ALIBABA CLOUD

阿里云

云原生关系型数据库PolarDB O引擎

性能调优指南

文档版本：20220525

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

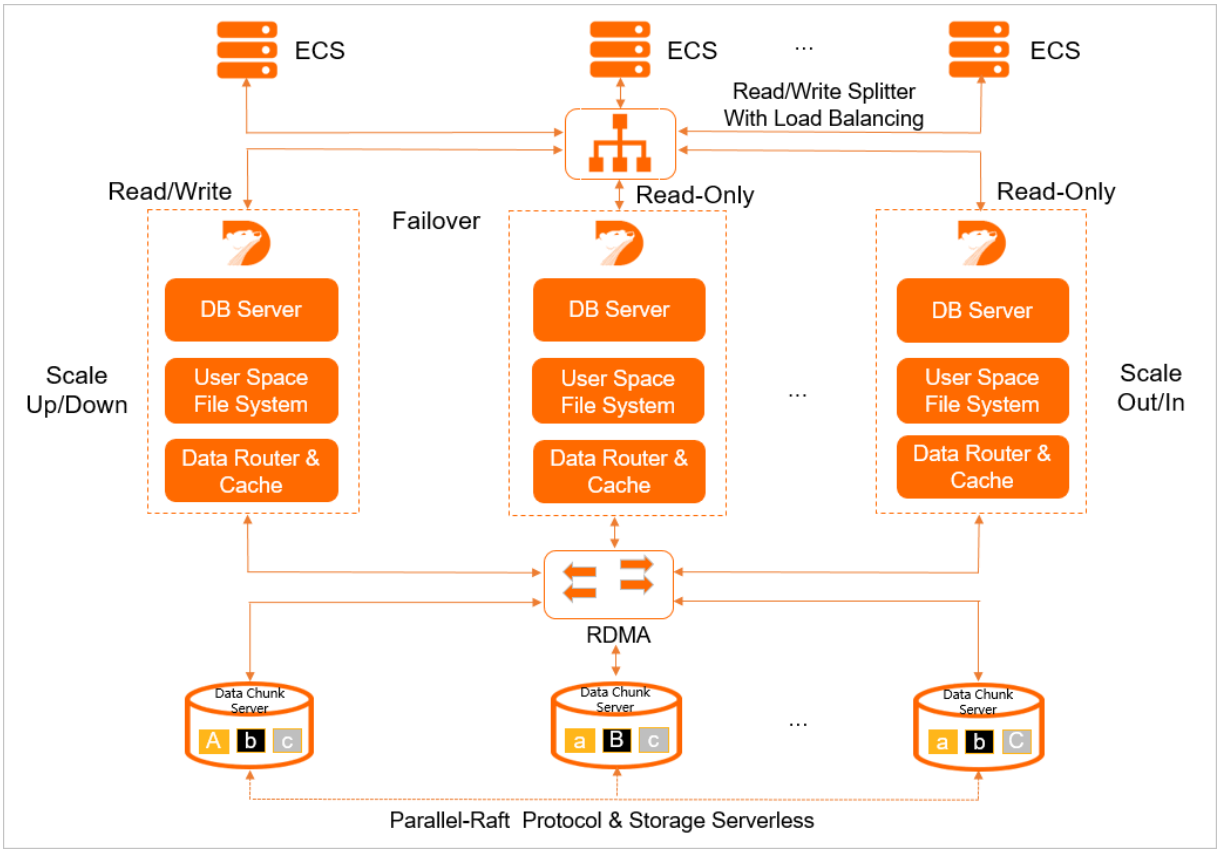
1.了解体系架构 06
1.1. 重要组件 06
1.2. 内部架构 07
1.3. 缓冲区管理 08
1.4. 文件存储 13
1.5. 堆表结构 15
1.6. 查询机制 16
1.7. 垃圾回收机制 18
2.定位性能问题 20
2.1. 监控信息概览 20
2.2. 查看视图 20
2.2.1. 数据库 20
2.2.2. SQL 44
2.2.3. 对象 48
2.3. 使用AWR 52
2.4. 使用性能洞察 52
3.优化集群性能 57
3.1. 参数说明 57
3.2. 设置参数 61
4.优化SQL语句 63
4.1. 使用索引 63
4.1.1. B-TREE索引 63
4.1.2. HASH索引 67
4.1.3. GIN索引 69
4.1.4. GiST索引 73
4.1.5. BRIN索引 75

4.2. 分析执行计划 77
4.2.1. EXPLAIN语法 77
4.2.2. EXPLAIN输出 78
4.2.3. 调整执行计划 84
4.3. 物理优化 88
4.3.1. 代价的相关概念 89
4.3.2. 代价计算 92
5.性能调优最佳实践 94
5.1. 有关性能调优 94
5.2. 实例问题分析 94
5.2.1. 问题分析方法论 94
5.2.2. 等待事件 94
5.2.3. CPU瓶颈 94
5.2.4. I/O瓶颈 95
5.2.5. 锁 96
5.2.6. SLRU 96
5.2.7. 内存诊断 96
5.2.8. 磁盘空间诊断 97
5.2.9. 日志位点监控 97
5.3. SQL性能分析 100
5.4. CPU使用率高 102
5.5. 高I/O 104
5.6. 网络问题 105
5.7. SQL调优思路 106

1.了解体系架构

1.1. 重要组件

本文为您介绍PolarDB O引擎的重要组件。



DB Server

即数据库进程（Polar Database, 简称PolarDB）。PolarDB数据库内核区分实例角色，目前包括三种角色，Primary，Standby和Replica。Primary即为拥有读写权限的读写库，Replica即为只读实例，仅仅拥有读取数据的权限（后台线程也不能修改数据），Primary和Replica采用Shared Everything架构，即底层共享同一份数据文件和日志文件。StandBy节点拥有一份独立的数据和日志文件。StandBy节点主要用来机房级别的容灾以及创建跨可用区的只读实例。由于只读实例的扩展不需要拷贝数据，创建新的只读实例不但速度快，而且很便宜，您只需要支付相应计算节点的成本即可。阿里云称StandBy和Replica节点为Slave节点，Primary节点也可称为Master节点。

User Space File System

即用户态文件系统（Polar File Sytem, 简称PolarFS）。由于多个主机的数据库实例需要访问块存储上的同一份数据，常用的Ext4等文件系统不支持多点挂载，PolarDB数据库团队自行研发了专用的用户态文件系统，提供常见的文件读写查看接口，支持类似O_DIRECT的非缓存方式读写数据，还支持数据页原子写，IO优先级等优秀的特性，为上层数据库的高性能提供了结实的保障。传统的文件系统，由于嵌入在操作系统内核中，每次系统文件读写操作都需要先陷入内核态，完成后再返回用户态，造成效率低下。PolarFS以函数库形式编译在PolarDB中，因此都运行在用户态，从而减少了操作系统切换的开销。

Data Router & Cache

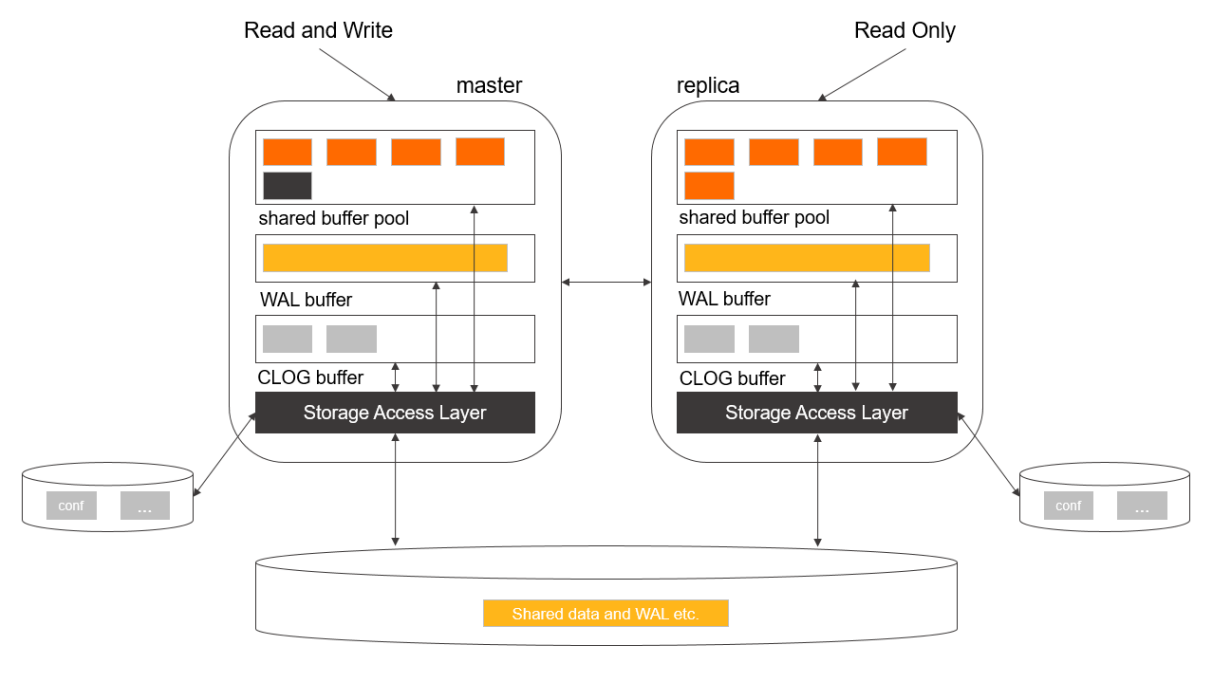
即块存储系统客户端（Polar Store Client，别名PolarSwitch）。PolarFS收到读写请求后，会通过共享内存的方式把数据发送给PolarSwitch，PolarSwith是一个计算节点主机维度的后台守护进程，接收主机上所有实例以及工具发来的读写块存储的请求。PolarSwith做简单的聚合，统计后分发给相应的存储节点上的守护进程。

Data Chunk Server

即块存储系统服务器端（Polar Store Server，别名ChunkSever）。上述三个部件都运行在计算节点上，这个部件则运行在存储节点上。主要负责相应数据块的读取。数据块的大小目前为10GB，每个数据块都有三个副本（位于三台不同的存储节点上），两个副本写成功，才给客户端返回成功。支持数据块维度的高可用，即如果一个数据块发生不可用，可以在上层无感知的情况下秒级恢复。此外，PolarStore使用了类似Copy On Write技术，支持秒级快照，即对数据库来说，不管底层数据有多大，都能快速完成全量数据备份，因此PolarDB支持高达100T的磁盘规格。计算节点和存储节点之间通过25G RDMA网络连接，保证数据传输不会出现瓶颈。

1.2. 内部架构

本节为您展示PolarDB O引擎内部架构图。



架构组件	说明
shared_buffer_pool	共享缓冲区，用于缓冲数据，加快数据处理的速度。
wal buffer	日志缓冲区，写入WAL日志时，需要先写入到日志缓冲区，根据日志产生的大小或者commit触发刷新到磁盘的操作。
clog buffer	主要用于存放clog日志。用于记录事务的提交状态，通过clog进行事务可见性的判断。
local storage	本地存储。

1.3. 缓冲区管理

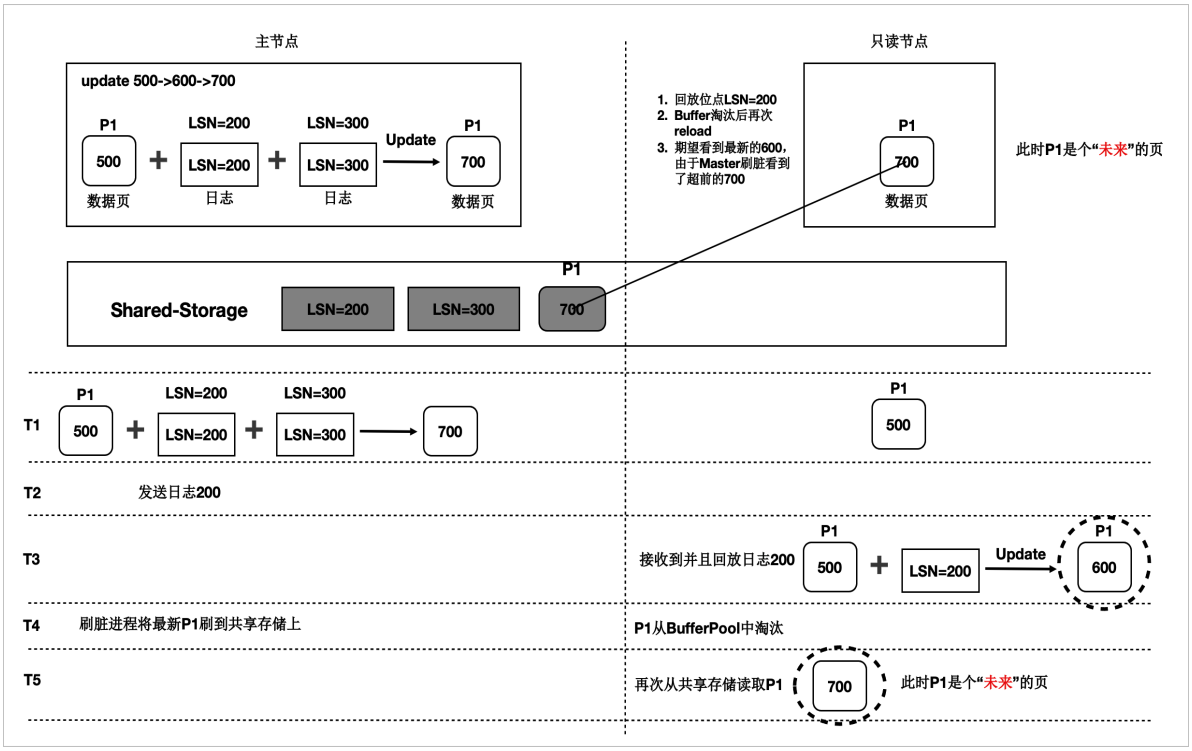
本文介绍了如何进行缓冲区管理。

背景信息

一写多读架构下，只读节点可能从共享存储中读到两类数据页：

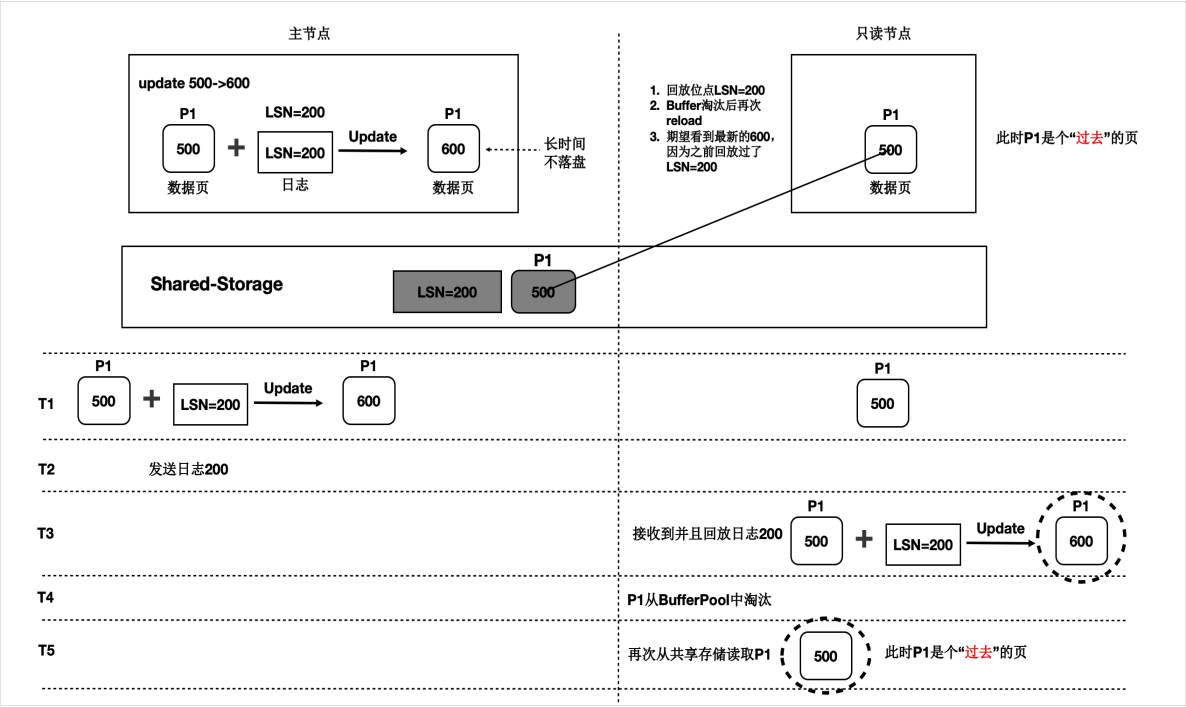
- **未来页**

  数据页中包含只读节点尚未回放到的数据。例如，只读节点回放LSN为200的WAL日志，但数据页中已经包含LSN为300的WAL日志对应的改动。此类数据页被称为未来页。



- **过去页**

数据页中未包含所有回放位点之前的改动。例如，只读节点将数据页回放到LSN为200的WAL日志，但该数据页在从Buffer Pool淘汰之后，再次从共享存储中读取的数据页中没有包含LSN为200的WAL日志的改动，此类数据页被称为过去页。



对于只读节点而言，只需要访问与其回放位点相对应的数据页。但如果读取到如上所述的未来页和过去页时，可以按照以下步骤处理：

- 对于过去页，只读节点需要回放数据页上截止回放位点之前缺失的WAL日志，对过去页的回放由每个只读节点根据自己的回放位点完成，属于只读节点回放功能，此处暂不赘述。
- 对于未来页，只读节点无法将未来的数据页转换为所需的数据页，因此需要在主节点将数据写入共享存储时考虑所有只读节点的回放情况，从而避免只读节点读取到未来页，这也是Buffer管理需要解决的主要问题。

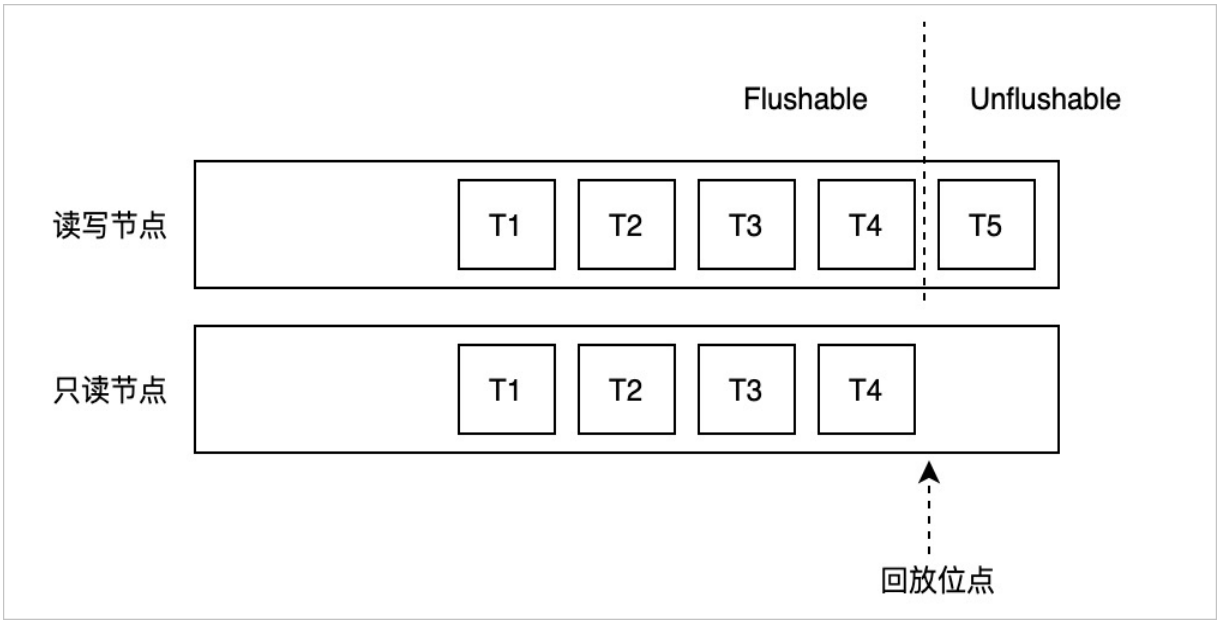
除此之外，Buffer管理还需要维护一致性位点，对于某个数据页，只读节点仅需回放一致性位点和当前回放位点之间的WAL日志即可，从而加速回放效率。

术语

名词	说明
Buffer Pool	缓冲池，一种用来存储最常访问的数据的内存结构，通常以页为单位来缓存数据。PolarDB中每个节点都有自己的Buffer Pool。
LSN	Log Sequence Number，日志序列号，是WAL日志的唯一标识。LSN在全局是递增的。
Apply LSN	回放位点，表示只读节点回放日志的位置，一般用LSN来标记。
Oldest Apply LSN最老回放位点	最老回放位点，表示所有只读节点中LSN最小的回放位点。

刷脏控制

为了避免只读节点读取到未来页，PolarDB引入刷脏控制功能，即在主节点将数据页写入共享存储时，判断所有只读节点是否均已回放到该数据页最近一次修改的对应的WAL日志。



主节点Buffer Pool中的数据页，根据是否包含未来数据（即只读节点的回放位点之后新产生的数据），可以分为两类：可以写入存储的数据页和不能写入存储的数据页。该判断依赖以下两个位点：

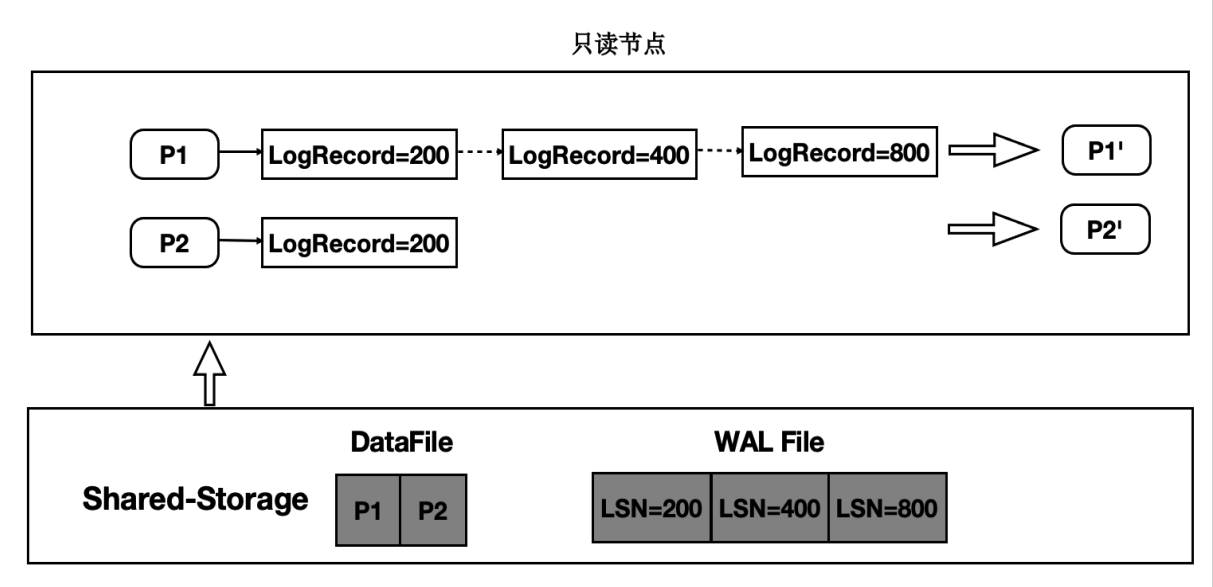
- Buffer最近一次修改对应的LSN，即Buffer Latest LSN。
- 最老回放位点，所有只读节点中最小的回放位点，即Oldest Apply LSN。

刷脏控制判断规则如下：

```
if buffer latest lsn <= oldest apply lsn
    flush buffer
else
    do not flush buffer
```

一致性位点

为了将数据页回放到指定的LSN位点，只读节点会维护数据页与该页上的LSN的映射关系，这种映射关系保存在LogIndex中。LogIndex可以理解为是一种可以持久化存储的HashTable。访问数据页时，会从该映射关系中获取数据页需要回放的所有LSN，依次回放对应的WAL日志，最终生成需要使用的数据页。



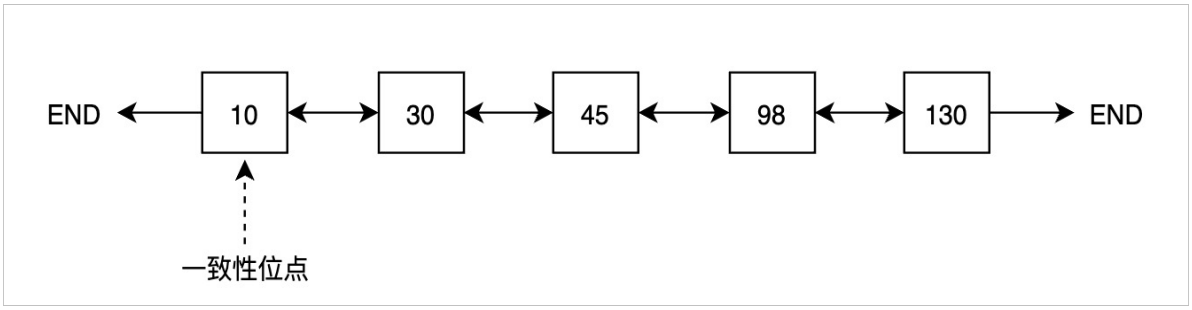
数据页上的修改越多，其对应的LSN也越多，回放所需耗时也越长。为了尽量减少数据页需要回放的LSN数量，PolarDB引入了一致性位点的概念。

一致性位点表示该位点之前的所有WAL日志修改的数据页均已经持久化到存储。主备节点之间，主节点向备节点发送当前WAL日志的写入位点和一致性位点，备节点向主节点发送当前回放的位点。由于一致性位点之前的WAL修改都已经写入共享存储，备节点无需再回放该位点之前的WAL日志。因此，可以将LogIndex中所有小于一致性位点的LSN清理掉，既加速回放效率，同时还能减少LogIndex占用的空间。

- **FlushList**

  为了维护一致性位点，PolarDB为每个Buffer引入了一个内存状态，即第一次修改该Buffer对应的LSN，称之为Oldest LSN，所有Buffer中最小的Oldest LSN即为一致性位点。

  一种获取一致性位点的方法是遍历Buffer Pool中所有Buffer，找到最小值，但遍历代价较大，CPU开销和耗时都很大。为高效获取一致性位点，PolarDB引入FlushList机制，将Buffer Pool中所有脏页按照Oldest LSN从小到大排序。借助FlushList，获取一致性位点的时间复杂度可以达到 O(1)。

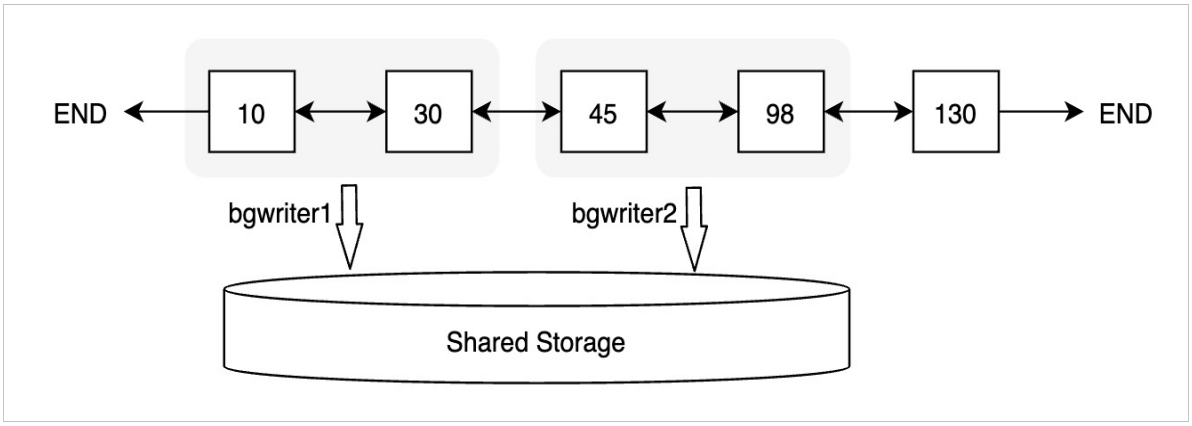
  

  第一次修改Buffer并将其标记为脏页时，将该Buffer插入到FlushList中，并设置其Oldest LSN。Buffer被写入存储时，将该内存中的标记清除。

  为高效推进一致性位点，PolarDB的后台刷脏进程（bgwriter）采用先被修改的Buffer先落盘的刷脏策略，即bgwriter会从前往后遍历FlushList，逐个刷脏，一旦有脏页写入存储，一致性位点就可以向前推进。如上图所示，如果Oldest LSN为10的Buffer落盘，一致性位点就可以推进到30。

- **并行刷脏**

为进一步提升一致性位点的推进效率，PolarDB实现了并行刷脏。每个后台刷脏进程会从FlushList中获取一批数据页进行刷脏。



热点页

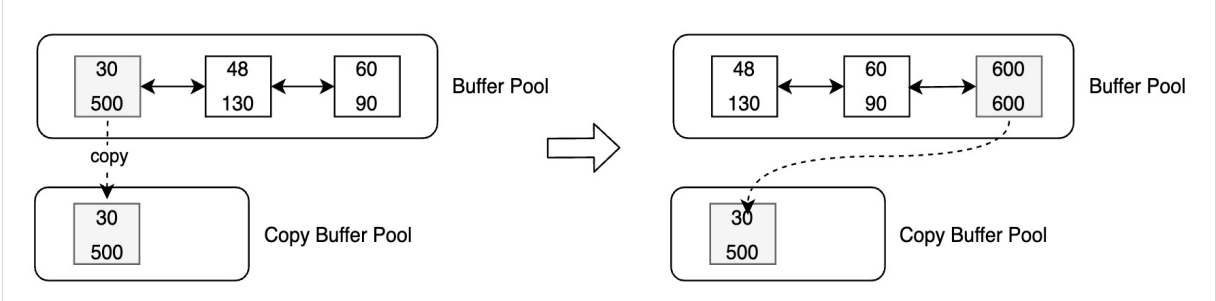
引入刷脏控制之后，仅满足刷脏条件的Buffer才能写入存储。如果某个Buffer修改非常频繁，可能导致Buffer Latest LSN总是大于Oldest Apply LSN，该Buffer始终无法满足刷脏条件，此类Buffer称之为热点页。热点页会导致一致性位点无法推进，为解决热点页的刷脏问题，PolarDB引入了Copy Buffer机制。

