



PostgreSQL 是一个开源的关系型数据库系统，它结合了关系型数据库的优秀特性与非关系型数据库的某些优势。相比其他开源数据库，PostgreSQL 在功能、性能、稳定性等方面都有出色的表现。

本书通过深入浅出地讲解 PostgreSQL 的核心概念和操作方法，帮助读者快速掌握 PostgreSQL 的使用技巧，从而能够高效地处理各种复杂的数据需求。

本书适合所有对 PostgreSQL 感兴趣的读者，特别是那些希望在企业级应用中使用 PostgreSQL 的开发者、架构师以及运维人员。

核心篇

核 心 篇

本书分为三个篇章：基础篇、进阶篇和核心篇。基础篇主要介绍 PostgreSQL 的基本概念、安装配置、数据类型、SQL 语句等基础知识；进阶篇主要介绍 PostgreSQL 的高级功能，如分区表、并行查询、事务与并发控制、NoSQL 特性等；核心篇则深入探讨 PostgreSQL 的体系结构、优化原理、故障恢复机制等核心话题。通过学习本书，读者将能够全面掌握 PostgreSQL 的各项技术，从而在实际工作中发挥更大的作用。

本书通过深入浅出地讲解 PostgreSQL 的核心概念和操作方法，帮助读者快速掌握 PostgreSQL 的使用技巧，从而能够高效地处理各种复杂的数据需求。

本书适合所有对 PostgreSQL 感兴趣的读者，特别是那些希望在企业级应用中使用 PostgreSQL 的开发者、架构师以及运维人员。

本书通过深入浅出地讲解 PostgreSQL 的核心概念和操作方法，帮助读者快速掌握 PostgreSQL 的使用技巧，从而能够高效地处理各种复杂的数据需求。

本书适合所有对 PostgreSQL 感兴趣的读者，特别是那些希望在企业级应用中使用 PostgreSQL 的开发者、架构师以及运维人员。

本书通过深入浅出地讲解 PostgreSQL 的核心概念和操作方法，帮助读者快速掌握 PostgreSQL 的使用技巧，从而能够高效地处理各种复杂的数据需求。

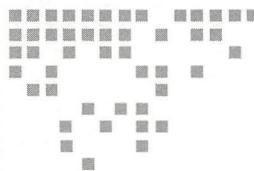
本书适合所有对 PostgreSQL 感兴趣的读者，特别是那些希望在企业级应用中使用 PostgreSQL 的开发者、架构师以及运维人员。

本书通过深入浅出地讲解 PostgreSQL 的核心概念和操作方法，帮助读者快速掌握 PostgreSQL 的使用技巧，从而能够高效地处理各种复杂的数据需求。

- 第 5 章 体系结构
- 第 6 章 并行查询
- 第 7 章 事务与并发控制
- 第 8 章 分区表
- 第 9 章 PostgreSQL 的 NoSQL 特性

本书通过深入浅出地讲解 PostgreSQL 的核心概念和操作方法，帮助读者快速掌握 PostgreSQL 的使用技巧，从而能够高效地处理各种复杂的数据需求。

本书适合所有对 PostgreSQL 感兴趣的读者，特别是那些希望在企业级应用中使用 PostgreSQL 的开发者、架构师以及运维人员。



Chapter 3

第5章

体 系 结 构

PostgreSQL 数据库是由一系列位于文件系统上的物理文件组成，在数据库运行过程中，通过整套高效严谨的逻辑管理这些物理文件。通常将这些物理文件称为数据库，将这些物理文件、管理这些物理文件的进程、进程管理的内存称为这个数据库的实例。在 PostgreSQL 的内部功能实现上，可以分为系统控制器、查询分析器、事务系统、恢复系统、文件系统这几部分。其中系统控制器负责接收外部连接请求，查询分析器对连接请求查询进行分析并生成优化后的查询解析树，从文件系统获取结果集或通过事务系统对数据做处理，并由文件系统持久化数据。本章将简单介绍 PostgreSQL 的物理和逻辑结构，同时介绍 PostgreSQL 实例在运行周期的进程结构。

5.1 逻辑和物理存储结构

在 PostgreSQL 中有一个数据库集簇（Database Cluster）的概念，也有一些地方翻译为数据库集群，它是指由单个 PostgreSQL 服务器实例管理的数据库集合，组成数据库集簇的这些数据库使用相同的全局配置文件和监听端口、共用进程和内存结构，并不是指“一组数据库服务器构成的集群”，在 PostgreSQL 中说的某一个数据库实例通常是指某个数据库集簇，这一点和其他常见的关系型数据库有一定差异，请读者注意区分。

5.1.1 逻辑存储结构

数据库集簇是数据库对象的集合，在关系数据库理论中，数据库对象是用于存储或引用数据的数据结构，表就是一个典型的例子，还有索引、序列、视图、函数等这些对象。在 PostgreSQL 中，数据库本身也是数据库对象，并且在逻辑上彼此分离，除数据库之外的



其他数据库对象（例如表、索引等）都属于它们各自的数据库，虽然它们隶属同一个数据库集簇，但无法直接从集簇中的一个数据库访问该集簇中的另一个数据库中的对象。

数据库本身也是数据库对象，一个数据库集簇可以包含多个 Database、多个 User，每个 Database 以及 Database 中的所有对象都有它们的所有者：User。图 5-1 显示了数据库集簇的逻辑结构。

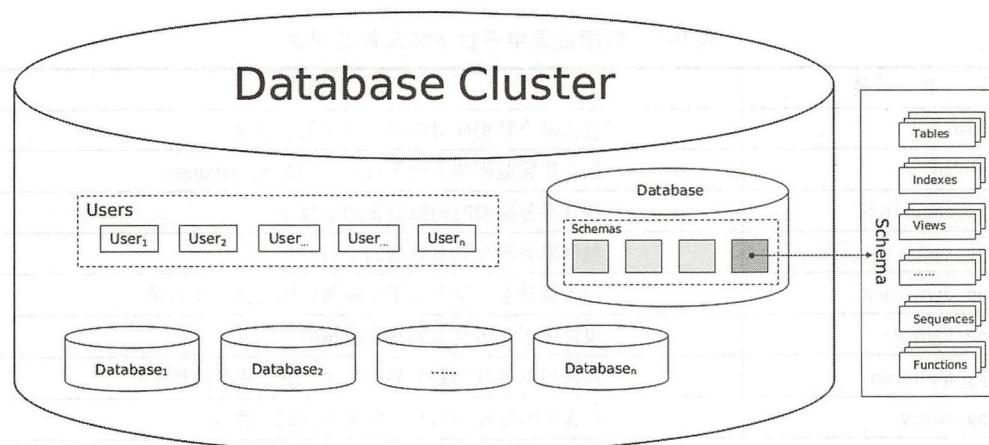


图 5-1 PostgreSQL 数据库集簇逻辑结构

创建一个 Database 时会为这个 Database 创建一个名为 public 的默认 Schema，每个 Database 可以有多个 Schema，在这个数据库中创建其他数据库对象时如果没有指定 Schema，都会在 public 这个 Schema 中。Schema 可以理解为一个数据库中的命名空间，在数据库中创建的所有对象都在 Schema 中创建，一个用户可以从同一个客户端连接中访问不同的 Schema。不同的 Schema 中可以有多个相同名称的 Table、Index、View、Sequence、Function 等数据库对象。

5.1.2 物理存储结构

数据库的文件默认保存在 initdb 时创建的数据目录中。在数据目录中有很多类型、功能不同的目录和文件，除了数据文件之外，还有参数文件、控制文件、数据库运行日志及预写日志等。

1. 数据目录结构

数据目录用来存放 PostgreSQL 持久化的数据，通常可以将数据目录路径配置为 PGDATA 环境变量，查看数据目录有哪些子目录和文件的命令如下所示：

```
[postgres@pghost1 ~]$ tree -L 1 -d /pgdata/10/data
/pgdata/10/data
└── base
```



表 5-1 对数据目录中子目录和文件的用途进行了说明。

表 5-1 数据目录中子目录和文件的用途

目 录	用 途
base	包含每个数据库对应的子目录的子目录
global	包含集簇范围的表的子目录，比如 pg_database
pg_commit_ts	包含事务提交时间戳数据的子目录
pg_xact	包含事务提交状态数据的子目录
pg_dynshmem	包含被动态共享内存子系统所使用文件的子目录
pg_logical	包含用于逻辑复制的状态数据的子目录
pg_multixact	包含多事务状态数据的子目录（用于共享的行锁）
pg_notify	包含 LISTEN/NOTIFY 状态数据的子目录
pg_repslot	包含复制槽数据的子目录
pg_serial	包含已提交的可序列化事务信息的子目录
pg_snapshots	包含导出的快照的子目录
pg_stat	包含用于统计子系统的永久文件的子目录
pg_stat_tmp	包含用于统计信息子系统临时文件的子目录
pg_subtrans	包含子事务状态数据的子目录
pg_tblspc	包含指向表空间的符号链接的子目录
pg_twophase	用于预备事务状态文件的子目录
pg_wal	保存预写日志
pg_xact	记录事务提交状态数据
文 件	用 途
PG_VERSION	PostgreSQL 主版本号文件
pg_hba.conf	客户端认证控制文件
postgresql.conf	参数文件
postgresql.auto.conf	参数文件，只保存 ALTER SYSTEM 命令修改的参数
postmaster.opts	记录服务器最后一次启动时使用的命令行参数

