



新手机 新应用 新娱乐

PostgreSQL 性能优化培训

Digoal.zhou

5/13/2014

<http://blog.163.com/digoal@126>



目录

- 授课环境
- SQL优化基础
- 如何让数据库输出好的执行计划
- 压力测试工具的使用和建模
- 性能分析工具的使用
- 综合优化案例

授课环境

- OS
 - CentOS 6.x x64
- DB
 - PostgreSQL 9.3.4
- Others
 - pgfincore
 - plproxy

授课环境搭建

- yum -y install lrzs sz sysstat e4fsprogs ntp readline-devel zlib zlib-devel openssl openssl-devel pam-devel libxml2-devel libxslt-devel python-devel tcl-devel gcc make smartmontools flex bison perl perl-devel perl-ExtUtils* OpenIPMI-tools systemtap-sdt-devel
- mkdir /opt/soft_bak

- cd /opt/soft_bak
- wget http://ftp.postgresql.org/pub/source/v9.3.4/postgresql-9.3.4.tar.bz2
- wget http://downloads.sourceforge.net/project/flex/flex-2.5.39.tar.bz2?r=http%3A%2F%2Fsourceforge.net%2Fprojects%2Fflex%2Ffiles%2F&ts=1397121679&use_mirror=jaist
- download http://git.postgresql.org/gitweb/?p=pgfincore.git;a=summary

- crontab -e
- -- 8 * * * * /usr/sbin/ntpdate asia.pool.ntp.org && /sbin/hwclock --systohc
- /usr/sbin/ntpdate asia.pool.ntp.org && /sbin/hwclock --systohc

- vi /etc/sysconfig/clock
- -- ZONE="Asia/Shanghai"
- UTC=false
- ARC=false

- rm /etc/localtime
- cp /usr/share/zoneinfo/PRC /etc/localtime

授课环境搭建

- vi /etc/sysconfig/i18n
 - -- LANG="en_US.UTF-8"

- vi /etc/ssh/sshd_config
 - UseDNS no
 - PubkeyAuthentication no

- vi /etc/ssh/ssh_config
 - GSSAPIAuthentication no

- vi /etc/sysctl.conf
 - kernel.shmmni = 4096
 - kernel.sem = 50100 64128000 50100 1280
 - fs.file-max = 7672460
 - net.ipv4.ip_local_port_range = 9000 65000
 - net.core.rmem_default = 1048576
 - net.core.rmem_max = 4194304
 - net.core.wmem_default = 262144
 - net.core.wmem_max = 1048576
 - net.ipv4.tcp_tw_recycle = 1

授课环境搭建

- net.ipv4.tcp_max_syn_backlog = 4096
- net.core.netdev_max_backlog = 10000
- vm.overcommit_memory = 0
- net.ipv4.ip_conntrack_max = 655360
- fs.aio-max-nr = 1048576
- net.ipv4.tcp_timestamps = 0

- sysctl -p

- vi /etc/security/limits.conf
- * soft nofile 131072
- * hard nofile 131072
- * soft nproc 131072
- * hard nproc 131072
- * soft core unlimited
- * hard core unlimited
- * soft memlock 50000000
- * hard memlock 50000000

- vi /etc/sysconfig/selinux
- SELINUX=disabled
- setenforce 0

授课环境搭建

- vi /etc/sysconfig/iptables
- -A INPUT -s 192.168.0.0/16 -j ACCEPT
- -A INPUT -s 10.0.0.0/8 -j ACCEPT
- -A INPUT -s 172.16.0.0/16 -j ACCEPT
- # or
- -A INPUT -m state --state NEW -m tcp -p tcp --dport 5432 -j ACCEPT

- service iptables restart

- cd /opt/soft_bak
- tar -jxvf flex-2.5.39.tar.bz2
- cd flex-2.5.39
- ./configure && make && make install

- useradd postgres
- cd /opt/soft_bak/
- tar -jxvf postgresql-9.3.4.tar.bz2
- tar -zxvf pgfincore.tar.gz
- mv pgfincore postgresql-9.3.4/contrib/

授课环境搭建

- vi /home/postgres/.bash_profile
- export PGPORT=5432
- export PGDATA=/home/postgres/pgdata
- export LANG=en_US.utf8
- export PGHOME=/opt/pgsql
- export LD_LIBRARY_PATH=\$PGHOME/lib:/lib64:/usr/lib64:/usr/local/lib64:/lib:/usr/lib:/usr/local/lib:\$LD_LIBRARY_PATH
- export DATE=`date +"%Y%m%d%H%M"``
- export PATH=\$PGHOME/bin:\$PATH:..
- export MANPATH=\$PGHOME/share/man:\$MANPATH
- export PGUSER=postgres
- export PGHOST=\$PGDATA
- export PGDATABASE=postgres
- alias rm='rm -i'
- alias ll='ls -lh'

授课环境搭建

- 安装systemtap环境
- http://debuginfo.centos.org/6/x86_64/
- vi /etc/yum.repos.d/CentOS-Debuginfo.repo
- enabled=1

- # uname -r
- 2.6.32-358.el6.x86_64
- yum install -y kernel-devel-2.6.32-358.el6.x86_64 kernel-debuginfo-2.6.32-358.el6.x86_64 kernel-debuginfo-common-x86_64-2.6.32-358.el6.x86_64 systemtap
- stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'
- <https://sourceware.org/systemtap/ftp/releases/> 或下载systemtap源码编译

授课环境搭建

- cd /opt/soft_bak/postgresql-9.3.4
- ./configure --prefix=/opt/pgsql9.3.4 --with-pgport=5432 --with-perl --with-tcl --with-python --with-openssl --with-pam --without-ldap --with-libxml --with-libxslt --enable-thread-safety --with-wal-blocksize=8 --with-blocksize=8 **--enable-dtrace --enable-debug --enable-cassert** && gmake world
- gmake install-world
- ln -s /opt/pgsql9.3.4 /opt/pgsql

- cd /opt/soft_bak/postgresql-9.3.4/contrib/pgfincore
- . /home/postgres/.bash_profile
- make clean
- make
- make install

- su - postgres
- initdb -D \$PGDATA -E UTF8 --locale=C -U postgres -W

- cd \$PGDATA
- vi postgresql.conf

授课环境搭建

- 调整数据库postgresql.conf参数. 打开日志, SQL统计, 跟踪, 以及性能参数, 便于优化过程中取证.
- 监听IPv4的所有IP.
- `listen_addresses = '0.0.0.0'`

- 最大允许1000个连接(测试的话100够了, 加大连接数同时需要调整shared buffer).
- `max_connections = 1000`

- 为超级用户保留13个可用连接.
- `superuser_reserved_connections = 13`

- 默认的unix socket文件放在/tmp, 修改为\$PGDATA, 以确保本地访问的安全性.
- `unix_socket_directory = ''`

- 默认的访问权限是0777, 修改为0700更安全.
- `unix_socket_permissions = 0700`

- TCP会话心跳包在Linux下面默认是2小时. 如果已经修改了系统的内核参数, 则不需要再修改这里.
- 为防止客户端和服务端之间的网络设备主动关闭空闲TCP会话, 设置以下参数.
 - `tcp_keepalives_idle = 60`
 - `tcp_keepalives_interval = 10`
 - `tcp_keepalives_count = 6`

授课环境搭建

- 大的shared_buffers需要大的checkpoint_segments,同时需要申请更多的System V共享内存资源. 并且增加共享内存管理的开销.
- 这个值不需要设的太大, 因为PostgreSQL还依赖操作系统的文件系统cache来提高读性能, 另外, 写操作频繁的数据库这个设太大反而会增加checkpoint压力.
- 在9.4版本中会增加mmap以及huge page table的支持以减少内存管理的开销.
- **shared_buffers = 512MB**

- 这个值越大, VACUUM, CREATE INDEX的操作越快, 当然大到一定程度瓶颈就不在内存了, 可能是CPU例如创建索引.
- 这个值是一个操作的内存使用上限, 而不是一次性分配出去的. 并且需要注意如果开启了autovacuum, 最大可能有autovacuum_max_workers*maintenance_work_mem的内存被系统消耗掉.
- **maintenance_work_mem = 512MB**

- 一般设置为比系统限制的略少,ulimit -a : stack size (kbytes, -s) 10240
- **max_stack_depth = 8MB**

- 手动执行vacuum操作时, 默认是没有停顿执行到底的, 为了防止VACUUM操作消耗太多数据库服务器硬件资源, 这个值是指vacuum在消耗多少资源后停顿多少时间,以便其他的操作可以使用更多的硬件资源.
- **vacuum_cost_delay = 10ms**
- **#vacuum_cost_page_hit = 1 # 0-10000 credits**
- **#vacuum_cost_page_miss = 10 # 0-10000 credits**
- **#vacuum_cost_page_dirty = 20 # 0-10000 credits**
- **vacuum_cost_limit = 10000 # 1-10000 credits**

授课环境搭建

- 默认bgwriter进程执行一次后会停顿200ms再被唤醒执行下一次操作, 当数据库的写操作很频繁的时候, 200ms可能太长, 导致其他进程需要花费过多的时间来进行bgwriter的操作. 短暂的停顿更利于将shared buffer中的脏块flush到磁盘, 降低backend 主动flush 以申请共享内存的情形. 后面使用explain时会讲到.
- **bgwriter_delay = 10ms**
- 另外还有几个和写脏块相关的参数, 即写多少脏块后开始休息.

