PostgreSQL 性能优化

作者：黄金华

2017/02/22

1 PostgreSQL 简介

1.1 功能简介

PostgreSQL 是一款开源的面向对象关系型数据库。除了支持标准 SQL 功能外，还支持许多现代功能：

- 复杂检索
- 外部键
- 触发器
- 事务完整性
- MVCC（多版本并发控制）

同时，PostgreSQL 还提供接口方便用户扩展功能，例如：

- 新增数据类型
- 新增函数
- 新增运算符
- 新增聚合功能
- 新增索引方法

1.2 与其他关系型数据库对比

就关系数据库而言，除去商业版 Oracle，目前广泛使用的就是开源的 MySQL 和 PostgreSQL。因此，主要对这两种主要的关系型数据库进行对比。

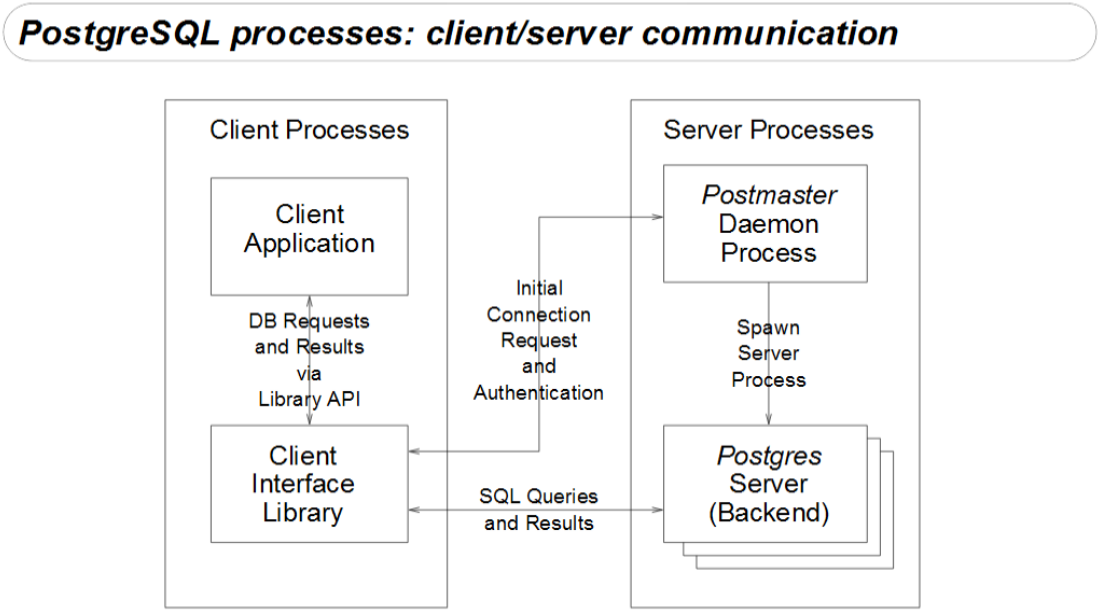
	MySQL(InnoDB)	PostgreSQL
连接处理	创建或复用线程处理新的连接请求。通常新线程创建开销小，同时可创建数目多。	创新新的进程处理连接请求。通常创建新进程开销大，同时可创建进程数目受系统资源影响较大，因而相对较少。因此，常借助 PgBouncer 或 PgPool-Ⅱ等共享连接，避免因新连接导致频繁的进程创建和销毁
缓存处理	主进程分配大块缓存（Buffer Pool）供进程来所有线程共享。同时支持 O_DIRECT 方式写入，避免文件系统再次缓存。	主进程与所有工作进程通过共享内存方式访问共同缓存（Shared Buffer）。因不直接支持 O_DIRECT，因此会同时存在 PostgreSQL 内部缓存和文件系统本身的缓存，这种双缓存既浪费了内存，又会造成因未及时写回磁盘导致数据丢失风险
数据类型	支持基本数据类型	除支持基本类型外， 还是支持复杂的数据类型，例如 JSON 等，同时支持扩展数量类型、函数以及索引等；
数据存储与索引	其内部索引通常是二级索引，也就	其内部通常是一级索引，也就是说

	是说，一级索引是索引 key 到主键，而二级索引是主键到物理存储位置。优点是，因物理存储位置发生变化时只需要更新二级索引，而缺点是增加了 I/O 请求。 同时，当数据修改更新时，是在原数据上直接更新，而如果原数据还正在被引用则会被复制到新的地方。 此外，其索引类型是 B-TREE	是从索引 Key 到磁盘位置直接建立索引。同时，对数据的更新修改是重新分配磁盘空间，而非在原来位置直接更新。其缺点是，当小量数据更新时，需要重建相关索引。 同时，除支持基本 B-Tree 索引类型外，还支持 Hash/GiST/SP-GiST/GIT/BRIN 等索引类型

简单来说，MySQL 是一种 Thin 数据库，也就是说数据库只是作为数据的存储提供简单的查询服务。而对应的，PostgreSQL 属于一种 Thick 数据库，其不仅提供数据存储，同时还提供复杂业务逻辑处理能力。

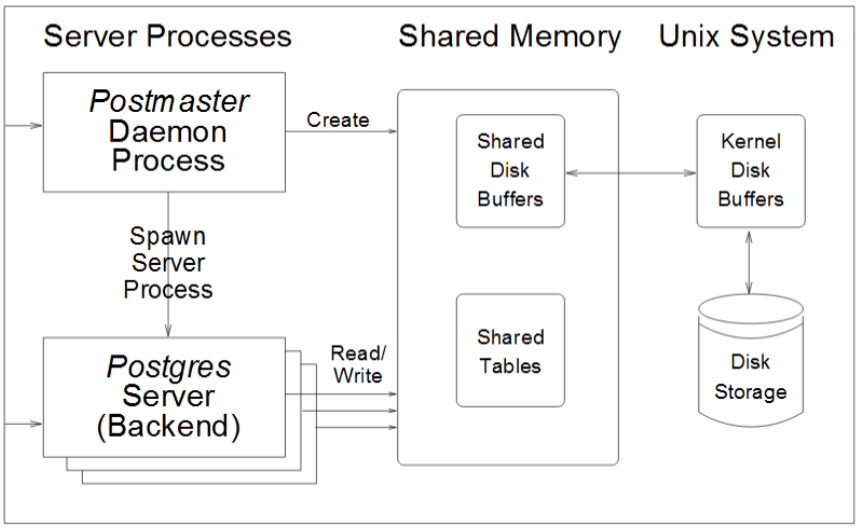
2 PostgreSQL 内部机制

2.1 架构总览

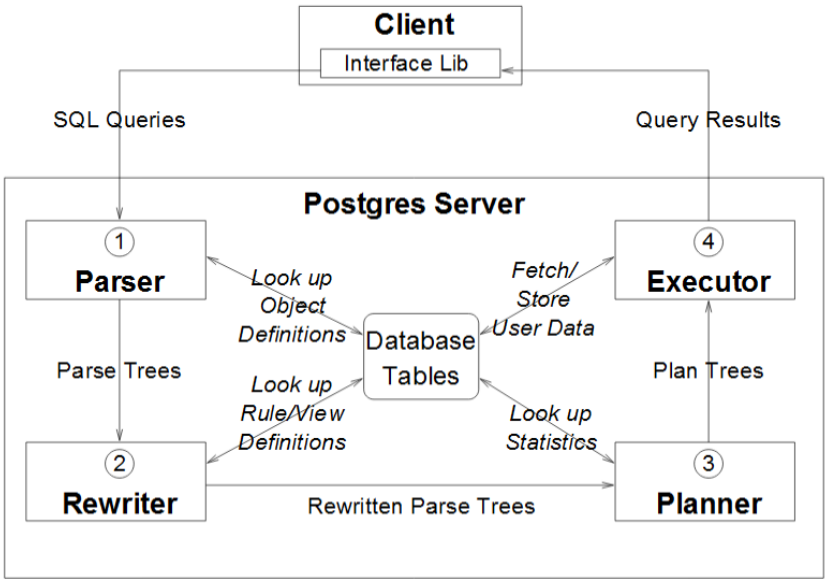


- Multiple client libraries offer different APIs: libpq, ODBC, JDBC,Perl DBD
- Client libraries insulate client applications from changes inon-the-wire protocol

