 或其他类型 Tuple 的数据格式。
- 结构定义在 src/include/executor/tuptable.h

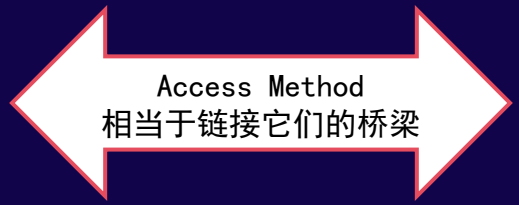
```c
/* base tuple table slot type */
typedef struct TupleTableSlot
{
    NodeTag     type;
#define FIELDNO_TUPLETABLESLOT_FLAGS 1
    uint16      tts_flags;      /* Boolean states */
#define FIELDNO_TUPLETABLESLOT_NVALID 2
    AttrNumber  tts_nvalid;     /* # of valid values in tts_values */
    const TupleTableSlotOps *const tts_ops; /* implementation of slot */
#define FIELDNO_TUPLETABLESLOT_TUPLEDESCRIPTOR 4
    TupleDesc   tts_tupleDescriptor;    /* slot's tuple descriptor */
#define FIELDNO_TUPLETABLESLOT_VALUES 5
    Datum       *tts_values;    /* current per-attribute values */
#define FIELDNO_TUPLETABLESLOT_ISNULL 6
    bool        *tts_isnull;    /* current per-attribute isnull flags */
    MemoryContext tts_mcxt;     /* slot itself is in this context */
    ItemPointerData tts_tid;    /* stored tuple's tid */
    Oid         tts_tableOid;   /* table oid of tuple */
} TupleTableSlot;
```



Tuple Table Slot (TTS)

Tuple Table Slot

- PostreSQL 内核的 Executor 使用的数据格式
- PostgerSQL 内部逻辑使用：
  - 列总数 natts
  - Tuple Descriptor
  - Flag 值
  - 数据数组 （Datum Array）
  - 空值数组（Isnull Array）
  - 。。。等等

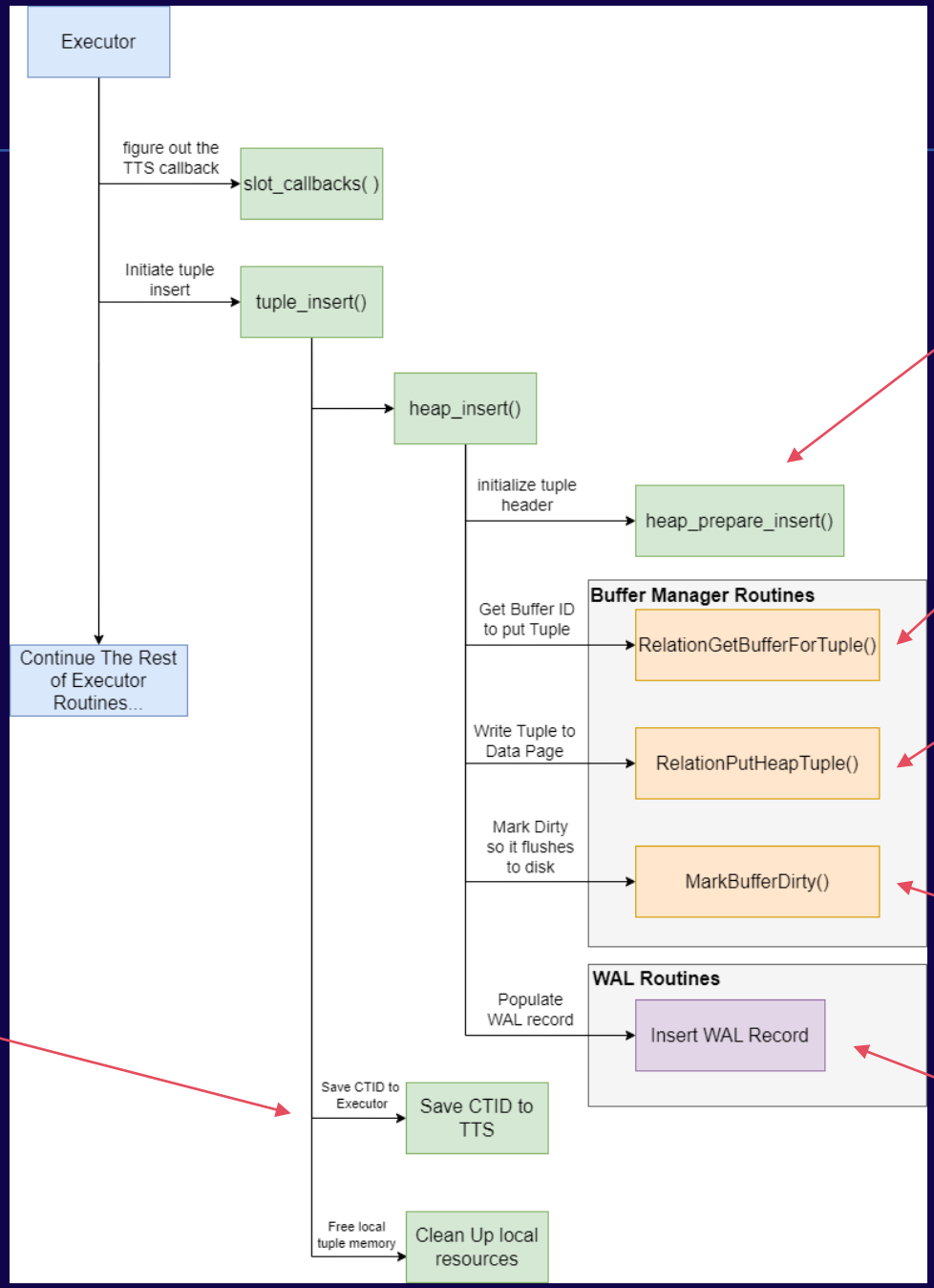


Heap Tuple

- Heap Access Method 使用的数据格式
- 真正存在磁盘上的数据格式，包含了：
  - Header 栏位
  - Flag 值
  - 偏移量
  - 用户数据等等
  - 。。。等等

数据插入

- 参与的 Access Method 接口：
  - slot_callbacks（）。
  - tuple_insert（）。
- 参与的 Buffer Manager 接口：
  - RelationGetBufferForTuple（）。
  - RelationPutHeapTuple（）。
  - MarkBufferDirty（）。