- 如果需要做数据库WAL日志备份的话至少需要设置成archive级别, 如果需要做hot_standby那么需要设置成hot_standby, 由于这个值修改需要重启数据库, 所以先设置成hot_standby比较好. 当然hot_standby意味着WAL记录得更详细, 如果没有打算做hot_standby设置得越低性能越好.
- **wal_level = hot_standby**

- wal buffers默认是-1 根据shared_buffers的设置自动调整shared_buffers*3% .最大限制是XLOG的segment_size.
- **wal_buffers = 16384kB**

- 多少个xlog file产生后开始checkpoint操作,
- 这个值越大, 允许shared_buffer中的被频繁访问的脏数据存储得更久. 一定程度上可以提高数据库性能. 但是太大的话会导致在数据库发生checkpoint的时候需要处理更多的脏数据带来长时间的IO开销(还要考虑bgwriter的存在).
- 太小的话会导致产生更多的WAL文件 (因为full page writes=on, CHECKPOINT后的第一次块的改变要写全块, checkpoint越频繁, 越多的数据更新要写全块导致产生更多WAL).
- **checkpoint_segments = 32**

- 这个和checkpoint_segments的效果是一样的, 只是触发的条件是时间条件.
- **checkpoint_timeout = 5min**

授课环境搭建

- 归档参数的修改也需要重启数据库, 所以就先打开吧.
- `archive_mode = on`
- 这个是归档调用的命令, 我这里用date代替, 所以归档的时候调用的是输出时间而不是拷贝wal文件.
`archive_command = '/bin/date' # cp %p /arch/%f`
- 如果要做hot standby这个必须大于0, 并且修改之后要重启数据库所以先设置为32.
- 表示允许建立多少个和流复制相关的连接.
`max_wal_senders = 32`
- 这是个standby 数据库参数, 为了方便角色切换, 我一般是所有的数据库都把他设置为on 的.
`hot_standby = on`
- 这个参数是说数据库中随机的PAGE访问的开销占seq_page_cost的多少倍 , seq_page_cost默认是1. 其他的开销都是seq_page_cost的倍数.
- 这些都用于基于成本的执行计划选择. 后面讲成本因子的调教时会详细说明.
`random_page_cost = 2.0`
- CPU相关的成本因子, 如果内存足够大, 大部分数据都在内存命中的话, 可以适当调大以下参数, 使得数据块扫描的成本和CPU成本更接近.
 - `#cpu_tuple_cost = 0.01 # same scale as above`
 - `#cpu_index_tuple_cost = 0.005 # same scale as above`
 - `#cpu_operator_cost = 0.0025`

授课环境搭建

- effective_cache_size只是个度量值, 不是实际分配使用的内存值.
- 表示系统有多少内存可以作为操作系统的cache. 越大的话, 数据库越倾向使用index这种适合random访问的执行计划.
- 一般设置为内存大小减去数据库的shared_buffer再减去系统和其他软件所需的内存.
- **effective_cache_size = 12000MB**

- 下面是日志输出的配置.
- **log_destination = 'csvlog'**
- **logging_collector = on**
- **log_directory = '/home/postgres/pg_log' # 需提前创建这个目录, 并赋予相应的写权限**
- **log_truncate_on_rotation = on**
- **log_rotation_age = 1d**
- **log_rotation_size = 10MB**

- 这个参数调整的是记录执行时间超过1秒的SQL到日志中, 一般用于跟踪哪些SQL执行时间长.
- **log_min_duration_statement = 1s**

- 记录每一次checkpoint到日志中.
- **log_checkpoints = on**

授课环境搭建

- 记录锁等待超过1秒的操作,一般用于排查业务逻辑上的问题.
- `log_lock_waits = on`
- `deadlock_timeout = 1s`

- 记录连接和端口连接,可以反映短连接的问题,同时也可以作为连接审计日志.
- `log_connections = on`
- `log_disconnections = on`

- 打开代码位置信息的输出,可以反映日志信息输出自哪个代码的什么函数.
- 在会话中可以使用`\set VERBOSITY verbose`开启
- `log_error_verbosity = verbose`

- 记录DDL语句,但是需要注意的是,创建用户,修改密码的语句也会被记录,所以敏感SQL执行前建议在会话中关闭这个审计.
- `log_statement = 'ddl'`

- 这个原本是1024表示跟踪的SQL在1024的地方截断,超过1024将无法显示全SQL. 修改为2048会消耗更多的内存(基本可以忽略),不过可以显示更长的SQL.
- `track_activity_query_size = 2048`

授课环境搭建

- 默认autovacuum就是打开的, log_autovacuum_min_duration = 0记录所有的autovacuum操作.
- autovacuum = on
- log_autovacuum_min_duration = 0

- 这个模块用于收集SQL层面的统计信息, 如SQL被执行了多少次, 总共耗时, IO的耗时, 命中率等.
- 一般用于发现业务上最频繁调用的SQL是什么, 有针对性的进行SQL优化.
- shared_preload_libraries = 'pg_stat_statements'
- pg_stat_statements.max = 1000
- pg_stat_statements.track = all
- pg_stat_statements.track_utility = off

- 其他, 如果要监控IO的时间, 可以打开以下参数. 会带来较大的开销, 一般不建议打开.
- track_io_timing = on

目录

- 授课环境
- SQL优化基础
- 如何让数据库输出好的执行计划
- 压力测试工具的使用和建模
- 性能分析工具的使用
- 综合优化案例

SQL优化基本常识

- 数据库统计信息介绍
- 成本因子介绍
- explain的使用和输出详解
- 成本因子的校准
- 执行计划跟踪
- 执行计划缓存管理
- 跟踪和调试
- 函数的三态
- 多版本并发控制

数据库统计信息

- <http://www.postgresql.org/docs/9.3/static/catalog-pg-statistic.html>
- <http://www.postgresql.org/docs/9.3/static/view-pg-stats.html>
- 非继承列的统计信息
 - select ... from only tbl 使用这项统计信息
 - NULL值百分比
 - 平均字节数
 - 唯一值百分比
 - 高频值以及每个值的占比
 - 排除高值后的平均分配区间值
 - 物理存储顺序和列值分布顺性
 - 如果该列为数组类型, 统计数组元素高值以及各百分比, 空值百分比, 元素唯一值个数的平均分布区间, 平均唯一值个数.
- 继承列统计信息. (包含继承表的采样统计)
 - select ... from tbl* 使用这项统计信息.
- 表和索引: 块, 行统计信息
 - pg_class.relpages, pg_class.reltuples.

数据库统计信息

■ 列统计信息例子：

- digoal=# create table t8(id int, info int[]);
- digoal=# insert into t8 select generate_series(1,10000),'{1,1,2,2,3,3,4,4}'::int[];
- digoal=# insert into t8 select generate_series(1,1000000),'{100,200}'::int[];
- digoal=# analyze t8;
- digoal=# select * from pg_stats where attname='info' and tablename='t8';
- schemaname | public
- tablename | t8
- attname | info
- inherited | f -- 非继承列的统计信息, 不包含继承列的采样.
- null_frac | 0 -- 空值百分比
- avg_width | 29 -- 列的平均长度(字节)

数据库统计信息

数据库统计信息

- 顺性例子：
 - digoal=# create table t10(id int);
 - digoal=# insert into t10 select generate_series(1,10);
 - digoal=# select ctid,id from t10;
 - ctid | id
 - -----+---
 - (0,1) | 1
 - (0,2) | 2
 - (0,3) | 3
 - (0,4) | 4
 - (0,5) | 5
 - (0,6) | 6
 - (0,7) | 7
 - (0,8) | 8
 - (0,9) | 9
 - (0,10) | 10
 - (10 rows)
- ctid表示物理行信息, blockid, itemid.