PostgreSQL processes: inter-server communication



Steps of query processing: overview



Key data structures: parse tree, plan tree

2.2 计划执行

计划本质上是一个“树”，其每个节点代表一种操作（例如：join、disk scan、sort 等），并且每个节点向自己的父节点返回自己的操作结果，而根节点操作返回最终结果。顺便提一句，无论 update 还是 insert 本质上都是采用类似 select 的 “树”型的计划，其最主要的差别是根节点操作不同。

正如“架构总览”一节所描述的那样，一个 SQL 语句执行是需要优化器根据计算备选计划

的代价（cost）来计算最优计划，并最终执行最优计划。

在计算每个计划 Cost 的时候，基于下列考虑：

- 假定磁盘 I/O 是所有 cost 中最主要的因素，同时随机 I/O 要远高于顺序 I/O；
- 根据 ANALYZE 统计的结果预测每个计划的中间结果大小从而估算 I/O 次数（或代价）；（注释：因此后续优化的时候要注意，有可能当前优化器所选的计划实际上并不是最优的。此时需要执行 ANALYZE 命令更新统计数据）
- 对 cost 的计算，一般分为”startup cost” (获取第一个 Tuple 的代价)和该计划的总代价；

目前绝大部分数据库采用 System-R 优化器算法。

$$\text{Cost} = \underbrace{c_{\text{seq}} n_{\text{seq}}}_{\textit{Seq. I/O}} + \underbrace{c_{\text{rand}} n_{\text{rand}}}_{\textit{Random I/O}} + \underbrace{c_{\text{tup}} n_{\text{tup}}}_{\textit{CPU cost per tuple}} + \cdots$$

$$= \mathbb{C} \cdot \mathbb{N}$$

$\mathbb{C}$ Cost of single operation

- seq_page_cost
- random_page_cost
- cpu_tuple_cost
- cpu_index_tuple_cost
- cpu_operator_cost
- (parallel_tuple_cost)

$\mathbb{N}$ Estimated number of each operation

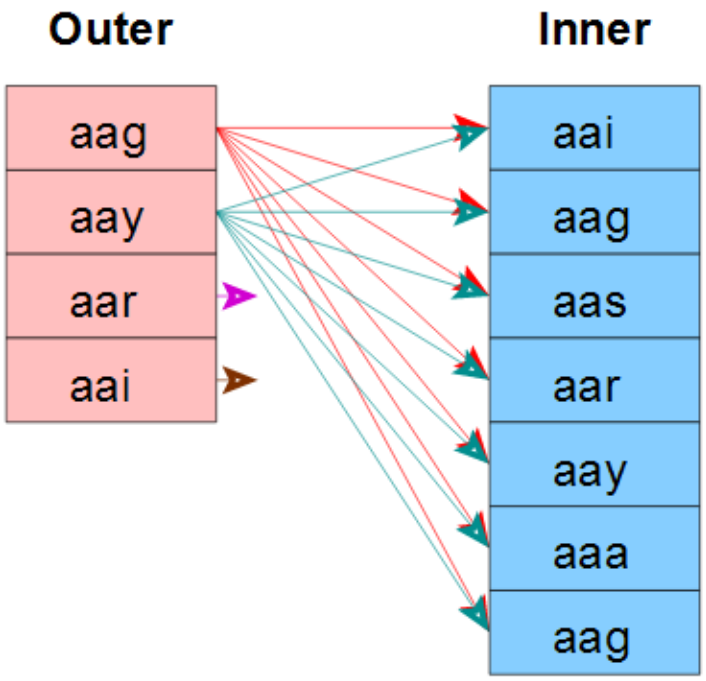
- Cardinality estimation with statistics
- Cost formula for each plan type
  - SeqScan, IndexScan
  - NestLoopJoin, HashJoin, MergeJoin, …

2

不同计划之间最主要的差别在于两种因素：

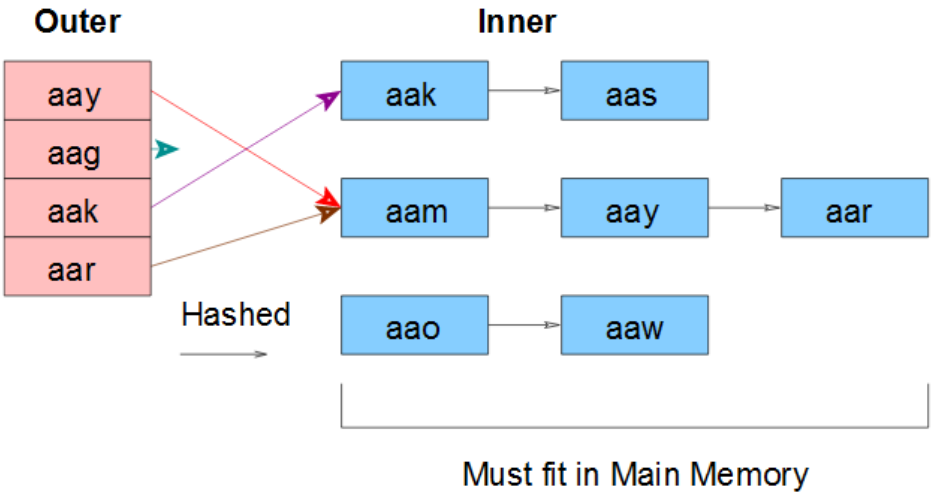
- Scan 方法
  - 顺序 Scan（也就是全表扫描）
  - 索引 Scan（具体索引机制如 2.2 节所示）
  - 位图索引 Scan（先索引扫描，然后把满足条件的行或块在内存中建立一个位图，同时在位图上执行某种操作（如 and、or 等），再根据位图计算结果到表的数据文件中读取对应的数据）
- Join 方法
  - Nested Loop（嵌套循环连接）

Nested Loop Join with Inner Sequential Scan



- Hash Join(散列连接)

Hash Join



- Merge Join(合并连接)

2.3 索引机制

在对内部机制说明之前，需要对 PostgreSQL 内部一些相关术语简单说明一下：

- Tuple：代表数据表中的一行，与“row”同义；
- Relation: 与“Table”同义，代表一个表；
- CTID(Current Tuple ID): 代表某 Tunple 存储的物理位置，通常由磁盘分区号和偏移地址组成；
- Heap：代表一个存储多个无序行列数据的文件（并不是常见“堆”），该文件通常由多个数据块组成，而每个数据块的常见大小是 8K。

同时，对所有索引而言（BRIN 索引例外），索引直接存储 TID 信息。同时，当数据修改时，会创建一个新的索引，而非修改原来索引。而对于废弃的索引，由 VACUUM(类似垃圾回收的机制)负责回收处理。

2.3.1 B-Tree

B-Tree 是一种平衡二叉树。其数据有序存放，并且是默认的索引方式，可满足大多数索引需求。通常可用 B-Tree 处理：

- 查找某个 key 等于 X 的数据；
- 查找某些 keys 小于或等于 X 的数据；
- 排序；
- 查找类似‘foo%’的模糊匹配；

2.3.2 Hash

目前 Hash 索引仅用户处理查找某个 key“等于”X 的情况。如果设置恰当，其 O(1)性能通常会优于 B-Tree 索引。不过，目前实现中并没有把 Hash 索引写入 WAL 日志，因此数据库重启或恢复或备份的时候可能需要重建 Hash 索引。