2. 数据文件布局

数据目录中的 base 子目录是我们的数据文件默认保存的位置，是数据库初始化后的默认表空间。在讨论 base 目录之前，我们先了解两个基础的数据库对象：OID 和表空间。



(1) OID

PostgreSQL 中的所有数据库对象都由各自的对象标识符 (OID) 进行内部管理，它们是无符号的 4 字节整数。数据库对象和各个 OID 之间的关系存储在适当的系统目录中，具体取决于对象的类型。数据库的 OID 存储在 pg_database 系统表中，可以通过如下代码查询数据库的 OID：

```
SELECT oid,datname FROM pg_database WHERE datname = 'mydb';
  oid  | datname
-----+-----
 16384 | mydb
(1 row)
```

数据库中的表、索引、序列等对象的 OID 存储在 pg_class 系统表中，可以通过如下代码查询获得这些对象的 OID：

```
mydb=# SELECT oid,relname,relkind FROM pg_class WHERE relname ~ 'tbl';
  oid  |      relname      | relkind
-----+-----+-----+
 16385 | tbl_id_seq      | S
 16387 | tbl              | r
 16396 | tbl_pkey         | i
 3455  | pg_class_tblspc_relfilenode_index | i
(4 rows)
```

(2) 表空间

在 PostgreSQL 中最大的逻辑存储单位是表空间，数据库中创建的对象都保存在表空间中，例如表、索引和整个数据库都可以被分配到特定的表空间。在创建数据库对象时，可以指定数据库对象的表空间，如果不指定则使用默认表空间，也就是数据库对象的文件的位置。初始化数据库目录时会自动创建 pg_default 和 pg_global 两个表空间。如下所示：

```
mydb=# \db
          List of tablespaces
   Name   |  Owner   | Location
-----+-----+-----+
 pg_default | postgres |
 pg_global  | postgres |
(2 rows)
```

- ❑ pg_global 表空间的物理文件位置在数据目录的 global 目录中，它用来保存系统表。
- ❑ pg_default 表空间的物理文件位置在数据目录中的 base 目录，是 template0 和 template1 数据库的默认表空间，我们知道创建数据库时，默认从 template1 数据库进行克隆，因此除非特别指定了新建数据库的表空间，默认使用 template1 的表空间，也就是 pg_default。

除了两个默认表空间，用户还可以创建自定义表空间。使用自定义表空间有两个典型的场景：



- 通过创建表空间解决已有表空间磁盘不足并无法逻辑扩展的问题；
- 将索引、WAL、数据文件分配在性能不同的磁盘上，使硬件利用率和性能最大化。

由于现在固态存储已经很普遍，这种文件布局方式反倒会增加维护成本。

要创建一个表空间，先用操作系统的 postgres 用户创建一个目录，然后连接到数据库，使用 CREATE TABLESPACE 命令创建表空间，如下所示：

```
[postgres@pghost1 ~]$ mkdir -p /pgdata/10/mytblspc
[postgres@pghost1 ~]$ /usr/pgsql-10/bin/psql -p 1921 mydb
psql (10.2)
Type "help" for help.
mydb=# CREATE TABLESPACE myspc LOCATION '/pgdata/10/mytblspc';
CREATE TABLESPACE
mydb=# \db
      List of tablespaces
   Name   |  Owner   |       Location
-----+-----+-----
  myspc | postgres | /pgdata/10/mytblspc
 pg_default | postgres |
 pg_global | postgres |
(3 rows)
```

当创建新的数据库或表时，便可以指定刚才创建的表空间，如下所示：

```
mydb=# CREATE TABLE t(id SERIAL PRIMARY KEY, ival int) TABLESPACE myspc;
CREATE TABLE
```

由于表空间定义了存储的位置，在创建数据库对象时，会在当前的表空间目录创建一个以数据库 OID 命名的目录，该数据库的所有对象将保存在这个目录中，除非单独指定表空间。例如我们一直使用的数据库 mydb，从 pg_database 系统表查询它的 OID，如下所示：

```
mydb=# SELECT oid,datname FROM pg_database WHERE datname = 'mydb';
      oid   | datname
-----+-----
  16384 | mydb
(1 row)
```

通过以上查询可知 mydb 的 OID 为 16384，我们就可以知道 mydb 的表、索引都会保存在 \$PGDATA/base/16384 这个目录中，如下所示：

```
[postgres@pghost1 ~]$ ll /pgdata/10/data/base/16384/
-rw----- 1 postgres postgres 16384 Nov 28 21:22 3712
...
...
...
-rw----- 1 postgres postgres 8192 Nov 28 21:22 3764_vm
```

(3) 数据文件命名

在数据库中创建对象，例如表、索引时首先会为表和索引分配段。在 PostgreSQL 中，每个表和索引都用一个文件存储，新创建的表文件以表的 OID 命名，对于大小超出 1GB 的



表数据文件，PostgreSQL 会自动将其切分为多个文件来存储，切分出的文件用 OID.<顺序号>来命名。但表文件并不是总是“OID.<顺序号>”命名，实际上真正管理表文件的是 pg_class 表中的 relfilenode 字段的值，在新创建对象时会在 pg_class 系统表中插入该表的记录，默认会以 OID 作为 relfilenode 的值，但经过几次 VACUUM、TRUNCATE 操作之后，relfilenode 的值会发生变化。

举例如下：

```
mydb=# SELECT oid,relfilenode FROM pg_class WHERE relname = 'tbl';
      oid |    relfilenode
-----+-----
      16387 |        16387
(1 row)
mydb=# \! ls -l /pgdata/10/data/base/16384/16387*
-rw----- 1 postgres postgres 8192 Mar 26 22:22 /pgdata/10/data/base/16384/16387
```

在默认情况下，tbl 表的 OID 为 16387，relfilenode 也是 16387，表的物理文件为“/pgdata/10/data/base/16384/16387”。依次 TRUNCATE 清空 tbl 表的所有数据，如下所示：

```
mydb=# TRUNCATE tbl;
TRUNCATE TABLE
mydb=# CHECKPOINT;
CHECKPOINT
mydb=# \! ls -l /pgdata/10/data/base/16384/16387*
ls: cannot access /pgdata/10/data/base/16384/16387*: No such file or directory
```

通过上述操作之后，tbl 表原先的物理文件“/pgdata/10/data/base/16384/16387”已经不存在了，那么 tbl 表的数据文件是哪一个？

```
postgres@160.40:1922/mydb=# select oid,relfilenode from pg_class where relname = 'tbl';
      oid |    relfilenode
-----+-----
      16387 |        24591
(1 row)
postgres@160.40:1922/mydb=# \! ls -l /pgdata/10/data/base/16384/24591*
-rw----- 1 postgres postgres 0 Apr  2 21:24 /pgdata/10/data/base/16384/24591
```

如上所示，再次查询 pg_class 表得知 tbl 表的数据文件已经成为“/pgdata/10/data/base/16384/24591”，它的命名规则为 <relfilenode>.<顺序号>。

在 tbl 测试表中写入一些测试数据，如下所示：

```
mydb=# insert into tbl (ival,description,created_time) select (random()*(2*10^9))::integer as ival,substr('abcdefghijklmnopqrstuvwxyz',1,(random()*26)::integer) as description,date(generate_series(now(), now() + '1 week', '1 day')) as created_time from generate_series(1,2000000);
INSERT 0 16000000
```

查看表的大小，如下所示：

```
mydb=# SELECT pg_size_pretty(pg_relation_size('tbl'::regclass));
```



```
pg_size_pretty
```

```
-----  
1068 MB  
(1 row)
```

通过上述命令看到 `tbl` 表的大小目前为 1068MB，执行一些 UPDATE 操作后再次查看数据文件，如下所示：

```
/mydb=# \! ls -lh /pgdata/10/data/base/16384/24591*  
-rwx----- 1 postgres postgres 1.0G Apr 7 08:44 /pgdata/10/data/base/16384/24591  
-rwx----- 1 postgres postgres 383M Apr 7 08:44 /pgdata/10/data/base/16384/24591.1  
-rwx----- 1 postgres postgres 376K Apr 7 08:44 /pgdata/10/data/base/16384/24591_fsm  
-rwx----- 1 postgres postgres 8.0K Apr 7 08:44 /pgdata/10/data/base/16384/24591_vm
```