数据库统计信息

- digoal=# analyze t10;
- digoal=# select correlation from pg_stats where attname='id' and tablename='t10';
- correlation
- -----
- 1
- (1 row)

- digoal=# truncate t10;
- digoal=# insert into t10 select generate_series(10,1,-1);
- digoal=# analyze t10;
- digoal=# select correlation from pg_stats where attname='id' and tablename='t10';
- correlation
- -----
- -1
- (1 row)

数据库统计信息

- digoal=# select ctid,id from t10;
- ctid | id
- -----+---
- (0,1) | 10
- (0,2) | 9
- (0,3) | 8
- (0,4) | 7
- (0,5) | 6
- (0,6) | 5
- (0,7) | 4
- (0,8) | 3
- (0,9) | 2
- (0,10)| 1
- (10 rows)

成本因子

- <http://www.postgresql.org/docs/9.3/static/runtime-config-query.html#RUNTIME-CONFIG-QUERY-CONSTANTS>
- src/backend/optimizer/path/costsize.c

- #seq_page_cost = 1.0 # measured on an arbitrary scale -- 连续数据块扫描的成本因子, 表示扫描1个块的成本. 例如全表扫描.
- #random_page_cost = 4.0 # same scale as above -- 离散数据块扫描的成本因子, 例如索引扫描.
- #cpu_tuple_cost = 0.01 # same scale as above -- 每一行的CPU开销
- #cpu_index_tuple_cost = 0.005 # same scale as above -- 每一个索引条目的CPU开销
- #cpu_operator_cost = 0.0025 # same scale as above -- 每一个操作符的CPU开销, 如果操作符的开销被重新定义了, 那么需要乘以这个因子.
- #effective_cache_size = 96GB # -- 评估操作系统缓存可能使用的内存大小, PostgreSQL planner用于评估索引扫描的开销, 大的值倾向使用索引扫描, 小的值倾向使用全表扫描. 一般设置为**内存大小减去shared buffer以及内核和其他软件占用的内存**.

- SSD硬盘建议random_page_cost调小到趋向seq_page_cost甚至相等.
- 如果内存极大, 或者已经把热点数据全部缓存到内存中, 那么可以把数据块相关的seq_page_cost和random_page_cost调整到趋向cpu相关的大小. 因为从内存中获取数据块成本又降了一个数量级.

EXPLAIN 输出结构

■ digoal=# explain (analyze,verbose,buffers,costs,timing) select id from t11 where id<10;

■ 节点信息

QUERY PLAN
成本信息(含启动成本,总成本)

真实时间开销(含启动时间,总时间), 输出多少行, 该节点循环次数

Bitmap Heap Scan on public.t11 (cost=2.36..18.15 rows=9 width=4) (actual time=0.013..0.014 rows=9 loops=1)

启动成本包含子节点的成本

Output: id

Recheck Cond: (t11.id < 10)

Buffers: shared hit=3

-> Bitmap Index Scan on t11_pkey (cost=0.00..2.36 rows=9 width=0) (actual time=0.006..0.006 rows=9 loops=1)

箭头表示节点信息

Index Cond: (t11.id < 10)

Buffers: shared hit=2

Total runtime: 0.046 ms

(8 rows)

planner最终目标是选择total_cost最低的执行计划

该节点的附属信息,在节点下缩进排列.

EXPLAIN 输出结构

- explain结构解说：
- plan nodes 树结构
 - 树的最底层是扫描节点
 - 从表返回行, 包括多种访问方法(索引扫描, 位图扫描, 全表扫描)
 - 从非表返回行, 包含VALUES子句, set-returning函数等
 - 扫描节点的上面可以有
 - 关联节点
 - 包括嵌套关联, hash关联, 合并关联
 - 聚合节点
 - 排序节点
 - ...等等
 - 树的最顶端对应最终的节点, 包含了这个执行计划的启动成本和最终成本, planner以降低这个最终成本为目的, 选择最优的执行计划.
 - 每个节点包含了节点类型, 节点的启动成本和总成本, 节点的行评估, 平均行字节数, 真实的执行时间等.
 - 每个节点的额外的属性以缩进的形式排列在各自节点的下面, TEXT格式输出的话, 节点的前面->符号区分.
 - 这些节点可能有多个层次, 例如关联后的结果又与其他数据合并, 那么关联节点上面还会有其他节点.

EXPLAIN输出结构

- 节点类型:
- ExplainNode@src/backend/commands/explain.c
- "Result"
- "Insert"
- "Update"
- "Delete"
- "Append"
- "Merge Append"
- "Recursive Union"
- "BitmapAnd"
- "BitmapOr"
- "Nested Loop"
- "Merge Join"
- "Hash Join"
- "Seq Scan"
- "Index Scan"
- "Index Only Scan"
- "Bitmap Index Scan"
- "Bitmap Heap Scan"
- "Tid Scan"
- "Subquery Scan"
- "Function Scan"
- "Values Scan"
- "CTE Scan"
- "WorkTable Scan"
- "Foreign Scan"
- "Materialize"
- "Sort"
- "Group"
- "Aggregate"
- "WindowAgg"
- "Unique"
- "Sorted"
- "HashSetOp"
- "LockRows"
- "Limit"
- "Hash"
-
- PostgreSQL选择总成本最低的节点组合作为最终的执行计划.

EXPLAIN语法

- EXPLAIN [(option [, ...])] statement
- EXPLAIN [ANALYZE] [VERBOSE] statement

- where option can be one of:
 - ANALYZE [boolean] -- 执行statement, 得到真实的运行时间以及统计信息.
 - VERBOSE [boolean] -- 输出详细信息, 如列,schema,trigger等信息. 默认关闭.
 - COSTS [boolean] -- 输出根据成本因子计算得出的cost值, 默认打开.(分为该节点输出第一行前的成本以及输出所有行的成本.)
 - src/backend/optimizer/path/costsize.c
 - BUFFERS [boolean] -- 输出本次QUERY shared/local/TEMP blocks的信息. The number of shared blocks hit, read, dirtied, and written, the number of local blocks hit, read, dirtied, and written, and the number of temp blocks read and written.
 - -- 包括命中/未命中读数据块, 产生的脏数据块, 写出了多少QUERY开始前的脏数据块. (需打开analyze, TEXT模式只输出非0项, "计数包含所有子节点的计数".)
 - TIMING [boolean] -- 输出每个节点的真实的时间开销, 总时间不包含网络开销,parser,rewriter,planer开销, (需打开analyze)
 - FORMAT { TEXT | XML | JSON | YAML } -- 输出格式, 默认TEXT.

- 需要特别注意analyze的使用, 会真的执行被评估的SQL, 所以一般不要使用, 特别是DML, 如果要使用的话, 请放在事务中使用并回滚事务.
- BEGIN;
- EXPLAIN ANALYZE QUERY;
- ROLLBACK;

EXPLAIN语法

- COSTS讲解
- 每个节点的成本输出分为两个部分显示, 各种节点的成本计算方法参考 `src/backend/optimizer/path/costsize.c`
- `startup_cost`:
 - 输出该节点的第一行前的成本(首先要包含所有子节点的成本), 例如排序节点或者聚合节点, 那么在输出第一行前需要排序或者聚合操作, 因此一般`startup_cost`大于0.
 - 有子节点的节点, 启动成本一般都大于0, 因为它包含了子节点的成本.
 - 还需要注意index scan和bitmap index scan的启动成本的差别. index scan节点输出的是tuple, 要先扫描索引页因此启动成本大于0, bitmap index scan 节点输出的是索引条目, 并非行的数据, 因此启动成本等于0 (或者说Bitmap Index Scan节点和Bitmap Heap Scan是一对的, 启动成本算在Bitmap Heap Scan上).
 - 对于IndexOnlyScan节点, 虽然是从index输出结果, 但是还要先检查visibility MAP, 因此`startup_cost`也大于0. 但是, 它的启动成本计算并未计入这部分开销. 而是和普通的index scan计算方法一样.
 - `src/backend/executor/nodeIndexonlyscan.c`
 - `src/backend/utils/adt/selfuncs.c` -- 索引访问成本计算函数
- `total_cost`:
 - 包含`startup_cost`, 以及输出所有行需要的成本.
 - 使用`limit`或者在`exists`子句中, 真实的成本将使用一个插值.
 - $$\text{actual_cost} = \text{startup_cost} + (\text{total_cost} - \text{startup_cost}) * \text{tuples_to_fetch} / \text{path->rows};$$

EXPLAIN语法

- 例子1：
 - digoal=# create table t11(id int primary key, info text);
 - digoal=# insert into t11 select generate_series(1,100000),'test';
 - digoal=# explain select * from t11 where id=1;

QUERY PLAN

- -----
 - Index Scan using t11_pkey on t11 (cost=0.29..4.31 rows=1 width=9) -- Index Scan节点输出heap tuple, 启动成本是0.29, 因为输出第一行前需要先扫描索引, 从index entry定位到heapID和itemID, 启动成本正是扫描索引的成本.
 - Index Cond: (id = 1)
 - (2 rows)
- 此例startup_cost计算方法
 - btcostestimate@src/backend/utils/adt/selfuncs.c

EXPLAIN语法

```
■ if (index->tuples > 1)      /* avoid computing log(0) */  
■ {  
■     descentCost = ceil(log(index->tuples) / log(2.0)) * cpu_operator_cost;  
■     costs.indexStartupCost += descentCost;  
■     costs.indexTotalCost += costs.num_sa_scans * descentCost;  
■ }  
■ descentCost = (index->tree_height + 1) * 50.0 * cpu_operator_cost;  
■ costs.indexStartupCost += descentCost;  
■ costs.indexTotalCost += costs.num_sa_scans * descentCost;
```

EXPLAIN语法

- digoal=# select reltuples from pg_class where relname='t11_pkey';
■ 100000
- digoal=# show cpu_operator_cost;
■ 0.0025
- digoal=# select (log(100000)/log(2.0))*0.0025; -- $\text{ceil}(\log(\text{index-} \gt \text{tuples}) / \log(2.0)) * \text{cpu_operator_cost}$
■ 0.041524101186092
- digoal=# select (1+1)*50*0.0025; -- $(\text{index-} \gt \text{tree_height} + 1) * 50.0 * \text{cpu_operator_cost};$
■ 0.2500
- digoal=# select 0.041524101186092+0.2500;
■ 0.291524101186092

EXPLAIN语法

■ 例子2：

■ digoal=# explain select id from t11 where id=1;

■ QUERY PLAN

■ Index Only Scan using t11_pkey on t11 (cost=0.29..4.31 rows=1 width=4) -- 启动成本和index scan计算方法一致,但是由于索引页不存储行的版本信息,输出前还需要VM扫描,只有不在VM中的数据块是需要查看版本信息的.