2.3.3 GiST(Generalized Search Tree)

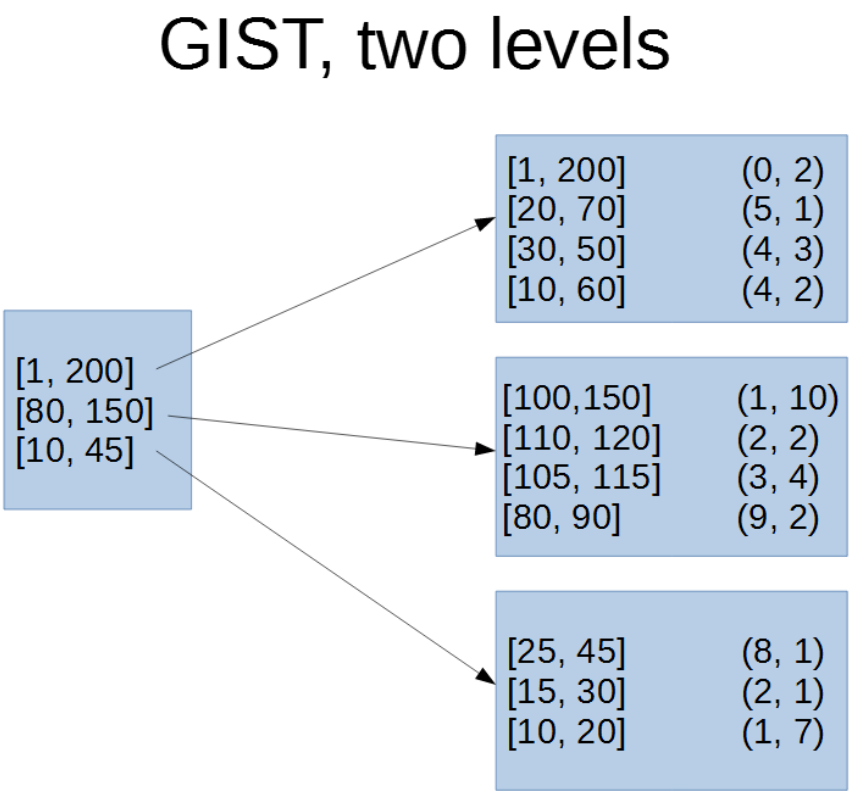
通用搜索树，是用户建立自定义索引的一个基础模板。也就是会说，用户只需要按照模板实现所要求的 GiST 操作类中一系列回调函数，则可实现自定义索引。B-tree, R-trees 以及许多其他索引都可以基于 GiST 实现。因此，GiST 的主要优势是让业务领域数据专家（而非数据库专家）开发自定义数据类型以及访问方法更加简洁方便。

其具有以下特点：

- 树形结构；
- 无序；
- 任何 Key 可以存储到树中任何位置，同时包含的内容允许有交叉；

特别的，PostgreSQL 本身已多多种复杂数据类型（如几何数据类型、Range 类型包括整数范围以及时间范围等）直接支持建立 GiST 和后续的 SP-GiST 索引。

例如，对于 Point 类型（或整型数据 Range 类型）的索引，其示例图如下：



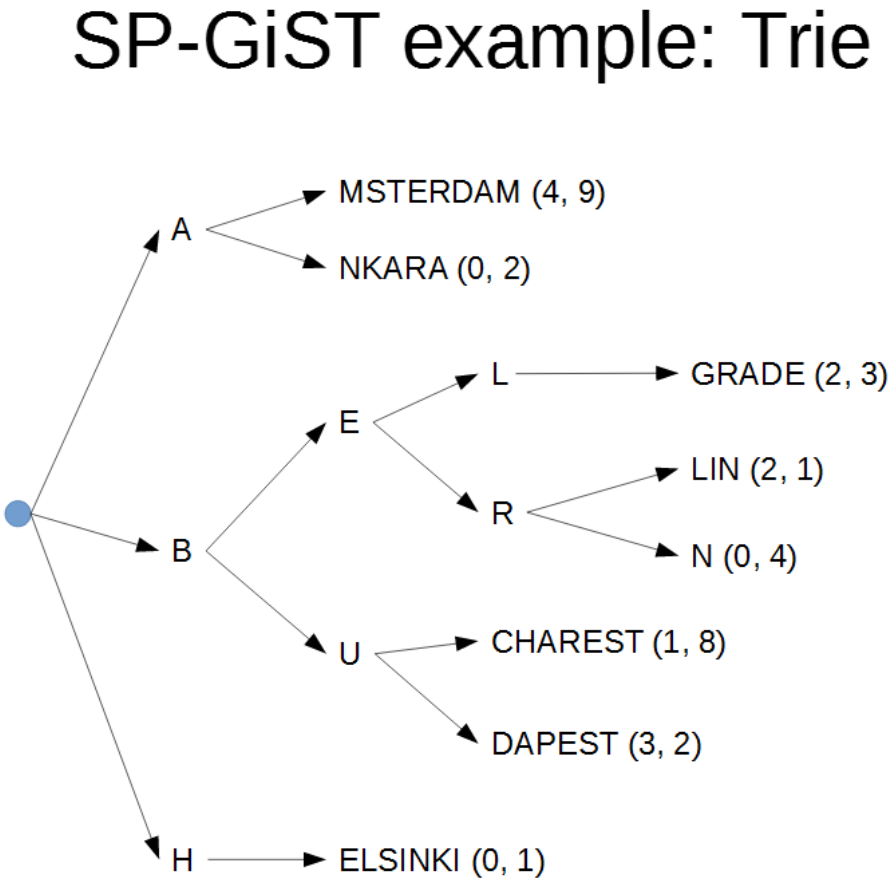
2.3.4 SP-GiST(Space-Partitioned GiST)