如前文所述，数据文件的命名规则为 `<relfilenode>.<顺序号>`，`tbl` 表的大小超过 1GB，`tbl` 表的 `relfilenode` 为 24591，超出 1GB 之外的数据会按每 GB 切割，在文件系统中查看时就是名称为 24591.1 的数据文件。在上述输出结果中，后缀为 `_fsm` 和 `_vm` 的这两个表文件的附属文件是空闲空间映射表文件和可见性映射表文件。空闲空间映射用来映射表文件中可用的空间，可见性映射表文件跟踪哪些页面只包含已知对所有活动事务可见的元组，它也跟踪哪些页面只包含未被冻结的元组。图 5-2 显示了 PostgreSQL 数据目录、表空间以及文件的结构概貌。

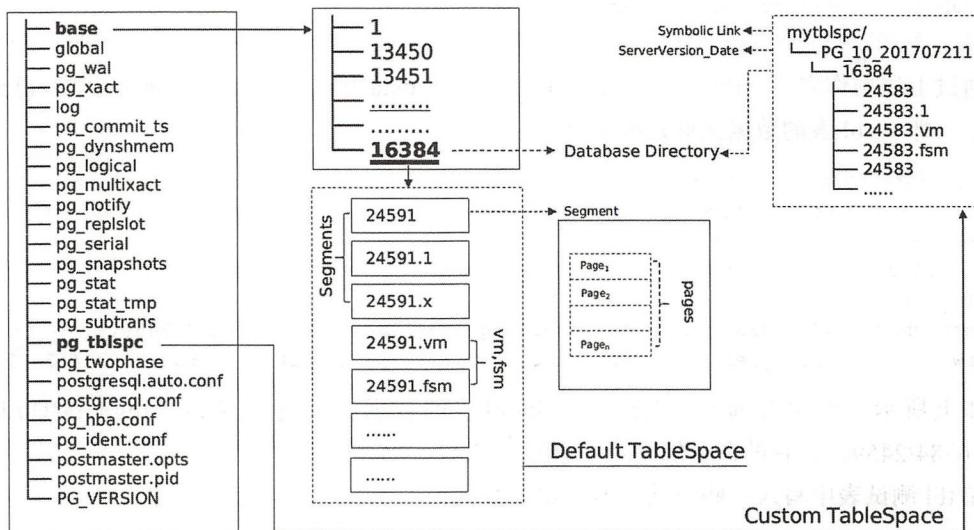


图 5-2 PostgreSQL 数据目录和文件结构

(4) 表文件内部结构

在 PostgreSQL 中，将保存在磁盘中的块称为 Page，而将内存中的块称为 Buffer，表和索引称为 Relation，行称为 Tuple，如图 5-3 所示。数据的读写是以 Page 为最小单位，每个 Page 默认大小为 8kB，在编译 PostgreSQL 时指定的 BLCKSZ 大小决定 Page 的大小。每个



表文件由多个 BLCKSZ 字节大小的 Page 组成，每个 Page 包含若干 Tuple。对于 I/O 性能较好的硬件，并且以分析为主的数据库，适当增加 BLCKSZ 大小可以小幅提升数据库性能。

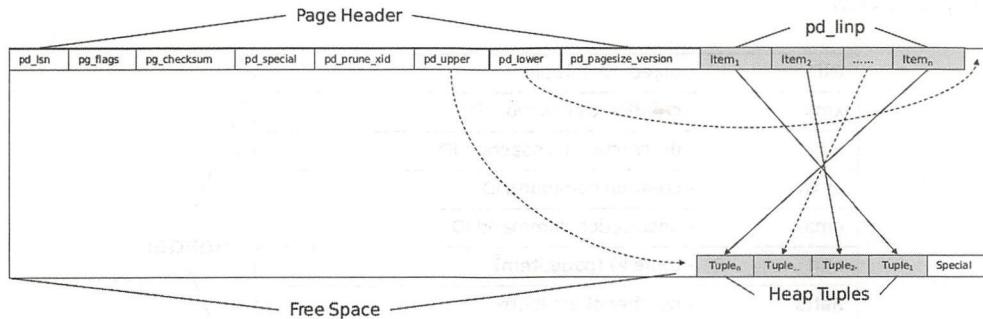


图 5-3 Page 内部结构

PageHeader 描述了一个数据页的页头信息，包含页的一些元信息。它的结构及其结构指针 PageHeader 的定义如下：

- ❑ pd_lsn：在 ARIES Recovery Algorithm 的解释中，这个 lsn 称为 PageLSN，它确定和记录了最后更改此页的 xlog 记录的 LSN，把数据页和 WAL 日志关联，用于恢复数据时校验日志文件和数据文件的一致性；pd_lsn 的高位为 xlogid，低位记录偏移量；因为历史原因，64 位的 LSN 保存为两个 32 位的值。
- ❑ pg_flags：标识页面的数据存储情况。
- ❑ pd_special：指向索引相关数据的开始位置，该项在数据文件中为空，主要是针对不同索引。
- ❑ pd_lower：指向空闲空间的起始位置。
- ❑ pd_upper：指向空闲空间的结束位置。
- ❑ pd_pagesize_version：不同的 PostgreSQL 版本的页的格式可能会不同。
- ❑ pd_lnp[1]：行指针数组，即图 5-3 中的 Item1, Item2, ..., Itemn，这些地址指向 Tuple 的存储位置。

如果一个表由一个只包含一个堆元组的页面组成。该页面的 pd_lower 指向第一行指针，并且行指针和 pd_upper 都指向第一个堆元组。当第二个元组被插入时，它被放置在第一个元组之后。第二行指针被压入第一行，并指向第二个元组。pd_lower 更改为指向第二行指针，pd_upper 更改为第二个堆元组。此页面中的其他头数据（例如，pd_lsn、pg_checksum、pg_flag）也被重写为适当的值。

当从数据库中检索数据时有两种典型的访问方法，顺序扫描和 B 树索引扫描。顺序扫描通过扫描每个页面中的所有行指针顺序读取所有页面中的所有元组。B 树索引扫描时，索引文件包含索引元组，每个元组由索引键和指向目标堆元组的 TID 组成。如果找到了正在查找的键的索引元组，PostgreSQL 使用获取的 TID 值读取所需的堆元组。



每个 Tuple 包含两部分的内容，一部分为 HeapTupleHeader，用来保存 Tuple 的元信息，如图 5-4 所示，包含该 Tuple 的 OID、xmin、xmax 等；另一部分为 HeapTuple，用来保存 Tuple 的数据。

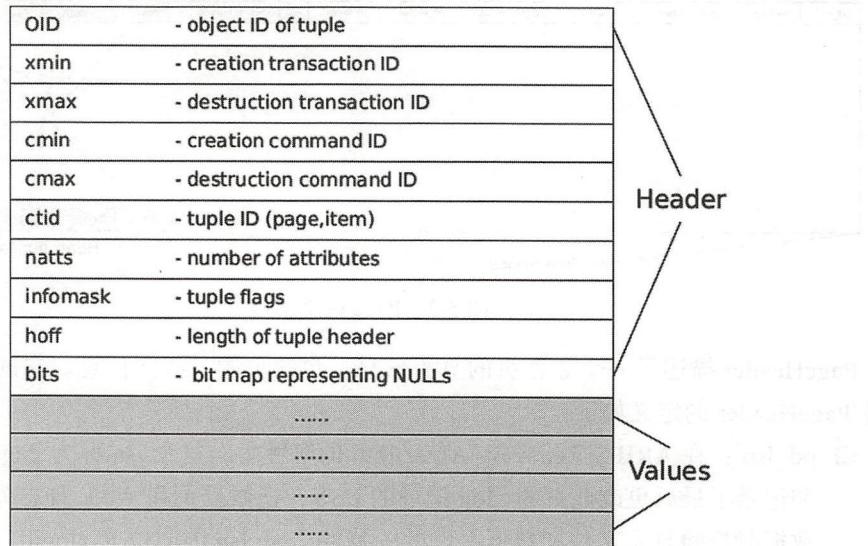


图 5-4 Tuple 内部结构

图 5-5 展示了一个完整的文件布局。

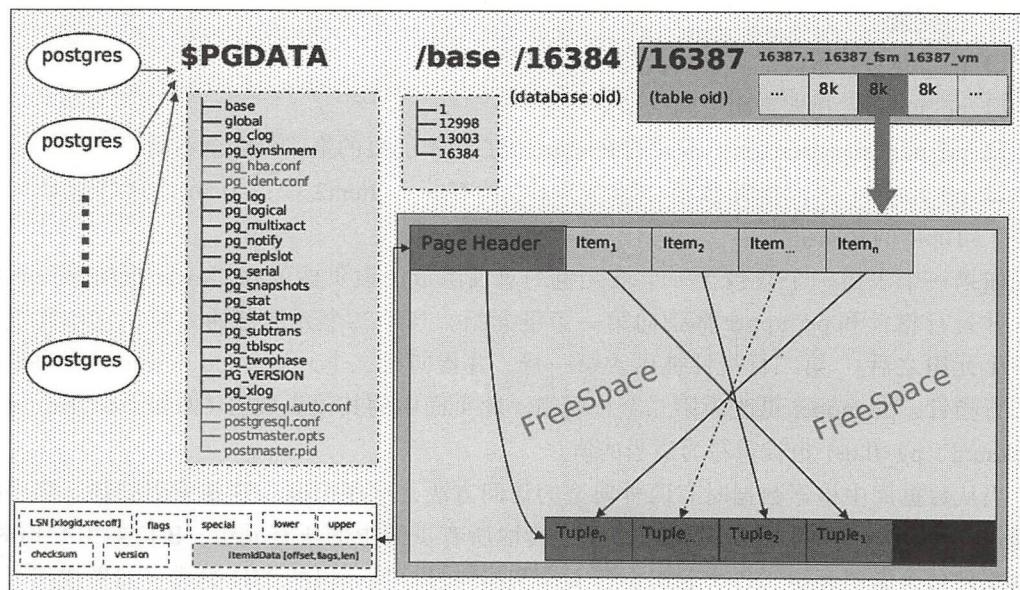


图 5-5 完整文件布局



5.2 进程结构

PostgreSQL 是一用户一进程的客户端 / 服务器的应用程序。数据库启动时会启动若干个进程，其中有 postmaster（守护进程）、postgres（服务进程）、syslogger、checkpointer、bgwriter、walwriter 等辅助进程。