初始化新 Tuple 的 Meta Data，像是 xmin，xmax，cid，relation OID 等等

根据 Relation，和 Tuple 信息，向 Buffer Manager 请求一个还有空闲位置的内存块/页（Buffer Block/Page）。

通过 Buffer Manager 把 Tuple 写进返回的内存块/页里面。

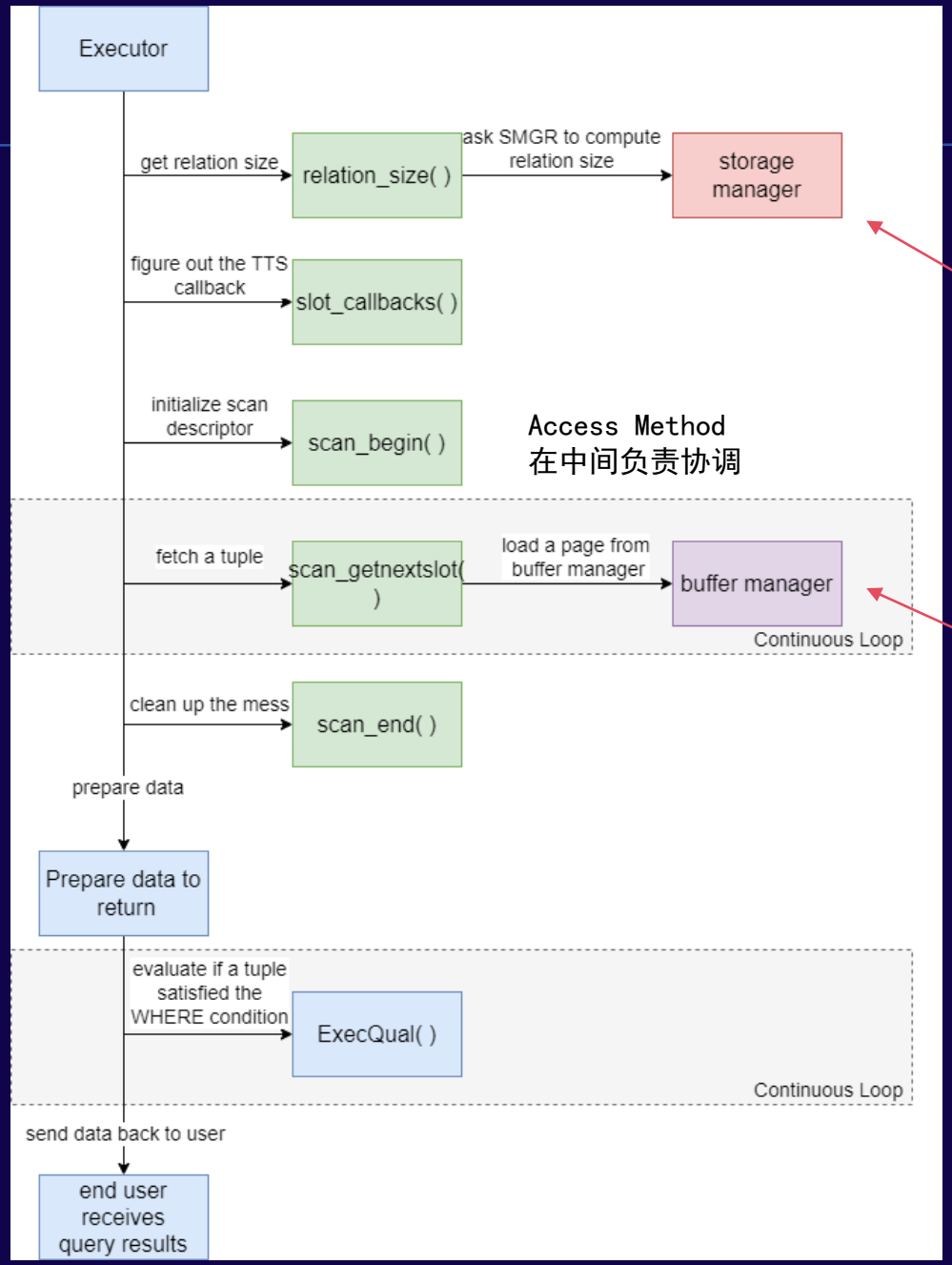
把此内存块/页标记为“脏”。代表它已被更新过，下次 Checkpoint 执行时，或是 Buffer Manager 需要腾出内存空间时会优先写入磁盘。

生成一条 WAL 记录，给系统崩溃恢复使用

数据插入完毕后，记录它的物理位置 CTID 值。索引就是通过 CTID 实现快速数据查询

顺序扫描

- 也叫 Sequential Scan。
- 参与的 Access Method 接口：
  - relation_size（）。
  - slot_callbacks（）。
  - scan_begin（）。
  - scan_getnextslot（）。
  - scan_end（）。