作为 GiST 的扩展，具有以下特点：

- Nodes 之间无交叉；
- 索引树深度是可变的；

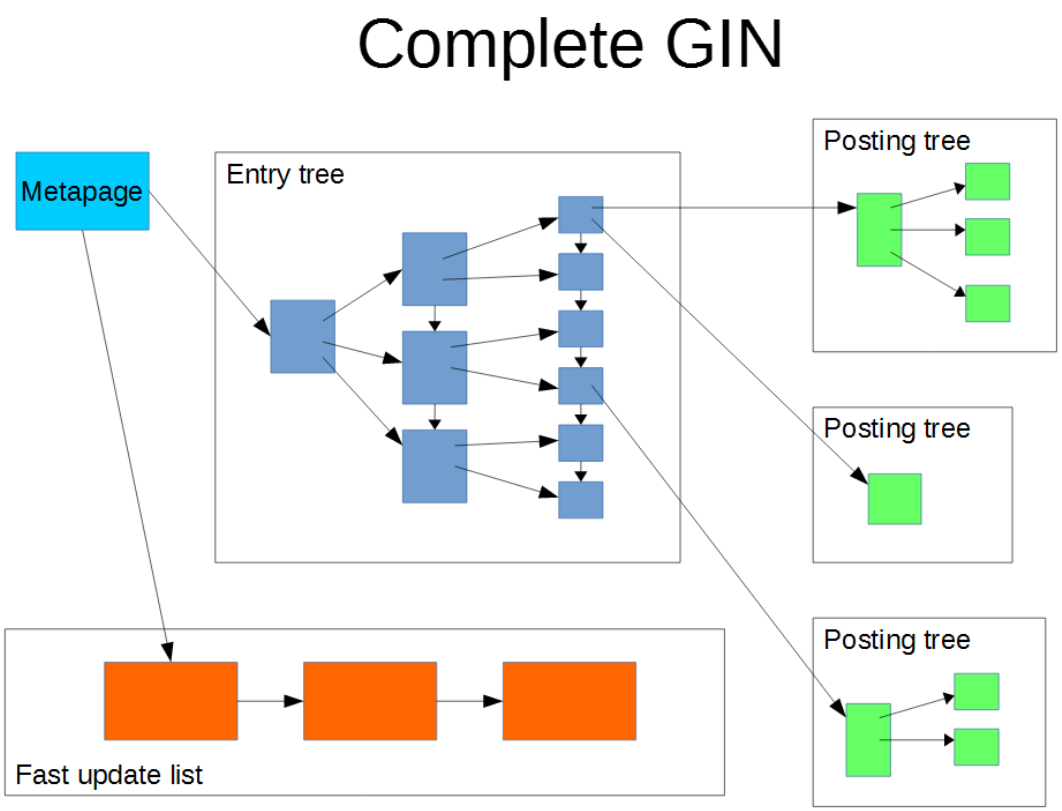
通常来说，可以将 GiST 当做

- Kd-tree(k-d 树是一个二叉树，其每个节点表示一个空间范围)；
- Trie(或 Prefix tree) ： 前缀树，



2.3.5 GIN(Generalized Inverted Index)

其本质上实际也是 B-Tree，但与 B-Tree 索引不同的是，其叶子节点可能是一个 TID，也可能是 TID 倒排列表，甚至可能是由 TID 组成的有序的倒排树。其基本组成结构如图：



鉴于对数据更新时涉及到对倒排索引多次读写，因此其索引更新通常是低效的。为此，单独增加“Fast Update List”来存储新增的索引。当索引列表过大或者被主动触发（例如VACUMM）时，会将该“Fast Update List”删除并更新到主索引中。

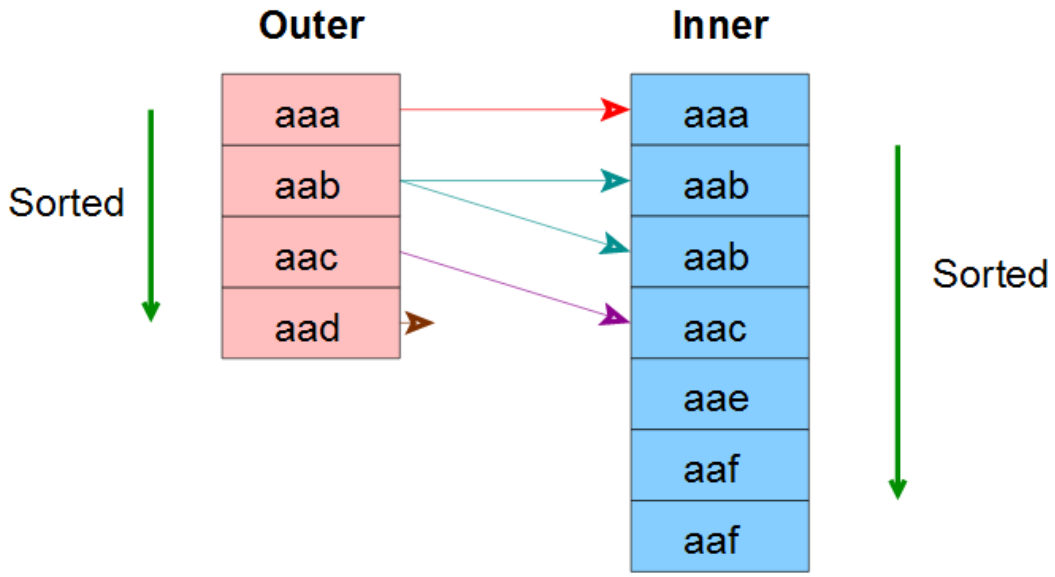
2.3.6 BRIN(Block Range Indexes)

其基本结构不再是树形结构，同时也并非对所有 TID 直接建立索引，而索引只是对物理上相互邻接存储的且行数据之间有存在某种关联的页表存储摘要信息。因此，BRIN 主要适合处理非常大的表，同时这种表的行之间存在某种关联，而且通常存储在紧密相连的磁盘数据块上。