5.2.1 守护进程与服务进程

首先从 postmaster（守护进程）说起。postmaster 进程的主要职责有：

- 数据库的启停。
- 监听客户端连接。
- 为每个客户端连接 fork 单独的 postgres 服务进程。
- 当服务进程出错时进行修复。
- 管理数据文件。
- 管理与数据库运行相关的辅助进程。

当客户端调用接口库向数据库发起连接请求，守护进程 postmaster 会 fork 单独的服务进程 postgres 为客户端提供服务，此后将由 postgres 进程为客户端执行各种命令，客户端也不再需要 postmaster 中转，直接与服务进程 postgres 通信，直至客户端断开连接，如图 5-6 所示。

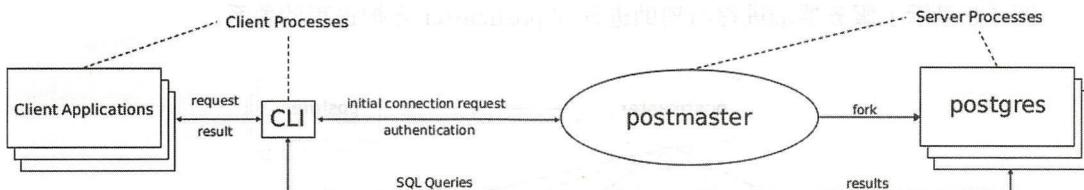


图 5-6 客户端与服务器端进程

PostgreSQL 使用基于消息的协议用于前端和后端（服务器和客户端）之间通信。通信都是通过一个消息流进行，消息的第一个字节标识消息类型，后面跟着的四个字节声明消息剩下部分的长度，该协议在 TCP/IP 和 Unix 域套接字上实现。服务器作业之间通过信号和共享内存通信，以保证并发访问时的数据完整性，如图 5-7 所示。

5.2.2 辅助进程

除了守护进程 postmaster 和服务进程 postgres 外，PostgreSQL 在运行期间还需要一些辅助进程才能工作，这些进程包括：

- background writer：也可以称为 bgwriter 进程，bgwriter 进程很多时候都是在休眠状态，每次唤醒后它会搜索共享缓冲池找到被修改的页，并将它们从共享缓冲池刷出。



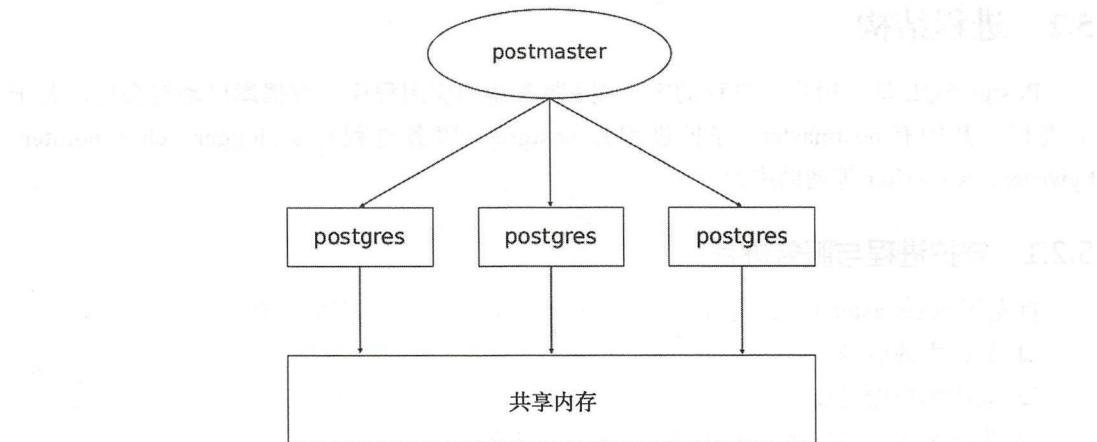


图 5-7 服务端进程与共享内存

- autovacuum launcher: 自动清理回收垃圾进程。
- WAL writer: 定期将 WAL 缓冲区上的 WAL 数据写入磁盘。
- statistics collector: 统计信息收集进程。
- logging collector: 日志进程, 将消息或错误信息写入日志。
- archiver: WAL 归档进程。
- checkpointer: 检查点进程。

图 5-8 显示了服务器端进程与辅助进程和 postmaster 守护进程的关系。

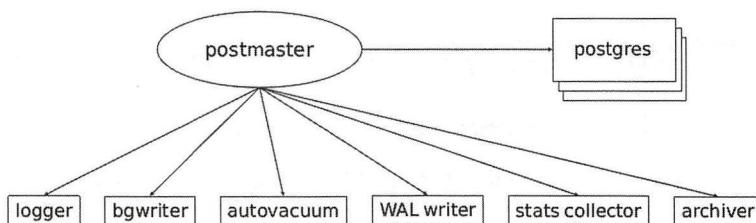


图 5-8 服务端进程与辅助进程

5.3 内存结构

PostgreSQL 的内存分为两大类：本地内存和共享内存，另外还有一些为辅助进程分配的内存等，下面简单介绍本地内存和共享内存的概貌。

5.3.1 本地内存

本地内存由每个后端服务进程分配以供自己使用，当后端服务进程被 fork 时，每个后



端进程为查询分配一个本地内存区域。本地内存由三部分组成：work_mem、maintenance_work_mem 和 temp_buffers。

- work_mem：当使用 ORDER BY 或 DISTINCT 操作对元组进行排序时会使用这部分内存。
- maintenance_work_mem：维护操作，例如 VACUUM、REINDEX、CREATE INDEX 等操作使用这部分内存。
- temp_buffers：临时表相关操作使用这部分内存。

5.3.2 共享内存

共享内存 在 PostgreSQL 服务器启动时分配，由所有后端进程共同使用。共享内存主要由三部分组成：

- shared buffer pool：PostgreSQL 将表和索引中的页面从持久存储装载到这里，并直接操作它们。
- WAL buffer：WAL 文件持久化之前的缓冲区。
- CommitLog buffer：PostgreSQL 在 Commit Log 中保存事务的状态，并将这些状态保留在共享内存缓冲区中，在整个事务处理过程中使用。

图 5-9 显示了内存的结构概貌。

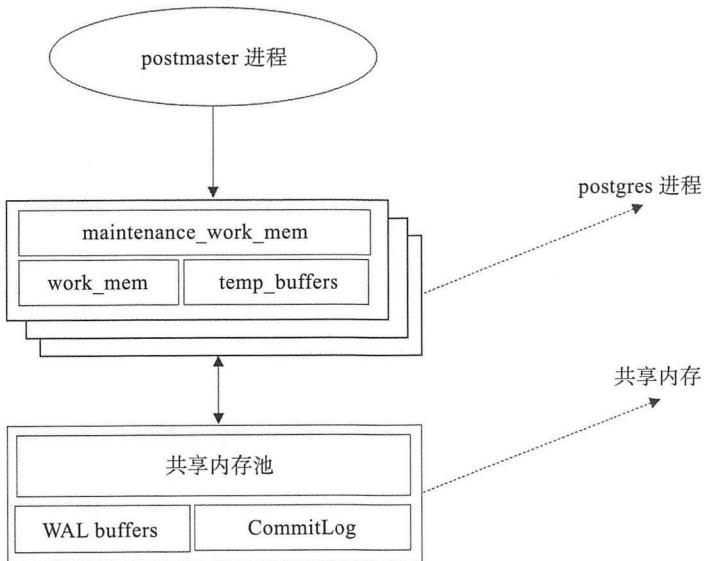


图 5-9 内存结构

5.4 本章小结

本章从全局角度简单讨论了 PostgreSQL 数据库的文件存储，介绍了构成数据库的参



数文件、数据文件布局，以及表空间、数据库、数据库对象的逻辑存储结构，简单介绍了 PostgreSQL 的守护进程、服务进程和辅助进程，介绍了客户端与数据库服务器连接交互方式，从数据库目录到表文件到最小的数据块，从大到小逐层分析了重要的数据文件。通过这些简单介绍，只能够窥探到 PostgreSQL 体系的冰山一角，PostgreSQL 体系结构中的每一个知识点都有足够丰富的内容，值得深入学习。PostgreSQL 有着多年的技术沉淀，清晰的代码结构，通过源代码深入学习可以事半功倍。



