ALIBABA CLOUD

阿里云

PolarDB Oracle

性能调优指南

文档版本：20201030

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PolarDB-O体系架构

1.1. 重要组件

本文为您介绍PolarDB-O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。

架构组件	说明
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. 文件存储

本文为您介绍PolarDB-O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。	

1.4. 堆表结构

本文为您展示PolarDB-O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。
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。

结构	说明
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。

1.5. 查询机制

本节介绍PolarDB-O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。

1.6. 垃圾回收机制

PolarDB-O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. 查看统计信息

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。通过这些信息您可以评估当前数据库中可以优化的地方，这些统计信息就类似一份体检报告，告诉您哪里存在问题，哪里需要进行调整。

pg_stat_database

列名	示例值	说明
datid	13510	数据库OID
datname	postgres	数据库名
numbackends	98	访问当前数据库连接数量
xact_commit	14291309	该数据库事务提交总量
xact_rollback	0	该数据库事务回滚总量
blks_read	536888	总磁盘物理读的块数
blks_hit	261717850	在shared_buffer命中的块数
tup_returned	58521416	对于表来说是全表扫描的行数，对于索引是通过索引方法返回的索引行数，如果这个值数量明显大于tup_fetched，说明当前数据库存在大量全表扫描的情况。
tup_fetched	57193639	指通过索引返回的行数
tup_inserted	14293061	插入的行数
tup_updated	42868451	更新的行数
tup_deleted	98	删除的行数
conflicts	0	与恢复冲突取消的查询次数（只会在备库上发生）
temp_files	0	产生临时文件的数量，如果这个值很高说明work_mem需要调大
temp_bytes	0	产生临时文件的大小
deadlocks	0	死锁的数量，如果这个值很大说明业务逻辑有问题。
blk_read_time	0	数据库中花费在读取文件的时间，这个值较高说明内存较小，需要频繁的从磁盘中读入数据文件。

列名	示例值	说明
blk_write_time	0	数据库中花费在写数据文件的时间
stats_reset	2019/11/9 14:06	无

性能监控中的TPS和扫描行数的相关信息是从pg_stat_database中获取的，从以下两个视图中可以简单看出数据库的基本使用状态，对于一些明显负载的变化可以很清楚的进行定位，如何打开性能监控页面请参见性能监控。

pg_stat_user_tables

列名	示例值	说明
relid	16390	表的OID
schemaname	public	模式名称
relname	pgbench_accounts	表名
seq_scan	0	这个表进行全表扫描的次数
seq_tup_read	0	全表扫描的数据行数，如果这个值很大说明对这个表进行SQL很有可能都是全表扫描。
idx_scan	29606482	索引扫描的次数
idx_tup_fetch	29606482	通过索引扫描返回的行数
n_tup_ins	0	插入的数据行数
n_tup_upd	14803241	更新的数据行数
n_tup_del	0	删除的数据行数
n_tup_hot_upd	14638544	hot update的数据行数，这个值与n_tup_upd越接近说明update的性能较好，更新数据时不会更新索引。
n_live_tup	100012319	活着的行数量
n_dead_tup	2403437	死亡的行数量
n_mod_since_analyze	0	这个表最后一次被分析后被修改的行的估计数量
last_vacuum	无	上次手动vacuum的时间
last_autovacuum	无	上次autovacuum的时间
last_analyze	无	上次analyze的时间

列名	示例值	说明
last_autoanalyze	2019/4/9 14:12	上次自动analyze的时间
vacuum_count	0	vacuum的次数
autovacuum_count	0	autovacuum的次数
analyze_count	0	analyze的次数
autoanalyze_count	1	自动analyze的次数