Storage Manager 查看指定表的物理大小

Buffer Manager 负责调出数据页（从磁盘读或是从共享内存）

得到数据后，Executor 决定如何返回给用户

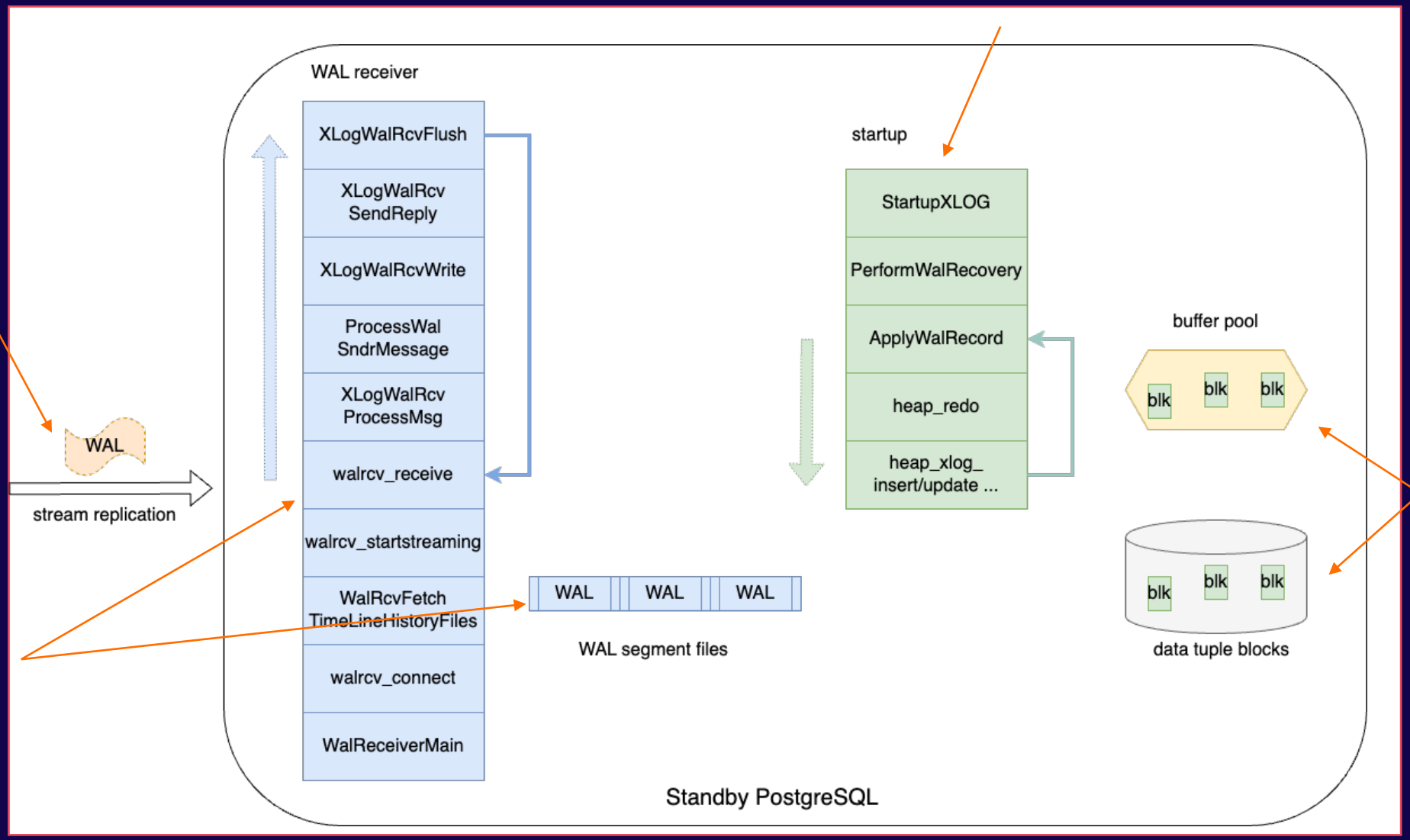
第五章：WAL 和 流复制

WAL 日志 REDO 流程

startup 进程负责对接收的 WAL 记录执行 REDO。

备节点通过流复制接收主节点的 WAL 日志的更新。

WAL receiver 进程负责更新主节点自己当前 WAL的位置，同时将接收的 WAL 日志记录写入磁盘。



REDO 后，记录先缓存在内存，然后存储到磁盘。

WAL REDO 函数

Operation	Resource manager
Heap tuple operations	RM_HEAP, RM_HEAP2
Index operations	RM_BTREE, RM_HASH, RM_GIN, RM_GIST, RM_SPGIST, RM_BRIN
Sequence operations	RM_SEQ
Transaction operations	RM_XACT, RM_MULTIXACT, RM_CLOG, RM_XLOG, RM_COMMIT_TS
Tablespace operations	RM_SMGR, RM_DBASE, RM_TBLSPC, RM_RELMAP
replication and hot standby operations	RM_STANDBY, RM_REPLORIGIN, RM_GENERIC_ID, RM_LOGICALMSG_ID

PG 内部定了多个 REDO 函数，通过资源管理器来管理日志重放，对于典型的 INSERT，UPDATE，COMMIT 操作：

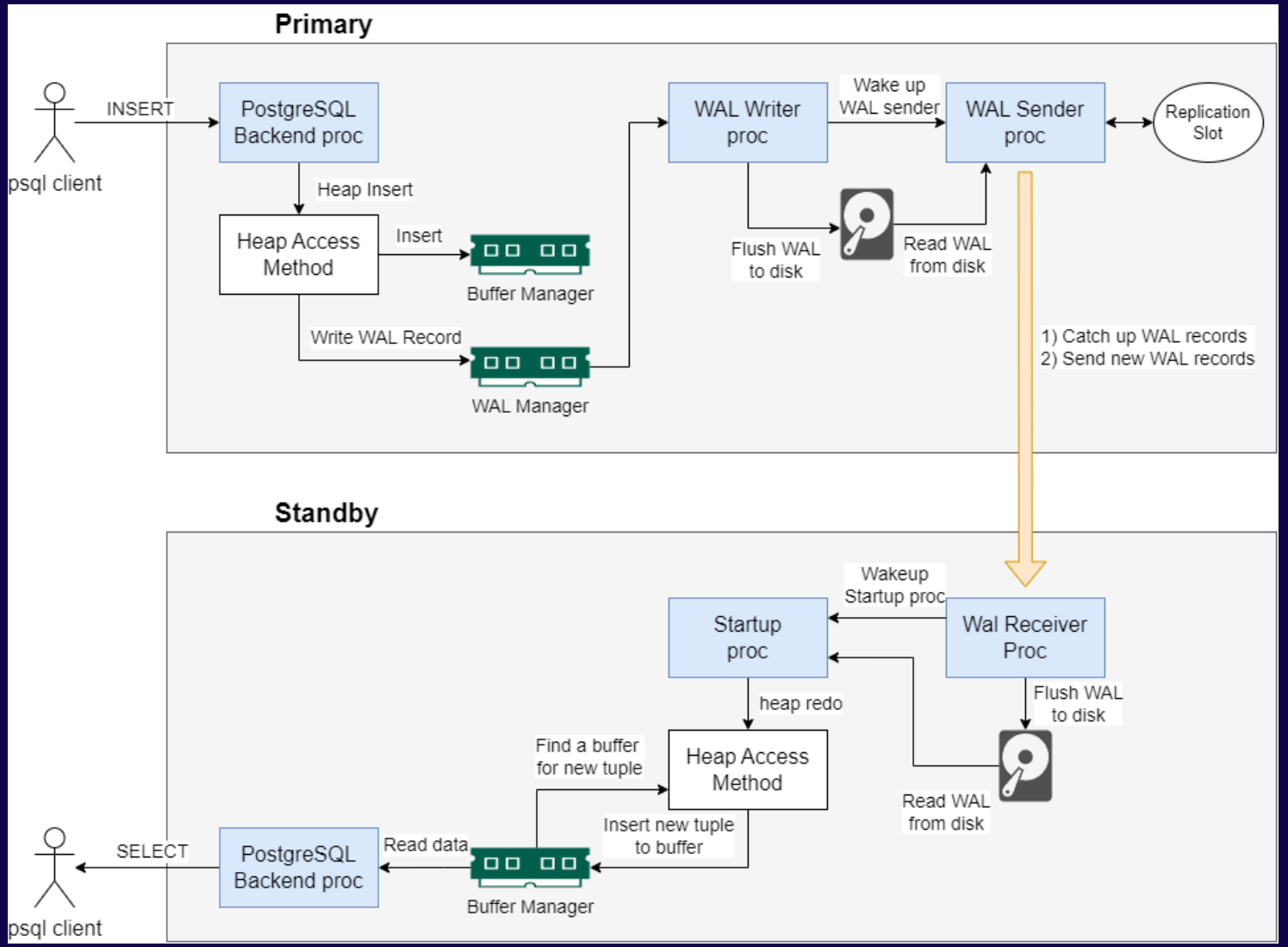
- INSERT：WAL 日志头部变量 xl_rmid 和 xl_info = 'RM_HEAP' 和 'XLOG_HEAP_INSERT'。使用对应的 heap_xlog_insert() 函数重放。
- UPDATE：WAL 日志头部变量 xl_rmid 和 xl_info = 'RM_HEAP' 和 'XLOG_HEAP_UPDATE'。使用对应的 heap_xlog_update() 函数重放。
- COMMIT：WAL 日志头部变量 xl_rmid 和 xl_info = 'RM_XACT' 和 'XLOG_XACT_COMMIT'。使用对应的 xact_redo_commit() 函数重放。