并行查询

了解 Oracle 的朋友应该知道 Oracle 支持并行查询，比如 SELECT、UPDATE、DELETE 大事务开启并行功能后能利用多核 CPU，从而充分发挥硬件性能，提升大事务处理效率，PostgreSQL 在 9.6 版本前还不支持并行查询，SQL 无法利用多核 CPU 提升性能，9.6 版本开始支持并行查询，只是 9.6 版本的并行查询所支持的范围非常有限，例如只在顺序扫描、多表关联、聚合查询中支持并行，10 版本增强了并行查询功能，例如增加了并行索引扫描、并行 index-only 扫描、并行 bitmap heap 扫描等，本章将介绍 PostgreSQL10 的并行查询功能。

6.1 并行查询相关配置参数

介绍 PostgreSQL 并行查询之前先来介绍并行查询的几个重要参数。

1. max_worker_processes(integer)

设置系统支持的最大后台进程数，默认值为 8，如果有备库，备库上此参数必须大于或等于主库上的此参数配置值，此参数调整后需重启数据库生效。

2. max_parallel_workers (integer)

设置系统支持的并行查询进程数，默认值为 8，此参数受 max_worker_processes 参数限制，设置此参数的值比 max_worker_processes 值高将无效。

当调整这个参数时建议同时调整 max_parallel_workers_per_gather 参数值。

3. max_parallel_workers_per_gather (integer)

设置允许启用的并行进程的进程数，默认值为 2，设置成 0 表示禁用并行查询，此参数受 max_worker_processes 参数和 max_parallel_workers 参数限制，因此并行查询的实际进程

数可能比预期的少，并行查询比非并行查询消耗更多的 CPU、IO、内存资源，对生产系统有一定影响，使用时需考虑这方面的因素，这三个参数的配置值大小关系通常如下所示：

```
max_worker_processes > max_parallel_workers > max_parallel_workers_per_gather
```

4. parallel_setup_cost(floating point)

设置优化器启动并行进程的成本，默认为 1000。

5. parallel_tuple_cost(floating point)

设置优化器通过并行进程处理一行数据的成本，默认为 0.1。

6. min_parallel_table_scan_size(integer)

设置开启并行的条件之一，表占用空间小于此值将不会开启并行，并行顺序扫描场景下扫描的数据大小通常等于表大小，默认值为 8MB。

7. min_parallel_index_scan_size(integer)

设置开启并行的条件之一，实际上并行索引扫描不会扫描索引所有数据块，只是扫描索引相关数据块，默认值为 512kb。

8. force_parallel_mode (enum)

强制开启并行，一般作为测试目的，OLTP 生产环境开启需慎重，一般不建议开启。

本章节中 postgresql.conf 配置文件设置了以下参数：

```
max_worker_processes = 16
max_parallel_workers_per_gather = 4      # taken from max_parallel_workers
max_parallel_workers = 8
parallel_tuple_cost = 0.1
parallel_setup_cost = 1000.0
min_parallel_table_scan_size = 8MB
min_parallel_index_scan_size = 512kB
force_parallel_mode = off
```

本章节将演示并行查询相关测试，测试环境为一台 4CPU、8GB 内存的虚拟机。



注意 并行查询进程数预估值由参数 `max_parallel_workers_per_gather` 控制，并行进程数预估值是指优化器解析 SQL 时执行计划预计会启用的并行进程数，而实际执行查询时的并行进程数受参数 `max_parallel_workers`、`max_worker_processes` 的限制，也就是说 SQL 实际获得的并行进程数不会超过这两个参数设置的值，比如 `max_worker_processes` 参数设置成 2，`max_parallel_workers_per_gather` 参数设置成 4，不考虑其他因素的情况下，并行查询实际的并行进程数将会是 2，另一方面并行进程数据会受 `min_parallel_table_scan_size` 参数的影响，即表的大小会影响并行进程数。并行查询执行计划中的 Workers Planned 表示执行计划预估的并行进程数，Worker Launched 表示并行查询实际获得的并行进程数。

6.2 并行扫描

上一小节介绍了 PostgreSQL 并行查询相关参数，接下来通过示例演示并行扫描，包括并行顺序扫描、并行索引扫描、并行 index-only 扫描、并行 bitmap heap 扫描场景，测试过程中会对上一小节提到的部分参数进行设置，通过实验了解这些参数的含义。

6.2.1 并行顺序扫描

介绍并行顺序扫描之前先介绍顺序扫描 (sequential scan)，顺序扫描通常也称之为全表扫描，全表扫描会扫描整张表数据，当表很大时，全表扫描会占用大量 CPU、内存、IO 资源，对数据库性能有较大影响，在 OLTP 事务型数据库系统中应当尽量避免。

首先创建一张测试表，并插入 5000 万数据，如下所示：

```
CREATE TABLE test_big1(
    id int4,
    name character varying(32),
    create_time timestamp without time zone default clock_timestamp();

INSERT INTO test_big1(id, name)
SELECT n, n || '_test' FROM generate_series(1, 50000000) n ;
```

一个顺序扫描的示例如下所示：

```
mydb=> EXPLAIN SELECT * FROM test_big1 WHERE name='1_test';
          QUERY PLAN
-----
 Seq Scan on test_big1  (cost=0.00..991664.00 rows=1 width=25)
   Filter: ((name)::text = '1_test'::text)
 (2 rows)
```

以上执行计划 Seq Scan on test_big1 说明表 test_big1 上进行了顺序扫描，这是一个典型的顺序扫描执行计划，PostgreSQL 中的顺序扫描在 9.6 版本开始支持并行处理，并行顺序扫描会产生多个子进程，并利用多个逻辑 CPU 并行全表扫描，一个并行顺序扫描的执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM test_big1 WHERE name='1_test';
          QUERY PLAN
-----
 Gather  (cost=1000.00..523914.10 rows=1 width=25)  (actual time=0.440..1362.675
rows=1 loops=1)
   Workers Planned: 4
   Workers Launched: 4
   -> Parallel Seq Scan on test_big1  (cost=0.00..522914.00 rows=1 width=25)  (actual
time=1083.280..1355.685 rows=0 loops=5)
      Filter: ((name)::text = '1_test'::text)
      Rows Removed by Filter: 10000000
Planning time: 0.085 ms
```



```
Execution time: 1367.248 ms
(8 rows)
```

注意以上执行计划加粗的三行，Workers Planned 表示执行计划预估的并行进程数，Worker Launched 表示查询实际获得的并行进程数，这里 Workers Planned 和 Worker Launched 值都为 4，Parallel Seq Scan on test_big1 表示进行了并行顺序扫描，Planning time 表示生成执行计划的时间，Execution time 表示 SQL 实际执行时间，从以上可以看出，开启 4 个并行时 SQL 实际执行时间为 1367 毫秒。接下来测试不开启并行的 SQL 性能，由于 max_parallel_workers_per_gather 参数设置成了 4，设置成 0 表示关闭并行，在会话级别设置此参数值为 0，如下所示：

```
mydb=> SET max_parallel_workers_per_gather =0;
SET
```

不开启并行，执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT * FROM test_big1 WHERE name='1_test';
          QUERY PLAN
-----
Seq Scan on test_big1  (cost=0.00..991664.00 rows=1 width=25)
  (actual time=0.022..5329.100 rows=1 loops=1)
    Filter: ((name)::text = '1_test'::text)
    Rows Removed by Filter: 49999999
  Planning time: 0.163 ms
  Execution time: 5329.136 ms
(5 rows)
```

不开启并行时此 SQL 执行时间为 5329 毫秒，比开启并行查询性能低了 3 倍左右。

6.2.2 并行索引扫描

介绍并行索引扫描之前，先简单介绍下索引扫描（index scan），在表上创建索引后，进行索引扫描的执行计划如下所示：

```
mydb=> EXPLAIN SELECT * FROM test_1 WHERE id=1;
          QUERY PLAN
-----
Index Scan using test_1_pkey on test_1  (cost=0.43..4.45 rows=1 width=26)
  Index Cond: (id = 1)
(2 rows)
```

Index Scan using 表示执行计划预计进行索引扫描，索引扫描也支持并行，称为并行索引扫描（Parallel index scan），本节演示并行索引扫描，首先在表 test_big1 上创建索引，如下所示：

```
mydb=> CREATE INDEX idx_test_big1_id ON test_big1 USING btree (id);
CREATE INDEX
```