Copy Buffer机制会将特定的、不满足刷脏条件的Buffer从Buffer Pool中拷贝至新增的Copy Buffer Pool中，Copy Buffer Pool中的Buffer不会再被修改，其对应的Latest LSN也不会更新，随着Oldest Apply LSN的推进，Copy Buffer会逐步满足刷脏条件，从而可以将Copy Buffer落盘。

引入Copy Buffer机制后，刷脏的流程如下：

1. 如果Buffer不满足刷脏条件，判断其最近修改次数以及距离当前日志位点的距离，超过一定阈值，则将当前数据页拷贝至Copy Buffer Pool中。
2. 下次再刷该Buffer时，判断其是否满足刷脏条件。如果满足，则将该Buffer写入存储并释放其对应的Copy Buffer。
3. 如果Buffer不满足刷脏条件，则判断其是否存在Copy Buffer。若存在且Copy Buffer满足刷脏条件，则将Copy Buffer落盘。
4. Buffer被拷贝到Copy Buffer Pool之后，如果对该Buffer进行修改，则会重新生成该Buffer的Oldest LSN，并将其追加到FlushList末尾。

如下图所示，[Oldest LSN, latest LSN]为[30, 500]的Buffer被认为是热点页，将当前Buffer拷贝至Copy Buffer Pool中，随后该数据页再次被修改。假设修改对应的LSN为600，则设置其Oldest LSN为600，并将其从FlushList中删除，然后追加至FlushList 末尾。此时，Copy Buffer中数据页不会再修改，其Latest LSN始终为500，如果满足刷脏条件，则可以将Copy Buffer写入存储。



> **说明** 引入Copy Buffer之后，一致性位点的计算方法则有所改变。FlushList中的Oldest LSN不再是最小的Oldest LSN，Copy Buffer Pool中可能存在更小的Oldest LSN。因此，除考虑FlushList中的Oldest LSN之外，还需要遍历Copy Buffer Pool，找到Copy Buffer Pool中最小的Oldest LSN，取两者的最小值即为一致性位点。

Lazy Checkpoint

PolarDB引入的一致性位点概念，与checkpoint的概念类似。PolarDB中checkpoint位点表示该位点之前的所有数据都已经落盘，数据库Crash Recovery时可以从checkpoint位点开始恢复，提升恢复效率。普通的checkpoint会将所有Buffer Pool中的脏页以及其他内存数据落盘，这个过程可能耗时较长且在此期间I/O吞吐较大，可能会对正常的业务请求产生影响。

借助一致性位点，PolarDB引入了一种特殊的checkpoint：Lazy Checkpoint。之所以称之为Lazy，是与普通的checkpoint相比，Lazy Checkpoint不会把Buffer Pool中所有的脏页落盘，而是直接使用当前的一致性位点作为checkpoint位点，极大地提升了checkpoint的执行效率。

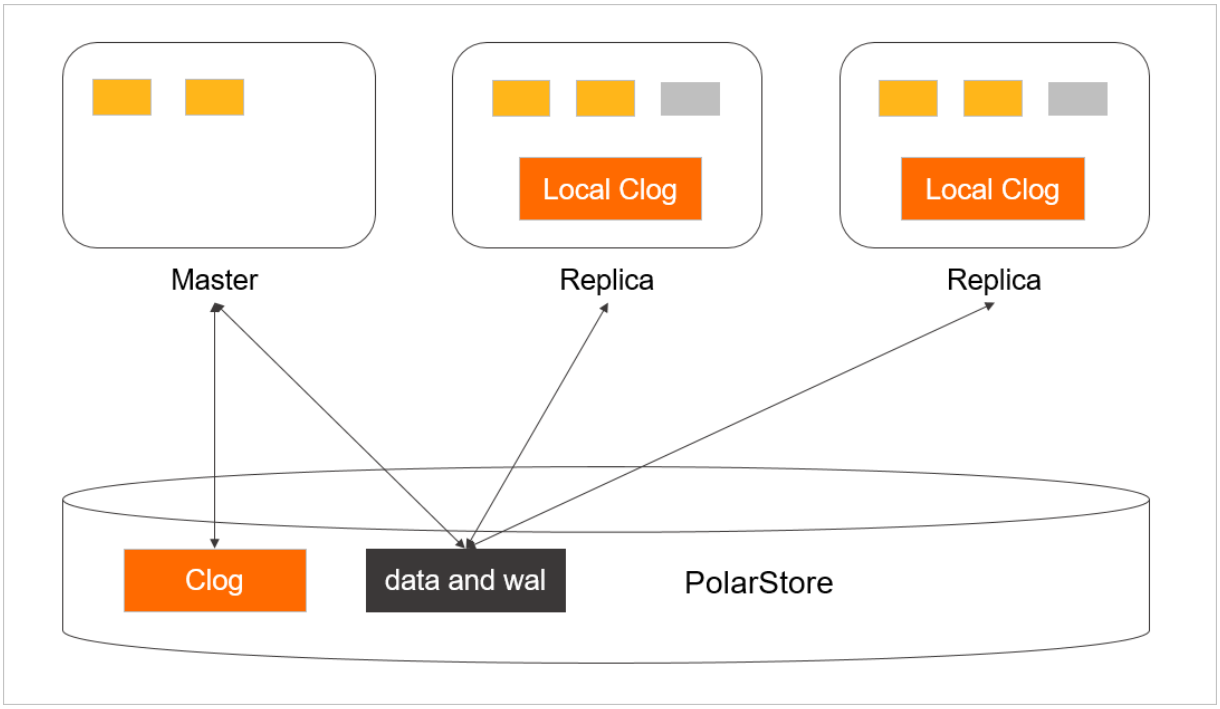
Lazy Checkpoint的整体思路是将普通checkpoint一次性刷大量脏页落盘的逻辑转换为后台刷脏进程持续不断落盘并维护一致性位点的逻辑。

> **说明** Lazy Checkpoint与PolarDB中Full Page Write的功能有冲突，开启Full Page Write之后会自动关闭该功能。

1.4. 文件存储

本文为您介绍PolarDB O引擎的文件存储。

- 主节点和只读节点的数据和WAL日志保存在共享存储中。
- 配置文件、Log文件和临时文件在每个节点都会单独保存。
- 主节点的CLOG文件存储在共享存储中；只读节点的CLOG文件会在各自节点上进行维护。
- 主节点的pg_control文件存储在共享存储中；只读节点的pg_control文件仅在启动时从共享存储中读取，启动后在内存中维护。

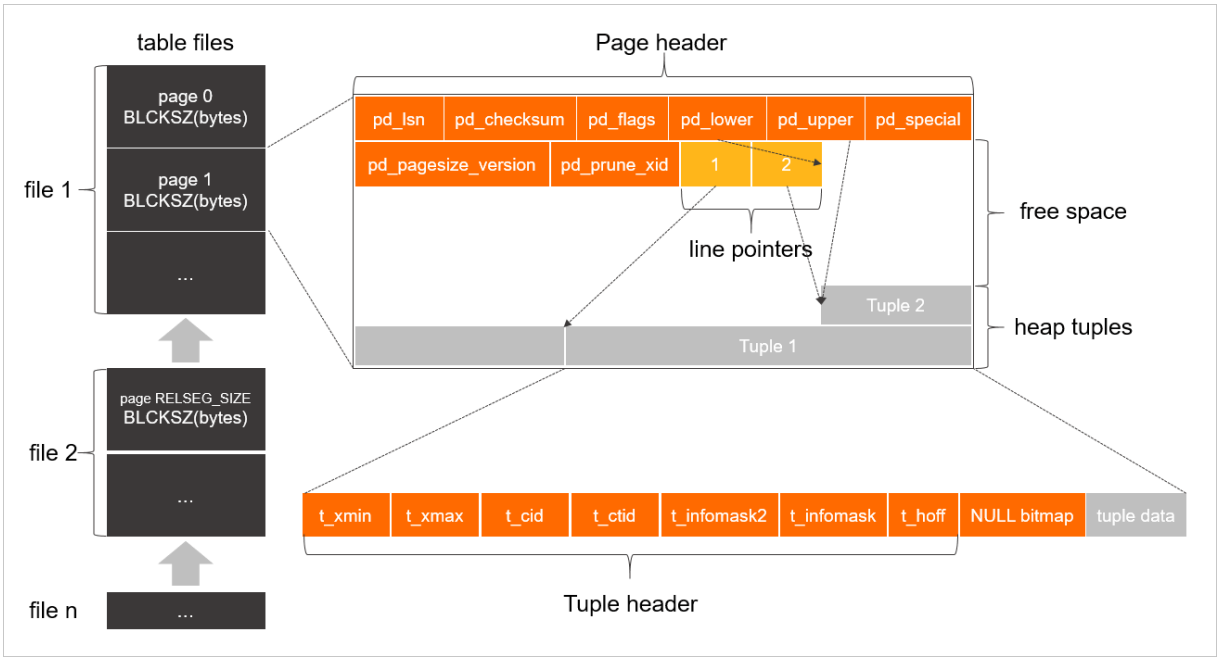


目录/文件	说明	存储位置
base	每个库的子目录。	共享存储
global	集群全局表的目录。	
pg_tblspc	表空间。	
pg_wal	WAL日志目录。	
pg_dynshmem	用于动态共享内存的文件。	所有节点存储一份
pg_snapshots	导出的快照（snapshot）。	
pg_replslot	replication slot数据。	
pg_stat_tmp	统计子系统的临时文件。	
pg_stat	统计子系统的持久化文件。	
pg_serial	已提交的Serializable级别的事务信息。	
pg_xact	事务日志文件。	• Master读写共享存储 • Replica本地存放一份
pg_commit_ts	已提交事务的时间。	
pg_multixact	子事务的信息。	
pg_version	版本信息。	

目录/文件	说明	存储位置
postgresql.auto.conf	参数配置文件，优先级大于postgresql.conf。	所有节点各自存储一份
postmaster.opts	记录服务器最后一次启动时的命令参数。	
postmaster.pid	一个锁文件，记录postmaster进程ID、数据库目录路径、postmaster进程启动时间、端口号等信息。	
postgresql.conf	参数配置文件。	
pg_hba.conf	客户端认证控制文件。	
pg_ident.conf	本地用户名映射文件配置。	

1.5. 堆表结构

本文为您展示PolarDB O引擎的堆表结构图。



结构	说明
table files	表文件，由一个或多个数据文件组成，文件数量与文件大小有关，超过1 GB就会生成一个新的数据文件。
file	数据文件，由多个数据块组成。
page	数据块，由多个数据行组成。
page header	数据块头，详细信息请参见数据块头信息。

结构	说明
tuple	数据行。
tuple header	数据行头，详细信息请参见数据行头信息。

数据块头信息

结构	说明
pd_lsn	最后修改这个块的WAL的位点。
pd_checksum	页面校验码。
pd_flags	标志位。
pd_lower	到空闲空间开头的偏移量。
pd_upper	到空闲空间结尾的偏移量。
pd_special	到特殊空间开头的偏移量。
pd_pagesize_version	页面大小和布局版本号信息。
pd_prune_xid	页面上最老未删除的xmax，如果没有则为0。

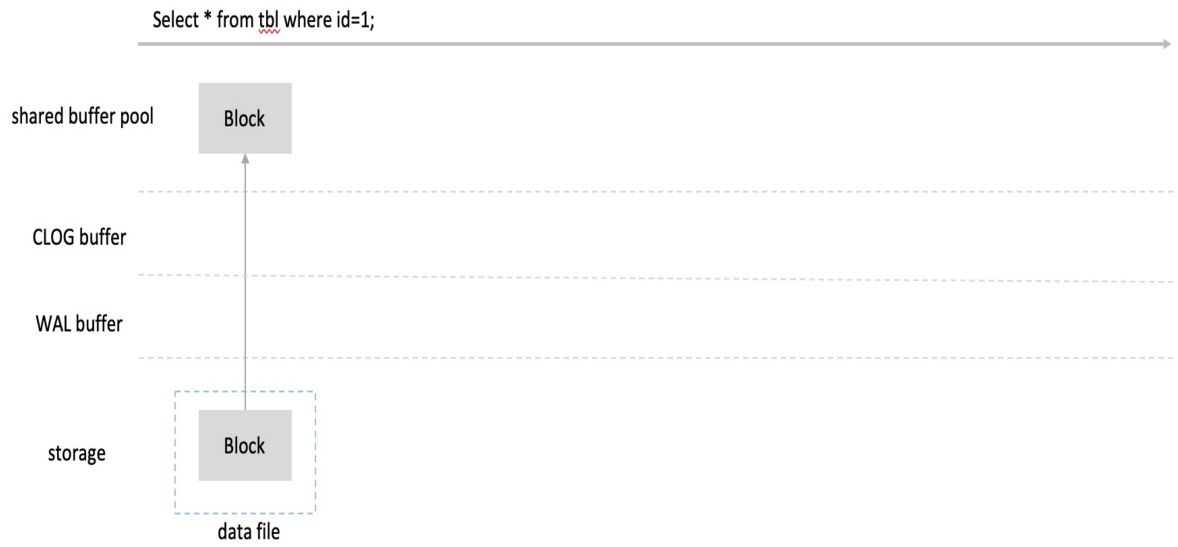
数据行头信息

结构	说明
t_xmin	插入事务ID（XID）。
t_xmax	删除事务ID（XID）。
t_cid	插入或删除CID。
t_ctid	当前版本的页面偏移量或者指向更新的版本。
t_infomask	标志位。
t_infomask2	额外标志位。
t_hoff	到用户数据的偏移量。
tuple data	实际的数据。

1.6. 查询机制

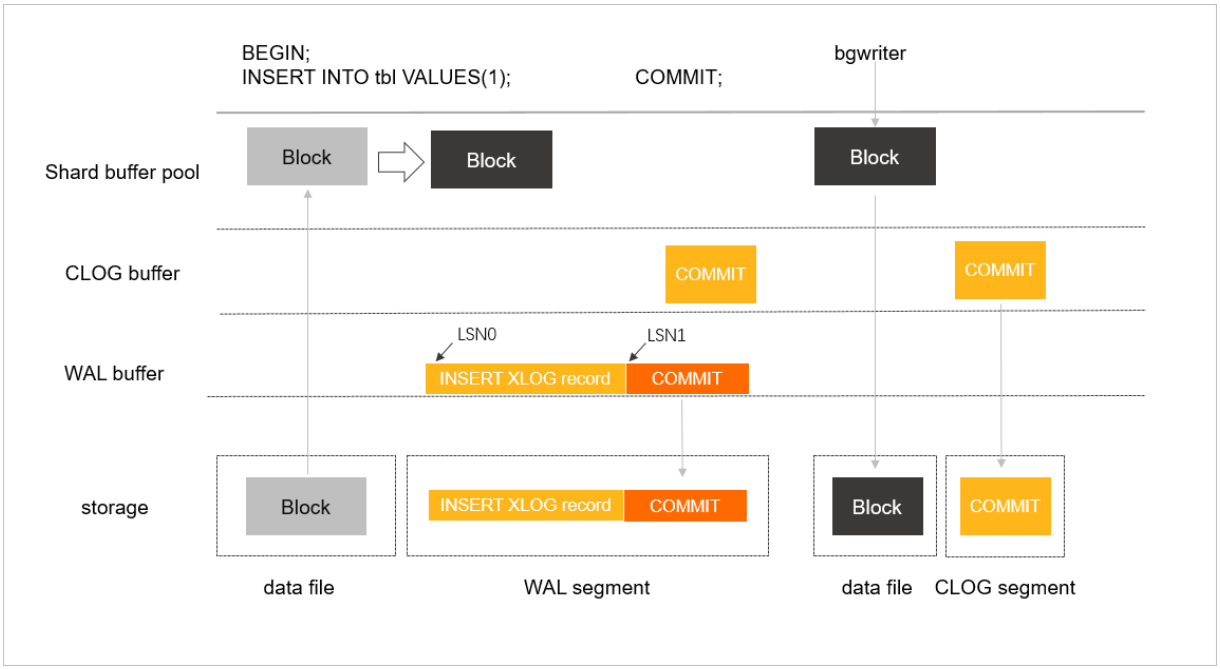
本节介绍PolarDB O引擎的查询机制。

SELECT



使用SELECT查询数据时，需要从磁盘中读取一个数据块到内存中，然后通过内存中数据块获取返回结果。

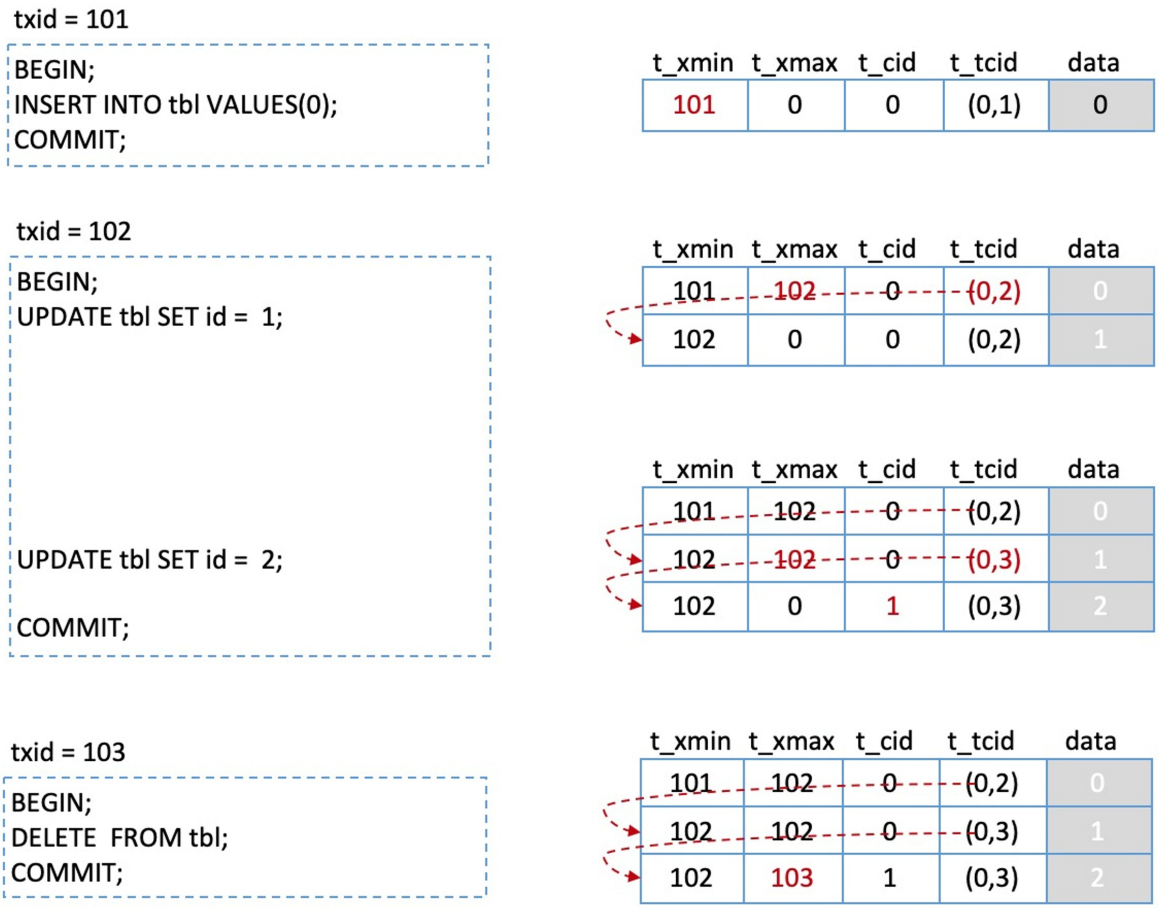
INSERT



当您使用INSERT插入一条数据时，具体过程如下：

1. 从数据文件中读取一个数据块到内存中，并将这行数据写入这个数据块，系统生成一条INSERT的WAL日志。
2. 当执行COMMIT命令后，系统会产生一条COMMIT的WAL日志。
3. INSERT WAL日志和COMMIT WAL日志产生后会被立即写入磁盘，CLOG缓存记录这个事务提交成功的信息。
4. 数据块中的数据通过bgwriter写入到数据文件中，同时更新CLOG文件。

UPDATE



上图中，txid表示事务号，t_xmin表示插入的事务号，t_xmax表示删除事务号，t_cid表示事务内命令的序号，t_tcid表示数据块内的偏移量，data表示实际数据存储的值。

- 事务101

  左侧SQL表示在tbl表中插入一行值为0的数据，右侧表示实际的数据块存储内容。事务号为101；由于没有删除事务，所以删除事务号为0；事务内命令的序号为0；偏移量为（0，1）表示数据块的第一个偏移位置。

- 事务102

  事务102更新了tbl表，设置ID等于1，然后继续更新设置ID等于2并提交。

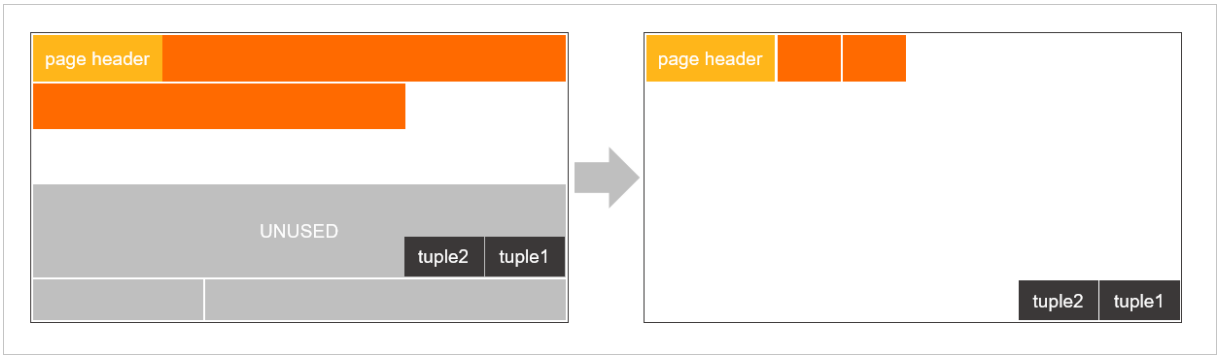
  - 从右侧上半张图可以看出，删除事务号从0变成了102，表示这行数据被102事务删除了；偏移量变成了（0，2），表示指向了0号数据块的第二个偏移位置；从第二个偏移位置可以看到事务号为102，表示事务102插入了这行数据，数据存储值为1。
  - 从右侧下半张图可以看出，事务号依旧是102，偏移量指向了（0，3），表示存储实际的数据为2；事务内命令的序号为1表示事务内的第二条命令进行了更新，与第二个UPDATE对应。

- 事务103

  从左侧SQL语句表示执行了删除tbl表的动作，删除事务号更新为103。

1.7. 垃圾回收机制

PolarDB O引擎支持通过vacuum命令对垃圾数据进行回收。回收垃圾数据释放的空间并不会返还给操作系统，但是新插入或更新的数据可以使用该空间。



在上图中，`tuple1` 和 `tuple2` 代表有效数据行，`unused` 代表无效数据行，通过垃圾回收后，无效数据行占用的空间得到释放，可以被重新利用。

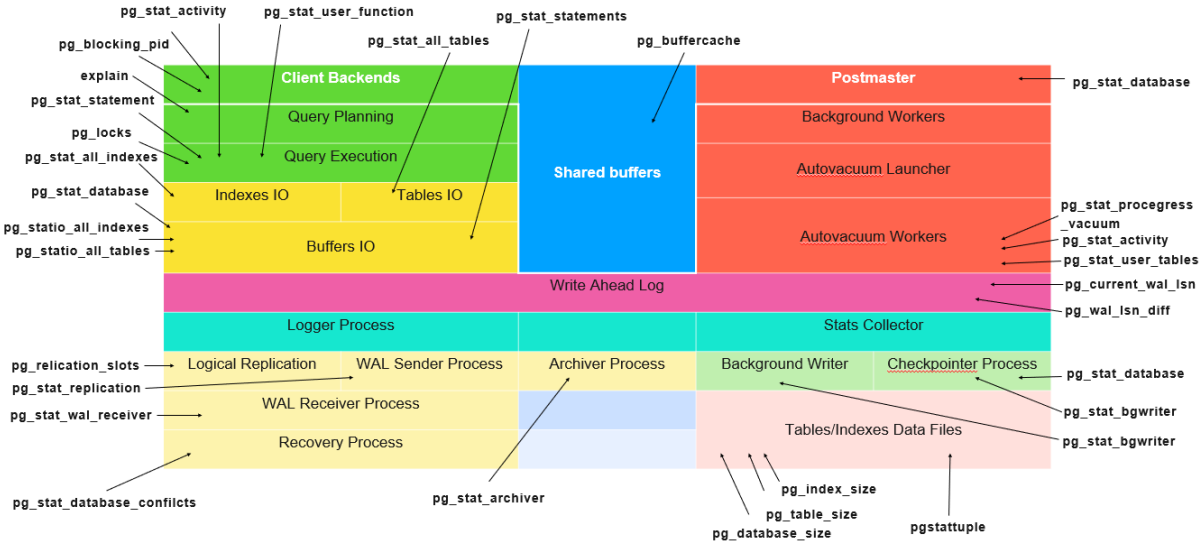
PolarDB O引擎通过多版本来实现MVCC机制，当执行UPDATE和DELETE操作时会产生一个新的数据行版本，老的数据行版本会变成无效版本，这些无效版本数据会占用数据块的空间，需要及时进行清理，否则会导致数据的膨胀。当无效数据行占总数据行数的一定数值时（您可以根据需求自定义设置无效数据行数占比），PolarDB O引擎会自动触发垃圾数据的回收动作，通过VACUUM进程对垃圾数据进行回收；您也可以手动执行 `vacuum table_name` 进行垃圾数据的回收。

2.定位性能问题

2.1. 监控信息概览

通过统计信息的相关指标您可以了解数据库整体的使用情况，例如SQL语句、表、索引以及后台进程的相关信息等等。

下图是整体监控大图，供问题定位时参考，更多PolarDB提供的监控视图请参见查看视图。



2.2. 查看视图

2.2.1. 数据库

本章节主要介绍PolarDB O引擎中的不同视图详细信息，可通过查询不同视图获取相关实例信息，进而方便定位性能问题。

数据库基本信息

- pg_stat_database

  pg_stat_database视图将显示集群中每一个数据库中的每一行数据库范围的统计信息。相关参数信息如下表所示：

参数	类型	描述
datid	oid	数据库OID。
datname	name	数据库名称。
numbackends	integer	当前连接到数据库的后端数量。这是在视图中唯一一个返回反映当前状态值的列。所有其他列返回从上次重置以来积累的值。
xact_commit	bigint	数据库中已经被提交的事务的数量。

参数	类型	描述
xact_rollback	bigint	数据库中已经被回滚的事务的数量。
blks_read	bigint	数据库中被读取的磁盘块的数量。
blks_hit	bigint	磁盘块被发现已经在缓冲区中的次数，这样不需要一次读取（这只包括PolarDB O引擎缓冲区中的命中，而不包括在操作系统文件系统缓冲区中的命中）。
tup_returned	bigint	数据库中被查询返回的行数。
tup_fetched	bigint	数据库中被查询取出的行数。
tup_inserted	bigint	数据库中被查询插入的行数。
tup_updated	bigint	数据库中被查询更新的行数。
tup_deleted	bigint	数据库中被查询删除的行数。
conflicts	bigint	由于与恢复冲突而在数据库中被取消的查询的数目。
temp_files	bigint	在数据库中被查询创建的临时文件的数量。所有临时文件都会被统计，忽略创建这些临时文件（例如排序或哈希）的原因，并且忽略 log_temp_files 的设置。
temp_bytes	bigint	在数据库中被查询写到临时文件中的数据总量。所有临时文件都会被统计，忽略创建这些临时文件的原因，并且忽略 log_temp_files 的设置。
deadlocks	bigint	在数据库中被检测到的死锁数。
blk_read_time	double precision	在数据库中后端花费在读取数据文件块的时间。单位：毫秒。
blk_write_time	double precision	在数据库中后端花费在写数据文件块的时间。单位：毫秒。
stats_reset	timestamp with time zone	统计信息最近一次被重置的时间。

- **pg_stat_bgwriter**

  pg_stat_bgwriter视图将只有单独的一行，包含集群的全局数据。

参数	类型	描述
checkpoints_timed	bigint	已经被执行的检查点的数量。
checkpoints_req	bigint	已经被执行的请求检查点的数量。

参数	类型	描述
checkpoint_write_time	double precision	在文件被写入磁盘的检查点处理部分花费的总时间。单位：毫秒。
checkpoint_sync_time	double precision	在文件被同步到磁盘中的检查点处理部分花费的总时间。单位：毫秒。
buffers_checkpoint	bigint	在检查点期间被写的缓冲区数目。
buffers_clean	bigint	被后台写进程写的缓冲区数目。
maxwritten_clean	bigint	后台写进程由于已经写了太多缓冲区而停止清洁扫描的次数。
buffers_backend	bigint	被后端直接写的缓冲区数量。
buffers_backend_fsync	bigint	执行fsync调用的次数（通常即使后端会自动进行写操作，后台写进程也会处理这些）。
buffers_alloc	bigint	被分配的缓冲区数量。
stats_reset	timestamp with time zone	这些统计信息上次被重置的时间。