流复制架构图

- WAL Sender 和 Receiver：
  - 负责追上备节点落后的 WAL 记录。
  - 负责传输和接收新的物理 WAL 记录。
  - 1 对 1 的关系。

- Startup Proc：
  - 负责处理接收的 WAL 记录，通过 Heap Access Method 提供的 heap_redo（）函数处理。

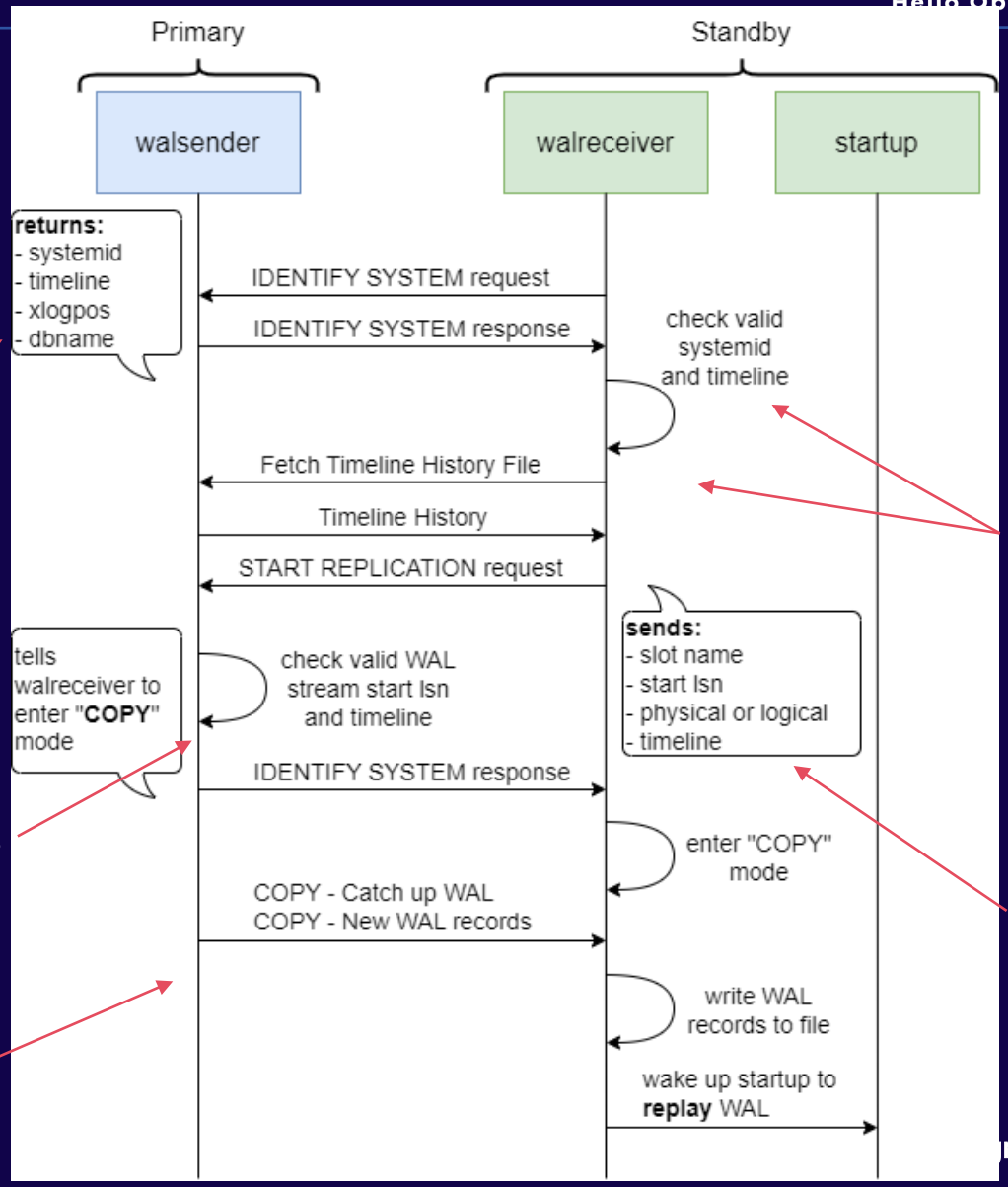
- Replication Slot （复制槽）：
  - 主节点可选择性的创建，备节点可选择性使用复制槽来做流复制。
  - 最大的好处就是可以防止主节点在所有备节点完成复制之前把 WAL 文件清理掉。



流复制启动流程

- 由 Standby 的 WAL receiver 负责发起一个流复制请求。
- 通过 libpq 提供的：
  - Replication Protocol 来做前期握手（Handshake）。
  - COPY Protocol 来传输 WAL 记录。

- 常用的 Replication Protocol 类型：
  - IDENTIFY_SYSTEM。
  - START_REPLICATION。
  - TIMELINE_HISTORY。
  - CREATE_REPLICATION_SLOT。
  - READ_REPLICATION_SLOT。