执行以下 SQL，统计 ID 小于 1 千万的记录数，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT count(name) FROM test_big1 WHERE id<10000000;
          QUERY PLAN
-----
Finalize Aggregate (cost=236183.98..236183.99 rows=1 width=8)
  (actual time=753.392..753.392 rows=1 loops=1)
    -> Gather (cost=236183.96..236183.97 rows=4 width=8)
      (actual time=750.133..753.384 rows=5 loops=1)
        Workers Planned: 4
        Workers Launched: 4
      -> Partial Aggregate (cost=235183.96..235183.97 rows=1 width=8) (actual
          time=746.344..746.344 rows=1 loops=5)
        -> Parallel Index Scan using idx_test_big1_id on test_big1
          (cost=0.56..228921.05 rows=2505162 width=13) (actual tim
          e=0.029..566.830 rows=2000000 loops=5)
            Index Cond: (id < 10000000)
Planning time: 0.116 ms
Execution time: 762.351 ms
(9 rows)
```

根据以上执行计划可以看出，进行了并行索引扫描，开启了 4 个并行进程，执行时间为 762 毫秒，在会话级别关闭并行查询，如下所示：

```
mydb=> SET max_parallel_workers_per_gather =0;
SET
```

再次执行以上查询，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT count(name) FROM test_big1 WHERE id<10000000;
          QUERY PLAN
-----
Aggregate (cost=329127.54..329127.55 rows=1 width=8) (actual time=2636.859..2636.859
rows=1 loops=1)
  -> Index Scan using idx_test_big1_id on test_big1 (cost=0.56..304075.92
rows= 10020649 width=13) (actual time=0.031..1654.500 ro
ws=9999999 loops=1)
    Index Cond: (id < 10000000)
Planning time: 0.132 ms
Execution time: 2636.920 ms
(5 rows)
```

从执行计划看出进行了索引扫描，没有开启并行，执行时间为 2636 毫秒，比并行索引扫描性能低很多。



PostgreSQL 10 对并行扫描的支持将提升范围扫描 SQL 的性能，由于开启并行将消耗更多的 CPU、内存、IO 资源，设置并行进程数时得合理考虑，另一方面，目前 PostgreSQL 10 暂不支持非 btree 索引类型的并行索引扫描。



6.2.3 并行 index-only 扫描

了解并行 index-only 扫描之前首先介绍下 index-only 扫描，顾名思义，index-only 扫描是指只需扫描索引，也就是说 SQL 仅根据索引就能获得所需检索的数据，而不需要通过索引回表查询数据。例如，使用 SQL 统计 ID 小于 100 万的记录数，在开始测试之前，先在会话级别关闭并行，如下所示

```
mydb=> SET max_parallel_workers_per_gather =0;
SET
```

之后执行以下 SQL，查看执行计划，如下所示：

```
mydb=> EXPLAIN SELECT count(*) FROM test_big1 WHERE id<1000000;
          QUERY PLAN
-----
Aggregate  (cost=36060.91..36060.92 rows=1 width=8)
    -> Index Only Scan using idx_test_big1_id on test_big1  (cost=0.56..33313.99
        rows=1098767 width=0)
        Index Cond: (id < 1000000)
(3 rows)
```

以上执行计划主要看 Index Only Scan 这一行，由于 ID 字段上建立了索引，统计记录数不需要再回表查询其他信息，因此进行了 index-only 扫描，接下来使用 EXPLAIN ANALYZE 执行此 SQL，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT count(*) FROM test_big1 WHERE id<1000000;
          QUERY PLAN
-----
Aggregate  (cost=35969.89..35969.90 rows=1 width=8)
(actual time=253.571..253.571 rows=1 loops=1)
    -> Index Only Scan using idx_test_big1_id on test_big1  (cost=0.56..33232.22
        rows=1095066 width=0) (actual time=0.038..179.005 rows=999999 loops=1)
        Index Cond: (id < 1000000)
        Heap Fetches: 999999
Planning time: 0.103 ms
Execution time: 253.617 ms
(6 rows)
```

执行时间为 253 毫秒，index-only 扫描支持并行，称为并行 index-only 扫描，接着测试并行 index-only 扫描，在会话级别开启并行功能，如下所示：

```
mydb=> SET max_parallel_workers_per_gather TO default;
SET
```

再次执行以下查询，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT count(*) FROM test_big1 WHERE id<1000000;
          QUERY PLAN
```



```
-- Finalize Aggregate (cost=26703.66..26703.67 rows=1 width=8)
--   (actual time=81. 285..81.285 rows=1 loops=1)
--     -> Gather (cost=26703.64..26703.65 rows=4 width=8)
--       (actual time=81. 121..81.277 rows=5 loops=1)
--         Workers Planned: 4
--         Workers Launched: 4
--           -> Partial Aggregate (cost=25703.64..25703.65 rows=1 width=8)
--             (actual time=75.778..75.778 rows=1 loops=5)
--               -> Parallel Index Only Scan using idx_test_big1_id on test_big1
--                 (cost=0.56..25019.22 rows=273766 width=0)
--                   (actual time=0.045..59.398 rows=200000 loops=5)
--                     Index Cond: (id < 1000000)
--                     Heap Fetches: 183366
-- Planning time: 0.113 ms
-- Execution time: 83.364 ms
(10 rows)
```

以上执行计划主要看 Parallel Index Only Scan 这段，进行了并行 index-only 扫描，执行时间降为 83 毫秒，开启并行后性能提升了不少。

6.2.4 并行 bitmap heap 扫描

介绍并行 bitmap heap 扫描之前先了解下 Bitmap Index 扫描和 Bitmap Heap 扫描，当 SQL 的 where 条件中出现 or 时很有可能出现 Bitmap Index 扫描，如下所示：

```
mydb=> EXPLAIN SELECT * FROM test_big1 WHERE id=1 OR id=2;
          QUERY PLAN
-----
Bitmap Heap Scan on test_big1 (cost=5.15..9.17 rows=2 width=25)
  Recheck Cond: ((id = 1) OR (id = 2))
    -> BitmapOr (cost=5.15..5.15 rows=2 width=0)
      -> Bitmap Index Scan on idx_test_big1_id (cost=0.00..2.57 rows=1 width=0)
        Index Cond: (id = 1)
      -> Bitmap Index Scan on idx_test_big1_id (cost=0.00..2.57 rows=1 width=0)
        Index Cond: (id = 2)
(7 rows)
```

从以上执行计划看出，首先执行两次 Bitmap Index 扫描获取索引项，之后将 Bitmap Index 扫描获取的结果合起来回表查询，这时在表 test_big1 上进行了 Bitmap Heap 扫描。Bitmap Heap 扫描也支持并行，执行以下 SQL，在查询条件中将 ID 的选择范围扩大。

```
EXPLAIN ANALYZE SELECT count(*) FROM test_big1 WHERE id <1000000 OR id > 49000000;
```

执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT count(*) FROM test_big1 WHERE id <1000000 OR id > 49000000;
```



QUERY PLAN

```
Finalize Aggregate (cost=406220.88..406220.89 rows=1 width=8)
  (actual time=241.186..241.186 rows=1 loops=1)
    -> Gather (cost=406220.46..406220.87 rows=4 width=8)
      (actual time=241.033..241.174 rows=5 loops=1)
        Workers Planned: 4
        Workers Launched: 4
      -> Partial Aggregate (cost=405220.46..405220.47 rows=1 width=8) (actual
          time=237.266..237.266 rows=1 loops=5)
        -> Parallel Bitmap Heap Scan on test_big1 (cost=28053.14..403933.27
            rows=514876 width=0) (actual time=83.059..189.073 rows=400000 loops=5)
          Recheck Cond: ((id < 1000000) OR (id > 49000000))
          Heap Blocks: exact=2982
        -> BitmapOr (cost=28053.14..28053.14 rows=2081101 width=0)
          (actual time=83.657..83.657 rows=0 loops=1)
          -> Bitmap Index Scan on idx_test_big1_id (cost=0.00..
              14219.03 rows=1094996 width=0)
            (actual time=42.187..42.187 rows=999999 loops=1)
            Index Cond: (id < 1000000)
          -> Bitmap Index Scan on idx_test_big1_id
            (cost=0.00.. 12804.35 rows=986105 width=0)
            (actual time=41.467..41.467 rows=1000000 loops=1)
            Index Cond: (id > 49000000)
Planning time: 0.157 ms
Execution time: 242.824 ms
(15 rows)
```

从以上执行计划看出进行了并行 Bitmap Heap 扫描，并行进程数为 4，执行时间为 242ms。在会话级关闭并行查询，如下所示：

```
mydb=> SET max_parallel_workers_per_gather =0;
SET
```

再次执行以上 SQL，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT count(*) FROM test_big1 WHERE id <1000000 OR id >
49000000;
```