活动状态

- polar_stat_activity

  描述所有进程的当前状态，polar_stat_activity视图统计的是累计值。

参数	类型	描述
datid	oid	数据库OID。
datname	name	数据库名称。
pid	integer	进程ID。
usesysid	oid	用户ID。
usename	name	用户名。
application_name	text	连接名称。
client_addr	inet	客户端IP。
client_hostname	text	客户端机器名。
client_port	integer	客户端端口。
backend_start	timestamp	客户端连接创建时间。

参数	类型	描述
xact_start	timestamp	事务开始时间。
query_start	timestamp	当前SQL开始执行时间。
state_change	timestamp	当前状态变更时间。
wait_event_type	text	等待事件类型，后端正在等待的事件类型，如果不存在则为 NULL。可能的值有： ○ LWLock：后端正在等待轻量级锁。每一个这样的锁保护着共享内存中的一个特殊数据结构。wait_event 将含有标识该轻量级锁目的的名称（一些锁具有特定的名称，其他是一组具有类似目的的锁中的一部分）。 ○ Lock：后端正在等待重量级锁。重量级锁，也称为锁管理器锁或者简单锁，主要保护 SQL 可见的对象，例如表。不过，它们也被用于确保特定内部操作的互斥，例如关系扩展。wait_event 将标识等待的锁的类型。 ○ BufferPin：服务器进程正在等待访问数据缓冲区，而此时没有其他进程正在检查该缓冲区。如果另一个进程持有一个最终从要访问的缓冲区中读取数据的打开的游标，缓冲区 pin 等待可能会被拖延。 ○ Activity：服务器进程处于闲置状态。这被用于在其主处理循环中等待活动的系统进程。wait_event 将标识特定的等待点。 ○ Extension：服务器进程正在扩展模块中等待活动。这个分类被用于要跟踪自定义等待点的模块。 ○ Client 服务器进程正在套接字上等待来自用户应用的某种活动，并且该服务器预期某种与其内部处理无关的事情发生。wait_event 将标识特定的等待点。 ○ IPC 服务器进程正在等待来自服务器中另一个进程的某种活动。wait_event 将标识特定的等待点。 ○ Timeout 服务器进程正在等待超时发生。wait_event 将标识特定的等待点。 ○ IO 服务器进程正在等待IO完成。wait_event 将标识特定的等待点。
wait_event	text	如果后端当前正在等待，则是等待事件的名称，否则为 NULL。
state	text	当前状态。

参数	类型	描述
backend_xid	xid	当前客户端xid。
backend_xmin	xid	当前客户端xmin。
query	text	SQL语句。
backend_type	text	当前backend类型。
queryid	bigint	SQL唯一标识
wait_object	text	如果有等待事件， 表示所等待的对象。
wait_type	text	如果有等待事件， 表示等待对象的类型。
wait_time_ms	double	如果有等待事件， 表示已经等待的时间。
cpu_user	bigint	当前backend用户态CPU。
cpu_sys	bigint	当前backend系统态CPU。
rss	bigint	当前backend内存占用。
pfs_read_ps	bigint	pfs读IO累计次数。
pfs_write_ps	bigint	pfs写IO累计次数。
pfs_read_throughput	bigint	pfs读IO累计吞吐量。
pfs_write_throughput	bigint	pfs写IO累计吞吐量。
pfs_read_latency_ms	double	pfs读IO延迟时间。
pfs_write_latency_ms	double	pfs写IO延迟时间。
local_read_ps	bigint	本地文件系统读IO累计次数。
local_write_ps	bigint	本地文件系统写IO累计次数。
local_read_throughput	bigint	本地文件系统读IO累计吞吐量。
local_write_throughput	bigint	本地文件系统写IO累计吞吐量。
local_read_latency_ms	double	本地文件系统读IO延迟时间。

参数	类型	描述
local_write_latency_ms	double	本地文件系统写IO延迟时间。

等待事件类型描述

等待事件类型	等待事件名称	描述
	ShmemIndexLock	正等待在共享内存中查找或者分配空间。
	OidGenLock	正等待分配或者赋予OID。
	XidGenLock	正等待分配或者赋予事务 ID。
	ProcArrayLock	正等待在事务结尾得到快照或者清除事务 ID。
	SInvalReadLock	正等待从共享无效消息队列中检索或者移除消息。
	SInvalWriteLock	正等待在共享无效消息队列中增加消息。
	WALBufMappingLock	正等待在 WAL 缓冲区中替换页面。
	WALWriteLock	正等待 WAL 缓冲区被写入到磁盘。
	ControlFileLock	正等待读取或者更新控制文件或创建新的 WAL 文件。
	CheckpointLock	正等待执行检查点。
	CLogControlLock	正等待读取或者更新事务状态。
	SubtransControlLock	正等待读取或者更新子事务信息。
	MultiXactGenLock	正等待读取或者更新共享多事务状态。
	MultiXactOffsetControlLock	正等待读取或者更新多事务偏移映射。
	MultiXactMemberControlLock	正等待读取或者更新多事务成员映射。
	RelCacheInitLock	正等待读取或者写入关系缓冲区初始化文件。
	CheckpointerCommLock	正等待管理 fsync 请求。
	TwoPhaseStateLock	正等待读取或者更新预备事务的状态。
	TablespaceCreateLock	正等待创建或者删除表空间。
	BtreeVacuumLock	正等待读取或者更新B-树索引的 vacuum 相关的信息。
	AddinShmemInitLock	正等待管理共享内存中的空间分配。

等待事件类型	等待事件名称	描述
LWLock	AutovacuumLock	自动清理工作者或者启动器正等待更新或者读取自动清理工作者的当前状态。
	AutovacuumScheduleLock	正等待确认选中进行清理的表仍需要清理。
	SyncScanLock	正等待为同步扫描得到表上扫描的开始位置。
	RelationMappingLock	正等待更新用来存储目录到文件节点映射的关系映射文件。
	AsyncCtlLock	正等待读取或者更新共享通知状态。
	AsyncQueueLock	正等待读取或者更新通知消息。
	SerializableXactHashLock	正等待检索或者存储有关可序列化事务的信息。
	SerializableFinishedListLock	正等待访问已结束可序列化事务的列表。
	SerializablePredicateLockListLock	正等待在由可序列化事务持有的所列表上执行操作。
	OldSerXidLock	正等待读取或者记录冲突的可序列化事务。
	SyncRepLock	正等待读取或者更新有关同步复制的信息。
	BackgroundWorkerLock	正等待读取或者更新后台工作者状态。
	DynamicSharedMemoryControlLock	正等待读取或者更新动态共享内存状态。
	AutoFileLock	正等待更新postgresql.auto.conf文件。
	ReplicationSlotAllocationLock	正等待分配或者释放复制槽。
	ReplicationSlotControlLock	正等待读取或者更新复制槽状态。
	CommitTsControlLock	正等待读取或者更新事务提交时间戳。
	CommitTsLock	正等待读取或者更新事务时间戳的最新设置值。
	ReplicationOriginLock	正等待设置、删除或者使用复制源头。
	MultiXactTruncationLock	正等待读取或者阶段多事务信息。
	OldSnapshotTimeMapLock	正等待读取或者更新旧的快照控制信息。

等待事件类型	等待事件名称	描述
	BackendRandomLock	正等待产生随机数。
	LogicalRepWorkerLock	正等待逻辑复制工作者上的动作完成。
	CLogTruncationLock	正等待截断预写式日志或者等待预写式日志截断操作完成。
	clog	正在等地clog （事务状态）缓冲区上的I/O。
	commit_timestamp	正等待提交时间戳缓冲区上的 I/O。
	subtrans	正等待子事务缓冲区上的 I/O。
	multixact_offset	正等待多事务偏移缓冲区上的 I/O。
	multixact_member	正等待多事务成员缓冲区上的 I/O。
	async	正等待 async（通知）缓冲区上的 I/O。
	oldserxid	正等待 oldserxid 缓冲区上的 I/O。
	wal_insert	正等待把 WAL 插入到内存缓冲区。
	buffer_content	正等待读取或者写入内存中的数据页。
	buffer_io	正等待一个数据页面上的 I/O。
	replication_origin	正等待读取或者更新复制进度。
	replication_slot_io	正等待复制槽上的 I/O。
	proc	正等待读取或者更新 fast-path 锁信息。
	buffer_mapping	正等待把数据块与缓冲池中的缓冲区关联。
	lock_manager	正等待增加或者检查用于后端的锁，或者正等待加入或者退出锁定组（并行查询使用）。
	predicate_lock_manager	正等待增加或者检查谓词锁信息。
	parallel_query_dsa	正等待并行查询动态共享内存分配锁。
	tbm	正等待TBM共享迭代器锁。
	parallel_append	在Parallel Append计划执行期间等待选择下一个子计划。
	parallel_hash_join	在Parallel Hash计划执行期间等待分配或交换一块内存或者更新计数器。
	relation	正等待获得关系上的锁。

等待事件类型	等待事件名称	描述
Lock	extend	正等待扩展关系。
	page	正等待获得关系上的页面的锁。
	tuple	正等待获得元组上的锁。
	transactionid	正等待事务结束。
	virtualxid	正等待获得虚拟 xid 锁。
	speculative token	正等待获取speculative insertion lock。
	object	正等待获得非关系数据库对象上的锁。
	userlock	正等待获得用户锁。
	advisory	正等待获得咨询用户锁。
BufferPin	BufferPin	正等待在缓冲区上加 pin。
Activity	ArchiverMain	正在归档进程的主循环中等待。
	AutoVacuumMain	正在autovacuum启动器进程的主循环中等待。
	BgWriterHibernate	正在后台写入器进程中等待，休眠中。
	BgWriterMain	正在后台写入器进程的后台工作者的主循环中等待。
	CheckpointerMain	正在检查点进程的主循环中等待。
	LogicalApplyMain	正在逻辑应用进程的主循环中等待。
	LogicalLauncherMain	正在逻辑启动器进程的主循环中等待。
	PgStatMain	正在统计收集器进程的主循环中等待。
	RecoveryWalAll	在恢复时等待来自于任意类型来源（本地、归档或流）的WAL。
	RecoveryWalStream	在恢复时等待来自于流的WAL。
	SysLoggerMain	正在系统日志进程的主循环中等待。
	WalReceiverMain	正在WAL接收器进程的主循环中等待。
	WalSenderMain	正在WAL发送器进程的主循环中等待。
	WalWriterMain	正在WAL写入器进程的主循环中等待。
	ClientRead	正等待从客户端读取数据。
	ClientWrite	正等待向客户端写入数据。

等待事件类型	等待事件名称	描述
Client	LibPQWalReceiverConnect	正在WAL接收器中等待建立与远程服务器的连接。
	LibPQWalReceiverReceive	正在WAL接收器中等待从远程服务器接收数据。
	SSLOpenServer	正在尝试连接期间等待SSL。
	WalReceiverWaitStart	正等待startup进程发送流复制的初始数据。
	WalSenderWaitForWAL	正在WAL发送器进程中等待WAL被刷写。
	WalSenderWriteData	在WAL发送器进程中处理来自WAL接收器的回复时等待任意活动。
Extension	Extension	正在扩展中等待。
	BgWorkerShutdown	正等待后台工作者关闭。
	BgWorkerStartup	正等待后台工作者启动。
	BtreePage	正等待继续并行B-树扫描所需的页号变得可用。
	ClogGroupUpdate	正等待组领袖在事务结束时更新事务状态。
	ExecuteGather	在执行Gather节点时等待来自子进程的活动。
	Hash/Batch/Allocating	正等待选出的Parallel Hash参与者分配哈希表。
	Hash/Batch/Electing	正在选出Parallel Hash参与者来分配哈希表。
	Hash/Batch/Loading	正等待其他Parallel Hash参与者完成装载哈希表。
	Hash/Build/Allocating	正等待选出的Parallel Hash参与者分配初始哈希表。
	Hash/Build/Electing	正在选出Parallel Hash参与者以分配初始哈希表。
	Hash/Build/HashingInner	正等待其他Parallel Hash参与者完成对内关系的哈希操作。
	Hash/Build/HashingOuter	正等待其他Parallel Hash参与者完成对外关系的哈希操作。
	Hash/GrowBatches/Allocating	正等待选出的Parallel Hash参与者分配更多批次。
	Hash/GrowBatches/Deciding	正在选出Parallel Hash参与者决定未来的批次增长。
	Hash/GrowBatches/Electing	正在选出Parallel Hash参与者分配更多批次。

等待事件类型	等待事件名称	描述
IPC	Hash/GrowBatches/Finishing	正在等待选出的Parallel Hash参与者决定未来的批次增长。
	Hash/GrowBatches/Repartitioning	正等待其他Parallel Hash参与者完成重新分区。
	Hash/GrowBuckets/Allocating	正等待选出的Parallel Hash参与者完成更多桶的分配。
	Hash/GrowBuckets/Electing	正在选出Parallel Hash参与者分配更多桶。
	Hash/GrowBuckets/Reinserting	正等待其他Parallel Hash参与者完成将元组插入到新桶的操作。
	LogicalSyncData	正等待逻辑复制的远程服务器发送用于初始表同步的数据。
	LogicalSyncStateChange	正等待逻辑复制的远程服务器更改状态。
	MessageQueueInternal	正等待其他进程被挂接到共享消息队列。
	MessageQueuePutMessage	正等待把协议消息写到共享消息队列。
	MessageQueueReceive	正等待从共享消息队列接收字节。
	MessageQueueSend	正等待向共享消息队列中发送字节。
	ParallelBitmapScan	正等待并行位图扫描被初始化。
	ParallelCreateIndexScan	正等待并行CREATE INDEX工作者完成堆扫描。
	ParallelFinish	正等待并行工作者完成计算。
	ProcArrayGroupUpdate	正等待组领袖在事务结束时清除事务ID。
	ReplicationOriginDrop	正等待复制源头变得不活跃以便被删除。
	ReplicationSlotDrop	正等待复制槽变得不活跃以便被删除。
	SafeSnapshot	正等待用于READ ONLY DEFERRABLE事务的快照。
	SyncRep	正在同步复制期间等待来自远程服务器的确认。
Timeout	BaseBackupThrottle	当有限流活动时在基础备份期间等待。
	PgSleep	正在调用pg_sleep的进程中等待。
	RecoveryApplyDelay	在恢复时等待应用WAL，因为它被延迟了。

等待事件类型	等待事件名称	描述
	BufFileRead	正等待从缓存的文件中读取。
	BufFileWrite	正等待向缓存的文件中写入。
	ControlFileRead	正等待从控制文件中读取。
	ControlFileSync	正等待控制文件到达稳定存储。
	ControlFileSyncUpdate	正等待对控制文件的更新到达稳定存储。
	ControlFileWrite	正等待对控制文件的写入。
	ControlFileWriteUpdate	正等待写操作更新控制文件。
	CopyFileRead	正在文件拷贝操作期间等待读操作。
	CopyFileWrite	正在文件拷贝操作期间等待写操作。
	DataFileExtend	正等待关系数据文件被扩充。
	DataFileFlush	正等待关系数据文件到达稳定存储。
	DataFileImmediateSync	正等待关系数据文件的立即同步到达稳定存储。
	DataFilePrefetch	正等待从关系数据文件中的一次异步预取。
	DataFileRead	正等待对关系数据文件的读操作。
	DataFileSync	正等待对关系数据文件的更改到达稳定存储。
	DataFileTruncate	正等待关系数据文件被截断。
	DataFileWrite	正等待对关系数据文件的写操作。
	DSMFillZeroWrite	等待向动态共享内存备份文件中写零字节。
	LockFileAddToDataDirRead	在向数据目录锁文件中增加一行时等待读操作。
	LockFileAddToDataDirSync	在向数据目录锁文件中增加一行时等待数据到达稳定存储。
	LockFileAddToDataDirWrite	在向数据目录锁文件中增加一行时等待写操作。
	LockFileCreateRead	在创建数据目录锁文件期间等待读取。
	LockFileCreateSync	在创建数据目录锁文件期间等待数据到达稳定存储。
	LockFileCreateWrite	在创建数据目录锁文件期间等待写操作。

等待事件类型	等待事件名称	描述
IO	LockFileReCheckDataDirRead	在重新检查数据目录锁文件的过程中等待读操作。
	LogicalRewriteCheckpointSync	在检查点期间等待逻辑重写映射到达稳定存储。
	LogicalRewriteMappingSync	在逻辑重写期间等待映射数据到达稳定存储。
	LogicalRewriteMappingWrite	在逻辑重写期间等待对映射数据的写操作。
	LogicalRewriteSync	正等待逻辑重写映射到达稳定存储。
	LogicalRewriteWrite	正等待对逻辑重写映射的写操作。
	RelationMapRead	正等待对关系映射文件的读操作。
	RelationMapSync	正等待关系映射文件到达稳定存储。
	RelationMapWrite	正等待对关系映射文件的写操作。
	ReorderBufferRead	在重排序缓冲区管理期间等待读操作。
	ReorderBufferWrite	在重排序缓冲区管理期间等待写操作。
	ReorderLogicalMappingRead	在重排序缓冲区管理期间等待对逻辑映射的读操作。
	ReplicationSlotRead	正等待对一个复制槽控制文件的读操作。
	ReplicationSlotRestoreSync	在把复制槽控制文件恢复到内存的过程中等待它到达稳定存储。
	ReplicationSlotSync	正等待复制槽控制文件到达稳定存储。
	ReplicationSlotWrite	正等待对复制槽控制文件的写操作。
	SLRUFlushSync	在检查点或者数据库关闭期间等待SLRU数据到达稳定存储。
	SLRURead	正等待对SLRU页面的读操作。
	SLRUSync	正等待SLRU数据在页面写之后到达稳定存储。
	SLRUWrite	正等待SLRU页面上的写操作。
	SnapbuildRead	正等待序列化历史目录快照的读操作。
	SnapbuildSync	正等待序列化历史目录快照到达稳定存储。
	SnapbuildWrite	正等待序列化历史目录快照的写操作。

等待事件类型	等待事件名称	描述
	TimelineHistoryFileSync	正等待通过流复制接收到的时间线历史文件到达稳定存储。
	TimelineHistoryFileWrite	正等待通过流复制接收到的时间线历史文件的读操作。
	TimelineHistoryRead	正等待时间线历史文件上的读操作。
	TimelineHistorySync	正等待新创建的时间线历史文件达到稳定存储。
	TimelineHistoryWrite	正等待新创建的时间线历史文件上的写操作。
	TwophaseFileRead	正等待两阶段状态文件的读操作。
	TwophaseFileSync	正等待两阶段状态文件到达稳定存储。
	TwophaseFileWrite	正等待两阶段状态文件的写操作。
	WALBootstrapSync	在自举期间等待WAL到达稳定存储。
	WALBootstrapWrite	在自举期间等待WAL页面的写操作。
	WALCopyRead	在通过拷贝已有WAL段创建新的WAL段时等待一个读操作。
	WALCopySync	正等待通过拷贝已有WAL段创建的新WAL段到达稳定存储。
	WALCopyWrite	在通过拷贝已有WAL段创建新的WAL段时等待写操作。
	WALInitSync	正等待新初始化的WAL文件到达稳定存储。
	WALInitWrite	在初始化新的WAL文件期间等待写操作。
	WALRead	正等待对WAL文件的读操作。
	WALSenderTimelineHistoryRead	在walsender的时间线命令期间等待对时间线历史文件的读操作。
	WALSyncMethodAssign	在指派WAL同步方法时等待数据到达稳定存储。
	WALWrite	正等待对WAL文件的写操作。

说明 polar_stat_activity视图依赖于 `polar_monitor` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行create extension polar_monitor命令使其生效。

- **polar_stat_activity_rt**

  描述所有进程的当前状态， **polar_stat_activity_rt** 视图统计的是实时数据。

参数	类型	描述
pid	integer	进程ID。
backend_type	text	当前backend类型。
cpu_user	bigint	当前backend用户态CPU。
cpu_sys	bigint	当前backend系统态CPU。
rss	bigint	当前backend内存占用。
local_read_ps	bigint	本地文件系统读IO累计次数。
local_write_ps	bigint	本地文件系统写IO累计次数。
local_read_throughput	bigint	本地文件系统读IO累计吞吐量。
local_write_throughput	bigint	本地文件系统写IO累计吞吐量。
local_read_latency_ms	double	本地文件系统读IO延迟时间。
local_write_latency_ms	double	本地文件系统写IO延迟时间。

> **说明** polar_stat_activity_rt视图依赖于 `polar_monitor` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 命令使其生效。

- **polar_delta**

> **注意** `polar_delta` 是一个函数，用于获取视图的增量值，依赖于 `polar_monitor` 插件。其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 命令使其生效。

使用说明如下：

i. 创建具有维度列和值列的视图。
   - 维度列名称必须以d开头。
   - 值列名称必须以v开头。

ii. 使用以下方法获取视图：

```
"select * from polar_delta (NULL::view_name)"
"\watch 1 select * from polar_delta (NULL::view_name)"
```

资源

- **CPU**

  会话级别的CPU指标已在**polar_stat_activity**视图中体现。

- **共享内存**

  PolarDB的buffer pool和latch等其它全局数据结构，均分配于共享内存中，它在启动时即静态申请完成，通过查看如下视图可获取相关信息。

  - **polar_stat_shmem**

    描述各类共享内存的监控信息。相关参数说明如下表所示：

参数	类型	描述
`shmname`	text	共享内存区域名。
`shmsize`	bigint	共享内存大小（字节数）。
`shmtype`	text	共享内存类型。

    > **说明** polar_stat_shmem视图依赖于 `polar_monitor` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 命令使其生效。

  - **polar_stat_shmem_total_size**

    描述共享内存的监控汇总信息。相关参数说明如下表所示：

参数	类型	描述
`shmsize`	bigint	共享内存大小（字节数）。
`shmtype`	text	共享内存类型。

    > **说明** polar_stat_shmem_total_size视图依赖于 `polar_monitor` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 使其生效。

- **会话私有内存**

  PolarDB的私有内存均在运行过程中动态分配或释放，其中会话级别的内存指标已经在**polar_stat_activity**中体现，更进一步，PolarDB中会使用内存上下文作为动态内存管理的基本单位，可以通过查看以下函数及视图获取。

- polar_get_mcxt()函数

参数	类型	描述
pid	integer	会话进程pid。
name	text	memory context名称。
level	int	层级。
nblocks	bigint	占用block数量。
freechunks	bigint	空闲block数量。
totalspace	bigint	总空间（字节数）。
freespace	bigint	空闲空间（字节数）。

- polar_backends_mcxt

  按照backend_type为粒度，统计内存上下文信息。相关参数说明如下表所示：

参数	类型	描述
pid	integer	进程ID。
name	text	memory context名称。
nblocks	bigint	占用block数量。
freechunks	bigint	空闲block数量。
totalspace	bigint	总空间（字节数）。
freespace	bigint	空闲空间（字节数）。

  说明 polar_backends_mcxt视图依赖于 `polar_monitor` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 命令使其生效。

- I/O

  会话级别的I/O指标已经在polar_stat_activity中体现，文件级别的I/O信息及I/O延迟分布可以通过查询以下视图获取。

- polar_stat_io_info

  按照文件类型为维度，描述I/O监控信息。相关参数说明如下表所示：

参数	类型	描述
filetype	text	文件类型。
fileloc	text	文件所在文件系统（目前有本地文件系统和共享文件系统PFS两种形式）。
open_count	numeric	文件打开总次数。
open_latency_us	double	文件打开总延迟。
close_count	numeric	文件关闭总次数。
read_count	numeric	读文件总次数。
write_count	numeric	写文件总次数。
read_throughput	numeric	读文件总吞吐。
write_throughput	numeric	写文件总吞吐。
read_latency_us	double	读文件总延迟 。单位：微秒。
write_latency_us	numeric	写文件总延迟。单位：微秒。
seek_count	numeric	seek总次数。
seek_latency_us	double	seek总延迟。
creat_count	numeric	创建文件总次数。
creat_latency_us	double	创建文件总延迟。
fsync_count	numeric	fsync总次数。
fsync_latency_us	double	fsync总延迟。
falloc_count	numeric	falloc总次数。
falloc_latency_us	double	falloc总延迟。

> **说明** polar_stat_io_info视图依赖于 `polar_monitor` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 命令使其生效。

- polar_stat_io_latency

  按照I/O类型为维度，统计I/O延时信息。相关参数说明如下表所示：

参数	类型	描述
iokind	text	文件操作类型 ■ fsync ： ■ creat ：创建文件。 ■ seek ： ■ open ：打开文件。 ■ read ：读文件。 ■ write ：写文件。 ■ falloc ：
num_lessthan200us	numeric	小于200微秒总次数。
num_lessthan400us	numeric	大于200微秒小于400微秒总次数。
num_lessthan600us	numeric	大于400微秒小于600微秒总次数。
num_lessthan800us	numeric	大于600微秒小于800微秒总次数。
num_lessthan1ms	numeric	大于800微秒小于1毫秒总次数。
num_lessthan10ms	numeric	大于1毫秒小于10毫秒总次数。
num_lessthan100ms	numeric	大于10毫秒小于100毫秒总次数。
num_morethan100ms	numeric	大于100毫秒总次数。

> **说明** polar_stat_io_latency视图依赖于 `polar_monitor` 插件。其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 命令使其生效。

- **网络**

  您可以通过查询本文描述的视图来获取网络监控信息。

- polar_proc_stat_network()函数

参数	类型	描述
pid	bigint	进程pid。
send_bytes	bigint	累计发送字节数。
send_count	bigint	累计发送次数。
recv_bytes	bigint	累计接收字节数。
recv_count	bigint	累计接收次数。
sendq	bigint	socket发送队列长度。
recvq	bigint	socket接收队列长度。
cwnd	bigint	socket滑动窗口大小。
rtt	bigint	TCP估算的网络rtt时间，单位：微秒。
retrans	bigint	累计重传次数。
tcpinfo_update_time	bigint	TCP socket相关监控信息的最近更新时间。实际为秒级时间戳，原则上每秒更新一次，上述 sendq ， recvq ， cwnd ， rtt ， retrans 指标会定期更新，更新时间为 tcpinfo_update_time 。

- polar_stat_network

  描述网络监控信息汇总数据。相关参数信息如下：

参数	类型	描述
send_count	bigint	发送总次数。
send_bytes	bigint	发送总字节数。
recv_count	bigint	接收总次数。
recv_bytes	bigint	接收总字节数。
retrans	bigint	总重传次数。

> **说明** polar_stat_network视图依赖于 polar_monitor 插件。其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行create extension polar_monitor命令使其生效。

- **锁**

  您可以通过本文描述的视图来获取相关锁信息。

- **LWLock**

  polar_stat_lwlock

  latch监控信息统计。相关参数信息如下：

参数	类型	描述
tranche	smallint	lwlock ID
name	text	锁名称。
sh_acquire_count	bigint	共享获取总次数。
ex_acquire_count	bigint	排它获取总次数。
block_count	bigint	获取阻塞总次数。
lock_nums	bigint	lwlock个数。
wait_time	bigint	总等待时间。

  说明 polar_stat_lwlock视图依赖于 `polar_monitor_preload` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行create extension polar_monitor_preload命令使其生效。

- **Lock**

- pg_locks

参数	类型	描述
locktype	text	可锁对象的类型： relation， extend， page， tuple， transactionid， virtualxid， object， userlock或advisory。
database	oid	锁目标存在的数据库的OID。如果目标是共享对象则为0，如果目标是事务ID则为空。
relation	oid	作为锁目标的关系的OID。如果目标不是关系或者只是关系的一部分，则值为空。
page	integer	作为锁目标的页在关系中的页号。如果目标不是关系页或元组，则值为空。
tuple	smallint	作为锁目标的元组在页中的元组号。如果目标不是元组，则值为空。
virtualxid	text	作为锁目标的事务虚拟ID。如果目标不是虚拟事务ID，则值为空。
transactionid	xid	作为锁目标的事务ID。如果目标不是事务ID，则值为空ID。
classid	oid	包含锁目标的系统目录的OID。如果目标不是普通数据库对象，则值为空。
objid	oid	锁目标在它的系统目录中的OID。如果目标不是普通数据库对象，则为空。
objsubid	smallint	锁的目标列号（classid和objid指表本身）。如果目标是某种其他普通数据库对象，则值为0。如果目标不是普通数据库对象，则值为空。
virtualtransaction	text	保持锁或者正在等待锁的事务的虚拟ID。
pid	integer	保持锁或者正在等待锁的服务器进程的PID。如果此锁被预备事务所持有，则值为空。
mode	text	此进程已持有或者希望持有的锁模式。
granted	boolean	如果锁已授予则为真。如果锁被等待则为假。
fastpath	boolean	如果锁通过快速路径获得则为真。通过主锁表获得则为假。

- **polar_stat_lock**

  统计锁信息。相关参数信息如下：

参数	类型	描述
id	integer	主键。
lock_type	text	锁类型。
invalid	numeric	无意义。
accesssharelock	numeric	access share lock锁次数。
rowsharelock	numeric	row share lock锁次数。
rowexclusivelock	numeric	row exclusive lock锁次数。
shareupdateexclusivelock	numeric	share update exclusive lock锁次数。
sharelock	numeric	share lock锁次数。
sharerowexclusivelock	numeric	share row exclusive lock锁次数。
exclusivelock	numeric	exclusive lock锁次数。
accessexclusivelock	numeric	access exclusive lock锁次数。
block_count	numeric	锁阻塞次数。
fastpath_count	numeric	本地快速锁次数。
wait_time	numeric	锁等待时间。

  > **说明** polar_stat_lock视图依赖于 `polar_monitor_preload` 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor_preload` 命令使其生效。

- **Slru**

  您可以通过查询 polar_stat_slru() 视图来获取相关slru监控信息。

  polar_stat_slru()