■ -- PostgreSQL启动成本并没有包含VM扫描的部分.

■ Index Cond: (id = 1)

■ (2 rows)

■ digoal=# explain (analyze,verbose,costs,buffers,timing) select id from t11 where id<10000;

■ QUERY PLAN

■ Index Only Scan using t11_pkey on public.t11 (cost=0.29..204.20 rows=9995 width=4) (actual time=0.088..3.623 rows=9999 loops=1)

■ Output: id

■ Index Cond: (t11.id < 10000)

■ Heap Fetches: 9999 -- 在没有vm文件之前,需要fetch所有的heap page. 这个开销实际上并不小.

■ Buffers: shared hit=22

■ Total runtime: 4.748 ms

■ (6 rows)

EXPLAIN语法

- 例子2：
- digoal=# vacuum ANALYZE t11; -- 初次创建的表, 需要使用vacuum生成vm文件, 或者等待自动vacuum.
- VACUUM
- digoal=# explain (analyze,verbose,costs,buffers,timing) select id from t11 where id<10000;
- QUERY PLAN
- -----
- Index Only Scan using t11_pkey on public.t11 (cost=0.29..191.64 rows=10020 width=4) (actual time=0.088..2.597 rows=9999 loops=1)
- Output: id
- Index Cond: (t11.id < 10000)
- Heap Fetches: 0 -- 生成vm文件之后, 通过查看vm得到所有的页都是可见的, 所以不需要fetch heap后取得行版本信息. 实际运行时间下降.
- Buffers: shared hit=9
- Total runtime: 3.634 ms
- (6 rows)

EXPLAIN语法

- 例子2：
- Index Only Scan在没有VM文件的情况下,速度比Index Scan要慢,因为要扫描所有的Heap page.
- digoal=# explain (analyze,verbose,costs,buffers,timing) select * from t11 where id<10000;

QUERY PLAN

- -----
- Index Scan using t11_pkey on public.t11 (cost=0.29..200.78 rows=9799 width=9) (actual time=0.074..3.530 rows=9999 loops=1)
 - Output: id, info
 - Index Cond: (t11.id < 10000)
 - Buffers: shared hit=22
 - Total runtime: **4.677 ms**
 - (5 rows)

EXPLAIN语法

- 例子3：
 - bitmap index scan节点直接从索引输出索引条目(包含heap page id), 上层节点按照heap page id排序后顺序获取heap tuple.
 - bitmap heap scan节点的启动成本包含子节点的成本以及index qual cost. 参src/backend/optimizer/path/costsize.c
 - 启动成本算在bitmap heap scan 节点上并不影响整体的执行计划, 因此在此bitmap index scan节点不计算/显示启动成本.
 - digoal=# set enable_indexscan=off;
 - SET
 - digoal=# explain (analyze,verbose,buffers,costs,timing) select id from t11 where id<10000;
 - **QUERY PLAN**

 - Bitmap Heap Scan on public.t11 (cost=93.95..354.20 rows=10020 width=4) (actual time=1.447..3.907 rows=9999 loops=1) -- 排序heap_page后, 顺序扫描heap page取数据.
 - Output: id
 - Recheck Cond: (t11.id < 10000)
 - Buffers: shared hit=22
 - -> Bitmap Index Scan on t11_pkey (cost=0.00..91.44 rows=10020 width=0) (actual time=1.370..1.370 rows=9999 loops=1) -- 输出索引条目
 - Index Cond: (t11.id < 10000)
 - Buffers: shared hit=8
 - Total runtime: 4.931 ms
 - (8 rows)

EXPLAIN语法

- 关于bitmap index scan startup_cost=0, TomLane的解释：
 - We don't bother to store/show the indexscan's startup cost in such cases,
 - since it has no effect whatsoever on subsequent planning: the total cost
 - of the indexscan will go into the parent's startup cost anyway.
- 因为bitmap heap scan和bitmap index scan 是成对出现的, 所以把startup cost计算到bitmap heap scan节点并不影响总体执行计划的选择.

EXPLAIN语法

- 例子4：
 - Seq Scan节点, 从Heap Page直接输出行, 因此没有启动成本.
 - digoal=# set enable_bitmapscan=off;
 - SET
 - digoal=# explain select id from t11 where id=1;
 - QUERY PLAN
 - -----
 - Seq Scan on t11 (cost=0.00..1385.00 rows=1 width=4) -- 全表扫描不计启动成本, 因为直接从heap page输出.
 - Filter: (id = 1)
 - (2 rows)

EXPLAIN语法

- BUFFERS讲解
 - Shared Hit Blocks -- 共享块包含表或索引或序列的数据块
 - Shared Read Blocks
 - Shared Dirtied Blocks
 - Shared Written Blocks
 - Local Hit Blocks -- 本地块包含临时表或索引的数据块
 - Local Read Blocks
 - Local Dirtied Blocks
 - Local Written Blocks
 - Temp Read Blocks -- 临时块包含短暂生存周期的块, 例如排序,hash,物化节点产生的work data.
 - Temp Written Blocks
- Hit Blocks, 表示在数据库shared buffer缓存中命中的数据块.
- Read Blocks, 表示未在数据库shared buffer缓存中命中的数据块, 但是有可能在OS Cache中命中.
- Dirtied Blocks, 表示之前在shared buffer中未修改过的数据块, 现在被当前QUERY修改了, 在shared buffer中变成脏块.
- Written Blocks, 表示之前在shared buffer中已经修改过的数据块, 现在被当前QUERY flush到磁盘, 并驱逐出shared buffer.

EXPLAIN语法

- 使用json格式输出详尽的报告(包含值为0的输出)
- digoal=# explain (analyze,verbose,buffers,costs,timing,format json)
select id from t11 where id<10000;
- QUERY PLAN
- -----
- [+
- { +
- "Plan": { +
- "Node Type": "Bitmap Heap Scan", +
- "Relation Name": "t11", +
- "Schema": "public", +
- "Alias": "t11", +
- "Startup Cost": 93.95, +
- "Total Cost": 354.20, +
- "Plan Rows": 10020, +
- "Plan Width": 4, +
- "Actual Startup Time": 1.343, +
- "Actual Total Time": 3.627, +
- "Actual Rows": 9999, +
- "Actual Loops": 1, +
- "Output": ["id"], +
- "Recheck Cond": "(t11.id < 10000)", +
- "Rows Removed by Index Recheck": 0, +
- "Shared Hit Blocks": 22, +
- "Shared Read Blocks": 0, +
- "Shared Dirtied Blocks": 0, +
- "Shared Written Blocks": 0, +
- "Local Hit Blocks": 0, +
- "Local Read Blocks": 0, +
- "Local Dirtied Blocks": 0, +
- "Local Written Blocks": 0, +
- "Temp Read Blocks": 0, +
- "Temp Written Blocks": 0, +

EXPLAIN语法

- "I/O Read Time": 0.000, +
- "I/O Write Time": 0.000, +
- "Plans": [+
- { +
- "Node Type": "Bitmap Index Scan", +
- "Parent Relationship": "Outer", +
- "Index Name": "t11_pkey", +
- "Startup Cost": 0.00, +
- "Total Cost": 91.44, +
- "Plan Rows": 10020, +
- "Plan Width": 0, +
- "Actual Startup Time": 1.282, +
- "Actual Total Time": 1.282, +
- "Actual Rows": 9999, +
- "Actual Loops": 1, +
- "Index Cond": "(t11.id < 10000)", +
- "Shared Hit Blocks": 8, +
- "Shared Read Blocks": 0, +
- "Shared Dirlted Blocks": 0, +