QUERY PLAN

```
Aggregate (cost=432494.42..432494.43 rows=1 width=8) (actual time=466.273.. 466.273
rows=1 loops=1)
  -> Bitmap Heap Scan on test_big1 (cost=28053.14..427345.66 rows=2059505
    width=0) (actual time=86.859..299.822 rows=1999999 loops=1)
    Recheck Cond: ((id < 1000000) OR (id > 49000000))
    Heap Blocks: exact=13724
  -> BitmapOr (cost=28053.14..28053.14 rows=2081101 width=0)
    (actual time=84.782..84.782 rows=0 loops=1)
```



```
--> Bitmap Index Scan on idx_test_big1_id (cost=0.00..14219.03 rows=1094996 width=0) (actual time=42.704..42.704 rows=999999 loops=1)
Index Cond: (id < 1000000)
--> Bitmap Index Scan on idx_test_big1_id (cost=0.00..12804.35 rows=986105 width=0) (actual time=42.076..42.076 rows=1000000 loops=1)
Index Cond: (id > 49000000)
Planning time: 0.152 ms
Execution time: 466.323 ms
(11 rows)
```

从以上执行计划看出进行了 Bitmap Heap 扫描，执行时间上升到 466 毫秒，不开启并行比开启并行性能低了不少。

6.3 并行聚合

上一小节介绍了 PostgreSQL 并行扫描，这一小节介绍并行聚合，并通过示例演示。聚合操作是指使用 count()、sum() 等聚合函数的 SQL，以下执行 count() 函数统计表记录总数，执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT count(*) FROM test_big1;
          QUERY PLAN
-----
Finalize Aggregate (cost=525335.91..525335.92 rows=1 width=8)
(actual time=2468.593.. 2468.594 rows=1 loops=1)
-> Gather (cost=525335.89..525335.90 rows=4 width=8)
(actual time=2468. 386..2468.585 rows=5 loops=1)
Workers Planned: 4
Workers Launched: 4
-> Partial Aggregate (cost=524335.89..524335.90 rows=1 width=8)
(actual time=2463.532..2463.532 rows=1 loops=5)
-> Parallel Seq Scan on test_big1 (cost=0.00..493083.91
rows= 12500791 width=0) (actual time=0.019..1456.506 rows=10000000
loops=5)
Planning time: 0.089 ms
Execution time: 2474.970 ms
(8 rows)
```

从以上执行计划看出，首先进行 Partial Aggregate，开启了四个并行进程，最后进行 Finalize Aggregate，此 SQL 执行时间为 2474 毫秒，在操作系统层面通过 top 命令能看到 EXPLAIN ANALYZE 命令的四个子进程，如图 6-1。

这个例子充分验证了聚合查询 count() 能够支持并行，为了初步测试并行性能，在会话级别关闭并行查询，如下所示：

```
mydb=> SET max_parallel_workers_per_gather = 0 ;
SET
```





```

top - 22:06:57 up 177 days, 8:01, 3 users, load average: 1.98, 0.60, 0.21
Tasks: 213 total, 6 running, 207 sleeping, 0 stopped, 0 zombie
Cpu(s): 88.3%us, 7.4%sy, 0.0%ni, 4.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8062340k total, 6643848k used, 1418492k free, 76912k buffers
Swap: 0k total, 0k used, 0k free, 5949344k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
28940	postgres	20	0	1233m	168m	167m	R	84.8	2.1	0:02.55	postgres: bgworker: parallel worker for PID 24988
28939	postgres	20	0	1233m	217m	216m	R	81.4	2.8	0:02.45	postgres: bgworker: parallel worker for PID 24988
24988	postgres	20	0	1246m	1.0g	987m	R	77.8	12.7	3:06.46	postgres: pguser mydb [local] EXPLAIN
28938	postgres	20	0	1233m	195m	194m	R	69.8	2.5	0:02.10	postgres: bgworker: parallel worker for PID 24988
28941	postgres	20	0	1233m	141m	140m	R	67.1	1.8	0:02.02	postgres: bgworker: parallel worker for PID 24988

图 6-1 top 命令查看并行进程

再次执行以上查询，如下所示：

```

mydb=> EXPLAIN ANALYZE SELECT count(*) FROM test_big1;
          QUERY PLAN
-----
Aggregate  (cost=993115.55..993115.56 rows=1 width=8)
(actual time=8655.614.. 8655.615 rows=1 loops=1)
->  Seq Scan on test_big1  (cost=0.00..868107.64 rows=50003164 width=0)
(actual time=0.019..4898.227 rows=50000000 loops=1)
Planning time: 0.106 ms
Execution time: 8655.666 ms
(4 rows)

```

从执行计划看出，关闭并行查询后进行了顺序扫描，执行时间由原来的 2474 毫秒上升为 8865 毫秒，性能降低近 3 倍，尝试将并行进程数更改为 2 再次进行性能对比。首先在会话级别修改以下参数，如下所示：

```

mydb=> SET max_parallel_workers_per_gather =2;
SET

```

再次执行以下 SQL，如下所示：

```

mydb=> EXPLAIN ANALYZE SELECT count(*) FROM test_big1;
          QUERY PLAN
-----
Finalize Aggregate  (cost=629509.16..629509.17 rows=1 width=8)
(actual time=3305. 585..3305.585 rows=1 loops=1)
->  Gather  (cost=629509.15..629509.16 rows=2 width=8)
(actual time=3305. 512..3305.578 rows=3 loops=1)
Workers Planned: 2
Workers Launched: 2
->  Partial Aggregate  (cost=628509.15..628509.16 rows=1 width=8) (actual
time=3302.362..3302.362 rows=1 loops=3)
->  Parallel Seq Scan on test_big1  (cost=0.00..576422.52 rows= 20834652
width=0) (actual time=0.021..2000.427 rows=16666667 loops=3)
Planning time: 0.112 ms
Execution time: 3314.371 ms
(8 rows)

```





从执行计划看出，开启了两个并行进程，执行时间为 3314 毫秒，之后我们测试并行进程数为 6、8 时此 SQL 的执行时间，执行时间汇总如表 6-1 所示。

表 6-1 不同并行进程数下的全表扫描执行时间

并行进程数	Count() 执行时间
0	8865 毫秒
2	3314 毫秒
4	2474 毫秒
6	2479 毫秒
8	2446 毫秒

从表 6-1 可以看出，并行进程数为 8 时执行时间最短，但与并行进程数为 4、6 时执行时间非常接近，当并行进程数设置比 4 高时，执行时间几乎没有变化，这是受数据库主机 CPU 核数限制的，本机测试环境为 4 核 CPU。

sum() 函数也能支持并行扫描，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT sum(hashtext(name)) FROM test_big1;
          QUERY PLAN
-----
      Finalize Aggregate  (cost=555163.25..555163.26 rows=1 width=8)
        (actual time=3307. 939..3307.939 rows=1 loops=1)
          -> Gather  (cost=555162.83..555163.24 rows=4 width=8)
            (actual time=3307. 694..3307.934 rows=5 loops=1)
              Workers Planned: 4
              Workers Launched: 4
            -> Partial Aggregate  (cost=554162.83..554162.84 rows=1 width=8) (actual
                time=3303.628..3303.628 rows=1 loops=5)
              > Parallel Seq Scan on test_big1  (cost=0.00..491663.22 rows=12499922
                width=13) (actual time=0.045..1837.554 rows=10000000 loops=5)

Planning time: 0.078 ms
Execution time: 3308.999 ms
(8 rows)
```

min()、max() 聚合函数也支持并行查询，这里不再测试。

6.4 多表关联

多表关联也能用到并行扫描，例如 nested loop、merge join、hash join，多表关联场景能够使用并行并不是指多表关联本身使用并行，而是指多表关联涉及的表数据检索时能够使用并行处理，这一小节将分别介绍三种多表关联方式使用并行的场景。



6.4.1 Nested loop 多表关联

多表关联 Nested loop 实际上是一个嵌套循环，伪代码如下所示：

```
for (i = 0; i < length(outer); i++)
    for (j = 0; j < length(inner); j++)
        if (outer[i] == inner[j])
            output(outer[i], inner[j]);
```

接着测试 Nested loop 多表关联场景下使用到并行扫描的情况，创建一张 test_small 小表，如下所示：

```
CREATE TABLE test_small(id int4, name character varying(32));

INSERT INTO test_small(id, name)
SELECT n, n|| '_small' FROM generate_series(1,8000000) n ;
```

创建索引并做表分析，如下所示：

```
mydb=> CREATE INDEX idx_test_small_id ON test_small USING btree (id);
CREATE INDEX
```

```
mydb=> ANALYZE test_small;
ANALYZE
```

ANALYZE 命令用于收集表上的统计信息，使优化器能够获得更准确的执行计划，两表关联执行计划如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT test_small.name
      FROM test_big1, test_small
     WHERE test_big1.id = test_small.id
       AND test_small.id < 10000;
                                         QUERY PLAN
-----
Gather  (cost=1138.18..31628.76 rows=10217 width=13) (actual time=1.036..16.207
  rows=9999 loops=1)
  Workers Planned: 3
  Workers Launched: 3
    -> Nested Loop (cost=138.18..29607.06 rows=3296 width=13)
        (actual time= 0.244..10.895 rows=2500 loops=4)
          -> Parallel Bitmap Heap Scan on test_small (cost=137.61..14796.65
              rows= 3296 width=17) (actual time=0.203..0.653 rows=2500 loops=4)
              Recheck Cond: (id < 10000)
              Heap Blocks: exact=10
                -> Bitmap Index Scan on idx_test_small_id (cost=0.00..135.06
                    rows=10217 width=0) (actual time=0.652..0.652 rows=9999
                    loops=1)
                    Index Cond: (id < 10000)
          -> Index Only Scan using idx_test_big1_id on test_big1 (cost=0.56..4.48
              rows=1 width=4) (actual time=0.003..0.003 rows=1 loops=9999)
              Index Cond: (id = test_small.id)
              Heap Fetches: 1674
```



```
Planning time: 0.427 ms
Execution time: 17.548 ms
(14 rows)
```