- seq_scan、seq_tup_read的值很高说明该表有大量的全表扫描动作，您需要找到问题SQL进行优化。
- n_dead_tup的值很高说明该表有大量的UPDATE和DELETE操作，产生了大量的垃圾数据，您需要对该表进行vacuum动作，同时表明autovacuum参数设置也不够合理需要您进行调整。遇到类似情况您可以手动执行vacuum table操作或调整autovacuum参数触发自动vacuum动作。

pg_stat_user_indexes

列名	示例值	说明
relid	16390	表的OID
indexrelid	16404	索引的OID
schemaname	public	模式名
relname	pgbench_accounts	表名
indexrelname	pgbench_accounts_pkey	索引名
idx_scan	29606482	通过索引扫描的次数，如果这个值很小，说明这个索引很少被用到，可以考虑进行删除。
idx_tup_read	29949698	通过任意索引方法返回的索引行数
idx_tup_fetch	29606482	通过索引方法返回的数据行数

通过pg_stat_user_indexes可以知道当前数据库中哪些是用的很频繁的索引，哪些是无效索引，无效索引可以进行删除，可以减少磁盘空间的使用和提升INSERT，UPDATE，DELETE性能。

pg_statio_user_tables

列名	示例值	说明
relid	16390	表的OID
schemaname	public	模式名
relname	pgbench_accounts	表名

列名	示例值	说明
heap_blks_read	414012	从磁盘中读入表的块数
idx_heap_blks_hit	44710713	指在shared_buffer中命中表的块数
idx_blks_read	67997	从磁盘中读入索引的块数
idx_blks_hit	89424015	在shared_buffer中命中的索引的块数
toast_blks_read	无	从磁盘中读入toast表的块数
toast_blks_hit	无	指在shared_buffer中命中toast表的块数
tidx_blks_read	无	从磁盘中读入toast表索引的块数
tidx_blks_hit	无	指在shared_buffer中命中toast表索引的块数

pg_stat_bgwriter

列名	示例值	说明
checkpoints_timed	1050	指超过checkpoint_timeout的时间后触发的检查点。
checkpoints_req	1	指手动触发的检查点或者因为wal文件数量到达max_wal_size大小时也会增加。
checkpoint_write_time	659728	指从shared_buffer中write到磁盘的时间。
checkpoint_sync_time	549	指checkpoint调用fsync将脏数据同步到磁盘花费的时间。
buffers_checkpoint	122383	checkpoint写入的脏块的数量
buffers_clean	60723	通过bgwriter写入的块的数量
maxwritten_clean	583	指bgwriter单次写入超过bgwriter_lru_maxpages时停止的次数。
buffers_backend	306521	通过backend写入的块数量
buffers_backend_fsync	0	指backend需要fsync的次数
buffers_alloc	317113	被分配的缓冲区数量
stats_reset	2019-03-28 16:54:45	统计重置的时间

通过pg_stat_bgwriter视图可以判断checkpoint以及max_wal_size的相关参数是否合理。也可以判断bgwriter相关的参数是否合理。

pg_stat_statements

列名	示例值	说明
userid	10	用户ID
dbida	12917	数据库OID
queryid	4390283800491518311	SQL进行归一化后的HASH值
query	select version()	SQL归一化后的内容
calls	1	执行次数
total_time	0.208	SQL总共的执行时间
min_time	0.208	SQL最小的执行时间
max_time	0.208	SQL最大的执行时间
mean_time	0.208	SQL平均的执行时间
stddev_time	0	在该语句中花费时间的总体标准偏差，以毫秒计
rows	1	SQL返回或者影响的行数
shared_blks_hit	0	SQL在shared_buffer中命中的块数
shared_blks_read	0	从磁盘中读取的块数
shared_blks_dirtied	0	SQL语句弄脏的shared_buffer的块数
shared_blks_written	0	SQL语句写入的块数
local_blks_hit	0	临时表中命中的块数
local_blks_read	0	临时表需要读的块数
local_blks_dirtied	0	临时表弄脏的块数
local_blks_written	0	临时表写入的块数
temp_blks_read	0	从临时文件读取的块数
temp_blks_written	0	从临时文件写入的数据块数
blk_read_time	0	从磁盘读取花费的时间
blk_write_time	0	从磁盘写入花费的时间