告诉备节点当前：
- 系统初始化 ID
- Timeline ID
- WAL 的 LSN 位置
- 数据库名

检查请求的合法性。

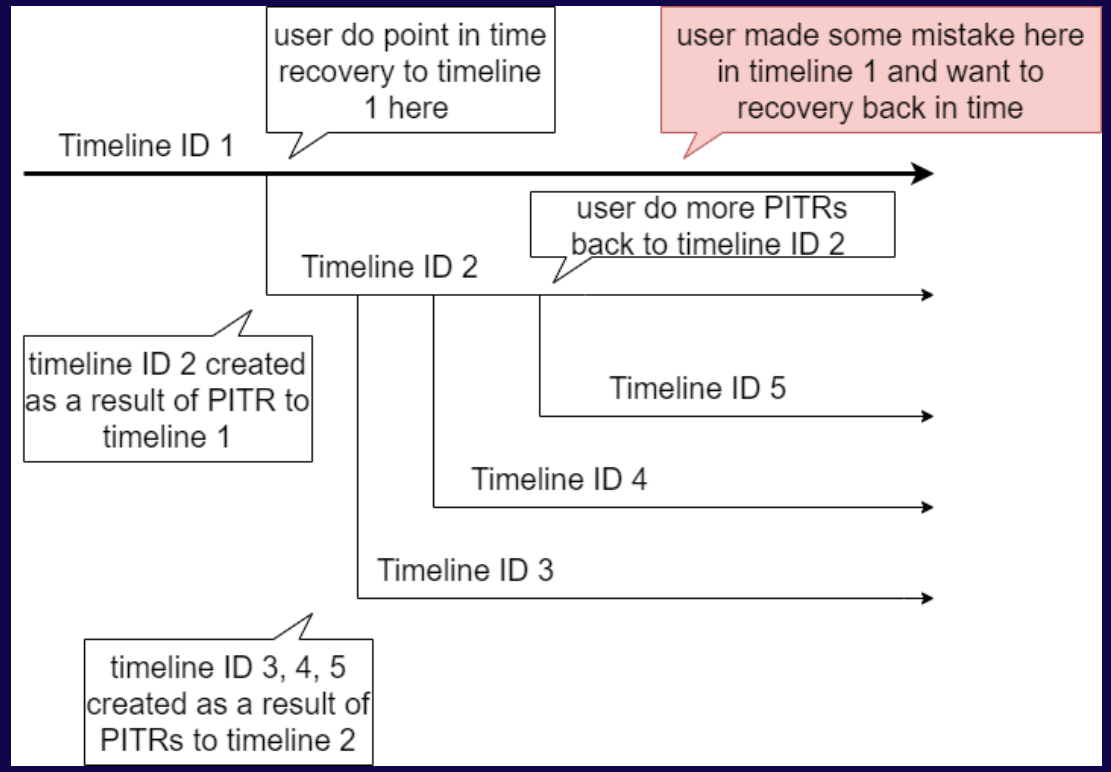
双方都进入 COPY 模式：
- 先做 WAL Catchup
- 发送新的 WAL 记录

检查 timeline ID 合法性，系统初始化 ID 是否和备节点相同等
获取 timeline history。

请求主节点要从某个 timeline ID 的某个 LSN 点上开始流复制。并提供想使用的复制槽名字（如果有）。

时间线（Timeline）

- 时间线（Timeline）是 WAL 机制里用于追踪数据库变化的一个概念。他表示 WAL 的连续序列。
- 新初始化的数据库，Timeline 从 1 开始：
  - Timeline += 1 当数据库做了 Point In Time Recovery (PITR)到某个 LSN。
  - Timeline += 1 当备节点被提升成主节点。



- WAL 文件的命名规则也包含了 Timeline 信息。

```
000000010000001300000E1    000000010000001300000E2
000000010000001300000E3    000000010000001300000E4
000000010000001300000E5    00000002000000130000E3
00000002000000130000E4     00000002000000130000E5
```

- 新的 Timeline 会产生一个对应的 History 文件：

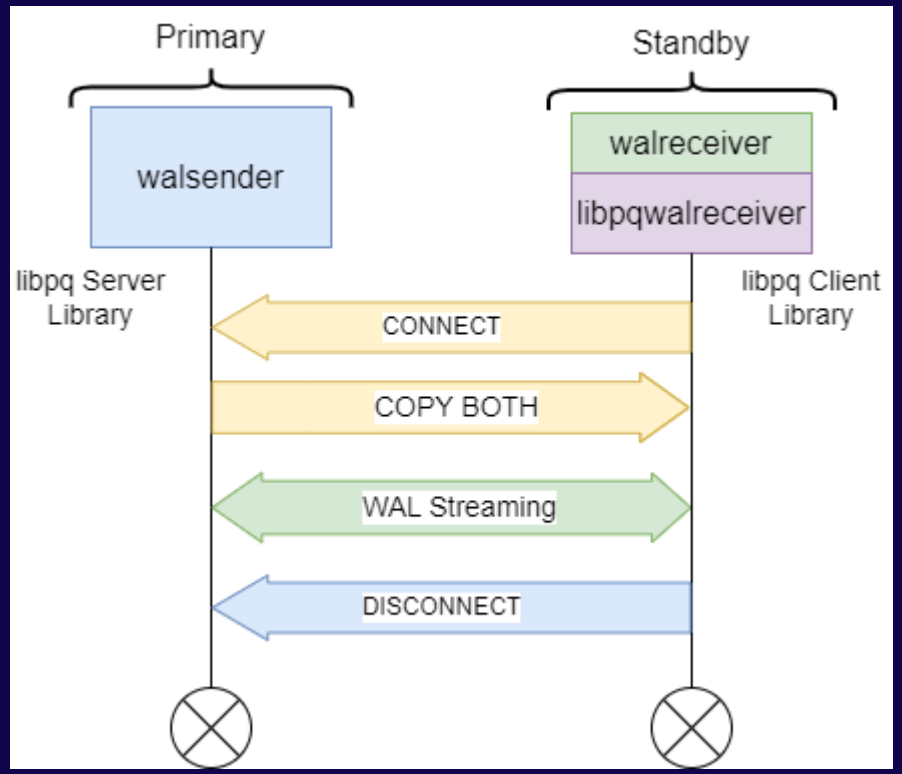
```
cat pg_wal/00000003.history
1 0/30000D8 no recovery target specified
2 0/3002B08 no recovery target specified
```

可理解为 Timeline 3 的历史由来。

Timeline 3 来自 Timeline 2 的 LSN 0/3002B08 点。
Timeline 2 来自 Timeline 1 的 LSN 0/30000D8 点。