从以上执行计划可以看出，首先在表 test_big1 上进行了 Index Only 扫描，用于检索 id 小于 10000 的记录，之后两表进行 Nested loop 关联同时在表 test_small1 上进行了并行 Bitmap Heap 扫描，用于检索 id 小于 10000 的记录，开启并行情况下这条 SQL 执行时间为 17.5 毫秒。如果关闭并行，如下所示：

```
mydb=> SET max_parallel_workers_per_gather =0;
SET
```

再次查看执行计划，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT test_small.name
      FROM test_big1, test_small
     WHERE test_big1.id = test_small.id
       AND test_small.id < 10000;
                                         QUERY PLAN
-----
Nested Loop  (cost=1.00..46213.80 rows=10217 width=13) (actual time=0.025..29.054
  rows=9999 loops=1)
    -> Index Scan using idx_test_small_id on test_small  (cost=0.43..304.23
      rows=10217 width=17) (actual time=0.014..2.284 rows=9999 loops=1)
          Index Cond: (id < 10000)
    -> Index Only Scan using idx_test_big1_id on test_big1  (cost=0.56..4.48
      rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=9999)
          Index Cond: (id = test_small.id)
          Heap Fetches: 9999
Planning time: 0.262 ms
Execution time: 29.606 ms
(8 rows)
```

从以上执行计划看出不开启并行此 SQL 执行时间为 29.6 毫秒左右，性能差别还是挺大的。

6.4.2 Merge join 多表关联

Merge join 多表关联首先将两个表进行排序，之后进行关联字段匹配，Merge join 示例如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT test_small.name
      FROM test_big1, test_small
     WHERE test_big1.id = test_small.id
       AND test_small.id < 200000;
```

QUERY PLAN

```
Gather  (cost=1001.79..195146.04 rows=204565 width=13) (actual time=0.308.. 160.192 rows=199999 loops=1)
  Workers Planned: 4
```



```

Workers Launched: 4
-> Merge Join  (cost=1.79..173689.54 rows=51141 width=13)
    (actual time=4.059.. 127.164 rows=40000 loops=5)
        Merge Cond: (test_big1.id = test_small.id)
            -> Parallel Index Only Scan using idx_test_big1_id on test_big1
                (cost=0.56..1017275.56 rows=12500000 width=4)
                (actual time=0.044..16.223 rows=40001 loops=5)
        Heap Fetches: 50875
-> Index Scan using idx_test_small_id on test_small  (cost=0.43..6010.32
    rows=204565 width=17) (actual time=0.034..64.427 rows=199999 loops=5)
    Index Cond: (id < 200000)
Planning time: 0.255 ms
Execution time: 171.755 ms
(11 rows)

```

开启了四个并行，执行时间为 171 毫秒左右。下面关闭并行进行比较，如下所示：

```
mydb=> SET max_parallel_workers_per_gather =0;
SET
```

再次查看执行计划，如下所示：

```

mydb=> EXPLAIN ANALYZE SELECT test_small.name
      FROM test_big1, test_small
     WHERE test_big1.id = test_small.id
       AND test_small.id < 200000;
                                         QUERY PLAN
-----
Merge Join  (cost=1.79..249728.34 rows=204565 width=13) (actual time=0.034..
198.331 rows=199999 loops=1)
    Merge Cond: (test_big1.id = test_small.id)
        -> Index Only Scan using idx_test_big1_id on test_big1  (cost=0.56..1392275.56
            rows=50000000 width=4) (actual time=0.018..60.445
            rows=200000 loops=1)
            Heap Fetches: 200000
        -> Index Scan using idx_test_small_id on test_small  (cost=0.43..6010.32
            rows=204565 width=17) (actual time=0.013..50.531 rows=199999 loops=1)
            Index Cond: (id < 200000)
Planning time: 0.264 ms
Execution time: 209.387 ms
(8 rows)

```

从以上执行计划看出不开启并行此 SQL 执行时间为 209 毫秒左右，执行时间比开启并行略长。

6.4.3 Hash join 多表关联

PostgreSQL 多表关联也支持 Hash join，当关联字段没有索引情况下两表关联通常会进行 Hash join，接下来查看 Hash join 的执行计划，先将两张表上的索引删除，同时关闭并行，如下所示：



```
mydb=> DROP INDEX idx_test_big1_id;
DROP INDEX
mydb=> DROP INDEX idx_test_small_id ;
DROP INDEX

mydb=> SET max_parallel_workers_per_gather =0;
SET
```

两表关联执行计划如下所示：

```
mydb=> EXPLAIN SELECT test_small.name
      FROM test_big1 JOIN test_small ON (test_big1.id = test_small.id)
      AND test_small.id < 100;
                                QUERY PLAN
-----
Hash Join  (cost=150871.78..1205043.77 rows=800 width=13)
  Hash Cond: (test_big1.id = test_small.id)
    -> Seq Scan on test_big1  (cost=0.00..866664.00 rows=50000000 width=4)
    -> Hash  (cost=150861.78..150861.78 rows=800 width=17)
          -> Seq Scan on test_small  (cost=0.00..150861.78 rows=800 width=17)
              Filter: (id < 100)
(6 rows)
```

下面实际执行此 SQL，查看性能，如下所示：

```
mydb=> EXPLAIN ANALYZE SELECT test_small.name
      FROM test_big1 JOIN test_small ON (test_big1.id = test_small.id)
      AND test_small.id < 100;
                                QUERY PLAN
-----
Hash Join  (cost=151317.06..1206944.59 rows=802 width=13)
  (actual time=735. 778..11259.744 rows=99 loops=1)
  Hash Cond: (test_big1.id = test_small.id)
    -> Seq Scan on test_big1  (cost=0.00..868107.64 rows=50003164 width=4) (actual
        time=0.015..5424.515 rows=50000000 loops=1)
    -> Hash  (cost=151307.04..151307.04 rows=802 width=17)
          (actual time=735. 750..735.750 rows=99 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 13kB
              -> Seq Scan on test_small  (cost=0.00..151307.04 rows=802 width=17)
                  (actual time=0.010..735.731 rows=99 loops=1)
                  Filter: (id < 100)
                  Rows Removed by Filter: 7999901
Planning time: 0.134 ms
Execution time: 11259.789 ms
(10 rows)
```

从以上看出，执行时间为 11 259 毫秒。如果开启 4 个并行，如下所示：

```
mydb=> SET max_parallel_workers_per_gather =4;
SET
```

再次执行 SQL，如下所示：



```

mydb=> EXPLAIN ANALYZE SELECT test_small.name
   FROM test_big1 JOIN test_small ON (test_big1.id = test_small.id)
   AND test_small.id < 100;
                                         QUERY PLAN
-----
Gather  (cost=152317.06..692280.94 rows=802 width=13)
  (actual time=1152.263..4304.757 rows=99 loops=1)
    Workers Planned: 4
    Workers Launched: 4
-> Hash Join  (cost=151317.06..691280.94 rows=200 width=13) (actual time=
   3667.973..4298.423 rows=20 loops=5)
      Hash Cond: (test_big1.id = test_small.id)
      -> Parallel Seq Scan on test_big1  (cost=0.00..493083.91 rows=12500791
          width=4) (actual time=0.025..1737.558 rows=10000000 loops=5)
      -> Hash  (cost=151307.04..151307.04 rows=802 width=17)
          (actual time=1106.665..1106.665 rows=99 loops=5)
          Buckets: 1024  Batches: 1  Memory Usage: 13kB
          -> Seq Scan on test_small  (cost=0.00..151307.04 rows=802 width=17)
              (actual time=863.670..1106.624 rows=99 loops=5)
              Filter: (id < 100)
              Rows Removed by Filter: 7999901
Planning time: 0.156 ms
Execution time: 4315.928 ms
(13 rows)

```

从以上执行计划看出，开启了 4 个并行，执行时间下降为 4315 毫秒。

6.5 本章小结

本章主要介绍了 PostgreSQL 并行查询相关内容，包括并行查询相关配置参数、并行顺序扫描、并行索引扫描、并行 index-only 扫描、并行 bitmap heap 扫描，同时介绍了多表关联场景中并行的使用。从本节测试示例可以看出，大部分能够启用并行的场景对 SQL 性能都有较大幅度提升，甚至是提升 3 到 4 倍，通过阅读本章读者能了解 PostgreSQL 并行查询支持的场景，同时对并行查询相关配置参数有一定理解。