  统计slru监控信息。相关参数信息如下：

参数	类型	描述
slru_type	text	主键

参数	类型	描述
slots_number	integer	page数量。
valid_pages	integer	使用page数。
empty_pages	integer	空page数。
reading_pages	integer	正在读的page数。
writing_pages	integer	正在写的page数。
wait_readings	integer	等待读数量。
wait_writings	integer	等待写数量。
read_count	bigint	read次数。
read_only_count	bigint	read only次数。
read_upgrade_count	bigint	升级读次数。
victim_count	bigint	直接淘汰次数。
victim_write_count	bigint	淘汰落盘次数。
write_count	bigint	写次数。
zero_count	bigint	清零次数。
flush_count	bigint	flush次数。
truncate_count	bigint	truncate次数。
storage_read_count	bigint	读存储次数。
storage_write_count	bigint	写存储次数。

说明 polar_stat_slru视图依赖于 `polar_monitor` 插件。其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 `create extension polar_monitor` 命令使其生效。

- **cgroup**

  您可以通过查询本文描述的视图来获取相关系统资源信息。

- polar_stat_cgroup

  统计各类系统资源、cgroup信息。相关参数说明如下表所示：

参数	类型	描述
subtype	text	cgroup类型（IO、Memory、CPU）。
infotype	text	cgroup信息。
count	bigint	计数。

> **说明** polar_stat_cgroup视图依赖于 polar_monitor_preload 和 polar_monitor 插件，其中 polar_monitor_preload 和 polar_monitor 插件在安装PolarDB O引擎数据库后已默认存在，但需分别执行create extension polar_monitor_preload和create extension polar_monitor命令使其生效。

- polar_cgroup_quota

  统计各类系统资源、cgroup配额信息。相关参数说明如下表所示：

参数	类型	描述
subtype	text	cgroup类型（IO、Memory、CPU）。
infotype	text	cgroup信息。
count	bigint	限额带下。

> **说明** polar_cgroup_quota视图依赖于 polar_monitor 插件，其中该插件在安装PolarDB O引擎数据库后已默认存在，但需执行 create extension polar_monitor 命令使其生效。

2.2.2. SQL

本章节主要介绍PolarDB O引擎中的不同视图详细信息，可通过查询不同视图获取相关SQL信息，进而方便定位性能问题。

SQL

您可以通过查询本文描述的视图来获取相关SQL信息。

- pg_stat_statements

> **说明** pg_stat_statements视图依赖于 pg_stat_statements 插件，需要执行create extension pg_stat_statements命令创建该插件。

相关参数说明如下表所示：

参数	类型	描述
userid	oid	执行该语句的用户的 OID。
dbid	oid	在其中执行该语句的数据库的 OID。
queryid	bigint	内部哈希码，从语句的解析树计算得来。
query	text	语句的文本形式。
calls	bigint	被执行的次数。
total_time	double precision	在该语句中花费的总时间。单位：毫秒。
min_time	double precision	在该语句中花费的最小时间。单位：毫秒。
max_time	double precision	在该语句中花费的最大时间。单位：毫秒。
mean_time	double precision	在该语句中花费的平均时间。单位：毫秒。
stddev_time	double precision	在该语句中花费时间的总体标准偏差。单位：毫秒。
rows	bigint	该语句检索或影响的行总数。
shared_blks_hit	bigint	该语句造成的共享块缓冲命中总数。
shared_blks_read	bigint	该语句读取的共享块的总数。
shared_blks_dirtied	bigint	该语句弄脏的共享块的总数。
shared_blks_written	bigint	该语句写入的共享块的总数。
local_blks_hit	bigint	该语句造成的本地块缓冲命中总数。
local_blks_read	bigint	该语句读取的本地块的总数。
local_blks_dirtied	bigint	该语句弄脏的本地块的总数。
local_blks_written	bigint	该语句写入的本地块的总数。
temp_blks_read	bigint	该语句读取的临时块的总数。
temp_blks_written	bigint	该语句写入的临时块的总数。

参数	类型	描述
blk_read_time	double precision	该语句花在读取块上的总时间。单位：毫秒。（如果track_io_timing被启用，否则为零）。
blk_write_time	double precision	该语句花在写入块上的总时间。单位：毫秒。（如果track_io_timing被启用，否则为零）。

- **polar_stat_sql**

> **说明** polar_stat_sql视图依赖于 `polar_stat_sql` 插件，需要执行create extension polar_stat_sql命令创建该插件，也可作为 `pg_stat_statements` 的扩展。

相关参数说明如下表所示：

参数	类型	描述
queryid	bigint	查询ID
datname	name	数据库名称。
rolname	name	用户名。
user_time	double	用户态时间。
system_time	double	系统态时间。
minflts	bigint	页面回收数（软页面错误）。
majflts	bigint	缺页错误数（硬缺页错误）。
nswaps	bigint	交换页面数（swaps）。
reads	bigint	磁盘读取字节数（bytes）。
reads_blks	bigint	磁盘读取块数（blocks）。
writes	bigint	磁盘写入字节数（bytes）。
writes_blks	bigint	磁盘写入块数（blocks）。
msgsnds	bigint	IPC 信息发送次数。
msgrcvs	bigint	IPC 信息接受次数。
nsignals	bigint	信号量接受次数。
nvcsws	bigint	自愿上下文切换次数。
nivcsws	bigint	非自愿上下文切换次数。
scan_rows	double	扫描节点读取行数。

参数	类型	描述
scan_time	double	扫描节点执行时间。
scan_count	bigint	扫描节点执行次数。
join_rows	double	join节点读取行数。
join_time	double	join节点执行时间。
join_count	bigint	join节点次数时间。
sort_rows	double	排序节点读取行数。
sort_time	double	排序节点执行时间。
sort_count	bigint	排序节点执行次数。
group_rows	double	group节点读取行数。
group_time	double	group节点执行时间。
group_count	bigint	group节点执行次数。
hash_rows	double	hash节点读取行数。
hash_memory	bigint	hash节点使用内存（bytes）。
hash_count	bigint	hash节点执行次数。
parse_time	double	parse时间。
analyze_time	double	analyze时间。
rewrite_time	double	rewrite时间。
plan_time	double	执行计划生成时间。
execute_time	double	执行时间。
lwlock_wait	double	lwlock等待时间。
rel_lock_wait	double	表锁等待时间。
xact_lock_wait	double	事务锁等待时间。
page_lock_wait	double	page锁等待时间。
tuple_lock_wait	double	行锁等待时间。
shared_read_ps	bigint	读IOPS。
shared_write_ps	bigint	写IOPS。

参数	类型	描述
shared_read_throughput	bigint	读吞吐，单位：bytes。
shared_write_throughput	bigint	写吞吐，单位：bytes。
shared_read_latency	double	读延时，单位：微秒。
shared_write_latency	double	写延时，单位：微秒。
io_open_num	bigint	open文件操作次数。
io_seek_count	bigint	seek文件操作次数。
io_open_time	double	open文件操作耗时，单位：微秒。
io_seek_time	double	seek文件操作耗时，单位：微秒。

- **polar_stat_query_count**

  > **说明** polar_stat_query_count视图依赖于 `polar_stat_sql` 插件，需要执行create extension polar_stat_sql命令创建该插件。

  相关参数说明如下表所示：

参数	类型	描述
`sqltype`	text	SQL类型： DQL DMLDDL DCL
`cmdtype`	text	SQL命令类型： SELECTINSERT UPDATE ...
`count`	bigint	执行总次数。

2.2.3. 对象

本章节主要介绍PolarDB O引擎中的不同视图详细信息，可通过查询不同视图获取相关对象信息，进而方便定位性能问题。

表

- **pg_stat_all_tables**

  pg_stat_all_tables视图将为当前数据库中的每一个表显示与该表访问相关的统计信息。相关参数说明如下表所示：

参数	类型	描述
`relid`	oid	表OID。
`schemaname`	name	表所在的Schema的名称。

参数	类型	描述
relname	name	表名称。
seq_scan	bigint	在表上发起的顺序扫描的次数。
seq_tup_read	bigint	被顺序扫描取得的活着的行的数量。
idx_scan	bigint	在表上发起的索引扫描的次数。
idx_tup_fetch	bigint	被索引扫描取得的活着的行的数量。
n_tup_ins	bigint	被插入的行数。
n_tup_upd	bigint	被更新的行数（包括 HOT 更新的行）。
n_tup_del	bigint	被删除的行数。
n_tup_hot_upd	bigint	被更新的 HOT 行数（即不要求独立索引更新的行更新）。
n_live_tup	bigint	活着的行的估计数量。
n_dead_tup	bigint	死亡行的估计数量。
n_mod_since_analyze	bigint	从表最近一次被分析后被修改的行的估计数量。
last_vacuum	timestamp with time zone	最近一次表被手动清理的时间（不统计VACUUM FULL）。
last_autovacuum	timestamp with time zone	最近一次表被自动清理守护进程清理的时间。
last_analyze	timestamp with time zone	最近一次表被手动分析的时间。
last_autoanalyze	timestamp with time zone	最近一次表被自动清理守护进程分析的时间。
vacuum_count	bigint	表被手工清理的次数（不统计VACUUM FULL）。
autovacuum_count	bigint	表被自动清理守护进程清理的次数。
analyze_count	bigint	表被手工分析的次数。
autoanalyze_count	bigint	表被自动清理守护进程分析的次数。

- pg_statio_all_tables

  pg_statio_all_tables视图将为当前数据库中的每个表指定有关 I/O 的统计信息。相关参数说明如下表所示：

参数	类型	描述
relid	oid	表的 OID。
schemaname	name	表所在的Schema的名称。
relname	name	表的名称。
heap_blks_read	bigint	从表读取的磁盘块数量。
heap_blks_hit	bigint	在表中的缓冲区命中数量。
idx_blks_read	bigint	从表上所有索引中读取的磁盘块数。
idx_blks_hit	bigint	在表上的所有索引中的缓冲区命中数量。
toast_blks_read	bigint	从表的 TOAST 表读取的磁盘块数。
toast_blks_hit	bigint	在表的 TOAST 表中的缓冲区命中数量。
tidx_blks_read	bigint	从表的 TOAST 表索引中读取的磁盘块数。
tidx_blks_hit	bigint	在表的 TOAST 表索引中的缓冲区命中数量。

索引

- **pg_stat_all_indexes**

  通过查询**pg_stat_all_indexes**视图可以显示当前数据库中的每个索引访问的统计信息。

  索引可以被简单扫描、位图扫描以及通过优化器来访问。在一次位图扫描中，多个索引的输出可以通过 AND 或 OR 规则组合，当使用一次位图扫描时不能将取得的个体与特定的索引关联起来。因此，一次位图扫描会增加它使用的索引的**pg_stat_all_indexes.idx_tup_read**数量，并且为每个表增加**pg_stat_all_tables.idx_tup_fetch**数量，但是它不影响**pg_stat_all_indexes.idx_tup_fetch**。如果所提供的常量值不在优化器统计信息记录的范围之内，优化器也会访问索引来检查。

参数	类型	描述
relid	oid	索引基表OID。
indexrelid	oid	索引OID。
schemaname	name	索引所在的Schema的名称。
relname	name	索引基表的名称。
indexrelname	name	索引名称。
idx_scan	bigint	在索引上发起的索引扫描次数。
idx_tup_read	bigint	在索引上由扫描返回的索引项数量。
idx_tup_fetch	bigint	被使用索引的简单索引扫描取得的活着的表行数量。

- pg_statio_all_indexes

  显示当前数据库中每个索引的磁盘块读取总数和缓冲区命中总数，以及指定索引上与 I/O 有关的统计信息。相关参数说明如下：

参数	类型	描述
relid	oid	索引基表的 OID。
indexrelid	oid	索引OID。
schemaname	name	索引所在Schema的名称。
relname	name	索引基表的名称。
indexrelname	name	索引名称。
idx_blks_read	bigint	从索引读取的磁盘块数。
idx_blks_hit	bigint	在索引中的缓冲区命中数量。

序列

pg_statio_all_sequences

显示当前数据库相关序列信息，以及指定序列上与 I/O 有关的统计信息。相关参数说明如下：

参数	类型	描述
relid	oid	序列OID。
schemaname	name	序列所在的Schema的名称。
relname	name	序列名称。
blks_read	bigint	从序列中读取的磁盘块数。
blks_hit	bigint	在序列中的缓冲区命中数量。

函数

pg_stat_user_functions

显示被跟踪函数有关的统计信息。

参数	类型	描述
funcid	oid	函数OID。
schemaname	name	函数所在的Schema名称。
funcname	name	函数名称。
calls	bigint	函数被调用的次数。

参数	类型	描述
total_time	double precision	在函数以及它所调用的其他函数中花费的总时间。单位：毫秒。
self_time	double precision	在函数本身花费的总时间，不包括被它调用的其他函数。单位：毫秒。

2.3. 使用AWR

自动负载信息库AWR（Automatic Workload Repository）是一种性能收集和分析的工具。通过AWR工具，您可以从数据库的动态视图等统计信息中生成某个时间段的统计分析报告以及数据库性能报告。

PolarDB除了提供查看视图中的实时视图和函数外，还提供对这些实时视图采集后保存的历史数据，通过这些数据可以生成类似Oracle的AWR报告，关于历史数据及AWR报告的详细说明请参见AWR使用说明。

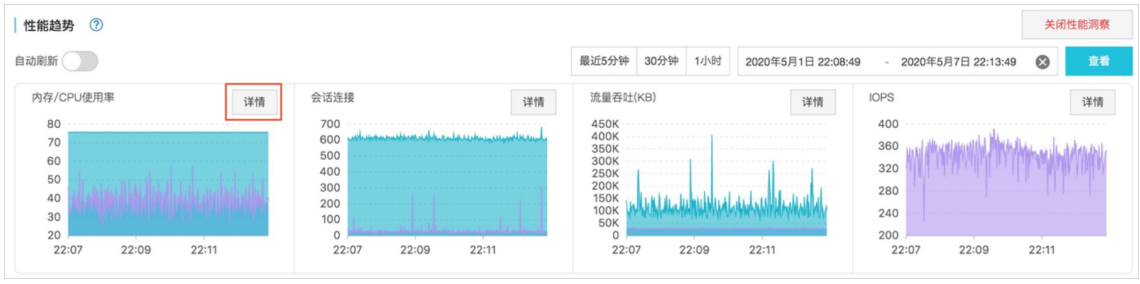
2.4. 使用性能洞察

性能洞察是一项专注于数据库性能调优、负载监控和关联分析的利器，使用直观简单的方式帮助您迅速评估数据库负载，找到性能问题的源头和对应的SQL语句，以此来指导您在何时、何处、采取何种行动进行数据性能优化。

操作步骤

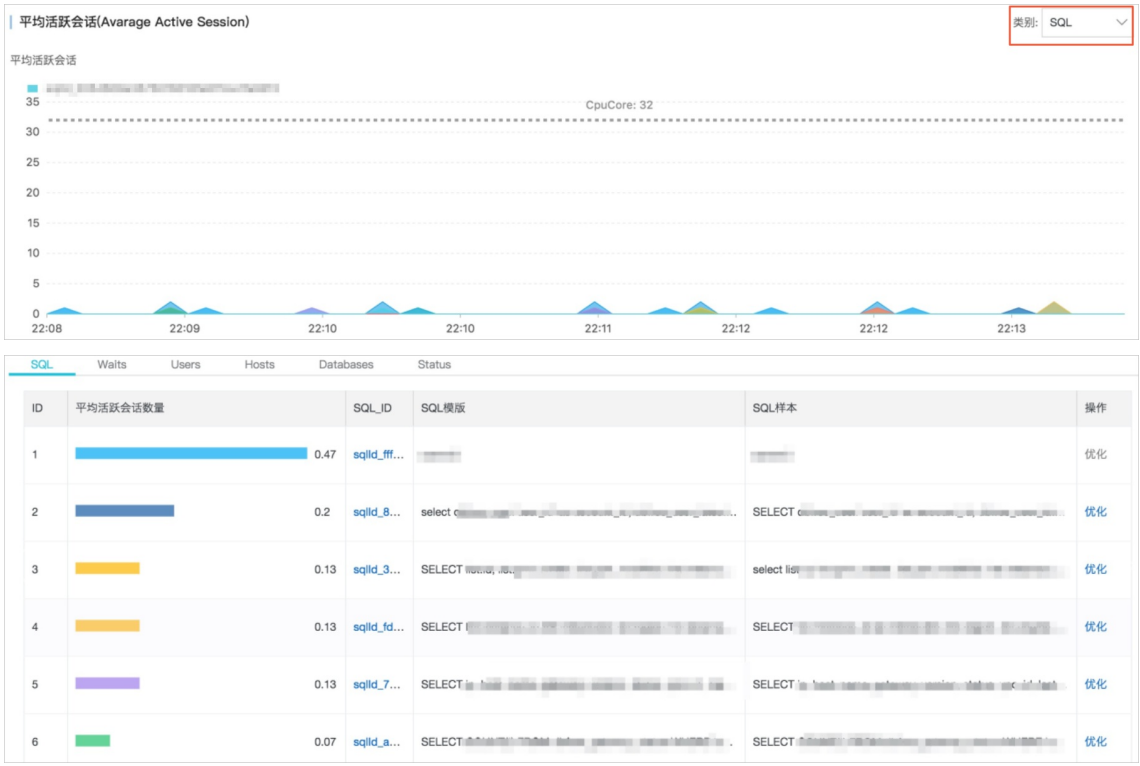
1. 登录PolarDB控制台。
2. 在控制台左上角，选择集群所在地域。
3. 在**集群列表**页，单击目标集群ID。
4. 在左侧导航栏中，选择**诊断与优化 > 一键诊断**。
5. 单击**性能洞察**页签。
6. 单击**开启性能洞察**。

7. 在弹出的对话框内，单击**确定**。
8. 您可以在性能洞察页面查看和管理如下信息：
   - 在**性能趋势**区域，您可以查看特定时段的数据库性能情况。若您需要查看某个具体性能（如CPU使用率），可以单击该性能名称右侧的**详情**按钮进行查看。



> **说明** 可供查看的时间范围不能超过7天。

- 在**平均活跃会话**区域，您可以查看不同类别（如SQL）会话的变化趋势图和相关多维负载信息列表，确定性能问题源头。



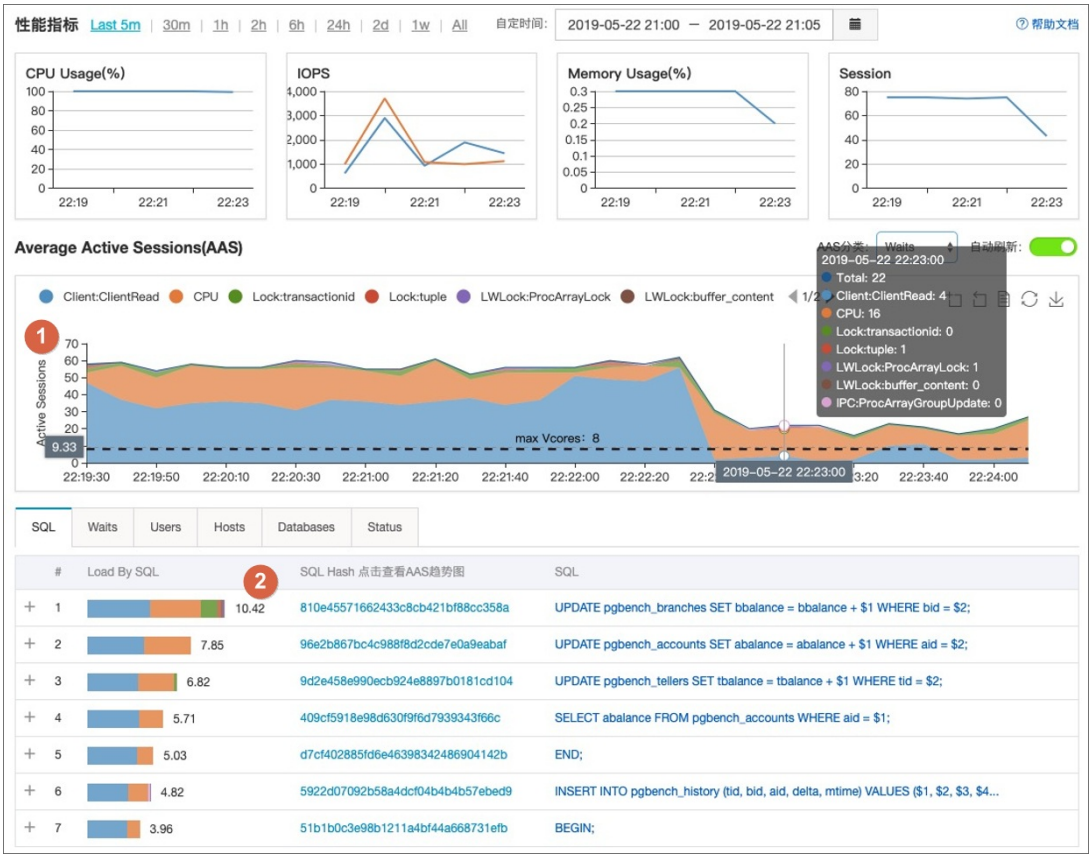
数据指标

性能洞察将pg_stat_activity视图作为数据源，采样后可以获取到以下信息：

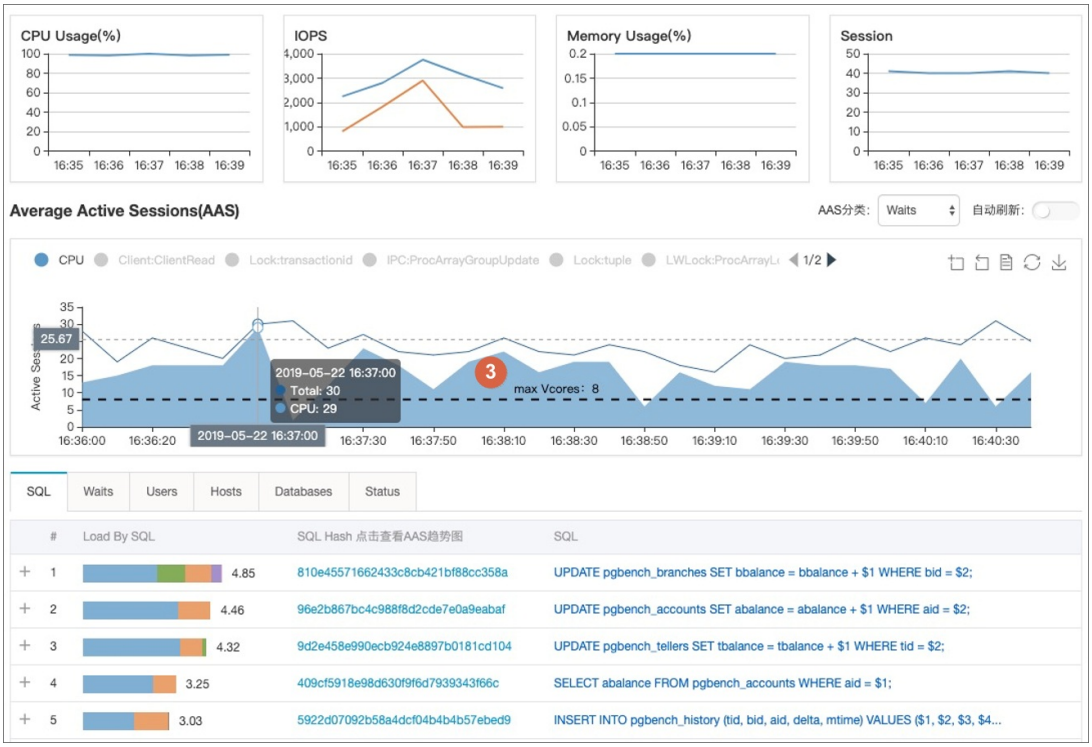
- 用户、等待事件
- SQL、 HOST
- 数据库、连接状态

以下图片和表格展示了PolarDB性能洞察页面以及部分参数的介绍。

性能洞察页面示例图一



性能洞察页面示例图二



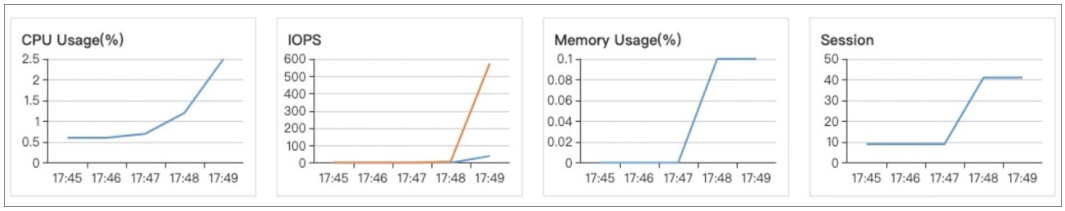
序号	参数	说明
①	AS：active session	当前活跃的会话个数。

序号	参数	说明
②	AAS：average active session	一段时间内，AS的平均值。
③	Max Vcores：8	• 当前实例的CPU个数。 • CPU事件超出该值可以认为存在性能瓶颈。

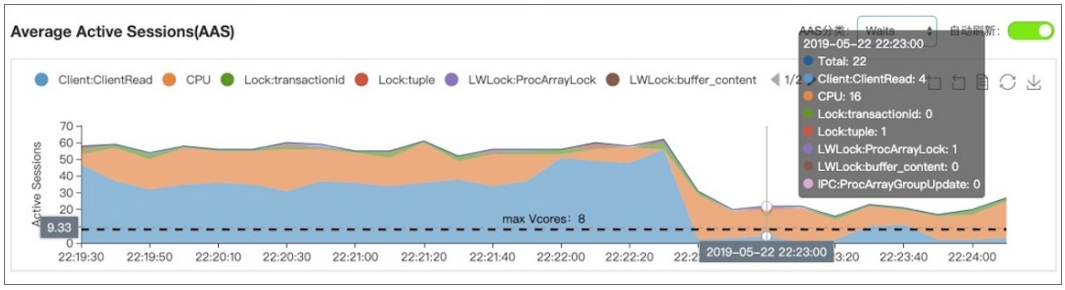
性能图表

性能洞察页面由以下三个部分组成：

- 关键的性能指标变化趋势图（实时）



- 实时AAS变化趋势图



上图为数据库实例中活跃会话变化趋势。

- 多维度实例负载信息（AAS）

上图为聚合30次采集结果。

性能洞察与常规排查过程的对比

问题	常规排查过程	使用性能洞察排查
在某个时间点原本时间很短UPDATE SQL变慢或审计日志中发现大量UPDATE。	• 排查性能监控 • 排查慢SQL日志 • 排查海量审计日志	• 选择问题出现时间范围 • 选择不同纬度反映问题的不同侧面 • 定位问题SQL • 定位SQL状态

问题	常规排查过程	使用性能洞察排查
同SQL在不用的客户端执行时间差距很大且客户端在不同机房。	• 排查性能监控 • 网络抓包分析	• 选择问题出现时间范围 • 主要观察host维度 • 综合多维度信息定位问题

3.优化集群性能

3.1. 参数说明

当您通过数据库统计信息、慢SQL等信息了解数据库当前的状态以及存在的问题后，可以针对发现的问题，进行调整和优化。PolarDB O引擎默认的参数模板适用于大多数通用的数据库场景，对于一些特殊的数据库场景，默认模板可能无法满足需求，您可以通过调整相关的数据库参数，对集群性能进行优化。

资源消耗参数

参数名	参数说明
shared_buffers	数据库使用的共享内存大小，越大的缓存区可以缓存的数据更多，PolarDB中该值与规格相关。
work_mem	指定在写到临时磁盘文件之前被内部排序操作和哈希表使用的内存量。注意对于一个复杂查询， 可能会并行运行好几个排序或者哈希操作；每个操作都会被允许使用这个参数指定的内存量，然后才会开始写数据到临时文件。同样，几个正在运行的会话可能并发进行这样的操作。因此被使用的总内存可能是work_mem值的好几倍，在选择这个值时一定要记住这一点。ORDER BY、DISTINCT和归并连接都要用到排序操作。哈希连接、基于哈希的聚集以及基于哈希的IN子查询处理中都要用到哈希表。
maintenance_work_mem	指定在维护性操作（例如VACUUM、CREATE INDEX和ALTER TABLE ADD FOREIGN KEY）中使用的最大的内存量。因为在一个数据库会话中，一个时刻只有一个这样的操作可以被执行，并且一个数据库安装通常不会有太多这样的操作并发执行， 把这个数值设置得比work_mem大很多是安全的。 更大的设置可以改进清理和恢复数据库转储的性能。 注意当自动清理运行时，可能会分配最多达这个内存的autovacuum_max_workers倍，因此要小心不要把该默认值设置得太高。 通过独立地设置autovacuum_work_mem可能会对控制这种情况 有所帮助。
autovacuum_work_mem	指定每个自动清理工作者进程能使用的最大内存量。其默认值为 -1，表示使用maintenace_work_mem的值。
temp_file_limit	指定一个进程能用于临时文件（如排序和哈希临时文件，或者用于保持游标的存储文件）的最大磁盘空间量。一个试图超过这个限制的事务将被取消。这个值以千字节计，-1意味着没有限制。
max_worker_processes	设置系统能够支持的后台进程的最大数量。这个参数只能在服务器启动时设置。
max_parallel_workers	设置系统为并行操作所支持的工作者的最大数量。要注意将这个值设置得大于max_worder_processes将不会产生效果。注意并行查询可能消耗比非并行查询更多的资源，因为每一个工作者进程时一个完全独立的进程，它对系统产生的影响大致和一个额外的用户会话相同。
max_parallel_workers_per_gather	设置单个Gather或者Gather Merge节点能够开始的工作者的最大数量。并行工作者会从max_worker_processes建立的进程池获取，数量由max_parallel_workers限制。
max_parallel_maintenance_workers	设置CREATE INDEX并行工作的最大数量。

autovacuum相关参数

参数	说明
autovacuum_max_workers	指定能同时运行的自动清理进程（除了自动清理启动器之外）的最大数量。默认值为3。该参数只能在服务器启动时设置。
autovacuum_naptime	指定自动清理在任意给定数据库上运行的最小延迟。在每一轮中后台进程检查数据库并根据需要为数据库中的表发出VACUUM和ANALYZE命令。延迟以秒计，且默认值为1分钟（1min）。
autovacuum_vacuum_threshold	指定能在一个表上触发VACUUM的被插入、被更新或被删除元组的最小数量。默认值为50个元组。可以通过修改表存储参数来覆盖该设置。
autovacuum_analyze_threshold	指定能在一个表上触发ANALYZE的被插入、被更新或被删除元组的最小数量。默认值为50个元组。可以通过修改表存储参数来覆盖该设置。
autovacuum_vacuum_scale_factor	指定一个表尺寸的分数，在决定是否触发VACUUM时将它加到autovacuum_vacuum_threshold上。默认值为0.05（表尺寸的5%）。可以通过修改表存储参数来覆盖该设置。
autovacuum_analyze_scale_factor	指定一个表尺寸的分数，在决定是否触发ANALYZE时将它加到autovacuum_analyze_threshold上。默认值为0.1（表尺寸的10%）。可以通过修改表存储参数来覆盖该设置。
autovacuum_vacuum_cost_delay	指定用于自动VACUUM操作中的代价延迟值。如果指定-1（默认值），则使用vacuum_code_delay值。可以通过修改表存储参数来覆盖该设置。
autovacuum_vacuum_cost_limit	指定用于自动VACUUM操作中的代价限制值。如果指定-1（默认值），则使用vacuum_cost_limit值。注意该值被按比例地分配到运行中的自动清理工作者上（如果有多个），因此每一个工作者的限制值之和不会超过这个变量中的值。可以通过修改表存储参数来覆盖该设置。