例如，其简单的索引方式如下：

Approximation #1

Merge Join

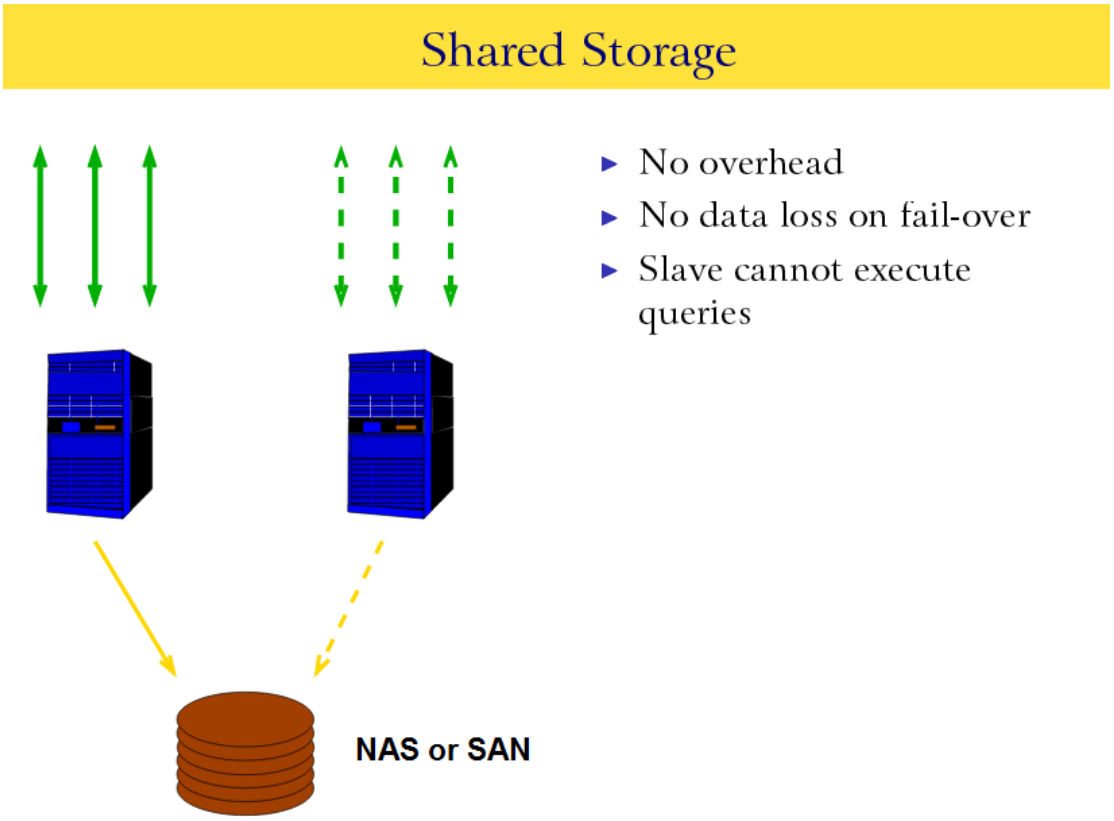


Ideal for Large Tables

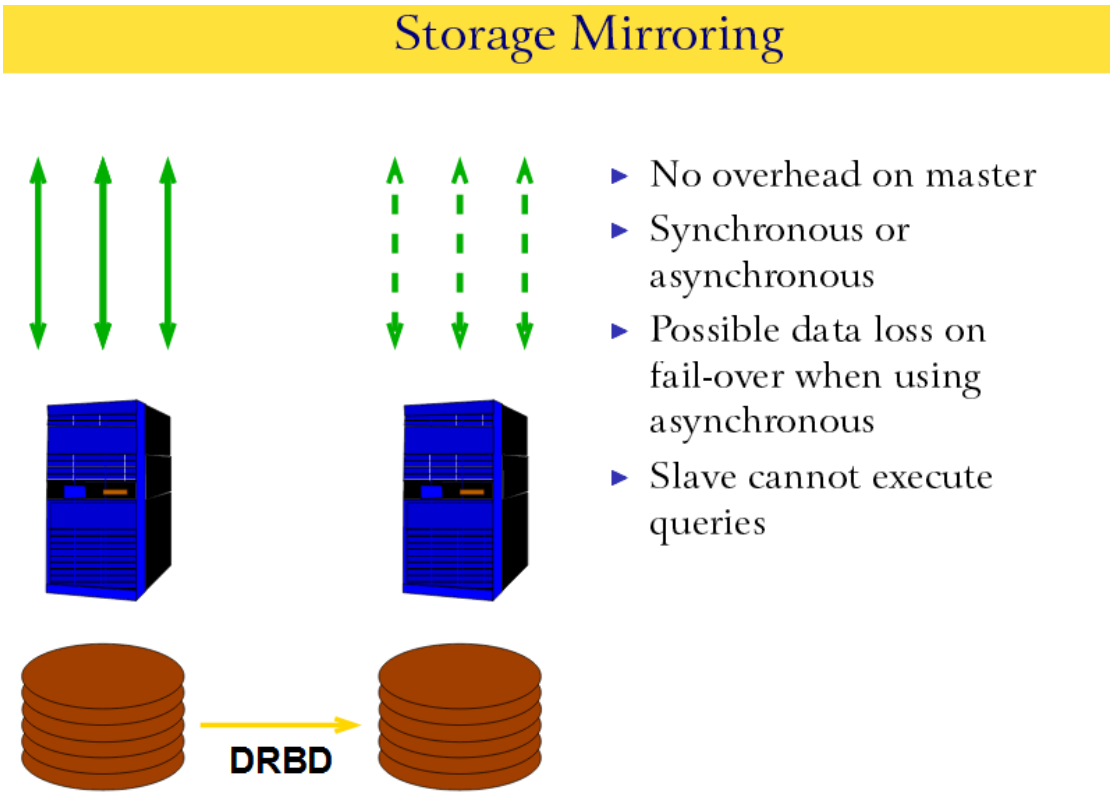
An Index Can Be Used to Eliminate the Sort

2.4 高可用、负载均衡以及复制

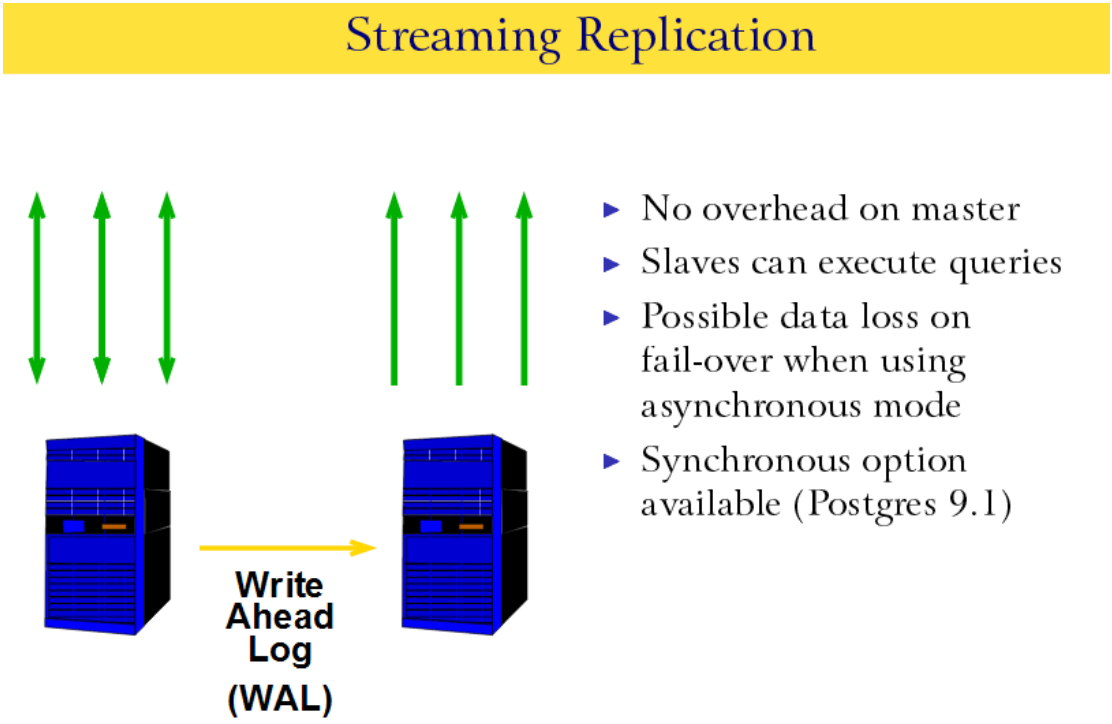
2.4.1 共享磁盘



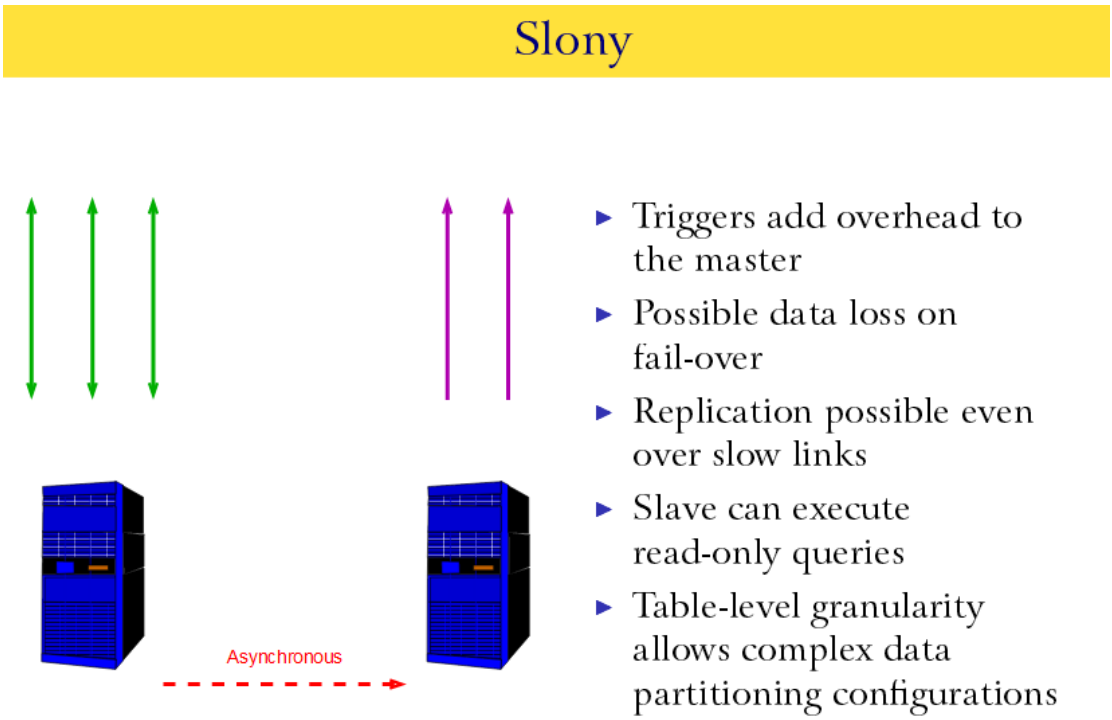
2.4.2 文件系统复制


Storage Mirroring
No overhead on master
Synchronous or asynchronous
Possible data loss on fail-over when using asynchronous
Slave cannot execute queries
DRBD