pg_stat_activity

列名	示例值	说明
datid	12630	后端连接到的数据库OID
datname	postgres	后端连接的数据库的名称
pid	19239	后端的进程 ID
usesysid	10	登录到后端的用户的OID
usename	postgres	登录到后端的用户的名称。
application_name	psql	连接到后端的应用的名称
client_addr	无	连接到后端的客户端的IP地址
client_hostname	无	已连接的客户端的主机名
client_port	-1	客户端和后端通信的TCP端口号
backend_start	2020-02-22 18:52:16	进程被启动的时间
xact_start	2020-02-23 15:25:46	进程的当前事务被启动的时间
query_start	2020-02-23 15:25:46	当前活动查询被开始的时间
state_change	2020-02-23 15:25:46	状态（state）上一次被改变的时间
wait_event_type	无	会话的等待事件类型
wait_event	无	具体的等待事件名称
state	active	后端会话状态：active，idle，idle in transaction, idle in transaction (aborted)。
backend_xid	无	后端的事务标识符
backend_xmin	1089	后端的xmin范围
query	select 1;	查询的SQL
backend_type	client backend	autovacuum launcher、autovacuum worker、client backend、checkpointer等

pg_locks

列名	示例值	说明

列名	示例值	说明
locktype	relation	可锁对象的类型：relation、extend，page、tuple、transactionid、virtualxid、object、userlock或advisory
database	12630	锁目标存在的数据库的OID，如果目标是一个共享对象则为0，如果目标是一个事务ID则为空。
relation	11645	锁目标的关系的OID，如果目标不是一个关系或者只是关系的一部分则此列为空。
page	无	锁目标的页在关系中的页号，如果目标不是一个关系页或元组则此列为空。
tuple	无	锁目标的元组在页中的元组号，如果目标不是一个元组则此列为空。
virtualxid	无	锁目标的事务虚拟ID，如果目标不是一个虚拟事务ID则此列为空。
transactionid	无	锁目标的事务ID，如果目标不是一个事务ID则此列为空ID。
classid	无	包含锁目标的系统目录的OID，如果目标不是一个普通数据库对象则此列为空。
objid	无	锁目标在它的系统目录中的OID，如果目标不是一个普通数据库对象则为空。
objsubid	无	锁的目标列号
virtualtransaction	3/220	保持这个锁或者正在等待这个锁的事务的虚拟ID。
pid	19239	保持这个锁或者正在等待这个锁的服务器进程的PID。
mode	AccessShareLock	此进程已持有或者希望持有的锁模式。
granted	t	此进程已持有或者希望持有的锁模式。
fastpath	t	如果锁通过快速路径获得则为真，通过主锁表获得则为假。

2.2. 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。
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
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。
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 a(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 a(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 a(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 a(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 a(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 a(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) from 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='c4ca4238a0b923820dcc509a6f75849bc4ca4238a0b923820dcc509a6f75849bc4ca4238a0b923820dcc509a6f75849bc4ca4238a0b923820dcc509a6f75849b';
```

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
- 按照任意列进行搜索
- 查找的数据比较稀疏

操作符

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

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];

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。Btree、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。

  
- 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。

  
- 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
- 使用索引和删除索引的区别示例一
- 使用索引和删除索引的区别示例二

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

本文介绍为您介绍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 O
RDER 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 l
oops=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 row
s=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on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 rows=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on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on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on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on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 ro
ws=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 time=0.015..0.016 ro
ws=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..0.025 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。

参数	说明
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。

4.3. 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::real)] 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. 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。

5.2. 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。

概念	说明
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。

5.3. 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。

5.4. 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
- 添加索引后

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
- 使用最优执行计划后

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
- SQL改写后