查询规划参数

以下参数不建议全局修改，当您在测试特定的查询时，在当前会话中修改。

参数	参数说明
enable_bitmapscan	允许或禁止查询规划器使用位图扫描计划类型。
enable_hashagg	允许或禁用查询规划器使用哈希聚集计划类型。
enable_hashjoin	允许或禁止查询规划器使用哈希连接计划类型。
enable_indexscan	允许或禁止查询规划器使用索引扫描计划类型。
enable_indexonlyscan	允许或禁止查询规划器使用只用索引扫描计划类型。
enable_seqscan	允许或禁止查询规划器使用顺序扫描计划类型。它不可能完全禁止顺序扫描，但是关闭这个变量将使得规划器尽可能优先使用其他方法。
enable_sort	允许或禁止查询规划器使用显式排序步骤。它不可能完全禁止显式排序，但是关闭这个变量将使得规划器尽可能优先使用其他方法。

参数	参数说明
enable_mergejoin	允许或禁止查询规划器使用归并连接计划类型。
enable_nestloop	允许或禁止查询规划器使用嵌套循环连接计划。它不可能完全禁止嵌套循环连接，但是关闭这个变量将使得规划器尽可能优先使用其他方法。
enable_parallel_append	允许或禁止查询规划器使用并行追加计划类型。
enable_parallel_hash	允许或禁止查询规划器对并行哈希使用哈希连接计划类型。如果哈希连接计划也没有启用，这个参数没有效果。

规划器代价参数

参数	参数说明
seq_page_cost	设置规划器计算一次顺序磁盘页面抓取的开销。默认值是1.0。
random_page_cost	设置规划器对一次非顺序获取磁盘页面的代价估计。默认值是 4.0。
cpu_tuple_cost	设置规划器对一次查询中处理每一行的代价估计。默认值是 0.01。
cpu_index_tuple_cost	设置规划器对一次索引扫描中处理每一个索引项的代价估计。默认值是 0.005。
cpu_operator_cost	设置规划器对于一次查询中处理每个操作符或函数的代价估计。默认值是 0.0025
parallel_setup_cost	设置规划器对启动并行工作者进程的代价估计。默认是 1000。
parallel_tuple_cost	设置规划器对于从一个并行工作者进程传递一个元组给另一个进程的代价估计。默认是0.1。
min_parallel_table_scan_size	为必须扫描的表数据量设置一个最小值，扫描的表数据量超过这一个值才会考虑使用并行扫描。对于并行顺序扫描，被扫描的表数据量总是等于表的尺寸，但是在使用索引时，被扫描的表数据量通常会更小。默认值是8MB。
min_parallel_index_scan_size	为必须扫描的索引数据量设置一个最小值，扫描的索引数据量超过这一个值时才会考虑使用并行扫描。注意并行索引扫描通常并不会触及整个索引，它是规划器认为该扫描会实际用到的相关页面的数量。默认值是512KB。

PolarDB调整参数

pg_setting详解

列名	值	解释
name	vacuum_cost_delay	参数名称
setting	10	参数当前值
unit	ms	参数的单位
category	Resource Usage / Cost-Based Vacuum Delay	参数类别

列名	值	解释
short_desc	Vacuum cost delay in milliseconds.	参数描述
extra_desc	无	附加详细描述
context	user	设置参数值的上下文
vartype	integer	参数类型（bool、enum、integer、real、string）
source	database	参数值来源
min_val	0	参数允许最小值
max_val	100	参数允许最大值
enumvals	无	枚举参数的允许值
boot_val	0	没有设置时，启动时的设定值
reset_val	10	会话中reset时设定的值
sourcefile	无	当前值设置的配置文件
sourceline	无	当前设置的配置文件的行号
pending_restart	f	修改该参数需要重启为true，否则为false

PolarDB资源消耗参数

参数名	参数说明
shared_buffers	数据库使用的共享内存大小，越大的缓存区，缓存的数据更多，PolarDB中该值与规格相关。
work_mem	指定在写到临时磁盘文件之前被内部排序操作和哈希表使用的内存量注意对于一个复杂查询， 可能会并行运行好几个排序或者哈希操作；每个操作都会被允许使用这个参数指定的内存量，然后才会开始写数据到临时文件。同样，几个正在运行的会话可能并发进行这样的操作。因此被使用的总内存可能是work_mem值的好几倍，在选择这个值时一定要记住这一点。ORDER BY、DISTINCT和归并连接都要用到排序操作。哈希连接、基于哈希的聚集以及基于哈希的IN子查询处理中都要用到哈希表。
maintenance_work_mem	指定在维护性操作（例如VACUUM、CREATE INDEX和ALTER TABLE ADD FOREIGN KEY）中使用的最大的内存量。因为在一个数据库会话中，一个时刻只有一个这样的操作可以被执行，并且一个数据库安装通常不会有太多这样的操作并发执行， 把这个数值设置得比work_mem大很多是安全的。 更大的设置可以改进清理和恢复数据库转储的性能。 注意当自动清理运行时，可能会分配最多达这个内存的autovacuum_max_workers倍，因此要小心不要把该默认值设置得太高。 通过独立地设置autovacuum_work_mem可能会对控制这种情况有所帮助。

参数名	参数说明
autovacuum_work_mem	指定每个自动清理工作者进程能使用的最大内存量。其默认值为 -1，表示使用maintenace_work_mem的值。
temp_file_limit	指定一个进程能用于临时文件（如排序和哈希临时文件，或者用于保持游标的存储文件）的最大磁盘空间量。一个试图超过这个限制的事务将被取消。这个值以千字节计，-1意味着没有限制。

3.2. 设置参数

本文将为您介绍如何调整PolarDB参数，您可以在PolarDB控制台修改参数，也可以通过客户端修改参数。

控制台修改参数

1. 登录PolarDB控制台。
2. 在控制台左上角，选择集群所在地域。
3. 找到目标集群，单击集群ID。
4. 在左侧导航栏中选择**配置与管理 > 参数配置**。
5. 修改一个或多个参数的**当前值**，单击**提交修改**。

6. 在弹出的**保存改动**d对话框中，单击**确定**。

更多信息请参见设置集群参数。

客户端修改参数

客户端支持以下几种修改方式：

命令	说明
`alter database`	可以修改数据库级别的参数，可以使不同数据库有不同的参数配置。需要会话重连才能生效。
`alter role`	可以修改用户级别的参数，可以使不用用户拥有不同参数，用户级别会覆盖数据库级别参数。需要会话重连才能生效。

命令	说明
`set`	可以修改会话级别参数，只对当前会话生效，如果会话断开，修改参数失效。

示例如下：

- `alter database`

```
test1=> alter database test1 set vacuum_cost_delay =10;
ALTER DATABASE
```

- `alter role`

```
test1=> alter role test1 pg_db_role_setting;
 setdatabase | setrole |       setcofig
 ------------+---------+-----------------------
       41891 |       0 | {vacuum_cost_delay=10}
       0 |     41284 | {random_page_cost=10}
(2 rows)
```

- `set`

```
test1=> set random_page_cost=100;
SET
test1=> show random_page_cost;
 random_page_cost
------------------
 100
(1 row)
```

4.优化SQL语句

4.1. 使用索引

4.1.1. B-TREE索引

B-TREE索引是最常用的索引，适合等值查询、范围查询、索引排序、多列条件、条件包含表达式等等场景。

操作符

操作符	示例
<	select * from test where id <1
<=	select * from test where id <=1
=	select * from test where id =1
>=	select * from test where id >=1
>	select * from test where id >1
between and	select * from test where id between 1 and 10
in	select * from test where id in (1,2,3)
like	select * from test where id like ‘abc%’

多列索引

多列索引用于定义在表的多个列上的索引，最多可以指定32个列。

- 表结构

```
create table test(id int,name varchar(10));
```

- 查询语句

```
select * from test where id=1 and name='a1’;
```

- 创建多列索引

```
create index ON test(id,name);
```

- 查看执行计划

```
postgres=# explain select * from test where id=1 and name='a1';
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Only Scan using a_id_name_idx on test  (cost=0.42..8.44 rows=1 width=10)
   Index Cond: ((id = 1) AND (name = 'a1'::text))
(2 rows)
```

以上示例中在条件没有包含ID的情况下不会走索引，示例如下。

```
postgres=# explain select * from test where name='a1';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test  (cost=0.00..1791.00 rows=1 width=10)
   Filter: ((name)::text = 'a1'::text)
(2 rows)
```

表达式索引

表达式索引用于索引的列不是物理表的一个列，是对表的一个列或者多列进行计算的函数或者表达式。

- 表结构

```
create table test(id int,name varchar(10));
```

- 查询语句

```
select * from test where lower(name)='a1’;
```

- 创建表达式索引

```
create index ON test (lower(name));
```

- 查看执行计划

```
postgres=# explain select * from test where lower(name)='a1';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=12.17..571.91 rows=500 width=10)
   Recheck Cond: (lower((name)::text) = 'a1'::text)
   ->  Bitmap Index Scan on test_lower_idx  (cost=0.00..12.04 rows=500 width=0)
         Index Cond: (lower((name)::text) = 'a1'::text)
(4 rows)
```

索引表达式的维护代价较为昂贵，在每一行被插入或更新时都得为它重新计算相应的表达式。

部分索引

当一个部分索引是建立在表的一个子集上，而该子集由一个条件表达式定义，索引中只包含符合谓词的表行的项，则可以使用部分索引。

- 表结构

```
create table test(id int,name varchar(10));
```

- 查询语句

```
select * from test where name='a1';
select * from test where name='a2';
```

- 创建部分索引

```
create index ON test(name) where name='a1';
```

- 执行计划

```
postgres=# explain select * from test where name='a1';
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using test_name_idx on test  (cost=0.12..8.14 rows=1 width=10)
(1 row)
postgres=# explain select * from test where name='a2';
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on test  (cost=0.00..1791.00 rows=1 width=10)
   Filter: ((name)::text = 'a2'::text)
(2 rows)
```

索引排序

索引除了简单查找返回行之外，还可以按照指定顺序返回不需要独立的排序步骤。

- 表结构

```
create table test(id int,name varchar(10));
```

- 查询语句

```
select * from test order by name desc;
```

- 创建索引前计划

```
postgres=# explain select * from test order by name desc;
                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=9845.82..10095.82 rows=100000 width=10)
   Sort Key: name DESC
   ->  Seq Scan on test  (cost=0.00..1541.00 rows=100000 width=10)
(3 rows)
```

- 创建索引

```
create index ON test (name desc);
```

- 查看执行计划

```
postgres=# explain select * from test order by name desc;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using test_name_idx on test  (cost=0.29..3666.46 rows=100000 width=10)
(1 row)
```

默认情况下，B-TREE索引将它的项以升序方式存储，并将空值放在最后。您可以在创建B-TREE索引时通过ASC、DESC、NULLS FIRST和NULLS LAST选项来改变索引的排序。

只使用索引扫描和覆盖索引

只查询索引相关字段，可以通过索引直接返回数据，无需访问具体的数据文件。

- 示例一

- 表结构

```
create table test(id int,name varchar(10));
```

- 查询语句

```
select name from test where name='a1';
```

- 没有索引时的执行计划

```
postgres=# explain select name from test where name='a1';
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on test  (cost=0.00..1791.00 rows=1 width=6)
   Filter: ((name)::text = 'a1'::text)
(3 rows)
```

- 创建索引

```
create index ON test (name);
```

- 有索引时的执行计划

```
postgres=# explain select name from test where name='a1';
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Only Scan using test_name_idx on test  (cost=0.29..8.31 rows=1 width=6)
   Index Cond: (name = 'a1'::text)
(2 rows)
```

- 示例二
  - 表结构（与示例一相同）

```
create table test(id int,name varchar(10));
```

  - 查询语句

```
select * from test where name='a1';
```

  - 没有索引时的执行计划

```
postgres=# explain select * from test where name='a1';
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on test  (cost=0.00..1791.00 rows=1 width=10)
   Filter: ((name)::text = 'a1'::text)
(2 rows)
```

  - 创建覆盖索引

```
create index ON test (name) include(id);
```

- 有索引时的执行计划

```
postgres=# explain select * from test where name='a1';
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Index Only Scan using test_name_id_idx on test  (cost=0.42..8.44 rows=1 width=10)
   Index Cond: (name = 'a1'::text)
(2 rows)
```

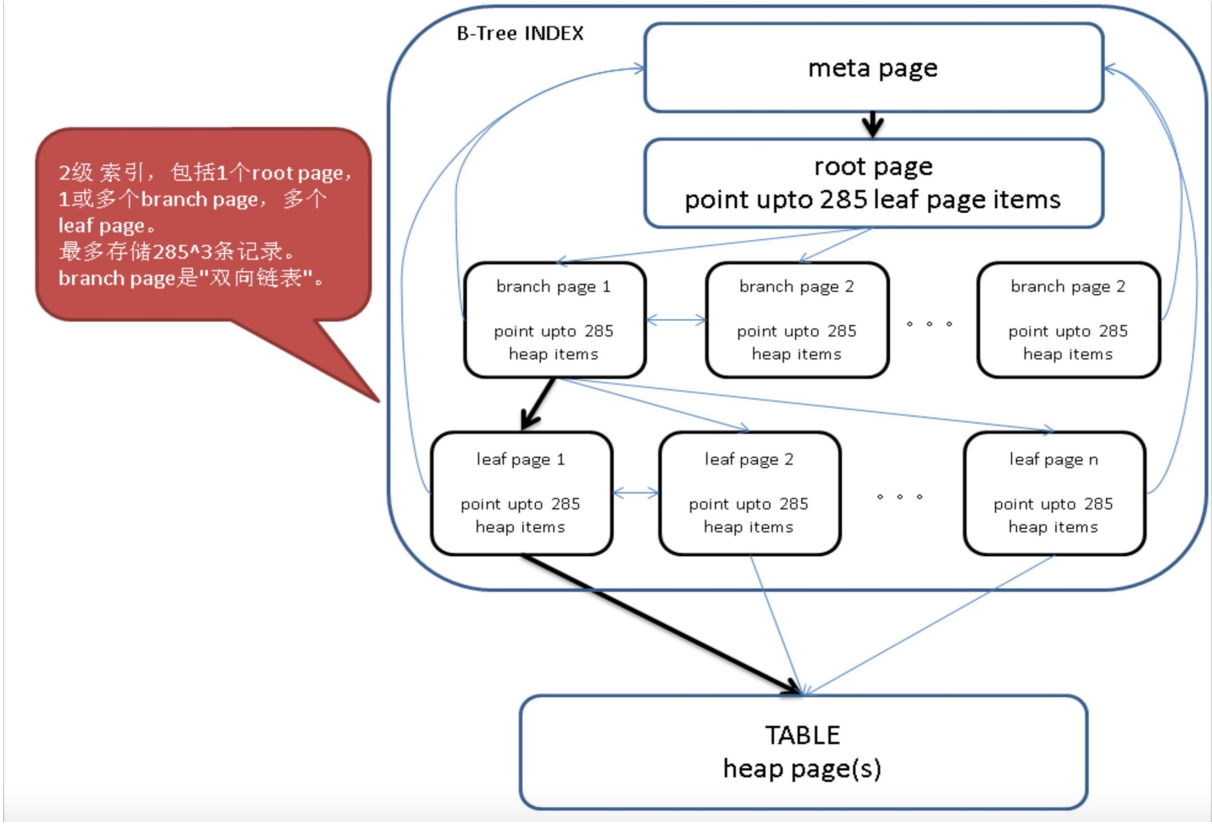
查询语句必须只引用存储在该索引中的列，才能使用覆盖索引，即只需要扫描索引，不需要去扫描表中数据就可以得到相应的结果。

索引页面类型

索引页面将简单介绍索引的内部架构，上述的索引功能都是基于内部架构实现。PolarDB的B-TREE索引页面分几个类型：

- meta page
- root page
- branch page
- leaf page

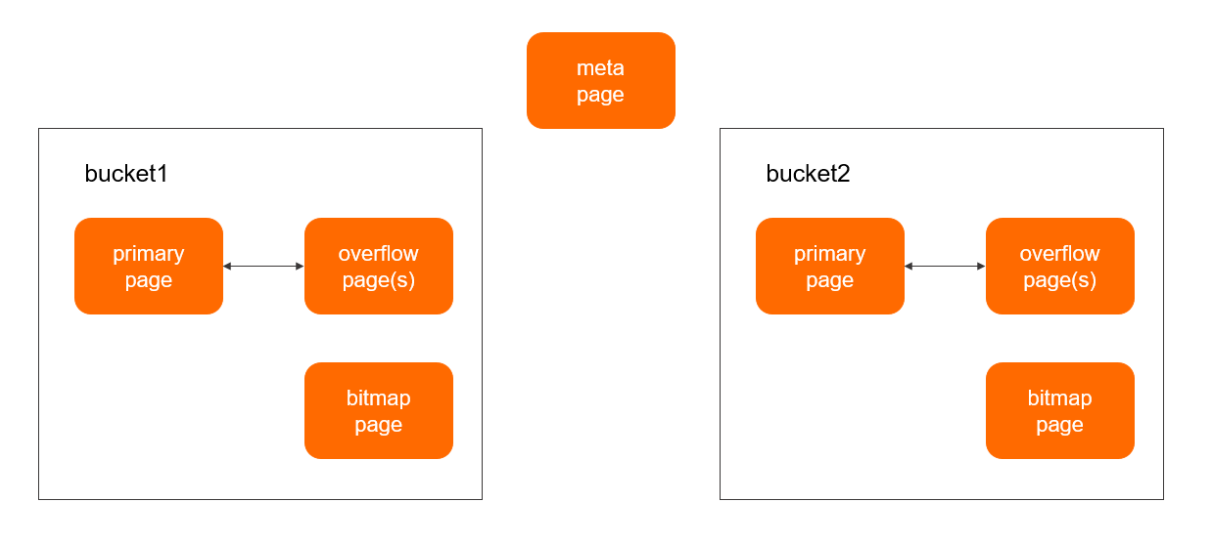
其中meta page类型和root page类型是必须有的，meta page需要一页来存储，表示指向root page的page id。随着记录数的增加，一个root page可能存不下所有的heap item，就会需要leaf page类型、branch page类型或多层的branch page类型。



4.1.2. HASH索引

HASH索引只支持等值查询。由于HASH索引只存储HASH值，不会存储实际的索引键值，所以适合字段长度较长，且字段选择性好的等值查询场景。

索引结构



- HASH值转换，HASH值映射到某个bucket。
- bucket数量为2的N次方。
- metapage包含索引内部的相关信息。
- 每个bucket内至少一个primary page。
- page中存放的是HASH值。
- overflow page不足一个page时作为bucket使用。
- bitmap page用于跟踪当前干净的overflow page。

操作符

HASH索引只支持=的操作符，意味着HASH索引只适合于等值查询的场景。

- 查询语句

```
select * from test where id=1;
```

- 创建HASH索引

```
create index ON test using hash(id);
```

- 查看执行计划

```
postgres=# explain select * from test where id=1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using test_id_idx on test  (cost=0.00..8.02 rows=1 width=10)
   Index Cond: (id = 1)
(2 rows)
```

示例

- 表结构

```
create table a(id int,name text);
```

- 插入数据

```
insert into test select id,md5(id::text)||md5(id::text)||md5(id::text)||md5(id::text) fro
m generate_series(1,3000000) t(id);
```

- 创建HASH索引

```
create index idx_test_hash ON test using hash (name);
```

- 创建BTREE索引

```
create index idx_test_btree on test(name);
```

- 查询语句

```
select * from test where name='c4ca4238a0b923820dcc509a6f75849bc4ca4238a0b923820dcc509a6f
75849bc4ca4238a0b923820dcc509a6f75849bc4ca4238a0b923820dcc509a6f75849b';
```

索引类型	索引大小	查询时间
HASH索引	224 MB	0.029 ms
BTREE索引	491 MB	0.103 ms

4.1.3. GIN索引

本文介绍PolarDB通用倒排索引GIN（Generalized Inverted Index）。

GIN是一个存储对（key、posting list）集合的索引结构，其中key是一个键值，posting list是一组出现过key的位置。如 `'hello', '14:2 23:4'` 中，表示hello在14:2和23:4这两个位置出现过，这些位置实际上就是元组的tid（行号，包括数据块ID，大小为32 bit；以及item point，大小为16 bit）。通过这种索引结构可以快速的查找到包含指定关键字的元组，因此GIN索引特别适用于多值类型的元素搜索。

应用场景

- 搜索多值类型，例如数组、全文检索

```
postgres=# create table t_gin1 (id int, arr int[]);

postgres=# do language plpgsql $$
postgres$# declare
postgres$# begin
postgres$#   for i in 1..10000 loop
postgres$#     insert into t_gin1 select i, array(select random()*1000 from generate_series(1,10));
postgres$#   end loop;
postgres$# end;
postgres$# $$;

postgres=# select * from t_gin1 limit 3;
 id |                   arr
----+-----------------------------------------
  1 | {128,700,814,592,414,838,615,827,274,210}
  2 | {284,452,824,556,132,121,21,705,537,865}
  3 | {65,185,586,872,627,330,574,227,827,64}

postgres=# create index idx_t_gin1_1 on t_gin1 using gin (arr);
CREATE INDEX

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gin1 where arr && array[1,2];
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on public.t_gin1  (cost=8.93..121.24 rows=185 width=65) (actual time=0.058..0.207 rows=186 loops=1)
   Output: id, arr
   Recheck Cond: (t_gin1.arr && '{1,2}'::integer[])
   Heap Blocks: exact=98
   Buffers: shared hit=103
   ->  Bitmap Index Scan on idx_t_gin1_1  (cost=0.00..8.89 rows=185 width=0) (actual time=0.042..0.042 rows=186 loops=1)
         Index Cond: (t_gin1.arr && '{1,2}'::integer[])
         Buffers: shared hit=5
 Planning time: 0.208 ms
 Execution time: 0.245 ms
(10 rows)

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gin1 where arr @> array[1,2];
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on public.t_gin1  (cost=7.51..9.02 rows=1 width=65) (actual time=0.022..0.022 rows=0 loops=1)
   Output: id, arr
   Recheck Cond: (t_gin1.arr @> '{1,2}'::integer[])
   Buffers: shared hit=5
   ->  Bitmap Index Scan on idx_t_gin1_1  (cost=0.00..7.51 rows=1 width=0) (actual time=0.020..0.020 rows=0 loops=1)
         Index Cond: (t_gin1.arr @> '{1,2}'::integer[])
         Buffers: shared hit=5
 Planning time: 0.116 ms
 Execution time: 0.044 ms
(9 rows)
```

- 按照任意列进行搜索

```
postgres=# create extension btree_gin;
CREATE EXTENSION
postgres=# create table t_gin2 (id int, c1 int);
CREATE TABLE
postgres=# insert into t_gin2 select generate_series(1,100000), random()*10 ;
INSERT 0 100000
postgres=# create index idx_t_gin2_1 on t_gin2 using gin (c1);
CREATE INDEX
postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gin2 where c1=1;
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on public.t_gin2  (cost=84.10..650.63 rows=9883 width=8) (actual time=0.925..3.685 rows=10078 loops=1)
   Output: id, c1
   Recheck Cond: (t_gin2.c1 = 1)
   Heap Blocks: exact=443
   Buffers: shared hit=448
   ->  Bitmap Index Scan on idx_t_gin2_1  (cost=0.00..81.62 rows=9883 width=0) (actual time=0.867..0.867 rows=10078 loops=1)
         Index Cond: (t_gin2.c1 = 1)
         Buffers: shared hit=5
 Planning time: 0.252 ms
 Execution time: 4.234 ms
(10 rows)
```

- 查找的数据比较稀疏

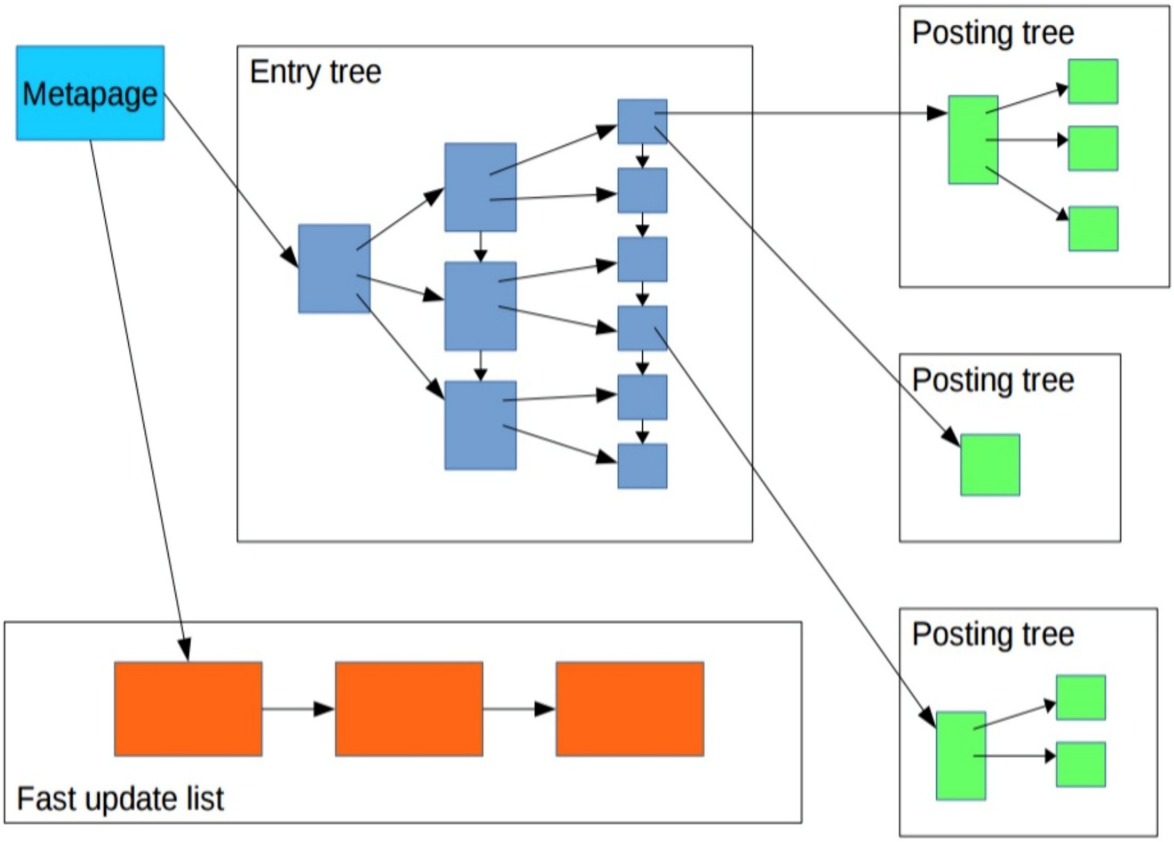
```
postgres=# create table t_gin3 (id int, c1 int, c2 int, c3 int, c4 int, c5 int, c6 int, c7 int, c8 int, c9 int);
CREATE TABLE
postgres=# insert into t_gin3 select generate_series(1,100000), random()*10, random()*20, random()*30, random()*40, random()*50, random()*60,
random()*70, random()*80, random()*90;
INSERT 0 100000
postgres=# create index idx_t_gin3_1 on t_gin3 using gin (c1,c2,c3,c4,c5,c6,c7,c8,c9);
CREATE INDEX

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gin3 where c1=1 or c2=1 and c3=1 or c4=1 and (c6=1 or c7=2) or c8=9 or
c9=10;
                                                                                         QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------
 Bitmap Heap Scan on public.t_gin3  (cost=154.03..1364.89 rows=12286 width=40) (actual time=1.931..5.634 rows=12397 loops=1)
   Output: id, c1, c2, c3, c4, c5, c6, c7, c8, c9
   Recheck Cond: ((t_gin3.c1 = 1) OR ((t_gin3.c2 = 1) AND (t_gin3.c3 = 1)) OR (((t_gin3.c4 = 1) AND (t_gin3.c6 = 1)) OR ((t_gin3.c4 = 1) AND (
t_gin3.c7 = 2))) OR (t_gin3.c8 = 9) OR (t_gin3.c9 = 10))
   Heap Blocks: exact=834
   Buffers: shared hit=867
   ->  BitmapOr  (cost=154.03..154.03 rows=12562 width=0) (actual time=1.825..1.825 rows=0 loops=1)
         Buffers: shared hit=33
         ->  Bitmap Index Scan on idx_t_gin3_1  (cost=0.00..83.85 rows=9980 width=0) (actual time=0.904..0.904 rows=10082 loops=1)
               Index Cond: (t_gin3.c1 = 1)
               Buffers: shared hit=6
         ->  Bitmap Index Scan on idx_t_gin3_1  (cost=0.00..9.22 rows=172 width=0) (actual time=0.355..0.355 rows=164 loops=1)
               Index Cond: ((t_gin3.c2 = 1) AND (t_gin3.c3 = 1))
               Buffers: shared hit=8
         ->  BitmapOr  (cost=21.98..21.98 rows=83 width=0) (actual time=0.334..0.334 rows=0 loops=1)
               Buffers: shared hit=13
               ->  Bitmap Index Scan on idx_t_gin3_1  (cost=0.00..7.92 rows=42 width=0) (actual time=0.172..0.172 rows=36 loops=1)
                     Index Cond: ((t_gin3.c4 = 1) AND (t_gin3.c6 = 1))
                     Buffers: shared hit=6
               ->  Bitmap Index Scan on idx_t_gin3_1  (cost=0.00..7.91 rows=41 width=0) (actual time=0.162..0.162 rows=27 loops=1)
                     Index Cond: ((t_gin3.c4 = 1) AND (t_gin3.c7 = 2))
                     Buffers: shared hit=7
         ->  Bitmap Index Scan on idx_t_gin3_1  (cost=0.00..14.38 rows=1317 width=0) (actual time=0.124..0.124 rows=1296 loops=1)
               Index Cond: (t_gin3.c8 = 9)
               Buffers: shared hit=3
         ->  Bitmap Index Scan on idx_t_gin3_1  (cost=0.00..12.07 rows=1010 width=0) (actual time=0.102..0.102 rows=1061 loops=1)
               Index Cond: (t_gin3.c9 = 10)
               Buffers: shared hit=3
 Planning time: 0.272 ms
 Execution time: 6.349 ms
(29 rows)
```