EXPLAIN语法

```
■ "Shared Written Blocks": 0,      +
■ "Local Hit Blocks": 0,      +
■ "Local Read Blocks": 0,      +
■ "Local Dirtied Blocks": 0,      +
■ "Local Written Blocks": 0,      +
■ "Temp Read Blocks": 0,      +
■ "Temp Written Blocks": 0,      +
■ "I/O Read Time": 0.000,      +
■ "I/O Write Time": 0.000      +
}
]
},
"Triggers": [
],
"Total Runtime": 4.678      +
```

+ }

] }

(1 row)

EXPLAIN语法

- TIMING讲解
- 如果打开的话，则包含每个节点执行的启动时间和总时间。启动时间含义和启动成本差不多。
- 如果关闭的话，则只包含整个被评估的SQL的执行时间 (total_time不包含parsing,rewriting,planing的时间以及网络数据传输的时间，因为analyze并不会传输数据，也不包含after触发器的执行时间。)



EXPLAIN输出信息的解读

- 综合例子1:
 - digoal=# explain (analyze, verbose, costs, buffers, timing) select count(*) from tbl_cost_align;
 - ### QUERY PLAN
 - Aggregate (cost=220643.00..220643.01 rows=1 width=0) (actual time=4637.754..4637.754 rows=1 loops=1)
 - Output: count(*) -- 这个节点的输出, 聚合, 输出第一行前的开销是220643.00.
 - -- 聚合的开销=220643.00 - 195393.00
 - Buffers: shared hit=4925 read=89468 -- 包含子节点的BUFFER统计项
 - -> Seq Scan on postgres.tbl_cost_align (cost=0.00..195393.00 rows=10100000 width=0) (actual time=0.018..3119.291 rows=10100000 loops=1) -- 这个节点的类型(全表扫描)
 - -- 0.00表示输出第一行前的成本, 如这里输出第一行前不需要排序为0.00. 后面是这个节点真实的时间.
 - Output: id, info, crt_time -- 这个节点输出的列
 - Buffers: shared hit=4925 read=89468 -- 这个节点的shared buffer命中4925个page, 从磁盘读取89468个page(如果shared buffer够大, 第二次执行的时候应该全部hit.)
 - Total runtime: 4637.805 ms -- 总的执行时间 (不包含parsing,rewriting,planing的时间以及网络数据传输的时间, 也不包含after触发器的执行时间.)
 - (7 rows)

EXPLAIN输出信息的解读

- 组合行集例子：
- digoal=# explain (analyze, verbose, costs, buffers, timing) select 1 **union** select 1; -- union去重
- QUERY PLAN
- **Unique** (cost=0.05..0.06 rows=2 width=0) (actual time=0.049..0.051 rows=1 loops=1)
- Output: (1)
- Buffers: shared hit=3
- -> **Sort** (cost=0.05..0.06 rows=2 width=0) (actual time=0.047..0.047 rows=2 loops=1)
 - Output: (1)
 - Sort Key: (1)
 - Sort Method: quicksort Memory: 25kB
 - Buffers: shared hit=3
 - -> **Append** (cost=0.00..0.04 rows=2 width=0) (actual time=0.006..0.007 rows=2 loops=1)
 - -> **Result** (cost=0.00..0.01 rows=1 width=0) (actual time=0.003..0.003 rows=1 loops=1)
 - Output: 1
 - -> **Result** (cost=0.00..0.01 rows=1 width=0) (actual time=0.000..0.000 rows=1 loops=1)
 - Output: 1
 - Total runtime: 0.136 ms
 - (14 rows)

EXPLAIN输出信息的解读

- 嵌套连接例子
- nested loop join: The right relation is scanned once for every row found in the left relation. This strategy is easy to implement but can be very time consuming. (However, if the right relation can be scanned with an index scan, this can be a good strategy. It is possible to use values from the current row of the left relation as keys for the index scan of the right.)

- for tuple in 外(左)表查询 loop
- 内(右)表查询(根据左表查询得到的行作为右表查询的条件依次输出最终结果)
- end loop;

- 适合内(右)表的关联列发生在唯一键值列或者主键列上的情况.

EXPLAIN输出信息的解读

- 嵌套连接例子
- digoal=# explain (analyze, verbose, costs, buffers, timing) select f.* from f,p where f.p_id=p.id and f.p_id<10;
- **QUERY PLAN**
-
- Nested Loop (cost=0.57..22.29 rows=9 width=49) (actual time=0.011..0.042 rows=9 loops=1) -- 计算成本时内表因为要多次扫描,所以要乘以循环次数(2.19*9)
 - Output: f.id, f.p_id, f.info, f.crt_time
 - Buffers: shared hit=31
 - -> Index Scan using idx_f_1 on postgres.f (cost=0.29..2.45 rows=9 width=49) (actual time=0.005..0.010 rows=9 loops=1) -- 外(左)表
 - Output: f.id, f.p_id, f.info, f.crt_time
 - Index Cond: (f.p_id < 10)
 - Buffers: shared hit=4
 - -> Index Only Scan using p_pkey on postgres.p (cost=0.29..2.19 rows=1 width=4) (actual time=0.002..0.003 rows=1 loops=9) -- 内(右)表
 - Output: p.id
 - Index Cond: (p.id = f.p_id)
 - Heap Fetches: 9
 - Buffers: shared hit=27
 - Total runtime: 0.072 ms
 - (13 rows)

EXPLAIN输出信息的解读

- 嵌套循环例子2
- EXPLAIN SELECT *
- FROM tenk1 t1, tenk2 t2
- WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

■ QUERY PLAN

-
- Nested Loop (cost=4.65..**49.46** rows=33 width=488)
 - Join Filter: (t1.hundred < t2.hundred)
 - -> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244)
 - Recheck Cond: (unique1 < 10)
 - -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.36 rows=10 width=0)
 - Index Cond: (unique1 < 10)
 - -> Materialize (cost=0.29..**8.51** rows=10 width=244) -- 物化节点, 小于work_mem时全部塞内存, 大时需要用到磁盘. 这里虽然是嵌套循环, 但是下面的索引扫描只有一次, 循环10次是在物化节点, 物化节点计算循环成本时, 只计算内存操作的成本.
 - -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..8.46 rows=10 width=244)
 - Index Cond: (unique2 < 10)

EXPLAIN输出信息的解读

- 物化节点的成本

```
/*
 * Whether spilling or not, charge 2x cpu_operator_cost per tuple to
 * reflect bookkeeping overhead. (This rate must be more than what
 * cost_rescan charges for materialize, ie, cpu_operator_cost per tuple;
 * if it is exactly the same then there will be a cost tie between
 * nestloop with A outer, materialized B inner and nestloop with B outer,
 * materialized A inner. The extra cost ensures we'll prefer
 * materializing the smaller rel.) Note that this is normally a good deal
 * less than cpu_tuple_cost; which is OK because a Material plan node
 * doesn't do qual-checking or projection, so it's got less overhead than
 * most plan nodes.
 */
run_cost += 2 * cpu_operator_cost * tuples;
```

EXPLAIN输出信息的解读

- 哈希连接例子
- hash join: the right relation is first scanned and loaded into a hash table, using its join attributes as hash keys. Next the left relation is scanned and the appropriate values of every row found are used as hash keys to locate the matching rows in the table.

- 首先内(右)表扫描加载到内存HASH表, hash key为JOIN列.
- 然后外(左)表扫描, 并与内存中的HASH表进行关联, 输出最终结果.

EXPLAIN输出信息的解读

- 哈希连接例子
- digoal=# explain (analyze, verbose, costs, buffers, timing) select f.* from f,p where f.p_id=p.id and f.p_id<10;
- Hash Join (cost=2.56..234.15 rows=9 width=49) (actual time=0.047..3.905 rows=9 loops=1)
 - Output: f.id, f.p_id, f.info, f.crt_time
 - Hash Cond: (p.id = f.p_id) **-- HASH join key, f.p_id.**
 - Buffers: shared hit=98
 - -> Seq Scan on postgres.p (cost=0.00..194.00 rows=10000 width=4) (actual time=0.014..2.016 rows=10000 loops=1) **-- 外(左)表**
 - Output: p.id, p.info, p.crt_time
 - Buffers: shared hit=94
 - -> Hash (cost=2.45..2.45 rows=9 width=49) (actual time=0.017..0.017 rows=9 loops=1)
 - Output: f.id, f.p_id, f.info, f.crt_time
 - Buckets: 1024 Batches: 1 Memory Usage: 1kB **-- 内(右)表加载到内存, hash key是join key f.p_id, 如果Batches大于1则会用到临时文件**
 - Buffers: shared hit=4
 - -> Index Scan using idx_f_1 on postgres.f (cost=0.29..2.45 rows=9 width=49) (actual time=0.005..0.012 rows=9 loops=1) **-- 内(右)表**
 - Output: f.id, f.p_id, f.info, f.crt_time
 - Index Cond: (f.p_id < 10)
 - Buffers: shared hit=4
 - Total runtime: 3.954 ms
 - (16 rows)

EXPLAIN输出信息的解读

- 合并连接例子
- merge join: Each relation is sorted on the join attributes before the join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is more attractive because each relation has to be scanned only once. The required sorting might be achieved either by an explicit sort step, or by scanning the relation in the proper order using an index on the join key.