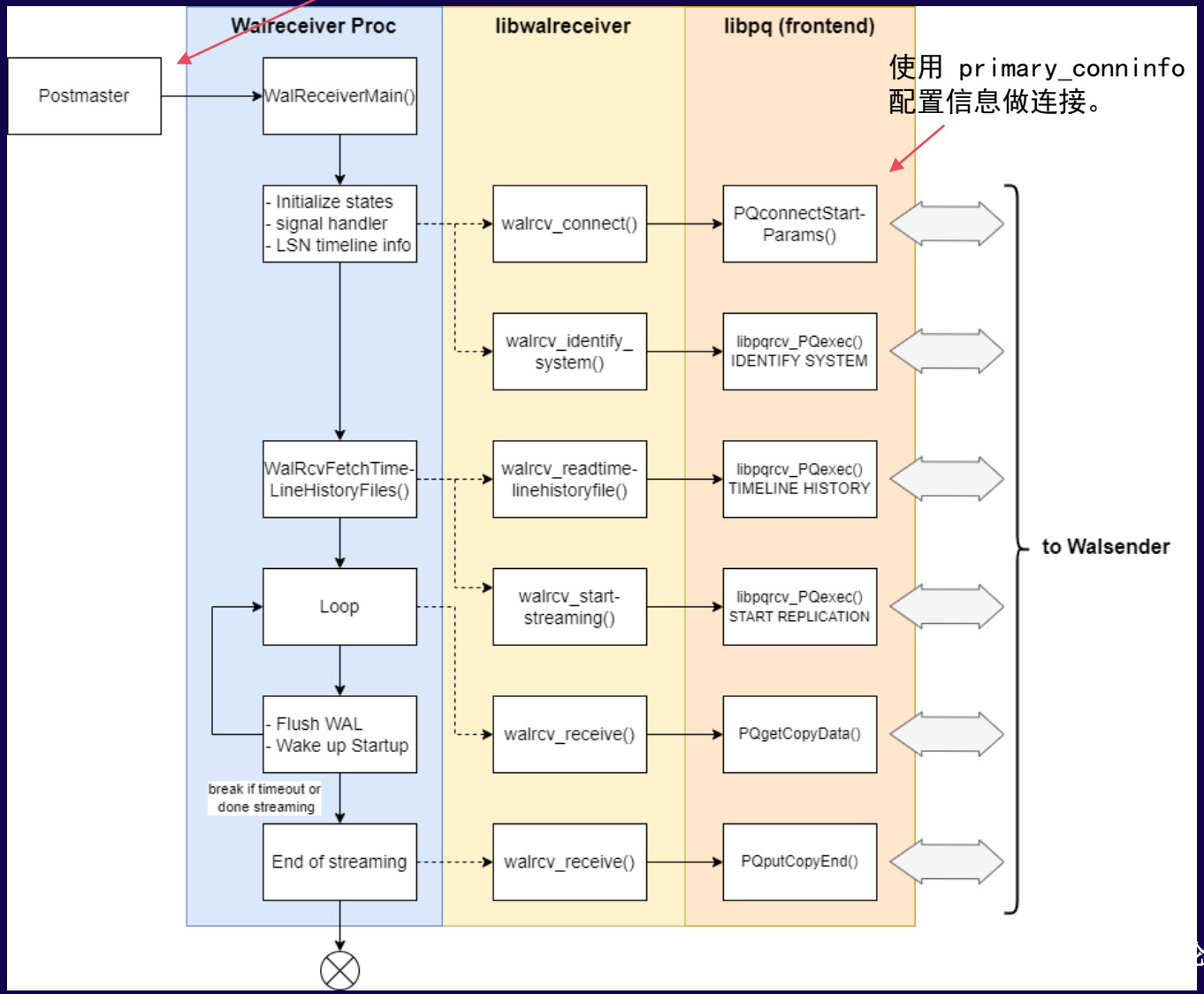
Libpq COPY 协议

- 流复制协议主要是协调主备节点之间的：
  - LSN 状态同步。
  - Timeline 之间的差别。
  - 复制范围（Replication Horizon）。
  - 检测主备节点的运行状态（Keep Alive）。
  - Replication Slot 的创建等等。
- WAL 的传输最终是由 COPY 协议完成的。
  - 流复制使用 COPY_BOTH 模式，允许双向数据的传输。
  - Walsender 和 Walreceiver 隶属于 PostgreSQL 的后台进程。
  - Walreceiver 等同一个 libpq 客户端（前端 libpq 库）。
  - Walsender 等同一个 libpq 服务器端（后端 libpq 库）。
  - 后台进程需要访问前台的链接库需要通过一个一共享库（Shared Library）形式的 wrapper 模块实现（libpqwalreceiver）。
  - 下一个章节我们会探讨 libpqwalreceiver 的细节以及如何自定义函数做额外的数据传输。



WAL Receiver 进程（流复制）

- 入口函数：WalReceiverMain（）- src/backend/replication/walreceiver.c。
- 负责发起流复制请求。
- 调用 libwalreceiver wrapper 来访问 libpq 前端函数服务。
- 负责接收 WAL，写入磁盘，叫醒 Startup 进程做 REDO。



第六章：逻辑复制和备份

背景信息 – 逻辑备份 vs 物理备份

逻辑备份

- 通过生成 SQL 和 Schema 脚本代表数据和架构备份。
- 通过执行 SQL 和 Schema 脚本还原数据。

案例

- 适合例行备份。
- 数据迁移和部分恢复。
- 选择性数据恢复。

备份大小

- 通常较小。

性能

- 通常比物理备份慢，因为涉及 SQL 语句执行。

特性

- 对数据迁移和维护有更高的灵活性。
- 相对简单。

物理备份

- 文件系统级别的备份（数据文件，配置等等）。
- 通过重新创建数据文件来实现恢复。

案例

- 适合灾难恢复场景。
- 硬件故障。
- 数据迁移。

备份大小

- 通常较大，因为它们包含所有数据文件。

性能

- 备份和恢复速度更快。

特性

- 时间点恢复 Point-In-Time Recovery。
- 流复制备节点基础。

背景信息 – 逻辑复制 vs 物理（流）复制

逻辑复制

- 选择性的数据复制，而非整个集群。

跨版本复制

- 可以在不同的 PostgerSQL 版本之间同步某个表。

数据聚合

- 将多个数据库中的数据整合到一个中央位置。

更改跟踪

- 跟踪对特定表所做的所有更改。

跨数据库复制

- 通过适当的逻辑复制解码插件，我们可以实现跨数据库平台的复制（PostgreSQL - Mongodb）。

物理（流）复制

- 对数据库集群整体进行的物理复制。

高可用

- 如果主节点发生故障，备用节点可以提升为新的主节点并继续数据库操作。Patroni 是用于此目的的流行工具。

数据备份

- 所有节点都存储相同数据的副本。

负载均衡

- 一写多读架构。
- 读和写的请求分离。 Pgpool 是用于此目的的流行工具。

可扩展性

- 可以添加更多的备用节点来分配更高的读请求负载。

什么是 Memory Context

- PostgreSQL 内核或逻辑复制插件开发时一定会遇到 Memory Context 的使用。它在内存管理扮演了重要的角色。
- C 语言中的内存管理是出了名的棘手；程序必须释放所有动态分配的内存。
- 不释放内存很容易导致内存泄漏（Memory Leak）。
- PostgreSQL 的开发我们不会直接使用 malloc（） 请求内存，而是通过 Memory Context 请求。
- 释放 Memory Context 的时候会自动释放所有已请求的内存。
- 创建自定义 Memory Context：

```
Wal2MongoData *data;

data = palloc0(sizeof(Wal2MongoData));
data->context = AllocSetContextCreate(ctx->context,
                                      "wal2mongo context",
                                      ALLOCSET_DEFAULT_SIZES);
```