操作符

操作符	示例
<@	select * from test where id <@ array[1,2];
@>	select * from test where id @> array[1,2];
=	select * from test where id = array[1,2];
&&	select * from test where id && array[1,2];

您也可以通过btree_gin插件，支持btree相关的操作符类。

索引结构



- entry：GIN索引中的一个元素。
- entry tree：在entry上构建的B树。
- posting list：entry物理位置的链表。
- posting tree：posting list构建的B树。
- pending list：索引元组的临时存储链表，用于fastupdate插入。

4.1.4. GiST索引

GiST表示通用搜索树。它是一种平衡的树结构的访问方法，它作为一种模板可用来实现任意索引模式。B tree、R tree和很多其他索引模式都可以在GiST中实现。

应用场景

- 几何类型：支持位置搜索（包含、相交、在上下左右等），按距离排序。

```
postgres=# create table t_gist (id int, pos point);
postgres=# insert into t_gist select generate_series(1,100000), point(round((random()*1000)::numeric, 2),
round((random()*1000)::numeric, 2));
postgres=# select * from t_gist  limit 3;
 id |       pos
----+-----------------
  1 | (325.43,477.07)
  2 | (257.65,710.94)
  3 | (502.42,582.25)
(3 rows)
postgres=# create index idx_t_gist_1 on t_gist using gist (pos);

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gist where circle '((100,100) 10)'  @> pos;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on public.t_gist  (cost=2.55..125.54 rows=100 width=20) (actual time=0.072..0.132 rows=46 loops=1)
   Output: id, pos
   Recheck Cond: ('<(100,100),10>'::circle @> t_gist.pos)
   Heap Blocks: exact=41
   Buffers: shared hit=47
   ->  Bitmap Index Scan on idx_t_gist_1  (cost=0.00..2.53 rows=100 width=0) (actual time=0.061..0.061 rows=46
   loops=1)
         Index Cond: ('<(100,100),10>'::circle @> t_gist.pos)
         Buffers: shared hit=6
 Planning time: 0.147 ms
 Execution time: 0.167 ms

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gist where circle '((100,100) 1)' @> pos
order by pos <-> '(100,100)' limit 10;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.28..14.60 rows=10 width=28) (actual time=0.045..0.048 rows=2 loops=1)
   Output: id, pos, ((pos <-> '(100,100)'::point))
   Buffers: shared hit=5
   ->  Index Scan using idx_t_gist_1 on public.t_gist  (cost=0.28..143.53 rows=100 width=28) (actual time=0.044..0.046
   rows=2 loops=1)
         Output: id, pos, (pos <-> '(100,100)'::point)
         Index Cond: ('<(100,100),1>'::circle @> t_gist.pos)
         Order By: (t_gist.pos <-> '(100,100)'::point)
         Buffers: shared hit=5
 Planning time: 0.092 ms
 Execution time: 0.076 ms
```

- 范围类型：支持位置搜索（包含、相交、在左右等）。
- IP类型：支持位置搜索（包含、相交、在左右等）。
- 空间类型（PostGIS）：支持位置搜索（包含、相交、在上下左右等），按距离排序。

```
postgres=# create extension btree_gist;
CREATE EXTENSION

postgres=# create index idx_t_btree_2 on t_btree using gist(id);
CREATE INDEX

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_btree order by id <-> 100 limit 1;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.15..0.19 rows=1 width=41) (actual time=0.046..0.046 rows=1 loops=1)
   Output: id, info, ((id <-> 100))
   Buffers: shared hit=3
   ->  Index Scan using idx_t_btree_2 on public.t_btree  (cost=0.15..408.65 rows=10000 width=41) (actual
   time=0.045..0.045 rows=1 loops=1)
         Output: id, info, (id <-> 100)
         Order By: (t_btree.id <-> 100)
         Buffers: shared hit=3
 Planning time: 0.085 ms
 Execution time: 0.076 ms
(9 rows)
```

- 标量类型：支持按距离排序。

操作符

- <<
- &<
- &>
- >>
- <<|
- &<|
- <@
- ~=
- &&
- |>>
- @>
- |&>

4.1.5. BRIN索引

BRIN 索引是块级索引，有别于B-TREE等索引，BRIN记录并不是以行号为单位记录索引明细，而是记录每个数据块或者每段连续的数据块的统计信息。因此BRIN索引空间占用特别的小，对数据写入、更新、删除的影响也很小。

操作符

- <
- <=
- =
- >=
- >

应用场景

流式日志数据，按时间顺序不断的顺序插入；索引占用空间小，性能要求高。

- 创建表以及插入数据等操作

```
postgres=# create table t_brin (id int, info text, crt_time timestamp);

postgres=# insert into t_brin select generate_series(1,1000000), md5(random()::text), clock_timestamp();

postgres=# select ctid,* from t_brin limit 3;
 ctid  | id |               info               |          crt_time
-------+----+----------------------------------+----------------------------
 (0,1) |  1 | e48a6cd688b6cc8e86ee858fa993b31b | 2017-06-27 22:50:19.172224
 (0,2) |  2 | e79c335c679b0bf544e8ba5f01569df7 | 2017-06-27 22:50:19.172319
 (0,3) |  3 | b75ec6db320891a620097164b751e682 | 2017-06-27 22:50:19.172323
(3 rows)

postgres=# create index idx_t_brin_1 on t_brin using brin (id) with (pages_per_range=1);
postgres=# create index idx_t_brin_2 on t_brin using brin (crt_time) with (pages_per_range=1);
postgres=# create index idx_t_brin_3 on t_brin(id);
postgres=# create index idx_t_brin_4 on t_brin(crt_time);

postgres=# \di+
                                 List of relations
  Schema  |     Name     | Type  |  Owner   | Table  |  Size  | Description
----------+--------------+-------+----------+--------+--------+-------------
 wangjian | idx_t_brin_1 | index | wangjian | t_brin | 272 kB |
 wangjian | idx_t_brin_2 | index | wangjian | t_brin | 352 kB |
 wangjian | idx_t_brin_3 | index | wangjian | t_brin | 21 MB  |
 wangjian | idx_t_brin_4 | index | wangjian | t_brin | 21 MB  |
```

- 使用索引和删除索引的区别示例一

```
postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_brin where id between 100 and 200;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
-------------
 Index Scan using idx_t_brin_3 on wangjian.t_brin  (cost=0.42..10.45 rows=101 width=45) (actual time=0.008..0.031
 rows=101 loops=1)
   Output: id, info, crt_time
   Index Cond: ((t_brin.id >= 100) AND (t_brin.id <= 200))
   Buffers: shared hit=5
 Planning Time: 1.530 ms
 Execution Time: 0.052 ms
(6 rows)

postgres=# drop index idx_t_brin_3;
DROP INDEX
postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_brin where id between 100 and 200;
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
---
 Bitmap Heap Scan on wangjian.t_brin  (cost=106.02..501.28 rows=95 width=45) (actual time=2.413..2.436 rows=101
 loops=1)
   Output: id, info, crt_time
   Recheck Cond: ((t_brin.id >= 100) AND (t_brin.id <= 200))
   Rows Removed by Index Recheck: 113
   Heap Blocks: lossy=2
   Buffers: shared hit=45
   ->  Bitmap Index Scan on idx_t_brin_1  (cost=0.00..106.00 rows=107 width=0) (actual time=2.394..2.394 rows=20
   loops=1)
         Index Cond: ((t_brin.id >= 100) AND (t_brin.id <= 200))
         Buffers: shared hit=43
 Planning Time: 0.317 ms
 Execution Time: 2.733 ms
(11 rows)
```

- 使用索引和删除索引的区别示例二

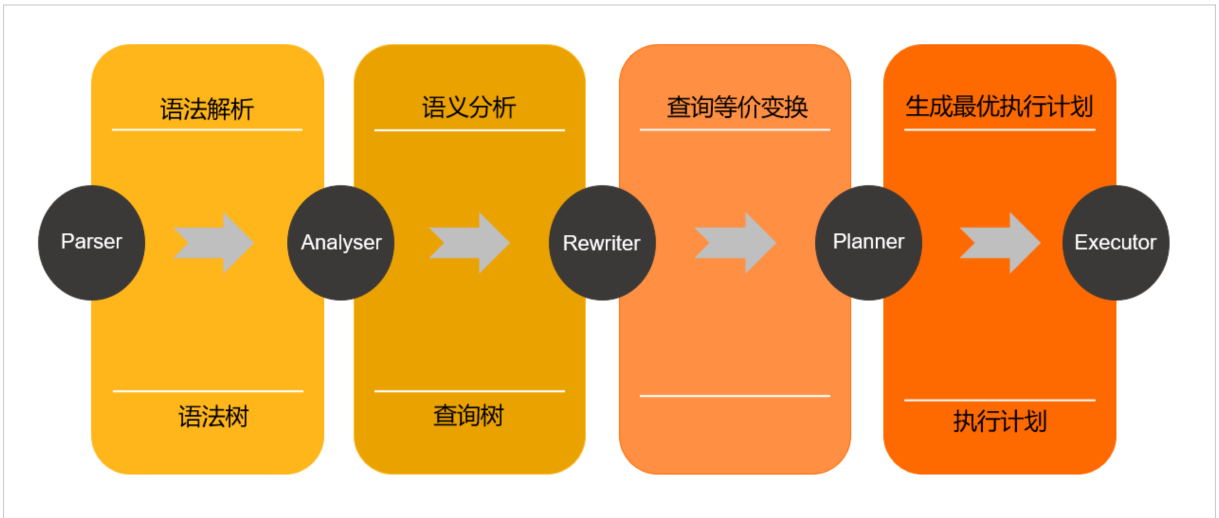
```
postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_brin where crt_time between '2017-06-27
22:50:19.172224' and '2017-06-27 22:50:19.182224';
                                                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------
 Index Scan using idx_t_brin_4 on wangjian.t_brin  (cost=0.42..8.45 rows=1 width=45) (actual time=0.007..0.007 rows=0
 loops=1)
   Output: id, info, crt_time
   Index Cond: ((t_brin.crt_time >= '2017-06-27 22:50:19.172224'::timestamp without time zone) AND (t_brin.crt_time
   <= '2017-06-27 22:50:19.182224'::timestamp without time zone))
   Buffers: shared hit=3
 Planning Time: 0.136 ms
 Execution Time: 0.020 ms
(6 rows)

postgres=# drop index idx_t_brin_4;
DROP INDEX
postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_brin where crt_time between '2017-06-27
22:50:19.172224' and '2017-06-27 22:50:19.182224';
                                                                                       QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------
 Bitmap Heap Scan on wangjian.t_brin  (cost=146.00..541.26 rows=1 width=45) (actual time=2.200..2.200 rows=0 loops=1)
   Output: id, info, crt_time
   Recheck Cond: ((t_brin.crt_time >= '2017-06-27 22:50:19.172224'::timestamp without time zone) AND (t_brin.crt_time
   <= '2017-06-27 22:50:19.182224'::timestamp without time zone))
   Buffers: shared hit=58
   ->  Bitmap Index Scan on idx_t_brin_2  (cost=0.00..146.00 rows=107 width=0) (actual time=2.198..2.198 rows=0
   loops=1)
         Index Cond: ((t_brin.crt_time >= '2017-06-27 22:50:19.172224'::timestamp without time zone) AND (
         t_brin.crt_time <= '2017-06-27 22:50:19.182224'::timestamp without time zone))
         Buffers: shared hit=58
 Planning Time: 0.318 ms
 Execution Time: 2.223 ms
(9 rows)
```

4.2. 分析执行计划

4.2.1. EXPLAIN语法

在使用数据库时，可能会出现表对应字段已经创建了索引，但是SQL语句执行慢的情况。您可以使用EXPLAIN命令查看下对应的查询计划，从而可以快速定位慢SQL。

下图为您展示优化器从开始解析到最终执行的过程。



语法

EXPLAIN 命令可以输出SQL 语句的查询计划，具体语法如下：

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
where option can be one of:
    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

- BUFFERS选项为TRUE会显示关于缓存的使用信息，默认为FALSE。该参数只能与ANALYZE 参数一起使用。缓冲区信息包括共享块（常规表或者索引块）、本地块（临时表或者索引块）和临时块（排序或者哈希等涉及到的短期存在的数据块）的命中块数，更新块数，挤出块数。
- COSTS选项为TRUE会显示每个计划节点的预估启动代价（找到第一个符合条件的结果的代价）和总代价，以及预估行数和每行宽度，默认为TRUE。
- VERBOSE选项为TRUE会显示查询计划的附加信息，默认为FALSE。附加信息包括查询计划中每个节点输出的列（Output），表的SCHEMA信息，函数的SCHEMA 信息，表达式中列所属表的别名，被触发的触发器名称等。
- ANALYZE选项为TRUE会实际执行SQL，并获得相应的查询计划，默认为FALSE。如果优化一些修改数据的SQL 需要真实的执行但是不能影响现有的数据，可以放在一个事务中，分析完成后可以直接回滚。
- FORMAT指定输出格式，默认为TEXT。各个格式输出的内容都是相同的，其中XML|JSON|YAML更有利于您通过程序解析SQL语句的查询计划。
- SUMMARY选项为TRUE会在查询计划后面输出总结信息，例如查询计划生成的时间和查询计划执行的时间。当ANALYZE 选项打开时，它默认为TRUE。
- TIMING选项为TRUE会显示每个计划节点的实际启动时间和总的执行时间，默认为TRUE。该参数只能与ANALYZE 参数一起使用。因为对于一些系统来说，获取系统时间需要比较大的代价，如果只需要准确的返回行数，而不需要准确的时间，可以把该参数关闭。

4.2.2. EXPLAIN输出

本文介绍EXPLAIN输出结构 、真实执行信息、输出节点、Index Only Scan、Bitmap Index Scan与Bitmap Heap Scan等信息。

EXPLAIN输出结构

EXPLAIN输出结构示例如下：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 t1, tenk2 t2 WHERE t1.unique1 < 100 AND t1.unique2 = t2
.unique2 ORDER BY t1.fivethous;
QUERY PLAN
------------------------------------------------------------------------------------------------
----------------------------------------
Sort (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
  Sort Key: t1.fivethous
  Sort Method: quicksort Memory: 77kB
  -> Hash Join (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100
loops=1)
       Hash Cond: (t2.unique2 = t1.unique2)
       -> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244) (actual time=0.007.
.2.583 rows=10000 loops=1)
       -> Hash (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100
loops=1)
            Buckets: 1024 Batches: 1 Memory Usage: 28kB
            -> Bitmap Heap Scan on tenk1 t1 (cost=5.07..229.20 rows=101 width=244) (actual
time=0.080..0.526 rows=100 loops=1)
                 Recheck Cond: (unique1 < 100)
                 -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101 width=0) (
actual time=0.049..0.049 rows=100 loops=1)
                      Index Cond: (unique1 < 100)
Planning time: 0.194 ms
Execution time: 8.008 ms
```

EXPLAIN命令的输出可以看做是一个树形结构，可以将其称之为查询计划树，树的每个节点包括对应的节点类型，作用对象以及其他属性例如cost、rows、width等。如果只显示节点类型，上面的例子可以简化为如下结构：

```
Sort
└── Hash Join
          ├── Seq Scan
          └── Hash
                    └── Bitmap Heap Scan
                              └── Bitmap Index Scan
```

PolarDB中SQL执行的一些特点：

- 按照查询计划树从底往上执行。
- 基于火山模型执行，即可以简单理解为每个节点执行返回一行记录给父节点（Bitmap Index Scan 除外）。

通过以上特征可以了解到EXPLAIN输出的是一个用户可视化的查询计划树，可以查看到您执行了哪些节点（操作），并且每个节点（操作）的代价预估。

真实执行信息

当EXPLAIN命令中ANALYZE选项为TRUE时，会在代价估计信息之后输出真实执行信息，包括：

- actual time：执行时间，格式为 `xxx..xxx` ，在 `..` 之前的是该节点实际的启动时间，即找到符合该节点条件的第一个结果实际需要的时间，在 `..` 之后的是该节点实际的执行时间。
- rows：指的是该节点实际的返回行数。
- loops：指的是该节点实际的重启次数。如果一个计划节点在运行过程中，它的相关参数值（如绑定变

量）发生了变化，就需要重新运行这个计划节点。

代价估计信息一般是和真实执行信息比较相近的，即预估代价和实际时间成正比且返回结果集的行数相近。但由于统计信息的时效性，有可能找到的预估代价最小的性能却很差，这就需要开发者调整参数或主动执行vacuum analyze命令对表的统计信息进行及时更新，保证PostgreSQL的执行优化器能够找到相对较优的查询计划树。

输出节点

- 节点类型

  在EXPLAIN命令的输出结果中可能包含多种类型的执行节点，可以大体分为几大类：

  - 控制节点（Control Node）
  - 扫描节点（ScanNode）
  - 物化节点（Materialization Node）
  - 连接节点（Join Node）

- 扫描节点

  扫描节点就是为了扫描表的元组，每次获取一条元组（Bitmap Index Scan除外）作为上层节点的输入。扫描节点不光可以扫描表，还可以扫描函数的结果集、链表结构、子查询结果集等。

  目前在PolarDB中支持：

  - Seq Scan：顺序扫描
  - Index Scan：基于索引扫描，但不只是返回索引列的值
  - IndexOnly Scan：基于索引扫描，并且只返回索引列的值，简称为覆盖索引
  - BitmapIndex Scan：利用Bitmap结构扫描
  - BitmapHeap Scan：把BitmapIndex Scan返回的Bitmap结构转换为元组结构
  - Tid Scan：用于扫描一个元组TID数组
  - Subquery Scan：扫描一个子查询
  - Function Scan：处理含有函数的扫描
  - TableFunc Scan：处理tablefunc相关的扫描
  - Values Scan：用于扫描Values链表的扫描
  - Cte Scan：用于扫描WITH字句的结果集
  - NamedTuplestore Scan：用于某些命名的结果集的扫描
  - WorkTable Scan：用于扫描Recursive Union的中间数据
  - Foreign Scan：用于外键扫描
  - Custom Scan：用于用户自定义的扫描

下面重点介绍常用的几个扫描节点：Seq Scan、Index Scan、IndexOnly Scan、BitmapIndex Scan、BitmapHeap Scan。

- Seq Scan

  Seq Scan是全表顺序扫描，一般查询没有索引的表需要全表顺序扫描，例如下面的EXPLAIN输出：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select * from class where st_no=2;
QUERY PLAN
------------------------------------------------------------------------------------------
---------
Seq Scan on public.class (cost=0.00..26.00 rows=1 width=35) (actual time=0.136..0.141 row
s=1 loops=1)
    Output: st_no, name
    Filter: (class.st_no = 2)
    Rows Removed by Filter: 1199
    Buffers: shared hit=11
Planning time: 0.066 ms
Execution time: 0.160 ms
```

其中：

参数	说明
`Seq Scan on public.class`	表明了这个节点的类型和作用对象，即在class表上进行了全表扫描。
`(cost=0.00..26.00 rows=1 width=35)`	表明了这个节点的代价估计。
`(actual time=0.136..0.141 rows=1 loops=1)`	表明了这个节点的真实执行信息，当EXPLAIN命令中的ANALYZE选项为TRUE时，会输出该项内容。
`Output: st_no, name`	表明了SQL的输出结果集的各个列，当EXPLAIN命令中的选项VERBOSE为TRUE时才会显示。
`Filter: (class.st_no = 2)`	表明了Seq Scan节点之上的Filter操作，即全表扫描时对每行记录进行过滤操作，过滤条件为 `class.st_no = 2` 。
`Rows Removed by Filter: 1199`	表明了过滤操作过滤了多少行记录，属于Seq Scan节点的VERBOSE信息，只有EXPLAIN命令中的VERBOSE选项为TRUE时才会显示。
`Buffers: shared hit=11`	表明了从共享缓存中命中了11个BLOCK，属于Seq Scan节点的BUFFERS信息，只有EXPLAIN命令中的BUFFERS选项为TRUE时才会显示。
`Planning time: 0.066 ms`	表明了生成查询计划的时间。
`Execution time: 0.160 ms`	表明了实际的SQL执行时间，其中不包括查询计划的生成时间。

- Index Scan

  Index Scan是索引扫描，主要用来在WHERE条件中存在索引列时的扫描，如上面Seq Scan中的查询如果在st_no上创建索引，则EXPLAIN输出如下：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select * from class where st_no=2;
QUERY PLAN
------------------------------------------------------------------------------------------
---------
Index Scan using no_index on public.class (cost=0.28..8.29 rows=1 width=35) (actual time=
0.022..0.023 rows=1 loops=1)
     Output: st_no, name
     Index Cond: (class.st_no = 2)
     Buffers: shared hit=3
Planning time: 0.119 ms
Execution time: 0.060 ms (6 rows)
```

其中：

参数	说明
`Index Scan using no_index on public.class`	表明是使用的public.class表的no_index索引对表进行索引扫描的。
`Index Cond: (class.st_no = 2)`	表明索引扫描的条件是class.st_no = 2。

通过示例可以看出，使用了索引之后，对相同表的相同条件的扫描速度变快了。这是因为从全表扫描变为索引扫描，通过Buffers: shared hit=3可以得出结果，需要扫描的元组（BLOCK）变少，所需要的代价就会变小，扫描速度就会变快。

Index Only Scan

IndexOnly Scan是覆盖索引扫描，所需的返回结果能被所扫描的索引全部覆盖，如Index Scan中的SQL把 `select *` 修改为 `select st_no` ，其EXPLAIN 结果输出如下：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select st_no from class where st_no=2;
QUERY PLAN
------------------------------------------------------------------------------------------
-------
Index Only Scan using no_index on public.class (cost=0.28..4.29 rows=1 width=4) (actual tim
e=0.015..0.016 rows=1 loops=1)
    Output: st_no Index Cond: (class.st_no = 2)
    Heap Fetches: 0
    Buffers: shared hit=3
Planning time: 0.058 ms
Execution time: 0.036 ms
(7 rows)
```

参数	说明
`Index Only Scan using no_index on public.class`	表明使用public.class表的no_index索引对表进行覆盖索引扫描。
`Heap Fetches`	表明需要扫描数据块的个数。

虽然Index Only Scan可以从索引直接输出结果。但是因为PostgreSQL MVCC机制的实现，需要对扫描的元组进行可见性判断，即检查visibility MAP文件。当新建表之后，如果没有进行过vacuum和autovacuum操作，这时还没有VM文件，而索引并没有保存记录的版本信息，索引Index Only Scan还是需要扫描数据块（Heap Fetches代表需要扫描的数据块个数）来获取版本信息，这个时候可能会比Index Scan慢。

Bitmap Index Scan与Bitmap Heap Scan

BitmapIndex Scan与Index Scan很相似，都是基于索引的扫描，但BitmapIndex Scan节点每次执行返回的是一个位图而不是一个元组，位图中每位代表了一个扫描到的数据块。而BitmapHeap Scan一般会作为BitmapIndex Scan的父节点，将BitmapIndex Scan返回的位图转换为对应的元组。这样做最大的好处就是把Index Scan的随机读转换成了按照数据块的物理顺序读取，在数据量比较大的时候，这会大大提升扫描的性能。

可以运行set enable_indexscan =off; 来指定关闭Index Scan ，上文中Index Scan中SQL 的EXPLAIN输出结果则变为：

```
postgres=> explain(ANALYZE,VERBOSE,BUFFERS) select * from class where st_no=2;
QUERY PLAN
------------------------------------------------------------------------------------------------
-------
Bitmap Heap Scan on public.class (cost=4.29..8.30 rows=1 width=35) (actual time=0.025..0.02
5 rows=1 loops=1) Output: st_no, name
Recheck Cond: (class.st_no = 2)
Heap Blocks: exact=1
Buffers: shared hit=3
-> Bitmap Index Scan on no_index (cost=0.00..4.29 rows=1 width=0) (actual time=0.019..0.019
rows=1 loops=1) Index Cond: (class.st_no = 2)
Buffers: shared hit=2
Planning time: 0.088 ms
Execution time: 0.063 ms
(10 rows)
```

其中：

参数	说明
`Bitmap Index Scan on no_index`	表明使用no_index索引进行位图索引扫描。
`Index Cond: (class.st_no = 2)`	表明位图索引的条件为class.st_no = 2。
`Bitmap Heap Scan on public.class`	表明对public.class表进行Bitmap Heap扫描。
`Recheck Cond: (class.st_no = 2)`	表明Bitmap Heap Scan的Recheck操作的条件是class.st_no = 2，这是因为Bitmap Index Scan节点返回的是位图，位图中每位代表了一个扫描到的数据块，通过位图可以定位到一些符合条件的数据块（这里是3，Buffers: shared hit=3），而Bitmap Heap Scan则需要对每个数据块的元组进行Recheck。
`Heap Blocks: exact=1`	表明准确扫描到数据块的个数是1。

- 大多数情况下，Index Scan要比Seq Scan快。但是如果获取的结果集占所有数据的比重很大时，这时Index Scan因为要先扫描索引再读表数据反而不如直接全表扫描来的快。

- 如果获取的结果集的占比比较小，但是元组数很多时，可能Bitmap Index Scan的性能要比Index Scan好。
- 如果获取的结果集能够被索引覆盖，则Index Only Scan因为不用去读数据，只扫描索引，性能一般最好。但是如果VM文件未生成，可能性能就会比Index Scan要差。

以上结论都是基于理论分析得到的结果，其实PostgreSQL的EXPLAIN命令中输出的cost，rows，width等代价估计信息中已经展示了这些扫描节点或者其他节点的预估代价，通过对预估代价的比较，可以选择出最小代价的查询计划树。

4.2.3. 调整执行计划

PolarDB O引擎提供了Outline功能，用于抓取、查看和删除固定的执行计划。

功能介绍

执行计划管理SPM（SQL Plan Management）是一种预防性机制，可以使优化器自动管理执行计划，确保数据库仅使用已知或经过验证的计划。

Outline是SPM的子集，仅支持固定执行计划，不支持演进执行计划。目前PolarDB的Outline功能仅支持固定PREPARE类型语句的执行计划。

PolarDB的Outline功能是通过polar_outline插件实现的，如需使用请提交工单联系技术支持进行开启。

创建插件

创建插件语句如下：

```
CREATE EXTENSION polar_outline;
```

配置参数

打开Outline固定执行计划功能，开启语句如下：

```
SET polar_outline.use_stored_plan TO ON;
```

由于Outline定义的函数和关系（Relation）都在polar_outline这个schema下，为了方便使用，您可以通过如下语句设置路径。

```
SET search_path To "$user",public,polar_outline;
```

准备测试数据

创建表t，并在表中插入数据。语句如下：

```
CREATE TABLE t(a INT,b INT);
INSERT INTO t SELECT i, i FROM generate_series(1, 1000)i;
```

更新表数据以确保优化器使用的统计信息正确。

```
ANALYZE t;
```

执行一个PREPARE类型的语句。

```
PREPARE test AS SELECT * FROM t WHERE a=$1;
```

> **说明**
>
> - 测试数据仅适用于本文中的示例，实际使用中请根据您的实际情况进行调整。
> - 您也可以使用hint功能来控制您想要被outline抓取的执行计划，从而使其他没有使用hint的语句也会被outline固定成您想要的计划。

抓取和固定执行计划

目前PolarDB提供了两种方法来抓取和固定执行计划，分别如下：

- 使用polar_outline_create函数抓取执行计划（推荐使用），语句如下：

  ```
  SELECT polar_outline_create('EXECUTE test(100)');
  ```

  返回信息如下：

  ```
   polar_outline_create
  ----------------------
   t
  (1 row)
  ```