2.4.3 事物日志（或流式复制）

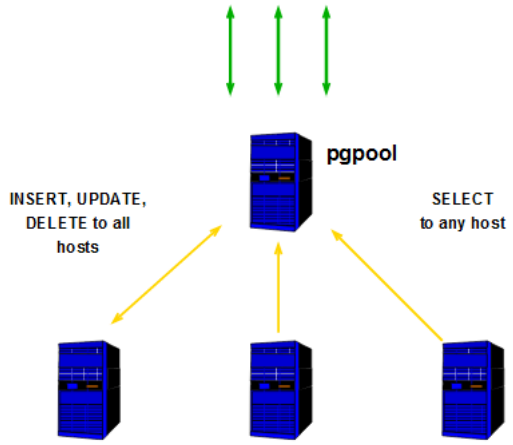


2.4.4 基于触发器主从复制模式



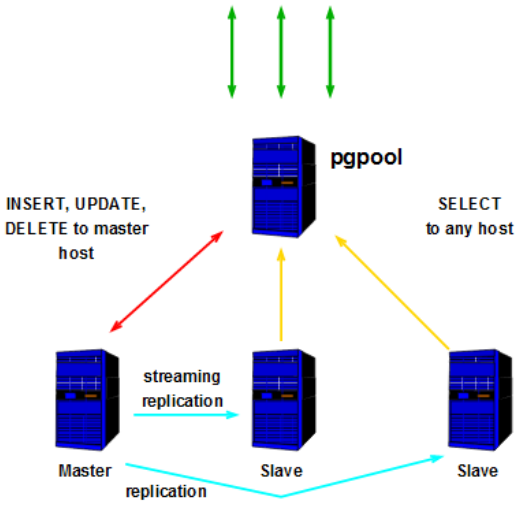
2.4.5 基于语句复制

Pgpool II



- Automatically load-balances read queries
- Queries with non-deterministic behavior can cause inconsistency
- Allows parallel query execution on all nodes
- Also does connection pooling and query caching

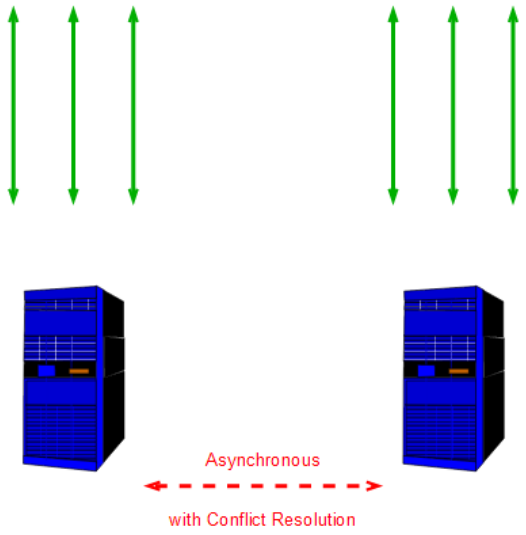
Pgpool II With Streaming Replication



Streaming replication avoids the problem of non-deterministic queries producing different results on different hosts.

2.4.6 多主节点（同步或异步）

Bucardo



- Similar to Slony, except multi-master with conflict resolution
- Conflict resolution rules are user-configurable

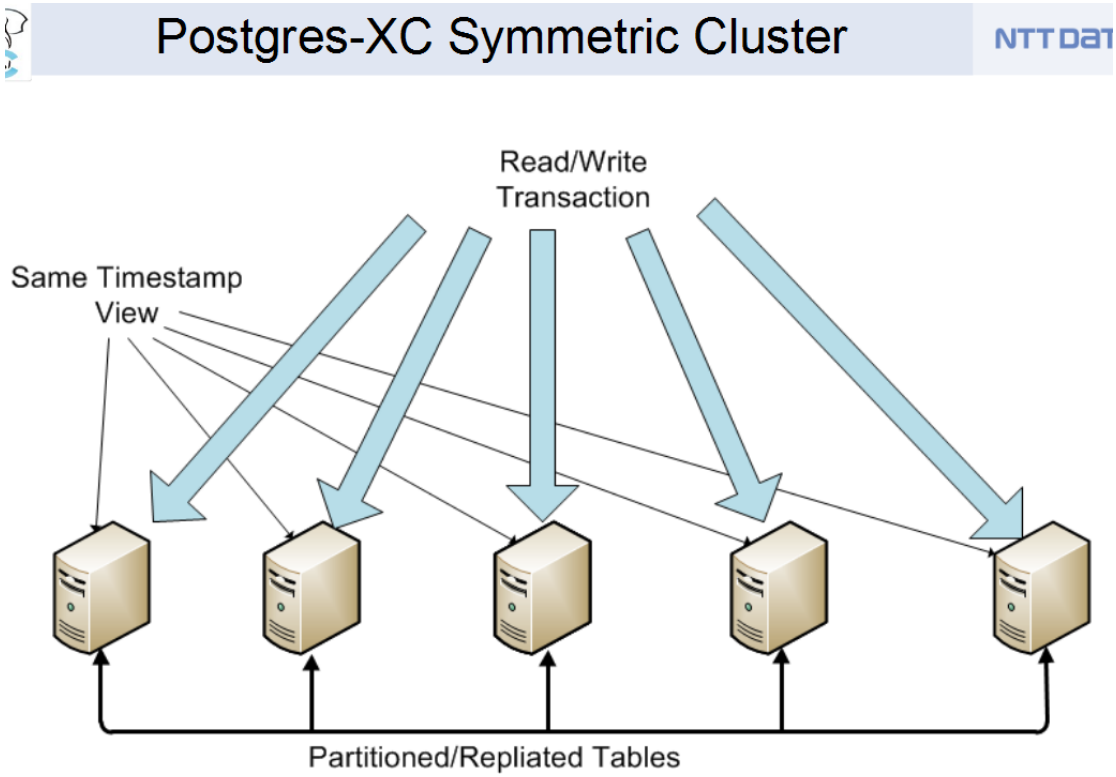
2.4.7 小结

Summary

Feature	Shared Disk Fail-over	File System Replic.	Transaction WAL Log Shipping	Trigger-based Replic.	Statement-Based Replication Middleware	Asynch-chronous Multi-Master Replic.	Synch-chronous Multi-Master Replic.
Most Popular Implementation	NAS	DRBD	Log shipping	Slony	pgpool-II	Bucardo	
Communication Method	shared disk	disk blocks	WAL	table rows	SQL	table rows	table rows & row locks
No Special hardware required		•	•	•	•	•	•
Allows multiple master servers					•	•	•
No master server overhead	•		•		•		
No waiting for multiple servers	•		•	•		•	
Master failure will never lose data	•	•			•		•
Slaves accept read-only queries			•	•	•	•	•
Per-table granularity				•		•	•
No conflict resolution necessary	•	•	•	•			•

2.5 分片（Sharding）

- 基于应用的分片
- PL/Proxy
- Postgre-XC/XL



- pg_shard

3 性能优化

3.1 基本方法论

对于性能优化的基本方法论，以及基本工具可参阅：https://zhuanlan.zhihu.com/p/25013051

3.2 基准测试

在性能优化之前需要衡量数据库系统性能，而对数据库而言，其性能的两个主要指标是：时延和吞吐量。为此，常需要借助基准测试来衡量系统性能，并且将时延或吞吐量作为优化目标。