- 首先两个JOIN的表根据join key进行排序
- 然后根据join key的排序顺序并行扫描两个表进行匹配输出最终结果.
- 适合大表并且索引列进行关联的情况.

EXPLAIN输出信息的解读

- 合并连接例子
 - ```
digoal=# explain (analyze, verbose, costs, buffers, timing) select f.* from f,p where f.p_id=p.id and f.p_id<10;
```
  - QUERY PLAN
  - Merge Join (cost=0.57..301.85 rows=9 width=49) (actual time=0.030..0.049 rows=9 loops=1)
    - Output: f.id, f.p\_id, f.info, f.crt\_time
    - Merge Cond: (f.p\_id = p.id)
    - Buffers: shared hit=8
    - -> Index Scan using idx\_f\_1 on postgres.f (cost=0.29..2.45 rows=9 width=49) (actual time=0.005..0.012 rows=9 loops=1)
      - Output: f.id, f.p\_id, f.info, f.crt\_time
      - Index Cond: (f.p\_id < 10)
      - Buffers: shared hit=4
    - -> Index Only Scan using p\_pkey on postgres.p (cost=0.29..274.29 rows=10000 width=4) (actual time=0.017..0.022 rows=10 loops=1)
      - Output: p.id
      - Heap Fetches: 10
      - Buffers: shared hit=4
    - Total runtime: 0.118 ms
    - (13 rows)

# EXPLAIN输出信息的解读

- 合并连接例子
- 如果全表扫描+排序的成本低于使用索引扫描的成本,那么将会选择全表扫描+排序的方式.
- EXPLAIN SELECT \*  
FROM tenk1 t1, onek t2  
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
  
- QUERY PLAN

---
- Merge Join (cost=198.11..268.19 rows=10 width=488)
  - Merge Cond: (t1.unique2 = t2.unique2)
  - -> Index Scan using tenk1\_unique2 on tenk1 t1 (cost=0.29..656.28 rows=101 width=244)
    - Filter: (unique1 < 100)
    - -> Sort (cost=197.83..200.33 rows=1000 width=244) -- 成本200.33
      - Sort Key: t2.unique2
      - -> Seq Scan on onek t2 (cost=0.00..148.00 rows=1000 width=244) -- 全表扫描后排序

# EXPLAIN输出信息的解读

- 合并连接例子
- 关闭排序功能.
- SET enable\_sort = off;
  
- EXPLAIN SELECT \*
- FROM tenk1 t1, onek t2
- WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
  
- **QUERY PLAN**
- 
- Merge Join (cost=0.56..292.65 rows=10 width=488)
- Merge Cond: (t1.unique2 = t2.unique2)
- -> Index Scan using tenk1\_unique2 on tenk1 t1 (cost=0.29..656.28 rows=101 width=244)
- Filter: (unique1 < 100)
- -> Index Scan using onek\_unique2 on onek t2 (cost=0.28..224.79 rows=1000 width=244) -- 变成走索引了. 成本224.79, 所以没有强制关闭排序的话, 会走全表+排序的方式.

# EXPLAIN输出信息的解读

- 外连接(Outer join)中: 查看执行计划时, 需要注意Join Filter和Filter的区别

- digoal=# create table j1(id int, info text);
- digoal=# create table j2(id int, info text);
- digoal=# insert into j1 values (1,'j1');
- digoal=# insert into j2 values (1,'j2');
- digoal=# explain select j1.\* , j2.\* from j1 left join j2 on (j1.id=j2.id and j1.info='no');
- QUERY PLAN



- 
- Merge Left Join (cost=728.10..2905.54 rows=4960 width=72)
- Merge Cond: (j1.id = j2.id)
- **Join Filter:** (j1.info = 'no'::text) -- 外连接, ON子句中的过滤条件作为Join Filter, 外表即使不符合这个条件的行也被显示.
- -> Sort (cost=364.05..376.45 rows=4960 width=36)
  - Sort Key: j1.id
  - -> Seq Scan on j1 (cost=0.00..59.60 rows=4960 width=36)
- -> Sort (cost=364.05..376.45 rows=4960 width=36)
  - Sort Key: j2.id
  - -> Seq Scan on j2 (cost=0.00..59.60 rows=4960 width=36)
- (9 rows)

# EXPLAIN输出信息的解读

- digoal=# select j1.\*,j2.\* from j1 left join j2 on (j1.id=j2.id and j1.info='no');
  - id | info | id | info
  - -----+-----+-----
  - 1 | j1 | |
  - (1 row)
  - digoal=# explain select j1.\*,j2.\* from j1 left join j2 on (j1.id=j2.id) where j1.info='no';
  - QUERY PLAN
  - -----
  - Hash Right Join (cost=72.31..162.91 rows=620 width=72)
  - Hash Cond: (j2.id = j1.id)
  - -> Seq Scan on j2 (cost=0.00..59.60 rows=4960 width=36)
  - -> Hash (cost=72.00..72.00 rows=25 width=36)
  - -> Seq Scan on j1 (cost=0.00..72.00 rows=25 width=36)
  - Filter: (info = 'no'::text) -- Filter 过滤条件, 外表也将被过滤, 因为这个节点是关联节点的子节点.
  - (6 rows)
  - digoal=# select j1.\*,j2.\* from j1 left join j2 on (j1.id=j2.id) where j1.info='no';
  - id | info | id | info
  - -----+-----+-----
  - (0 rows)
- 
- The diagram illustrates a query execution plan. A red bracket labeled "Filter" points to the "Filter" node in the plan, which corresponds to the WHERE clause in the SQL statement. The plan shows a Hash Right Join with two child nodes: a Hash Cond node and a Seq Scan on j2. The Hash Cond node has a child Seq Scan on j2, and the Hash node has a child Seq Scan on j1. The Filter node is positioned above the Hash Cond node, indicating it is a child of both the join and the filter condition.

# EXPLAIN输出信息的解读

- 使用inner join 时, Join Filter和Filter效果一样. 不符合条件的都会被过滤掉.

■ digoal=# explain select j1.\* ,j2.\* from j1 inner join j2 on (j1.id=j2.id and j1.info='no');

■ QUERY PLAN

-----

■ Merge Join (cost=110.88..117.43 rows=37 width=72)

■   Merge Cond: (j1.id = j2.id)

■   -> Sort (cost=25.45..25.47 rows=6 width=36)

■    Sort Key: j1.id

■    -> Seq Scan on j1 (cost=0.00..25.38 rows=6 width=36)

■       Filter: (info = 'no'::text)

■   -> Sort (cost=85.43..88.50 rows=1230 width=36)

■    Sort Key: j2.id

■    -> Seq Scan on j2 (cost=0.00..22.30 rows=1230 width=36)

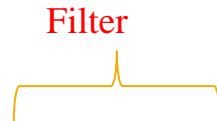
■ (9 rows)

Join Filter



# EXPLAIN输出信息的解读

- digoal=# explain select j1.\* ,j2.\* from j1 inner join j2 on (j1.id=j2.id) where j1.info='no';
- QUERY PLAN
- -----
- Merge Join (cost=110.88..117.43 rows=37 width=72)
- Merge Cond: (j1.id = j2.id)
- -> Sort (cost=25.45..25.47 rows=6 width=36)
- Sort Key: j1.id
- -> Seq Scan on j1 (cost=0.00..25.38 rows=6 width=36)
- Filter: (info = 'no'::text)
- -> Sort (cost=85.43..88.50 rows=1230 width=36)
- Sort Key: j2.id
- -> Seq Scan on j2 (cost=0.00..22.30 rows=1230 width=36)
- (9 rows)



# EXPLAIN 成本计算举例

- 成本计算相关的参数和系统表或视图
  - pg\_stats
  - pg\_class -- 用到relpages和reltuples
- 表或视图
- 参数
  - seq\_page\_cost -- 全表扫描的单个数据块的代价因子
  - random\_page\_cost -- 索引扫描的单个数据块的代价因子
  - cpu\_tuple\_cost -- 处理每条记录的CPU开销代价因子
  - cpu\_index\_tuple\_cost -- 索引扫描时每个索引条目的CPU开销代价因子
  - cpu\_operator\_cost -- 操作符或函数的开销代价因子
  - effective\_cache\_size -- 预知的可用缓存

# EXPLAIN 成本计算举例

- 全表扫描的成本计算
- digoal=# explain select \* from f;
- QUERY PLAN
- -----
- Seq Scan on f (cost=0.00..**12999.00** rows=640000 width=49)
- (1 row)
  
- Cost是怎么得来的? 全表扫描的成本计算只需要用到pg\_class.
- digoal=# select relpages,reltuples from pg\_class where relname='f';
- relpages | reltuples
- -----+-----
- 6599 | 640000
- (1 row)