- 使用执行计划抓取功能，该方法更适合批量抓取执行计划，操作方式如下：

  > **注意** 开启执行计划抓取功能会导致计划缓存（plancache）

  i. 开启抓取执行计划功能，语句如下：

  ```
  SET polar_outline.capture_plan TO ON;
  ```

  ii. 抓取执行计划，语句如下：

  ```
  EXECUTE test(100);
  ```

  返回信息如下：

  ```
    a  |  b
  -----+-----
   100 | 100
  (1 row)
  ```

  iii. 关闭抓取执行计划功能，语句如下：

  ```
  SET polar_outline.capture_plan TO OFF;
  ```

参数配置

参数	说明

参数	说明

use_stored_plan	打开或关闭Outline固定执行计划。
	• 打开命令如下：
	`SET polar_outline.use_stored_plan TO ON;`
	• 关闭命令如下：
	`SET polar_outline.use_stored_plan TO OFF;`

capture_plan	打开或关闭Outline计划抓取开关。
	• 打开命令如下：
	`SET polar_outline.capture_plan TO ON;`
	• 关闭命令如下：
	`SET polar_outline.capture_plan TO OFF;`

log_usage	控制outline内部日志的输出级别。
	`set polar_outline.log_usage To none;`
	`set polar_outline.log_usage To debug;`
	`set polar_outline.log_usage To debug1;`
	`set polar_outline.log_usage To debug2;`
	`set polar_outline.log_usage To debug3;`
	`set polar_outline.log_usage To debug4;`
	`set polar_outline.log_usage To debug5;`
	`set polar_outline.log_usage To log;`
	`set polar_outline.log_usage To info;`
	`set polar_outline.log_usage To notice;`
	`set polar_outline.log_usage To warning;`

通过outline实现查询改写

在PolarDB查询改写不符合预期时，您可以使用outline来实现SQL的查询改写。

> **说明** 错误的使用可能会导致PolarDB输出错误的执行计划，返回错误的结果集，请确保改写计划的等价。

如下所示，SQL1和SQL2是等价的。

```
SQL1:
select t.a, t2.avg_b
from t join (select avg(b) as avg_b, a
             from t2
             group by a) t2
on t2.a = t.a and t.c < $1
order by t.a;
SQL2:
select t.a, t2.avg_b
from t join lateral (select avg(b) as avg_b
                     from t2
                     where t2.a = t.a) as t2
on t.c < $1
order by t.a;
```

通过以下步骤展示如何通过outline将SQL1查询改写成SQL2的。

1. 执行以下命令，准备数据。

```
CREATE TABLE t(a int, b int, c int);
insert into t select i % 100000, i, i from generate_series(1, 1000000) i;
create table t2 as select * from t;
create index on t(c);
create index on t2(a);
ANALYZE t,t2;
```

2. 执行以下命令，将准备改写的原SQL和改写后的SQL抓取进outline中。

```
prepare s1 as select t.a, t2.avg_b from t join (select avg(b) as avg_b, a from t2 group
by a) t2
on t2.a = t.a and t.c < $1 order by t.a;
prepare s2 as select t.a, t2.avg_b from t join lateral (select avg(b) as avg_b from t2
where t2.a = t.a) as t2
on t.c < $1 order by t.a;
SELECT polar_outline.polar_outline_create('EXECUTE s1(5)');
 polar_outline_create
----------------------
 t
(1 row)
SELECT polar_outline.polar_outline_create('EXECUTE s2(5)');
 polar_outline_create
----------------------
 t
(1 row)
```

3. 执行以下命令，使用polar_outline_switch功能交换两个SQL的计划。

> **说明** 输入的参数为两个计划在outline中的ID，原SQL的计划即被改写为改写后的SQL的计划。

```
SELECT polar_outline.polar_outline_switch(1,2);
 polar_outline_switch
----------------------
 t
(1 row)
```

4. 执行以下命令，删除outline中改写后的SQL。

> **说明** 计划改写完成后，需要清除outline中SQL2的无用计划。

```
SELECT polar_outline.polar_outline_delete(2);
```

5. 通过outline实现查询改写的结果如下所示：

查询改写前：s1是改写前的计划

```
EXPLAIN (COSTS FALSE) EXECUTE s1(5);
                QUERY PLAN
------------------------------------------
 Sort
   Sort Key: t.a
   ->  Hash Join
         Hash Cond: (t.a = t2.a)
         ->  Index Scan using t_c_idx on t
               Index Cond: (c < $1)
         ->  Hash
               ->  HashAggregate
                     Group Key: t2.a
                     ->  Seq Scan on t2
(10 rows)
SELECT polar_outline_switch(1,2);
 polar_outline_switch
----------------------
 t
(1 row)
```

查询改写后：s1是改写后s2的计划

```
EXPLAIN (COSTS FALSE) EXECUTE s1(5);
                     QUERY PLAN
-------------------------------------------------------
 Sort
   Sort Key: t.a
   ->  Nested Loop
         ->  Index Scan using t_c_idx on t
               Index Cond: (c < $1)
         ->  Aggregate
               ->  Bitmap Heap Scan on t2
                     Recheck Cond: (a = t.a)
                     ->  Bitmap Index Scan on t2_a_idx
                           Index Cond: (a = t.a)
(10 rows)
```

4.3. 物理优化

4.3.1. 代价的相关概念

本节为您介绍什么是代价以及代价的相关概念。

概述

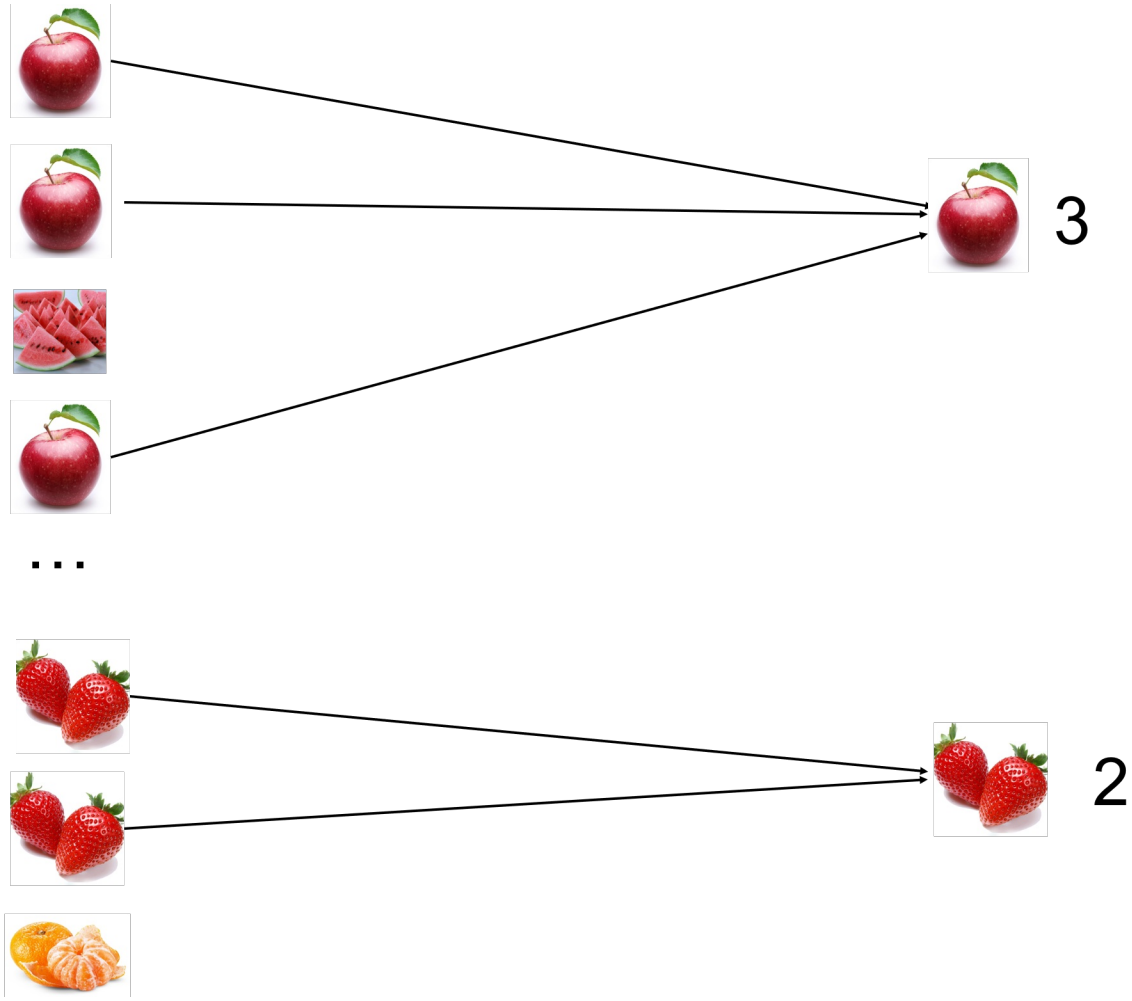
物理优化是基于代价的查询优化，执行代价由IO代价和CPU代价组成。

- IO代价的评估方式请参见
- CPU代价的评估方式请参见

统计信息

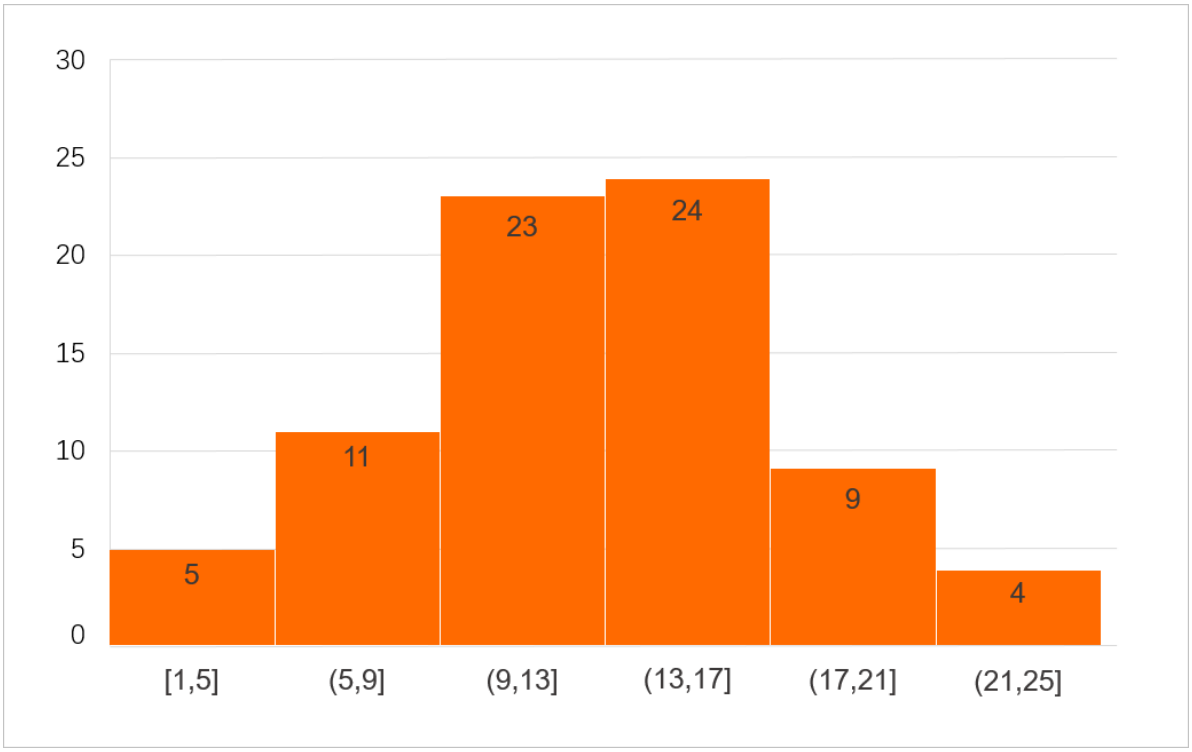
- 高频值

  表示常见值，例如在表t1中，a字段大小是1~100，其中1~10的值占据了95%，1-10的值就称为高频值。高频值用于等值查询，进行评估选择性。



- 直方图

  表示数据值的分布情况，例如在表t1中，a字段大小是1~100，可以分为4个桶，1~25的值有30个，26~50的值有20个，51~75的值有25个，76~100的值有25个。



- 相关系数

  表示某一列的物理顺序和逻辑顺序的相关性，相关性越高，走索引扫描离散块扫描代价越低。

- 其他统计信息
  - 唯一值个数
  - Null值比率
  - 表的行数
  - 表的页面数

选择率

- 无条件查询

```
EXPLAIN SELECT * FROM tenk1;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
 SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
 relpages | reltuples
   ----------+-----------
        358 |     10000
```

- 范围查询

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

- 范围查询计算公式

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';
                   histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

- 等值查询

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

- 等值查询计算公式

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';
null_frac         | 0
n_distinct        | 676
most_common_vals|{EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
selectivity = mcf[3]  = 0.003
rows = 10000 * 0.003 = 30
## 备注：如果值不在most_common_vals里面，计算公式为selectivity = (1 - sum(mvf))/(num_distinct
- num_mcv)
```

4.3.2. 代价计算

本节介绍代价计算和相关计算公式。

代价计算模型

操作成本	默认值	说明
seq_page_cost	1.0	顺序扫描一个页面的成本。
random_page_cost	4.0	随机扫描一个页面的成本。
cpu_tuple_cost	0.01	CPU处理一行的成本。
cpu_index_tuple_cost	0.005	CPU处理一个索引项的成本。
cpu_operator_cost	0.0025	CPU处理每个函数或操作符的成本。

全表扫描

```
postgres=# explain select * from test where id <1000;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test  (cost=0.00..1693.00 rows=5680 width=4)
   Filter: (id < 1000)
(2 rows)
postgres=# select relpages,reltuples from pg_class where relname='test';
 relpages | reltuples
 ------------+-----------
        443  |    100000
(1 row)
```

计算公式如下：

- TOTAL_COST = CPU代价 + IO代价
- CPU代价 = cpu_tuple_cost * reltuples + cpu_operator_cost * reltuples
- CPU代价 = 0.01 * 100000 + 0.0025 * 100000 = 1250.0
- IO代价 = seq_page_cost * relpages
- IO代价 = 1.0 * 443 = 443
- TOTAL_COST = 1250 + 443 = 1693

索引扫描

```
postgres=# explain select * from test where id =1414;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Only Scan using test_id_idx on test  (cost=0.29..72.66 rows=30 width=4)
   Index Cond: (id = 1414)
(2 rows)
```

计算公式如下：

- RUN_COST = CPU代价 + IO代价
- CPU代价 = 索引 CPU代价 + 表 CPU代价
- 索引CPU代价 = 选择率 * 索引行数 * (cpu_index_tuple_cost + cpu_operator_cost)
- 表 CPU代价 = 选择率 * 数据行数 * cpu_tuple_cost

索引CPU代价计算

```
postgres=# select most_common_freqs[array_position((most_common_vals::text)::real[],1414::r
eal)] from pg_stats where tablename ='test';
 most_common_freqs
-------------------
            0.0003
## 选择率 = 0.0003
postgres=# select relpages,reltuples from pg_class where relname='test_id_idx';
 relpages | reltuples
    ----------+-----------
          276 |    100000
## 索引行数 = 100000 ，索引页面数 = 276
postgres=# select relpages,reltuples from pg_class where relname='test';
 relpages | reltuples
    ----------+-----------
          443 |    100000
## 表行数 = 100000 ，表页面数 = 443
```

通过以上计算示例得出CPU代价如下：

- 索引CPU代价 = 0.0003 * 100000 * (0.0025 + 0.005) = 0.22500000
- 表CPU代价 = 0.0003 * 100000 * 0.01 = 0.300000
- CPU代价 = 0.300000 + 0.22500000 = 0.52500000

索引IO代价计算

```
索引IO代价 = ceil (选择率 * 索引页面数) * random_page_cost
表IO代价 = max_io_cost + 相关系数 * 相关系数 * (min_io_cost - max_io_cost)
max_io_cost = 选择率 * 数据行数 * random_page_cost = 120
min_io_cost = 1 * random_page_cost + (ceil(选择率 * 表页面数) - 1) * seq_page_cost = 4
postgres=# select correlation from pg_stats where tablename ='test';
 correlation
-------------
    0.670535
相关系数 = 0.670535
IO代价 = 4 + 67.844406397900 = 71.844406397900
```

通过以上计算示例得出IO代价如下：

- RUN_COST = 71.844406397900 + 0.52500000 = 72.369406397900
- START_COST = 0.29
- TOTAL_COST = 72.369406397900 + 0.29 = 72.659406397900 ≈ 72.66

5.性能调优最佳实践

5.1. 有关性能调优

本章节中提及的所有视图，均可在视图中找到对应内容。 在进行具体问题诊断之前，需确保polar_monitor、polar_monitor_preload、pg_stat_statements、polar_stat_sql 插件已经安装。

5.2. 实例问题分析

5.2.1. 问题分析方法论

实例问题分析的基本方法论是寻找性能瓶颈， 即哪种资源的消耗最多，什么原因导致该种资源消耗较多，并持续将问题做细粒度拆解。

对于PolarDB O引擎数据库，使用的基本资源有以下两种：

- 系统资源：包括CPU、I/O、网络、内存和磁盘资源。
- 数据库资源：锁（lwlock和lock）和缓存（buffer pool和slru）资源。

通用的调优思路，即通过数据库的等待事件，判断瓶颈资源，再根据具体资源进一步做原因定位。

5.2.2. 等待事件

等待事件一般表示SQL执行过程中等待获取某种资源。

可通过**polar_stat_activity**视图查看哪类等待事件较多：

```
SELECT
    CASE WHEN wait_event_type IS NULL THEN 'CPU' ELSE wait_event_type END,
    CASE WHEN wait_event IS NULL THEN 'CPU' ELSE wait_event END,
    COUNT(*) AS wait_count
FROM
    polar_stat_activity
WHERE
    state='active' AND backend_type='client backend'
GROUP BY wait_event_type, wait_event ORDER BY wait_count DESC; \watch 1
```

- 如果CPU等待事件持续较多，即持续大于等于CPU核数的80%，则可认为CPU存在瓶颈。
- 如果I/O事件持续出现且较多，则认为I/O存在瓶颈。
- 如果lwlock事件持续出现且较多，则认为lwlock争抢严重，lwlock保护的是数据库内部的数据结构。
- 如果lock事件持续出现且较多，则认为lock争抢严重，lock是数据库的表锁或者行锁。

5.2.3. CPU瓶颈

CPU等待事件持续较多，出现CPU瓶颈，需进行具体查询分析。

可观察**polar_stat_activity_rt** 中哪类进程CPU比较高，数值单位是CPU使用率。

```
SELECT
    backend_type,SUM(cpu_sys cpu_user) AS cpu_rate
FROM polar_stat_activity_rt
GROUP BY backend_type
ORDER BY cpu_rate desc; \watch 1
```

如果CPU等待事件的数量持续大于CPU核数，说明并发数量过高。

可通过**polar_stat_activity**查看是哪些SQL造成CPU过高。

```
SELECT
    query, COUNT(*) AS wait_count
FROM polar_stat_activity
WHERE state='active' AND backend_type='client backend' AND wait_event_type IS NULL
GROUP BY query
ORDER BY wait_count DESC; \watch 1
```

也可通过**pg_stat_database**中的 xact_commit xact_rollback 增量判断是否TPS超过预期，或者是通过 **polar_stat_query_count** 增量判断是否QPS超过预期。

确定哪些SQL造成CPU高后，可以通过观察**pg_stat_statements**和**polar_stat_sql**视图来获取SQL的统计信息。

5.2.4. I/O瓶颈

I/O事件持续出现且较多，出现I/O瓶颈，需进行具体查询分析。

遇到I/O瓶颈时，可通过以下三种方式来进行分析观察：

- 观察 **polar_stat_activity_rt** 中哪类进程I/O比较高。

  ```
  SELECT
      backend_type,
      SUM(local_read_iops   local_write_iops) iops,
      SUM(local_read_iothroughput   local_write_iothroughput) throughput,
      SUM(local_read_latency local_write_latency) latency
  FROM polar_stat_activity_rt
  GROUP BY
      backend_type
  ORDER BY
      iops
  DESC; \watch 1
  ```

  - 如果是client backend进程I/O高，则可进一步通过**polar_stat_activity**查看是哪些SQL和等待事件造成I/O高。

    ```
    SELECT
        query, COUNT(*) AS wait_count
    FROM polar_stat_activity
    WHERE state='active' AND backend_type='client backend' AND wait_event_type = 'IO'
    GROUP BY query
    ORDER BY wait_count DESC;\watch 1
    ```

  - 如果是checkpoint进程I/O高，说明写入数据量太大。

- 如果是bgwriter进程I/O高，说明存储写入太慢。
- 如果是vacuum进程，可以进一步通过观察**pg_stat_progress_vacuum**查看清理表的进度。

```
SELECT * FROM pg_stat_progress_vacuum;
```

- 可以通过观察**polar_stat_io_info**视图，按照文件类型为维度，记录I/O监控信息，具体内容请参见polar_stat_io_info。
- 可以通过观察**polar_stat_io_latency**视图来分析问题，该视图按照文件操作类型为维度，记录各类文件I/O操作的延时信息。具体内容请参见polar_stat_io_latency。

5.2.5. 锁

锁分为lwlock和lock，当出现锁争抢严重时，需根据具体情况分析。

- lwlock：等待事件中的 wait_event 描述了具体的lwlock，对应详细描述请参见lwlock。
- lock
  - **polar_stat_activity**中的 wait_object 字段表示是哪个进程在持有锁。
  - 也可在**pg_locks**视图中进行锁分析。

5.2.6. SLRU

SLRU可以获取相关监控信息。

可直接通过调用polar_stat_slru函数来获取相关监控信息。

```
select * from polar_stat_slru();
```

5.2.7. 内存诊断

内存诊断有普通内存诊断和共享内存诊断两种方式，具体内存诊断方式根据需求而定。

普通内存

可以通过观察**polar_stat_activity_rt**视图，如下所示：

```
SELECT pid, rss
FROM polar_stat_activity_rt
ORDER BY rss
DESC; \watch 1
```

进而可通过**polar_backends_mcxt**查看该进程中哪个内存上下文占用最多。

> **说明** 由于当前未有关于PolarDB O引擎的内存上下文的详细介绍，当前只能通过内核代码或SQL进行更进一步的分析。

共享内存

共享内存可直接通过**polar_stat_shmem**查看分布情况。

5.2.8. 磁盘空间诊断

PolarDB O引擎数据库主体空间主要被用户数据和WAL日志占用。

查询数据空间

查询数据空间包括以下内容：

- 数据库

  通过PG客户端执行\l ，可以查看每个数据库的空间占用情况注意该操作需要遍历目录树，会有较高的CPU和IO开销。

- 表

  对于占用空间较大的数据库，可以通过客户端连接到该数据库，执行\d 查看该数据库每个表的空间占用情况。

查询WAL日志

执行以下SQL可查看当前WAL日志占用空间：

```
SELECT pg_size_pretty(SUM(size)) FROM pg_ls_waldir();
```

一般WAL占用空间较大有以下几种原因：

- 参数 wal_keep_segments 设置过大。参数 wal_keep_segments 表示最少保留的WAL文件数量。

  解决思路：可通过 SHOW wal_keep_segments检查参数设置。此外可通过SHOW wal_segment_size获知单个WAL文件大小，帮助预估WAL最少会占用多少空间。

- 在 archive_mode=on 的情况下，WAL文件没有及时归档。

  解决思路：可观察配置项 archive_command 是否配置正常。

- pg_replication_slots 同步延迟过大或存在活跃状态为 false 的复制槽。

  解决思路：需进一步检查复制槽连接是否正常。

- checkpoint进程处理不及时。

  解决思路：需及时清理。

5.2.9. 日志位点监控

本文介绍了日志位点的监控信息和延迟情况，以及延迟的处理方法。

PolarDB中存在以下几个比较重要的WAL日志位点：

- 主节点当前日志写入位点，可以通过 `pg_current_wal_insert_lsn()` 函数查询。
- 所有备节点中最小的日志回放位点，可以通过 `polar_oldest_apply_lsn()` 函数查询。
- 主节点的一致性位点，可以通过 `polar_consistent_lsn()` 函数查询。

  > **说明** 关于一致性位点的背景介绍，请参见一致性位点。

- checkpoint位点，即数据库崩溃恢复时回放日志的起始位点，可以通过 `pg_control_checkpoint()` 函数的redo_lsn字段获取。

监控方法

您可以通过以下命令监控相关位点以及延迟情况：

```
select *,
  pg_size_pretty(pg_wal_lsn_diff(WP, AP)) as "WP-AP",
  pg_size_pretty(pg_wal_lsn_diff(AP, CP)) as "AP-CP",
  pg_size_pretty(pg_wal_lsn_diff(WP, CP)) as "WP-CP",
  pg_size_pretty(pg_wal_lsn_diff(WP, redo_lsn)) as "WP-REDO"
from
  (select pg_current_wal_insert_lsn() as WP,
          polar_oldest_apply_lsn() as AP,
          polar_consistent_lsn() as CP,
          redo_lsn
   from pg_control_checkpoint()
  ) as lsn_info;
```

> **说明** 以上命令中 `polar_*` 函数需要安装 `polar_monitor` 插件后才可以使用。

备节点回放延迟

根据以上监控结果，如果 `WP-AP` 较大，说明备节点回放延迟较大，需要及时关注。

备节点回放延迟会导致如下问题：

- 主节点无法重用WAL日志，导致WAL日志大量堆积，数据库集群占用的存储空间上涨。该异常可以通过监控数据库集群的存储空间或WAL日志存储空间发现。
- 由于备节点查询存在较大的延迟，导致备节点无法查询到主节点最近更新的数据。
- 如果使用了读写分离组件（Proxy），Proxy可能将所有请求都路由到主节点进行处理，无法发挥读写分离的作用。
- 导致一致性位点和checkpoint位点延迟，影响集群的可用性。

通常的处理方法如下：

1. 执行以下命令，查询复制槽，确认主备同步是否中断。

   ```
   select * from polar_replication_slots;
   ```

   查看是否存在 `active` 为 `f` 的复制槽，如果存在，则说明该备节点与主节点的同步发生了中断，需要进一步排查主备同步中断的原因。

   > **说明** PolarDB根据不同的部署形态，其复制槽可能分为两类，即replica和standby。复制槽类型与PolarDB集群中的节点类型相对应，replica节点与主节点共用一份数据，主节点与replica节点的同步如果断开，对主节点的影响较大，需要及时处理。standby节点与主节点类似传统PostgreSQL的主备，各自有独立的数据。复制槽类型可以通过其名称（上述查询中的 `slot_name` 字段）进行区分。

2. 如果主备同步正常，执行以下命令，查看主备延迟情况。

```
select pid, usename, application_name, client_addr, state,
       pg_wal_lsn_diff(pg_current_wal_insert_lsn(), sent_lsn) as sent_delay,
       pg_wal_lsn_diff(pg_current_wal_insert_lsn(), write_lsn) as write_delay,
       pg_wal_lsn_diff(pg_current_wal_insert_lsn(), flush_lsn) as flush_delay,
       pg_wal_lsn_diff(pg_current_wal_insert_lsn(), replay_lsn) as replay_delay
from pg_stat_replication;
```

查询结果可能存在以下四类延迟：

- `sent_delay`：主节点发送日志的延迟。
- `write_delay`：备节点将接收到的日志写入存储的延迟。
- `flush_delay`：备节点将接收到的日志flush到存储的延迟。
- `replay_delay`：备节点回放日志的延迟。

> **说明** 如果备节点的回放延迟（replay_delay）较大，通常需要登录备节点所在主机，查看备节点的 `startup` 进程的运行状况，视具体情况进一步排查。

根据以上命令，查询延迟较大的备节点，进而排查对应备节点延迟较大的原因。

一致性位点延迟

如果 `WP-CP` 较大，说明一致性位点延迟较大，需要及时关注。一致性位点延迟会导致无法及时有效地执行checkpoint，进而导致数据库集群崩溃恢复的耗时变长，影响集群的可用性。

通常的处理方法如下：

1. 如果同时存在备节点延迟大的情况，请优先处理备节点延迟问题，因为备节点延迟会导致一致性位点延迟。
2. 是否存在批量导入或者批量修改数据的场景，导致短时间内产生大量脏数据，此时数据库集群可能无法及时将脏页落盘，导致一致性位点延迟。此类情况属于正常的业务请求导致一致性位点延迟变大，延迟情况应该会在此类业务请求过后恢复到较小的水位。
3. 是否存在集群规格和shared_buffer较大的情况。shared_buffer越大，数据库的buffer pool越大，其能缓存的buffer越多，可能存在的脏页也越多。PolarDB的刷脏能力是有上限的（取决于磁盘IO能力以及并行刷脏的效率），一旦达到上限，将buffer pool中的脏页落盘的速率也就达到了上限，此时如果业务请求产生脏页的速率大于刷脏的速率，则可能导致buffer pool中的脏页越来越多，进而导致一致性位点延迟逐步加大。

   此时，可以适当调整后台刷脏相关的参数，以提升刷脏效率，包括如下参数：

   - `polar_parallel_bgwriter_workers`：并行刷脏进程的数量，默认是5个。

     > **说明** 该参数值设置的是默认启动的并行刷脏进程的数据，PolarDB会根据主备延迟情况，动态启停一些并行刷脏进程，可能存在的并行刷脏进程为min(2*polar_parallel_bgwriter_workers, 16)。

   - `polar_parallel_bgwriter_delay`：并行刷脏进程每轮刷脏之间的休息时间，默认是10ms。

   可以适当增加 `polar_parallel_bgwriter_workers`，适当减小 `polar_parallel_bgwriter_delay`，以提升刷脏效率。

> **说明** 随着刷脏进程增多，会对集群整体吞吐有一定影响。刷脏占用IO变大，可能导致其他请求写入WAL日志的效率下降，进而影响集群执行SQL的效率。

checkpoint位点延迟

如果 `WP-REDO` 较大，表示checkpoint位点延迟较大，checkpoint位点直接影响集群Crash Recovey时回放的日志量，回放的日志量越多，耗时越长，对集群的可用性影响可能越大。

以下几种情况可能导致checkpoint位点延迟：

- 一致性位点延迟会导致checkpoint位点延迟，需要优先排查一致性位点延迟的原因。
- 导致一致性位点延迟的原因均可能导致checkpoint位点延迟，例如业务中有批量数据修改或者刷脏不及时。
- checkpoint执行周期较长，周期时间内有较多的数据修改。您可以适当调整 `checkpoint_timeout` 或者 `max_wal_size` 提高执行checkpoint的频率。

其他监控项

- 脏页数量

  执行以下命令，查看Buffer Pool中脏页数量。