关于基准测试用例配置、部署等，可参考自动化部署脚本。

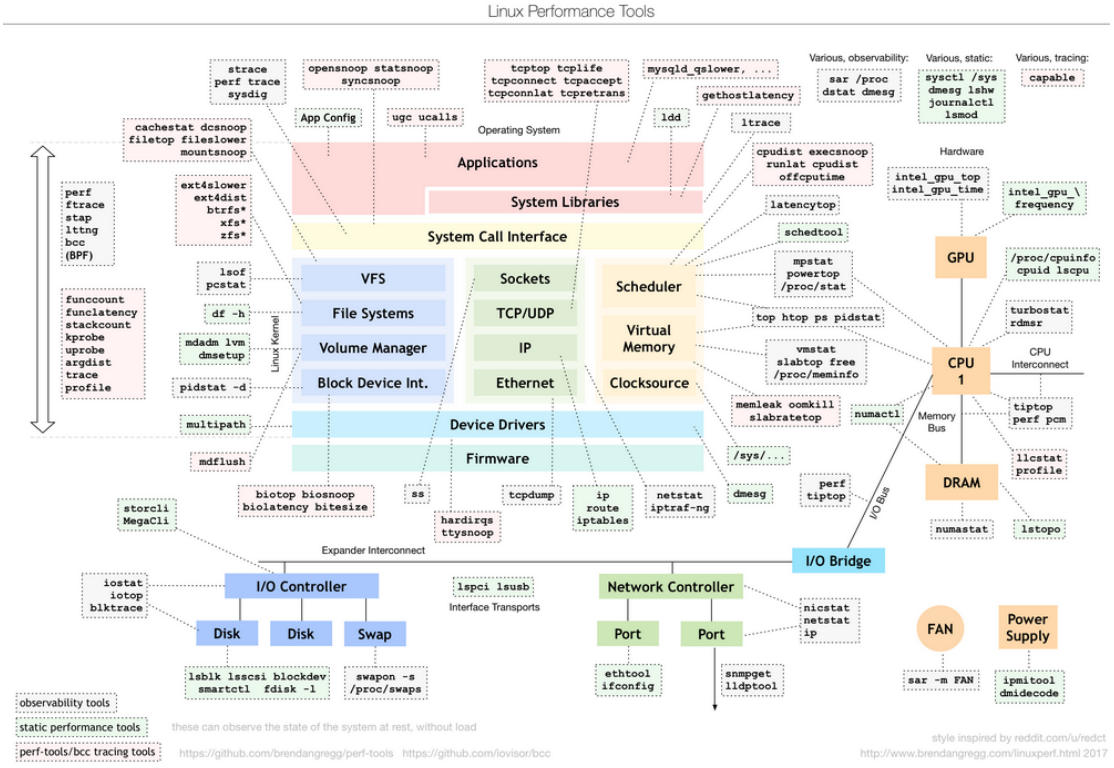
https://github.com/sjtuhjh/applications/tree/master/apps/postgresql/postgresql_test1

3.3 性能监控

3.3.1 基础性能监控

对 CPU/Memory/Disk/Networking/Kernel 等监控，可采用 perf/ftrace/bcc/top/dstat 等多种工具，既可以支持静动跟踪，也支持动态跟踪。

具体工具可参考 http://github.com/sjtuhjh/perftools，在此不再赘述。



3.3.2 应用内部监控

总体性能分析：

更多分析工具，可参考 https://github.com/sjtuhjh/perftools/tree/master/apptools/postgresql

3.4 基础优化

3.4.1 BIOS/CPU/Memory

在对应用优化之前，要确保 BIOS/CPU/Memory 配置恰当，否则后续优化往往事倍功半。为此，需要借助一些测试工具来验证硬件功能是否工作正常（或者最优）。值得注意的是，开始测试的时候常常无法根据数据来确定是否工作正常，因此可将数据与历史优化后的数据或者对应的 X86 测试数据进行对比。常见的测试工具如下：

- **CPU**:
  SpecInt2006(https://github.com/sjtuhjh/applications/tree/master/apps/cpu/spec_cpu2006)

- **Memory**: lmbench/Stream
- **DISK : I/O** : fio 测试

Location	Disks	RAID Level	Purpose
/ (root)	2	1	Operating system
$PGDATA	6+	10	Database
$PGDATA/pg_xlog	2	1	WAL
Tablespace	1+	None	Temporary files

  - RAID1：将两块硬盘构成 RAID 磁盘阵列，其容量仅相当于一块磁盘，但另一块磁盘总是保持完整的数据备份。一般支持“热交换”。
  - RAID10：首先建立两个独立的 RAID1，然后将两个独立的 RAID1 再组成一个 RAID0

  建议配备 RAND10，同时支持 BBU(Battery Backup Unit)和 WRITE-BACK 功能的存储设备。这样既可以有良好的读写性能，又能防止意外情况下（例如突然断电）存储内容丢失。

- **BIOS 电源管理**：将电源管理设置成“性能”模式以便最大发挥 CPU 性能；

常见的基准测试工具也可参考 https://github.com/sjtuhjh/applications/tree/master/apps。

3.4.2 OS/Kernel

3.4.2.1文件系统

通常建议采用 EXT4 或 XFS 文件系统。

- Barriers I/O (Barrier 之前的所有 I/O 请求必须在 Barrier 之前结束，并持久化到非易失性介质中。而 Barrier 之后的 I/O 必须在 Barrier 之后执行。如果本身存储设备自带电池足以预防在突然掉电时其缓存内容依然可以被正确持久化，则可以关闭

Barrier 一般提高性能)。一般文件系统挂载时提供选项可以关闭 barriers,例如：mount –o nobarrier /dev/sda /u01/data

- 打开 noatime，减少不必要的 I/O 操作，例如：mount –o nobarrier,noatime …
- 调整 vm.overcommit (将其设置为 2，尽量避免 OOM(Out of Memory killer))
- 写缓存优化（需要根据需要调整参数，当脏页刷写太频繁，会导致过多 I/O 请求，而刷写频率过低，又可能导致所占内存过多，而在不得不刷写脏页时导致应用性能波动）：
  - vm.dirty_background_ratio: 当脏页数量达到一定比例时，触发内核线程异步将脏页写入磁盘；
  - vm.dirty_ratio: 当脏页数量达到一定比例时，系统不得不将脏页写入缓存。此过程可能导致应用的 I/O 被阻塞。
- I/O 调度
  - 对于普通磁盘，建议使用 deadline 调度方式，而固态硬盘各种调度方式影响并不大；
- Read-ahead 调整：
  - blockdev –getra /dev/sda 查看 sda
  - blockdev –setra <sectors> /dev/sda （设置预读缓冲大小）
- vm.swappiness=0 （尽量避免使用 SWAP）

3.4.2.2NUMA

一般建议关闭 NUMA（或者使用交织调度并关闭 NUMA BALANCING）性能会更好。主要原因是，PostgreSQL 后台进程使用共享机制共同访问一大块内存，而该块共享内存一般无法在一个 NUMA 内存节点分配。因此必须在多个内存节点分配共享内存。同时，PostgreSQL 后台进程可能是随机访问共享内存内某块数据，而在随机访问模式下，关闭 NUMA（包括 NUMA BALANCING）性能会更好。

3.4.2.3调度策略

适当增大 kernel.sched_migration_cost_ns 和 关闭 kernel.sched_autogroup_enabled （=0）。主要原因是 autogroup 会根据终端的响应时间要求自动对进程分组并优先处理响应要求更高的分组，而这种机制对于与伪终端相关的 PostgreSQL 后台进程而言，有可能会影响其调度从而降低性能。

3.4.2.4HugePage

可以通过“cat /proc/meminfo | grep PageTables”可以查看页表大小。当页表过大时，建议使用 HugePage，从而可以提升 TLB 命中率，进而提升系统性能。

3.4.2.5存储设备设置优化

3.4.2.5.1 分区对齐

分区对齐的目的是让逻辑块与物理块对齐从而减少 I/O 操作。通常可以借助 parted 命令来自动对齐，例如：

device=/dev/sdb

parted –s –a optimal $device mklable gpt

parted –s –a optimal $device mkpart primary ext4 0% 100%

其中 optimal 表示要尽量分区对齐从而达到最好性能。

3.4.2.5.2 SSD 优化

SSD 支持 TRIM 功能，打开该功能可以防止未来读写性能恶化。通过 Btrfs, EXT4, JFS 和 XFS 文件系统都支持该功能。可通过下列命令进行配置：