# EXPLAIN 成本计算举例

- digoal=# show seq\_page\_cost;
- seq\_page\_cost
- -----
- 1
- (1 row)
- digoal=# show cpu\_tuple\_cost;
- cpu\_tuple\_cost
- -----
- 0.01
- (1 row)
- COST值：
- digoal=# select 6599\*1+640000\*0.01;
- ?column?
- -----
- 12999.00
- (1 row)

# EXPLAIN 成本计算举例

- 从柱状图评估行数的例子
- 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)
- 在8.3以及以前版本default\_statistics\_target默认是10, 也就是10个bucket.
- 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}

# EXPLAIN 成本计算举例

- 这个例子的行选择性如下
- $$\begin{aligned} \text{selectivity} &= (1 + (1000 - \text{bucket}[2].\text{min}) / (\text{bucket}[2].\text{max} - \text{bucket}[2].\text{min})) / \text{num\_buckets} \\ &= (1 + (1000 - 993) / (1997 - 993)) / 10 \\ &= 0.100697 \end{aligned}$$
- 最终得到的行数是：
- $$\begin{aligned} \text{rows} &= \text{rel\_cardinality} * \text{selectivity} \\ &= 10000 * 0.100697 \\ &= 1007 \text{ (rounding off)} \end{aligned}$$
- 这里 $\text{rel\_cardinality} = \text{pg\_class.reltuples}$ .

# EXPLAIN 成本计算举例

- 从MCV(most common values)评估行数的例子
- 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)
  
- 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 |  
  {EJAAAAA,BBAAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAAA,MCAAAA,NAAAAAA,WGAAAA}
- most\_common\_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}

# EXPLAIN 成本计算举例

- 行选择性如下, most common vals对应的占比most common freqs.
- $\text{selectivity} = \text{mcf}[3]$
- $= 0.003$
  
- 得到行数
- $\text{rows} = 10000 * 0.003$
- $= 30$

# EXPLAIN 成本计算举例

- 从MCV(most common values)和distinct值个数评估行数的例子
- EXPLAIN SELECT \* FROM tenk1 WHERE stringu1 = 'xxx';

- QUERY PLAN

- Seq Scan on tenk1 (cost=0.00..483.00 **rows=15** width=244)
- Filter: (stringu1 = 'xxx'::name)

- 1减去所有MCV的占比, 再乘以 distinct值的个数减去MCV的个数

$$\begin{aligned} \text{selectivity} &= (1 - \sum(\text{mvf})) / (\text{num\_distinct} - \text{num\_mcv}) \\ &= (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + \\ &\quad 0.003 + 0.003 + 0.003 + 0.003)) / (676 - 10) \\ &= 0.0014559 \end{aligned}$$

$$\begin{aligned} \text{rows} &= 10000 * 0.0014559 \\ &= 15 \text{ (rounding off)} \end{aligned}$$

# EXPLAIN 成本计算举例

- 从MCV(most common values)和柱状图评估行数的例子
- 条件中即包含了MCV又落在柱状图中的情况, 柱状图的统计中不包含MCV的值, 所以从柱状图中计算行的选择性时, 要乘以一个系数, 这个系数是1减去MCF的总和.
- EXPLAIN SELECT \* FROM tenk1 WHERE stringu1 < 'IAAAAAA';

## ■ QUERY PLAN

- Seq Scan on tenk1 (cost=0.00..483.00 rows=3077 width=244)
- Filter: (stringu1 < 'IAAAAAA'::name)
- SELECT histogram\_bounds FROM pg\_stats
- WHERE tablename='tenk1' AND attnname='stringu1';

## ■ histogram\_bounds

- {AAAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAAA,XLAAAA,ZZAAAA}

# EXPLAIN 成本计算举例

- 本例选择性MCV的占比
- $\text{selectivity} = \text{sum(relevant mvfs)}$ 
  - =  $0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003$
  - = 0.01833333
- 柱状图占比
- $\text{digoal} = \# \text{ select } 1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003)$ 
  - 
  - 0.96966667
- 0.298387是柱状图中通过类似 $(2 + ('IAAAAA' - \text{bucket[3].min}) / (\text{bucket[3].max} - \text{bucket[3].min})) / \text{num\_buckets}$ 的算法得到
- $\text{selectivity} = \text{mcv\_selectivity} + \text{histogram\_selectivity} * \text{histogram\_fraction}$ 
  - =  $0.01833333 + 0.298387 * 0.96966667$
  - = 0.307669
- $\text{rows} = 10000 * 0.307669$ 
  - = 3077 (rounding off)

# EXPLAIN 成本计算举例

- 多个列查询条件的选择性相乘评估例子
- EXPLAIN SELECT \* FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';
- QUERY PLAN
- -----
- Bitmap Heap Scan on tenk1 (cost=23.80..396.91 rows=1 width=244)
- Recheck Cond: (unique1 < 1000)
- Filter: (stringu1 = 'xxx'::name)
- -> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..23.80 rows=1007 width=0)
- Index Cond: (unique1 < 1000)
- 多列的选择性相乘得到最终的选择性
- $$\text{selectivity} = \text{selectivity}(\text{unique1} < 1000) * \text{selectivity}(\text{stringu1} = 'xxx')$$
- $$= 0.100697 * 0.0014559$$
- $$= 0.0001466$$
  
- $$\text{rows} = 10000 * 0.0001466$$
- $$= 1 \text{ (rounding off)}$$

# EXPLAIN 成本计算举例

- 索引扫描的成本计算
- 索引扫描时 ,和全表扫描不同, 扫描PAGE的开销是 $\text{pages} * \text{random\_page\_cost}$
- 另外, 索引扫描一般都涉及操作符, 例如大于, 小于, 等于.
- 这些操作符对应的函数的COST乘以 $\text{cpu\_operator\_cost}$ 就得到这个操作符的代价因子. 乘以实际操作的行数就得到CPU操作符开销. Rows Explain中可能无输出.
- 索引的CPU开销则是实际扫描的索引条目数乘以 $\text{cpu\_index\_tuple\_cost}$ . Rows Explain中无输出.
- 最后的一个开销是实际返回或丢给上层的TUPLE带来的CPU开销.  $\text{cpu\_tuple\_cost} * \text{实际扫描的行数}$ . Rows Explain中有输出

# EXPLAIN 成本计算举例

- 成本和实际的偏差举例
- 嵌套循环连接, 外表的关联字段存在大量重复值时, 总成本存在偏差.
- 合并连接, 关联字段少部分相交, 总成本远小于子节点的成本和. 因为任何一个表都只需要扫描到匹配结束.
- 使用LIMIT限制时, 总成本也将小于节点成本.
  
- digoal=# create table n1(id int, info text);
- CREATE TABLE
- digoal=# create table n2(id int, info text);
- CREATE TABLE
- digoal=# insert into n1 select generate\_series(1,100), 'test' from generate\_series(1,10);
- INSERT 0 1000
- digoal=# insert into n2 values (1);
- INSERT 0 1

# EXPLAIN 成本计算举例

- digoal=# explain (analyze,verbose,buffers,timing,costs) select \* from n1,n2 where n1.id=n2.id and n2.id=1;

QUERY PLAN

---

Nested Loop (cost=0.00..437.00 rows=250 width=45) (actual time=0.042..0.177 rows=10 loops=1) -- 计算嵌套循环总成本时, 按照25次循环计算. 所以比实际成本大很多. 实际时间0.177仅仅比0.16+0.01略大. 因为实际只循环一次.

Output: n1.id, n1.info, n2.id, n2.info

Buffers: shared hit=3

-> Seq Scan on public.n2 (cost=0.00..72.00 rows=25 width=36) (actual time=0.010..0.010 rows=1 loops=1) -- 评估25行, 实际1行

Output: n2.id, n2.info

Filter: (n2.id = 1)

Buffers: shared hit=1

-> Seq Scan on public.n1 (cost=0.00..14.50 rows=10 width=9) (actual time=0.027..0.160 rows=10 loops=1) -- 评估需要循环25次, 实际循环1次

Output: n1.id, n1.info

Filter: (n1.id = 1)

Rows Removed by Filter: 990

Buffers: shared hit=2

Total runtime: 0.214 ms

(13 rows)

# EXPLAIN 成本计算举例

- digoal=# truncate n1;
- TRUNCATE TABLE
- digoal=# truncate n2;
- TRUNCATE TABLE
- digoal=# insert into n1 select generate\_series(1,10),'test';
- INSERT 0 10
- digoal=# insert into n2 select generate\_series(1,100000),'test';
- INSERT 0 100000
- digoal=# analyze n1;
- ANALYZE
- digoal=# analyze n2;
- ANALYZE
- digoal=# create index idx\_n1 on n1(id);
- CREATE INDEX
- digoal=# create index idx\_n2 on n2(id);
- CREATE INDEX