  ```
  select size from polar_flushlist;
  ```

  `polar_flushlist` 记录FlushList的一些统计信息，上述查询结果中的FlushList中的buffer的数量，仅脏页才会放入FlushList中，因此， `size * 每个页的大小（默认为8 KB）` 即为buffer pool中脏页的大小。

- Copy Buffer Pool

  执行以下命令，查看Copy Buffer Pool中的buffer数量：

  ```
  select copy-release as cbuf_size from polar_cbuf;
  ```

  `polar_cbuf` 记录Copy Buffer Pool的一些统计信息，其中 `copy` 表示拷贝进Copy Buffer Pool的数量， `release` 表示释放的buffer的数量，两者相减即为当前仍在Copy Buffer Pool中的数量。

5.3. SQL性能分析

执行SQL语句时，如果出现慢SQL或SQL占用系统内存的情况，需进行具体查询分析。

查询慢SQL

查询慢SQL语句分为以下两种情况：

- 查看历史慢SQL语句：

  通过观察pg_stat_statements视图，执行时间较长的SQL语句。

  ```
  SELECT total_time / calls AS  avg, query
  FROM pg_stat_statements ORDER BY avg DESC LIMIT 10;
  ```

- 查看当前慢SQL语句。

  查询执行时间超过1秒的SQL：

```
select * from pg_stat_activity where state<>'idle' and now()-query_start > interval '1 s'
order by query_start ;
```

使用EXPLAIN查询SQL计划

通过使用**EXPLAIN**查询SQL计划，可分析SQL的快慢或资源占用情况。

使用**EXPLAIN**执行以下语句，可获得SQL的查询计划。具体语法示例如下：

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
where option can be one of:
    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

上述示例对应的命令参数选项含义如下：

参数	说明
ANALYZE	会实际执行SQL，并获得相应的查询计划，默认为FALSE。如果优化一些修改数据的SQL 需要真实的执行但是不能影响现有的数据，可以放在一个事务中，分析完成后可以直接回滚。
VERBOSE	会显示查询计划的附加信息，默认为FALSE。附加信息包括查询计划中每个节点输出的列（Output），表的SCHEMA信息，函数的SCHEMA 信息，表达式中列所属表的别名，被触发的触发器名称等。
COSTS	会显示每个计划节点的预估启动代价（找到第一个符合条件的结果的代价）和总代价，以及预估行数和每行宽度，默认为TRUE。
BUFFERS	会显示关于缓存的使用信息，默认为FALSE。该参数只能与ANALYZE 参数一起使用。缓冲区信息包括共享块（常规表或者索引块）、本地块（临时表或者索引块）和临时块（排序或者哈希等涉及到的短期存在的数据块）的命中块数，更新块数，挤出块数。
FORMAT	指定输出格式，默认为TEXT。各个格式输出的内容都是相同的，其中XML | JSON | YAML更有利于您通过程序解析SQL语句的查询计划。

视图分析

可以通过观察视图**pg_stat_statements**和**polar_stat_sql**进行分析SQL执行需占用的资源情况，这两个视图需创建相关插件，并打开开关。

```
CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION polar_stat_sql;
postgres=# alter system set polar_stat_sql.enable_stat=on;
ALTER SYSTEM
postgres=# alter system set polar_stat_sql.enable_qps_monitor=on;
ALTER SYSTEM
postgres=# select pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

- pg_stat_statements视图说明请参见pg_stat_statements。
- polar_stat_sql视图说明。

  sql级别监控视图信息包含4个部分内容：

  - 执行计划节点统计信息（扫描计划节点、join计划节点、聚集计划节点、排序计划节点、hash计划节点）。
  - 收集sql级别的CPU运行时间（分为系统态和用户态）、IO（读写字节数）、memory（请求页数，交换页数）、ipc（ipc message接受量和请求量）、CPU上下文切换（分为自愿切换和非自愿）。
  - SQL的各个执行阶段耗时统计，包括parse时间、analyze时间、rewrite时间、执行计划生成时间、执行时间。
  - latch和锁信息统计。

  polar_stat_sql视图是记录的累计数据，在执行sql之前需进行清空。

```
postgres=# select polar_stat_sql_reset();
 polar_stat_sql_reset
----------------------
(1 row)
```

5.4. CPU使用率高

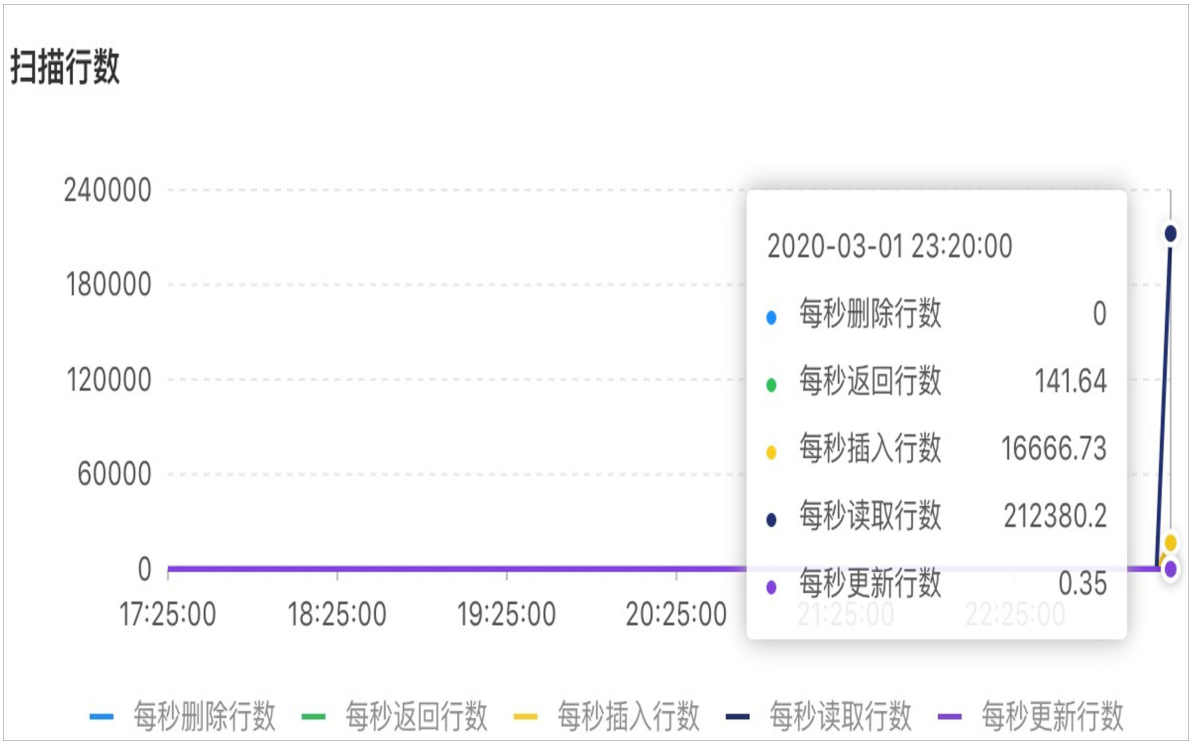
本文介绍CPU使用率的相关概念以及常见CPU使用率高的原因。

CPU相关概念

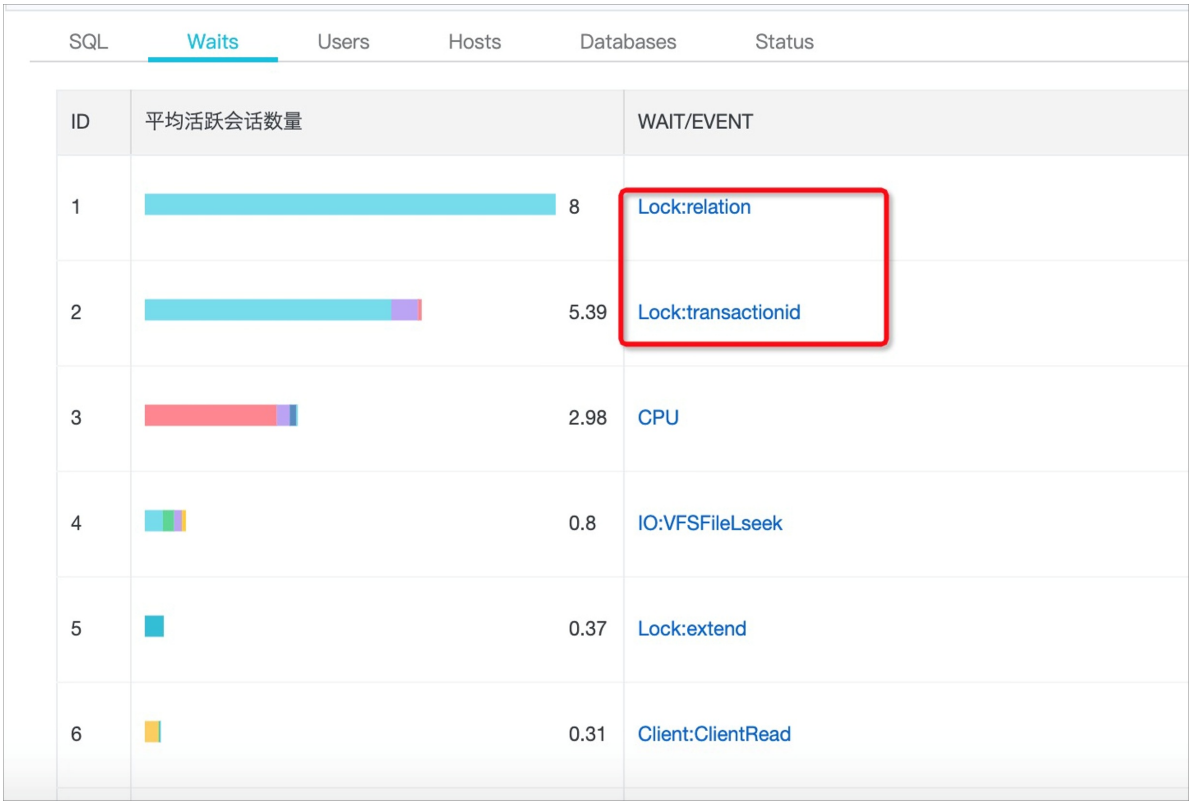
概念	说明
CPU使用率	CPU使用率指的是CPU执行工作的时间比例，包含了所有符合条件的活动的时钟周期，比如停滞等待IO而导致较高的使用率，CPU使用率被分为内核时间和用户时间。
用户时间	执行用户态程序的时间被称为用户时间。
内核时间	执行内核态代码时间为内核时间，包含系统调用，内核线程和中断的时间。
上下文切换	内核程序切换CPU让其在不同的地址空间上操作。
中断	由物理设备发送给内核的信号，通常是请求I/O服务。

常见CPU使用率高的原因

- 扫描行数突然变多，说明当前存在不合理的SQL、缺少索引或者有大量统计类SQL在执行。这种情况需要您找到问题SQL，查看并分析执行计划，对相关SQL进行优化，查看扫描行数请参见性能监控。



- 锁等待导致的相关会话阻塞造成SQL堆积，请参见性能洞察。



5.5. 高I/O

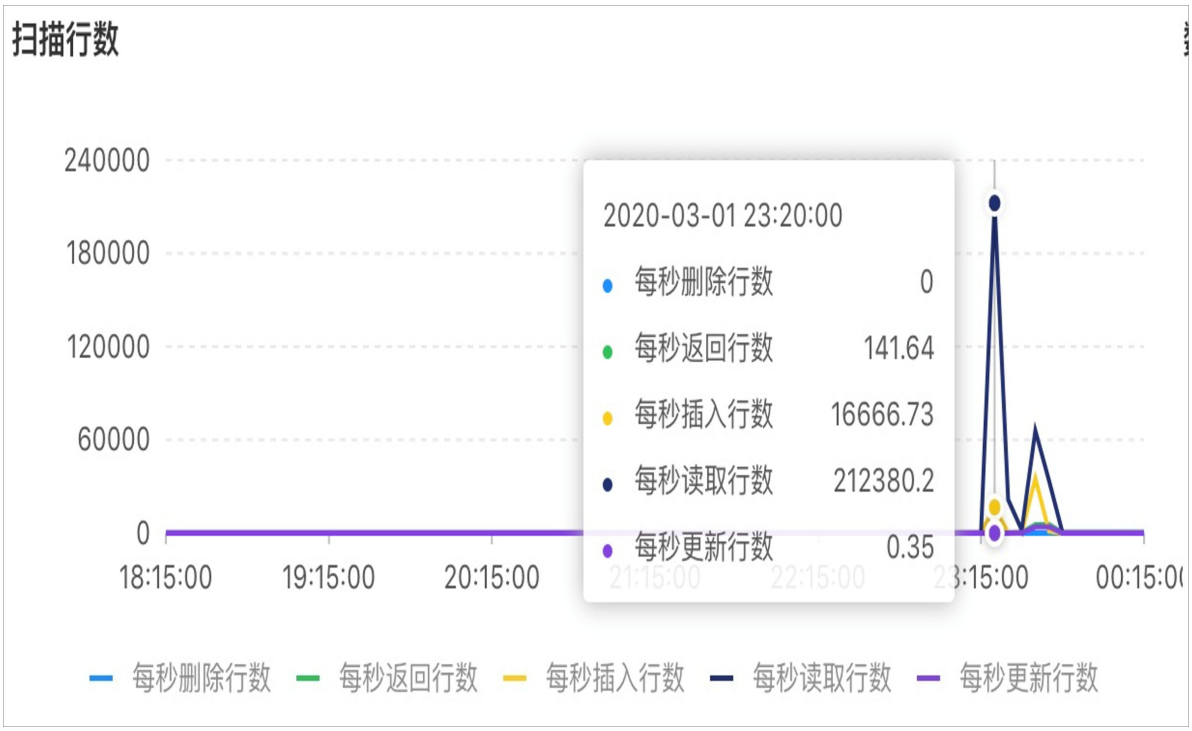
本文介绍高I/O的相关概念以及常见高I/O的原因。

I/O的相关概念

概念	说明
I/O	对磁盘进行读写的动作。
I/O延时	一个I/O操作的执行时间。
逻辑I/O	由应用程序发给文件系统的I/O。
物理I/O	由文件系统发给磁盘的I/O。
顺序I/O	顺序的从磁盘进行读写操作。
随机I/O	随机的访问磁盘进行读写操作。
同步写	需要等数据完全写入磁盘才进行返回。
异步写	无需等待数据写入磁盘进行返回，释放CPU资源。

常见高I/O的原因

- 执行了大量扫描行数多的SQL，导致shared_buffer无法缓存所有数据，需要大量的物理I/O。需要您定位问题SQL，并对问题SQL进行优化，查看问题SQL请参见性能洞察。



- vacuum动作触发的I/O高。主要原因是vacuum相关参数设置不够合理，需要您调整vacuum相关参数，例如vacuum_cost_delay、 vacuum_cost_limit等。设置参数请参见设置参数。

5.6. 网络问题

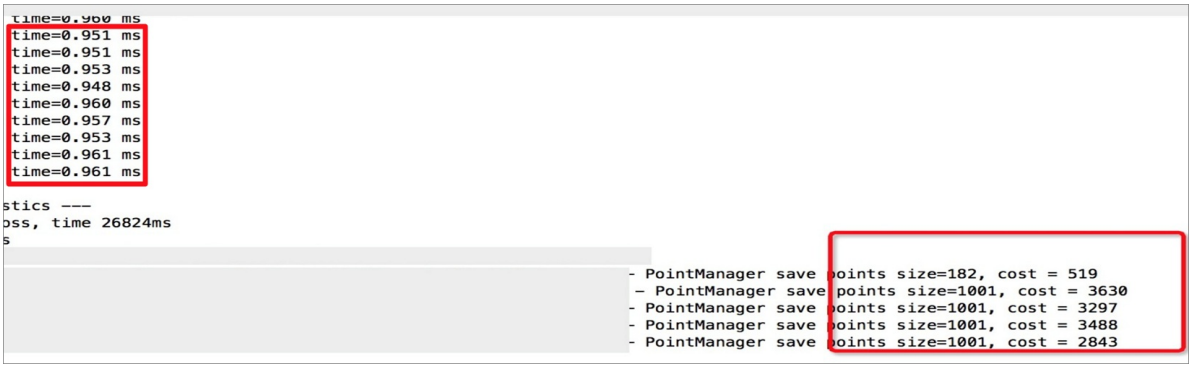
本文介绍网络问题对数据库性能的影响。

前端协议

- 简单查询：一个简单查询周期是由前端发送一条Query消息给后端进行初始化开始。这条消息包含一个用文本字符串表达SQL命令（或者一些命令）。后端根据查询命令串的内容发送一条或者更多条响应消息给前端，并且最后是一条ReadyForQuery响应消息。
- 扩展查询：在扩展协议里，前端首先发送一个Parse消息，它包含一个文本查询字符串，如果成功创建了一个命名的预备语句对象，那么它将持续到当前会话结束，一旦一个预备语句存在，就可以很使用Bind消息使之进入执行状态，Bind消息会传递相关参数，如果该语句被反复执行,服务器可能会保存创建好的计划并在后续对同一个预备语句的Bind消息中重用。之后可以用一个Execute消息执行它。最后会发出表示源SQL命令结束的CommandComplete消息。

网络延迟不同引起的性能差异

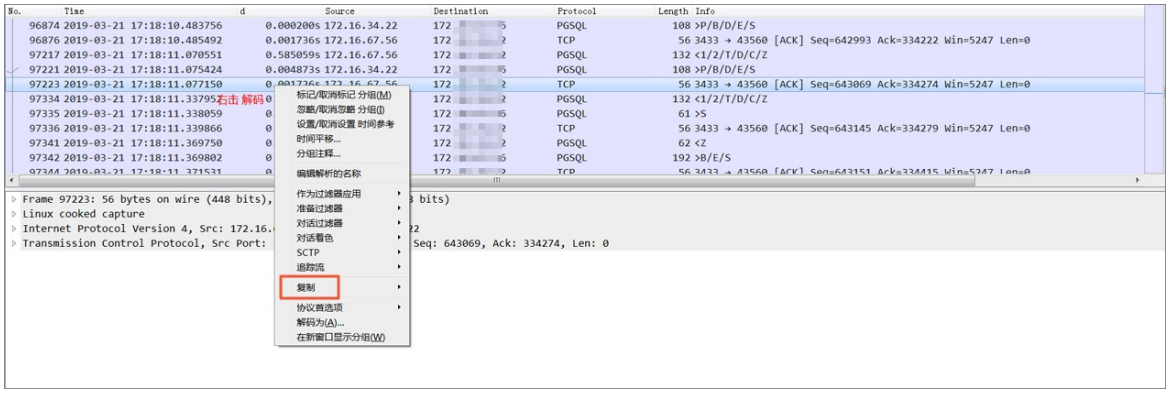
- Ping延迟1毫秒，1000条插入耗时4000毫秒。



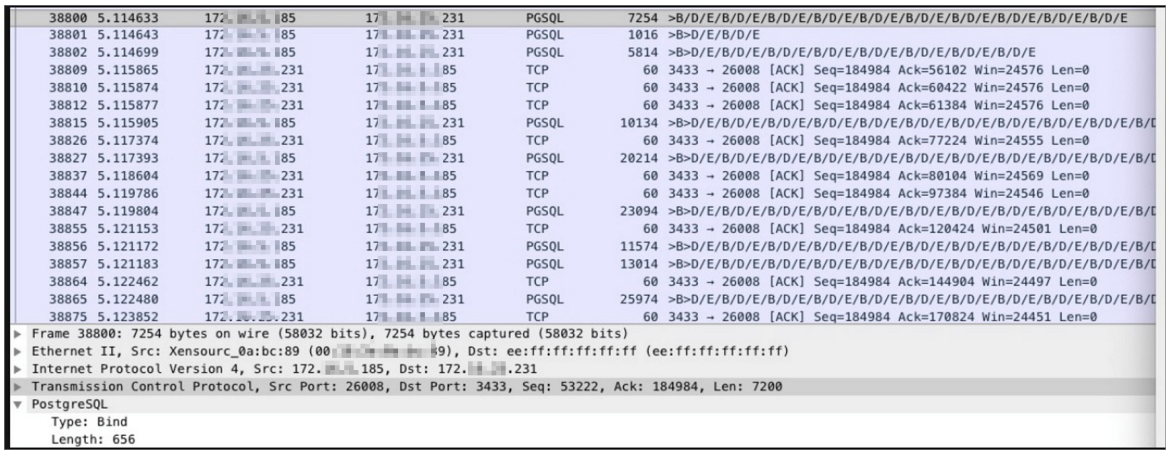
- Ping延迟0.1毫秒，1000条插入耗时500毫秒。

在插入数据时进行网络抓包，网络包示例如下。

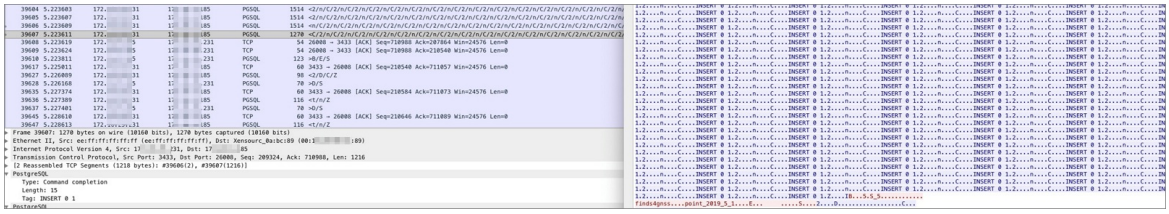
1. 在网络包中搜索 `tcpdump -i eth0 port 3433 -s 0 -w t.cap` 。



2. B代表bind（绑定）；D代表Describe（描述）；E代表execute（执行）。



3. C代表command complete（命令完成）。



通过以上测试可以得出如下结论：

- 结论：

  单行插入1条网络消耗时间就达到2毫秒，1000条INSERT语句，消耗在网络的响应时间（RT）就达到2秒。

- 解法：

  使用 `insert into values(),(),()` 的方式批量进行发送，避免网络的多次交互。

5.7. SQL调优思路

本文介绍多种SQL调优的思路。

缺少索引

索引缺失，通过执行计划发现SQL没有使用索引，查询效率低，创建索引后性能提升。示例如下：

- 添加索引前

```
postgres=# create table a(id int);
CREATE TABLE
postgres=# insert into a select generate_series(1,1000000);
INSERT 0 1000000
postgres=# explain analyze select * from a where id =10;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Gather  (cost=0.00..10301.95 rows=5642 width=4) (actual time=0.491..41.268 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on a  (cost=0.00..10301.95 rows=2351 width=4) (actual time=19.613..32.577 rows=0 loops=3)
         Filter: (id = 10)
         Rows Removed by Filter: 333333
 Planning Time: 1.089 ms
 Execution Time: 41.741 ms
(8 rows)
```

- 添加索引后

```
postgres=# create index ON a (id);
CREATE INDEX
postgres=# explain analyze select * from a where id =10;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Index Only Scan using a_id_idx on a  (cost=0.42..8.44 rows=1 width=4) (actual time=0.097..0.098 rows=1 loops=1)
   Index Cond: (id = 10)
   Heap Fetches: 1
 Planning Time: 0.986 ms
 Execution Time: 0.130 ms
(5 rows)
```

最优执行计划

执行计划并非最优执行计划，SQL执行时间未达到预期速度，通过执行计划发现SQL有调优空间，创建合适的索引。示例如下：

- 使用最优执行计划前

```
postgres=# create table test(id int,name int);
CREATE TABLE
postgres=# insert into test select random()*(id/100),random()*(id/100) from generate_series(1,100000) t(id)
postgres-# ;
INSERT 0 100000
postgres=# create index ON test (id);
CREATE INDEX
postgres=# explain analyze select * from test where id =1 and name=10;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=13.64..485.12 rows=4 width=8) (actual time=1.168..1.436 rows=6 loops=1)
   Recheck Cond: (id = 1)
   Filter: (name = 10)
   Rows Removed by Filter: 718
   Heap Blocks: exact=203
   ->  Bitmap Index Scan on test_id_idx  (cost=0.00..13.64 rows=713 width=0) (actual time=1.080..1.080 rows=724 loops=1)
         Index Cond: (id = 1)
 Planning Time: 0.381 ms
 Execution Time: 2.496 ms
(9 rows)
```

- 使用最优执行计划后

```
postgres=# create index ON test (id,name);
CREATE INDEX
postgres=# explain analyze select * from test where id =1 and name=10;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Index Only Scan using test_id_name_idx on test  (cost=0.29..17.42 rows=4 width=8) (actual time=0.080..0.085 rows=6 loops=1)
   Index Cond: ((id = 1) AND (name = 10))
   Heap Fetches: 6
 Planning Time: 1.797 ms
 Execution Time: 0.102 ms
(5 rows)
```

SQL改写

SQL执行时间慢，通过创建索引没有提升空间，需要改写SQL针对性的优化。示例如下：

- SQL改写前

```
SELECT *
FROM (
    SELECT row_.*, rownum AS rownum_
    FROM (
        SELECT DISTINCT device0_.id AS id1_1_, device0_.app_id AS app_id2_1_, device0_.app_version AS app_version3_1_,
        device0_.created_time AS created_time4_1_, device0_.fcm_token AS     fcm_token5_1_
            , device0_.member_id AS member_id12_1_, device0_.name AS name6_1_, device0_.notification AS notification7_1_,
            device0_.sys_version AS sys_version8_1_, device0_.token AS    token9_1_
            , device0_.type AS type10_1_, device0_.updated_time AS updated_time11_1_
        FROM dm_device device0_
        WHERE device0_.notification = 1
            AND device0_.token IS NOT NULL
        ORDER BY device0_.id ASC
    ) row_
    WHERE rownum <= 1000
)
WHERE rownum_ > 10

Subquery Scan on __unnamed_subquery_0  (cost=417184.72..473136.18 rows=133218 width=723) (actual time=2000.039..2888.102 rows=990
loops=1)
   Filter: (__unnamed_subquery_0.rownum_ > 10)
   Rows Removed by Filter: 10
   Buffers: shared hit=52736, temp read=33458 written=33473
   ->  Subquery Scan on row_  (cost=417184.72..468140.52 rows=399653 width=723) (actual time=2000.022..2887.974 rows=1000 loops=1)
         Filter: (rownum <= 1000)
         Rows Removed by Filter: 1190532
         Buffers: shared hit=52736, temp read=33458 written=33473
         ->  Unique  (cost=417184.72..453153.52 rows=1198960 width=715) (actual time=2000.016..2804.040 rows=1191532 loops=1)
               Buffers: shared hit=52736, temp read=33458 written=33473
               ->  Sort  (cost=417184.72..420182.12 rows=1198960 width=715) (actual time=2000.014..2416.502 rows=1191532 loops=1)
                     Sort Key: device0_.id, device0_.app_id, device0_.app_version, device0_.created_time, device0_.fcm_token,
                     device0_.member_id, device0_.name, device0_.sys_vers
ion, device0_.token, device0_.type, device0_.updated_time
                     Sort Method: external merge  Disk: 267664kB
                     Buffers: shared hit=52736, temp read=33458 written=33473
                     ->  Seq Scan on dm_device device0_  (cost=0.00..72905.11 rows=1198960 width=715) (actual time=0.100..608.593
                     rows=1191532 loops=1)
                           Filter: ((token IS NOT NULL) AND (notification = '1'::numeric))
                           Rows Removed by Filter: 421997
                           Buffers: shared hit=52736
 Planning Time: 22.127 ms
 Execution Time: 2954.789 ms
(20 rows)
```

- SQL改写后

```
SELECT *
FROM (
    SELECT row_.*, rownum AS rownum_
    FROM (
        SELECT  device0_.id AS id1_1_, device0_.app_id AS app_id2_1_, device0_.app_version AS app_version3_1_, device0_.created_time
        AS created_time4_1_, device0_.fcm_token AS     fcm_token5_1_
            , device0_.member_id AS member_id12_1_, device0_.name AS name6_1_, device0_.notification AS notification7_1_,
            device0_.sys_version AS sys_version8_1_, device0_.token AS    token9_1_
            , device0_.type AS type10_1_, device0_.updated_time AS updated_time11_1_
        FROM dm_device device0_
        WHERE device0_.notification = 1
            AND device0_.token IS NOT NULL
        ORDER BY device0_.id ASC limit 1000
    ) row_
    WHERE rownum <= 1000
)
WHERE rownum_ > 10;
                                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Subquery Scan on __unnamed_subquery_0  (cost=0.43..93.61 rows=111 width=723) (actual time=0.032..1.071 rows=990 loops=1)
   Filter: (__unnamed_subquery_0.rownum_ > 10)
   Rows Removed by Filter: 10
   Buffers: shared hit=59
   ->  Subquery Scan on row_  (cost=0.43..89.45 rows=333 width=723) (actual time=0.020..0.942 rows=1000 loops=1)
         Filter: (rownum <= 1000)
         Buffers: shared hit=59
         ->  Limit  (cost=0.43..76.95 rows=1000 width=715) (actual time=0.019..0.771 rows=1000 loops=1)
               Buffers: shared hit=59
               ->  Index Scan using dm_device_pkey on dm_device device0_  (cost=0.43..91742.47 rows=1198960 width=715) (actual
               time=0.018..0.675 rows=1000 loops=1)
                     Filter: ((token IS NOT NULL) AND (notification = '1'::numeric))
                     Rows Removed by Filter: 431
                     Buffers: shared hit=59
 Planning Time: 1.459 ms
 Execution Time: 1.147 ms
(15 rows)

Time: 2.977 ms
```