- 释放自定义 Memory Context：

```
Wal2MongoData *data = ctx->output_plugin_private;

/* cleanup our own resources via memory context reset */
MemoryContextDelete(data->context);
```

- 函数开始时，切换到适当的 Memory Context：

```
Wal2MongoData *data;
MemoryContext old;

data = ctx->output_plugin_private;

/* Avoid leaking memory by using and resetting our own context */
old = MemoryContextSwitchTo(data->context);
```

- 使用 palloc 来请求 Memory Context 内存：

```
scan->rs_base.rs_key = (ScanKey) palloc(sizeof(ScanKeyData) * nkeys);
```

- 也可以重置 Memory Context：

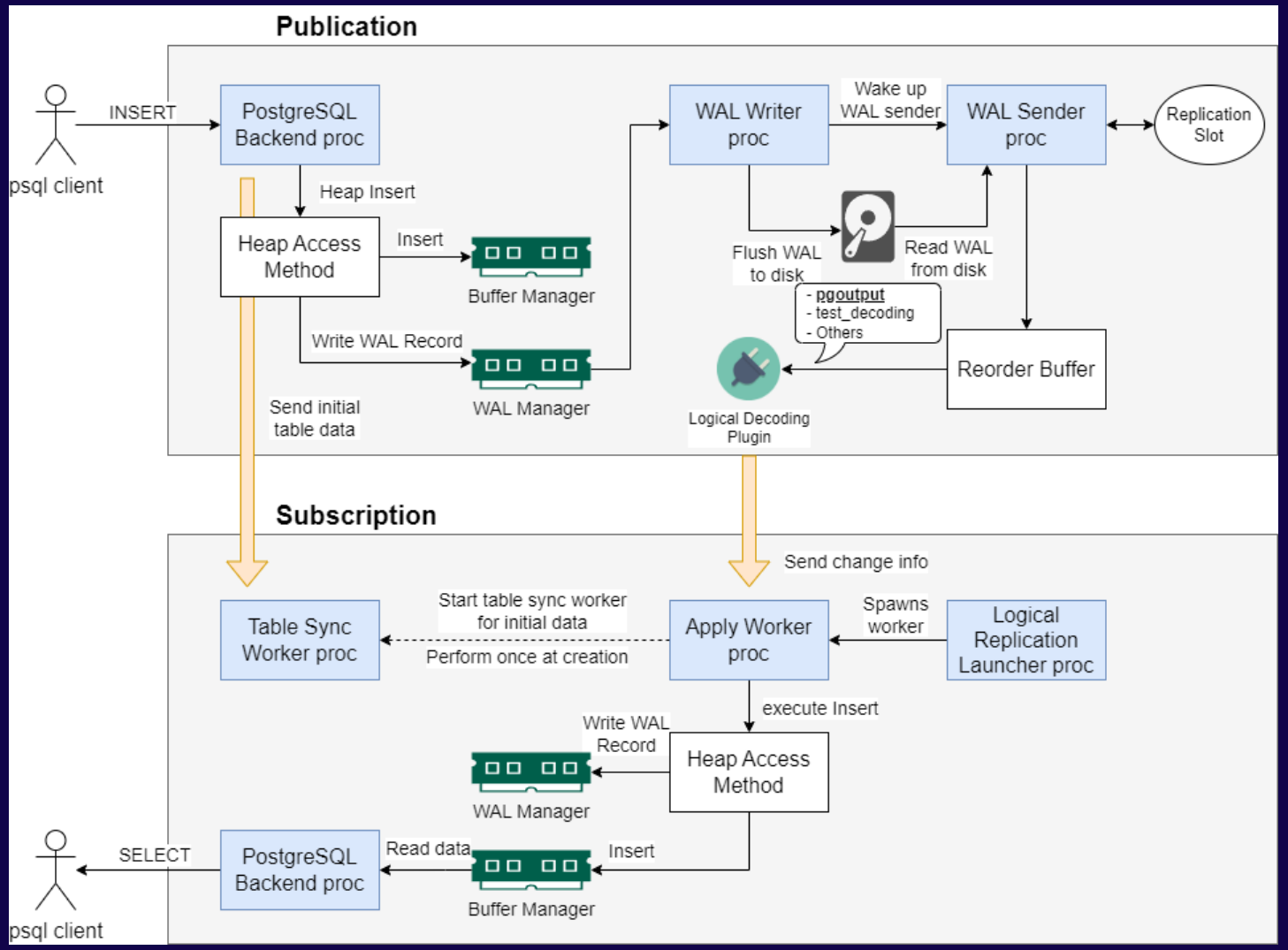
```
MemoryContextReset(data->context);
```