- 普通磁盘: 在 mount 的时候添加 discard 选项，例如：
  /dev/sda / ext4 rw,discard 0 1
- LVM (Logical Volumes)： 在/etc/lvm/lvm.conf 中添加 issue_discards=1；

3.4.2.5.3 I/O 调度算法

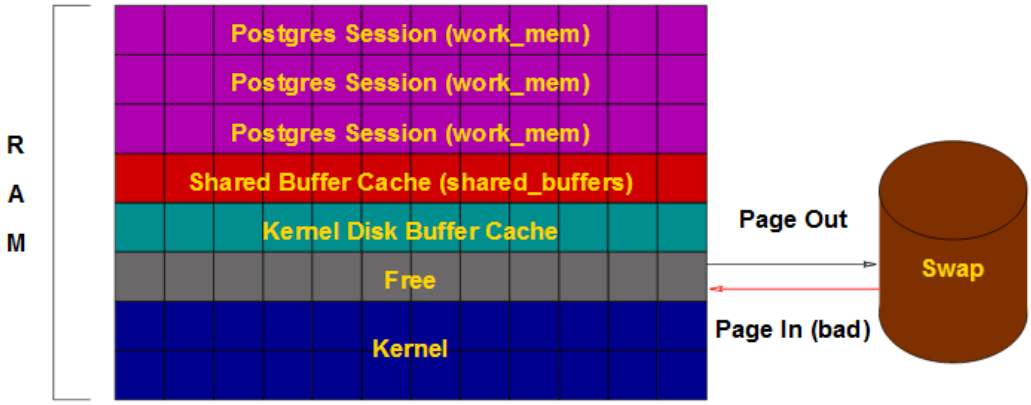
一般建议将/sys/block/<设备名>/queue/scheduler 设置为 deadline 或 noop。尤其是对普通机械硬盘，不要将其设置为 cfq 算法。

3.4.2.6SWAP

尽量禁止使用 SWAP，从而避免性能波动。可在/etc/sysctl.conf 中添加 vm.swappiness=0 并通过 sysctl –p 使其生效。

3.5 PostgreSQL 配置调优

3.5.1 内存配置优化



3.5.1.1动态共享内存机制

对 Linux 系统而言，PostgreSQL 提供了三种内存共享实现方式，其默认实现方式是 POSIX：

	POSIX	SYS-V	MMap
实现机制	调用 shm_open、Shm_unlink 以及 Mmap 等系统接口实现内存共享	调用 shmget、shmat、shmdt 以及 shmctl 等接口实现内存共享	创建一些列文件，并将其映射到对应的地址空间。操作系统负责文件之间的同步
优缺点	● 无文件 I/O 开销 ● 命名空间扁平化，容易造成命名冲突 ● 允许扩展共享内存大小 ● 共享内存永久存在，除非被进程显式删除，因此如果进程无故 Crash 会导致共享内存残留；	● 最早实现的内存共享方式，但命名空间同样受到限制，容易造成冲突 ● 无关进程之间的内存共享 ● 共享内存大小一开始便已固定，	● 并非真正意义上的共享内存，会涉及到文件 I/O 操作，因此会影响到系统性能，因此一般仅当其他方式不支持时使用 ● 匿名页共享：仅限于通过 fork 创建的一系列进程之间相互共

		后续无法扩展	享 ● 文件共享映射：适合无关进程之间共享 ● 共享内存大小允许动态扩展

系统默认建议使用 POSIX，但因命名冲突等因素不能保证 shm_open 都能成功，因此如果尝试超过 10 次则使用 SYS-V 共享内存。

3.5.1.2配置优化

- Shared_buffers：设置共享内存的大小，与 MySQL 中的 buffer_pool 类似。对整体性能有着重要的影响；
- effective_cache_size: 设置 PostgreSQL 能够使用的最大缓存（但没有直接分配内存）；
- wal_buffers:
- work_mem：为每个进程单独分配的内存，主要用于排序、hash 等操作；
- Maintence_work_mem：为每个进程单独分配的内存，主要用于维护操作，例如 vacuum
- Huge_pages = try(或 on): 建议使用 HugePage，从而提升 TLB 命中率；

3.5.2 检查点和 WAL 日志优化

- checkpoint_segments / checkpoint_timeout : 每写完 checkpoint_segments 个 WAL 日志或者每过 checkpoint_timeout 秒，就创建一个检查点；
- checkpoint_completion_target:（0~1.0）两个检查点间隔的多少时间内完成所有脏页的刷新；
- synchronous_commit：声明一个事务是否需要等到操作被写到 WAL 日志才返回。当设置为 off 时会提升性能，但是可能在突然断电时造成事务已经被提交但没有及时写回 WAL 日志中。此外，该参数支持在 session 级别设置；
- wal_buffers:
- wal_sync_method: 建议 open_sync
- wal_write_delay:声明 WAL 写进程周期；

3.5.3 VACUUM 与 ANALYZE

注释：VACUUM 通常用来回收 UPDATE 和 DELETE 操作遗留下的内存便于后续使用。而 ANALYZE 主要搜集表的统计信息，用于后续优化器对计划 COST 的估算，从而选择最优计划。

- autovacuum
- default_statistics_target:增大该值会消耗更多时间用于信息搜集分析，但是如果没

有足够的信息又会让优化器选择的“最优”计划实际上并不是最优的

3.5.4 其他配置优化

- max_connections: 该值决定允许的最大连接数。鉴于每个连接会创建一个新的进程，因此其值的大小与系统资源（例如内存等）密切相关；
- seq_page_cost/random_pange_cost : 默认情况下是假定是需要 I/O 操作读取数据的。而实际上如果数据已经在缓存中，则适当降低该值，从而可以使用其他索引方式提升系统性能；
- fsync/full_page_writes: 一般情况下不建议关闭。当然如果数据库损坏时可以通过其他途径快速恢复（例如重建等），则关闭该值会提升性能；
- max_prepared_transactions: 根据业务需要具体调整；
- enable_seqscan 等: 具体根据业务需要，当确定某个索引对性能有较大损耗时，可直接关闭该索引；

3.6 PostgreSQL 语句优化

- 分析查看 Explain
- COPY 与 INSERT : 加载大量数据时优先考虑使用 COPY(因为一个 COPY 可以处理多条数据，因此仅涉及单个事务，而对应的 INSERT 需要多条语句，故涉及到多个事务事务处理)
- LIMIT 与 CURSOR: CURSOR 可以显示声明被重复利用，从而可以提升性能；
- TRUNCATE 与 DELETE: TRUNCATE 相当于重新定义一个新的表，而将原来的表丢弃，因此比使用 DELETE 一条一条删除数据性能优越；
- Expression Indexes（表达式索引）：根据业务需要，针对字段进行某些运算后的结果创建索引，进而提升性能；
- Partial Indexed（部分索引）：根据业务需要，仅在部行列上建立索引，从而提升性能；
- Prepared Queries : 事先将 QUERY 解析、分析和改写，但并不执行，等待后续 EXECUTE 命令显式执行。因此可以重复利用 Prepared Queries 避免解析、改写等开销，从而提升性能；
- INTERSECT 与 AND(selfjoin)
- UNION 与 OR
- 多列索引：在多个列上建议索引；

3.7 PostgreSQL 代码优化

3.7.1 算法优化

3.7.2 CPU 相关代码优化

待补充。

4 附录

[1] www.postgresql.org
[2] http://momjian.us/main/writings/psql
[3] High Performance PostgreSQL