- 函数结束前，切换回原始的 Memory Context：

```
MemoryContextSwitchTo(old);
```

逻辑复制架构图

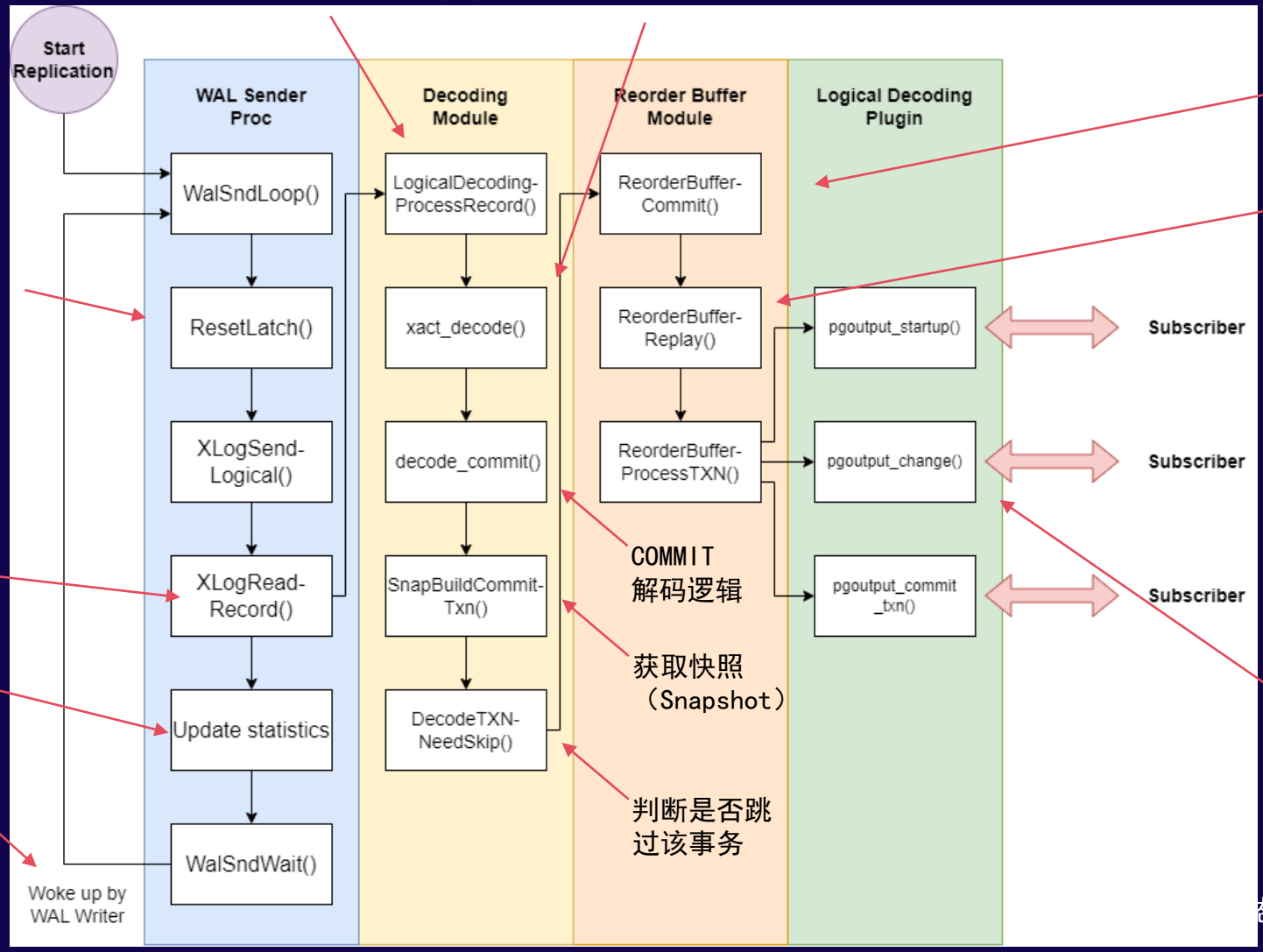
- **Logical Decoding Plugin（逻辑解码插件）：**
  - 发布端的核心组件。
  - 负责把数据库的改动解码成不同的格式。
  - Publication 和 Subscription 使用的是内置的 pgoutput 解码插件（src/backend/replication/pgoutput）。
- **Replication Slot（复制槽）：**
  - WAL Sender 通过 Slot 记录当前复制状态。
  - 避免重复发送数据，允许故障恢复后的逻辑复制继续问题。
- **Table Sync Worker：**
  - Subscription 创建时启动。
  - 目的是通过 libpq COPY 协议拷贝 Publication 原始表数据。



WAL Sender – COMMIT 流程

在这里调出对应的事务号和子事务号

解码入口函数，根据提交类型调用相对应的处理函数：COMMIT，COMMIT PREPARED， ABORT，ABORT PREPARED 等等



Reorder Buffer 入口函数：创建 ReorderBufferTXN 结构描述所有事务号的顺序关系

照顺序重播所有事务号和子事务号

重置每一次环路的 latch 状态

读取新写的 WAL Segments

更新发送统计信息

WAL Writer 负责“叫醒” WAL Sender

COMMIT 解码逻辑

获取快照（Snapshot）

判断是否跳过该事务

根据改动的性质（INSERT, UPDATE, DELETE, TRUNCATE），做相对应的解码。并发送到订阅端

目标表的信息和 Replica Identity 也是通过 decoding plugin 送出

解析 Tuple -> 字符输出

- 系统传给我们 HeapTuple 和 Relation 的信息，我们接下来就是要从这两个结构里面解析它们的值。

1. 首先，我们要先获取 TupleDesc 结构。它描述了一个表的所有列的信息。这个可以从 Relation 结构里获取。

```
TupleDesc tupdesc = RelationGetDescr(relation);
```

2. 从 TupleDesc 我们可以知道这个表有几个栏位（Column），他们的名字，数据类型等等 。

```
tupedesc->natt                  栏位总数
tupedesc->atttypid              类型ID
tupedesc->attname->data         栏位名
```

3. 我们就可以根据栏位 Index 号（从 1 开始），调出栏位的 Datum 值，如果 isnull 为 true，那代表没有值。

```
Datum val = heap_getattr(tuple, 1, tupdesc, &isnull);
```

解析 Tuple -> 字符输出

4. 我们也可以把数据类型以字符形式表示。例如 integer，boolean，bigint 等等：

```
format_type_be(tupedesc->atttypid);                    返回数据类型字符
```

5. 根据 tupedesc->atttypid，调出合适的输出函数 OID，并判断是否为可变长度类型（像是 TEXT 类型）：

```
Oid typoutput;       输出函数 OID
bool typisvarlena;   是否为可变长度类型标记

getTypeOutputInfo(tupedesc->atttypid, &typoutput, &typisvarlena);
```

6. 根据 typisvarlena 标记，调用相对应的输出函数把 Datum 以字符形式表示：

```
if (typisvarlena)
    OidOutputFunctionCall(typoutput, val)                    转换 Datum -> char*
else
{
    Datum val2;
    val2 = PointerGetDatum(PG_DETOAST_DATUM(val));           “DETOAST” 可变长度值
    OidOutputFunctionCall(typoutput, val2));                 转换 Datum -> char*
}
```

总结

总结

- 所有设计的培训内容都已在 2025 年交付。
- 课程的 ppt 可以从我这里获取。

- 课程使用的代码补丁可以在 github 上下载：

git clone git@github.com:HighgoSoftware/pg-lecture-resources.git。

- 我的邮箱：

hcary328@gmail.com

HOW 2026
Hello Open-source World

以开源之道·见致远之志
开源生态大会暨PostgreSQL高峰论坛

Thanks

谢谢