# EXPLAIN 成本计算举例

- digoal=# set enable\_sort=off;
- SET
- digoal=# explain (analyze,verbose,buffers,timing,costs) select \* from n1,n2 where n1.id=n2.id;
- **QUERY PLAN**
- -----
- **Merge Join** (cost=0.43..3.88 rows=10 width=18) (actual time=0.043..0.059 rows=10 loops=1) -- 总成本小于合并连接相加的成本. 因为只有10条记录匹配, 其中10万条记录的表也不需要全面扫描.
- Output: n1.id, n1.info, n2.id, n2.info
- Merge Cond: (n1.id = n2.id)
- Buffers: shared hit=5
- -> Index Scan using idx\_n1 on public.n1 (cost=0.14..3.29 rows=10 width=9) (actual time=0.005..0.008 rows=10 loops=1)
  - Output: n1.id, n1.info
  - Buffers: shared hit=2
- -> Index Scan using idx\_n2 on public.n2 (cost=0.29..1705.29 rows=100000 width=9) (actual time=0.030..0.034 rows=11 loops=1)
  - Output: n2.id, n2.info
  - Buffers: shared hit=3
- Total runtime: 0.098 ms
- (11 rows)

# EXPLAIN 成本计算举例

■ digoal=# explain (analyze,verbose,buffers,timing,costs) select \* from n2 limit 2;

■            QUERY PLAN

■ Limit (cost=0.00..**0.02** rows=2 width=9) (actual time=0.033..0.034 rows=2 loops=1) -- 使用限制后, 总成本低于子节点的成本. 只算十万分之2

■        Output: id, info

■        Buffers: shared hit=1

■        -> Seq Scan on public.n2 (cost=0.00..**1135.00** rows=**100000** width=9) (actual time=0.031..0.031 rows=2 loops=1)

■            Output: id, info

■            Buffers: shared hit=1

■        Total runtime: 0.059 ms

■        (7 rows)

■ digoal=# select 1135.00/50000;

■        ?column?

■        -----

■        0.02270000000

# EXPLAIN成本因子校准

- <http://blog.163.com/digoal@126/blog/static/163877040201310255717379/>
- 不同的硬件环境CPU性能, IO性能各不相同, 所以默认的代价因子可能不适合实际的硬件环境.
- 校准方法是求未知数的过程. 其中要用到第三方的工具得到一些比较容易得到的值.
- 这里有个例子, 根据SQL实际的执行时间, 计算代价因子的值.
- seq\_page\_cost和cpu\_tuple\_cost的校准 :
- seq\_page\_cost通过stap测试得到.
- cpu\_tuple\_cost通过公式得到.
  
- 安装systemtap环境
- vi /etc/yum.repos.d/CentOS-Debuginfo.repo
- enabled=1
  
- # uname -r
- 2.6.32-358.el6.x86\_64
- yum install -y kernel-devel-2.6.32-358.el6.x86\_64 kernel-debuginfo-2.6.32-358.el6.x86\_64 kernel-debuginfo-common-x86\_64-2.6.32-358.el6.x86\_64 systemtap
- stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'
- <https://sourceware.org/systemtap/ftp/releases/> 或下载systemtap源码编译

# EXPLAIN成本因子校准

- 创建测试表, 插入测试数据

- digoal=# create table tbl\_cost\_align (id int, info text, crt\_time timestamp);

- CREATE TABLE

- digoal=# insert into tbl\_cost\_align select (random()\*2000000000)::int, md5(random()::text), clock\_timestamp() from generate\_series(1,100000);

- INSERT 0 100000

- digoal=# insert into tbl\_cost\_align select (random()\*2000000000)::int, md5(random()::text), clock\_timestamp() from generate\_series(1,10000000);

- INSERT 0 10000000

- 分析表

- digoal=# analyze tbl\_cost\_align;

- ANALYZE

# EXPLAIN成本因子校准

- 得到表的PAGE数
- digoal=# select relpages from pg\_class where relname='tbl\_cost\_align';
- relpages
- -----
- 94393
- (1 row)
- 检查点
- digoal=# checkpoint;
- CHECKPOINT
- 停库
- pg93@db-172-16-3-150-> pg\_ctl stop -m fast
- waiting for server to shut down.... done
- server stopped
- 把操作系统的缓存刷入硬盘
- [root@db-172-16-3-150 ssd1]# sync; echo 3 > /proc/sys/vm/drop\_caches
- 以1号CPU亲和启动数据库, 0号CPU会带来一定的额外开销问题.
- pg93@db-172-16-3-150-> taskset -c 1 /home/pg93/pgsql9.3.1/bin/postgres >/dev/null 2>&1

# EXPLAIN成本因子校准

- 启动一个客户端
- pg93@db-172-16-3-150-> psql
- psql (9.3.1)
- Type "help" for help.
- digoal=# select pg\_backend\_pid();
- pg\_backend\_pid
- -----
- 5727
- (1 row)

# EXPLAIN成本因子校准

- 使用stap跟踪, 得到seq\_page\_cost. (替换对应的bin路径)
- [root@db-172-16-3-150 ~]# taskset -c 7 stap -e '
- global a
- probe process("/home/pg93/pgsql9.3.1/bin/postgres").mark("query\_start") {
- delete a
- println("query\_start ", user\_string(\$arg1), "pid:", pid())
- }
- probe vfs.read.return {
- t = gettimeofday\_ns() - @entry(gettimeofday\_ns())
- # if (execname() == "postgres" && devname != "N/A")
- a[pid()] <<< t
- }
- probe process("/home/pg93/pgsql9.3.1/bin/postgres").mark("query\_done") {
- if (@count(a[pid()]))
- printf(" \*\* ", pid(), @count(a[pid()]), @avg(a[pid()]))
- println("query\_done ", user\_string(\$arg1), "pid:", pid())
- if (@count(a[pid()])) {
- # 未完

# EXPLAIN成本因子校准

- `println(@hist_log(a[pid()]))`
- `#println(@hist_linear(a[pid()],1024,4096,100))`
- `}`
- `delete a`
- `}' -x 5727`
  
- 等stap模块加载完成后，`lsmod|grep stap`
- 执行SQL
- `diggoal=# explain (analyze,verbose,costs,buffers,timing) select * from tbl_cost_align;`
- `QUERY PLAN`
- 
- 
- 

---

- Seq Scan on postgres.tbl\_cost\_align (cost=0.00..195393.00 rows=10100000 width=45) (actual time=0.839..3260.695 rows=10100000 loops=1)
- Output: id, info, crt\_time
- Buffers: shared read=94393 -- 注意这个read指的是未命中shared buffer, 如果是命中的话会有hit=?
- Total runtime: 4325.885 ms
- (4 rows)



# EXPLAIN成本因子校准

# EXPLAIN成本因子校准

- 验证公式正确性
- digoal=# show seq\_page\_cost;
- seq\_page\_cost
- -----
- 1
- (1 row)
- digoal=# show cpu\_tuple\_cost;
- cpu\_tuple\_cost
- -----
- 0.01
- (1 row)
- $195393 = (\text{shared read=})94393 * 1(\text{seq\_page\_cost}) + (\text{rows=})10100000 * 0.01(\text{cpu\_tuple\_cost})$
- digoal=# select 94393+10100000\*0.01;
- ?column?
- -----
- 195393.00
- (1 row)

# EXPLAIN成本因子校准

- 从stap中我们得到io的平均响应时间是14329纳秒(0.014329毫秒). 得到了seq\_page\_cost.
- 真实的执行时间是3260.695套用到公式中, 求得cpu\_tuple\_cost :
- $3260.695 = 94393 * 0.014329 + 10100000 * \text{cpu\_tuple\_cost}$
- $\text{cpu\_tuple\_cost} = 0.00018892452504950495$
- 重启数据库, 并刷系统缓存后, 调整这两个代价因子
- digoal=# set seq\_page\_cost=0.014329;
- SET
- digoal=# set cpu\_tuple\_cost=0.00018892452504950495;
- SET
- 得到的cost和实际执行时间基本一致.
- digoal=# explain (analyze,verbose,costs,buffers,timing) select \* from tbl\_cost\_align;
- QUERY PLAN
- -----
- Seq Scan on postgres.tbl\_cost\_align (cost=0.00..**3260.695** rows=10100000 width=45) (actual time=0.915..**3318.443** rows=10100000 loops=1)
  - Output: id, info, crt\_time
  - Buffers: shared read=94393
  - Total runtime: 4380.828 ms

# EXPLAIN成本因子校准

- random\_page\_cost, cpu\_index\_tuple\_cost, cpu\_operator\_cost的校准.
- random\_page\_cost 本文还是通过stap跟踪来获得.
- cpu\_index\_tuple\_cost 和 cpu\_operator\_cost 两个未知数需要两个等式求得,
- 除了公式以外, 本文利用cpu\_index\_tuple\_cost 和 cpu\_operator\_cost的比例得到第二个等式.
- 首先我们还是要确定公式准确性, 为了方便公式验证, 把所有的常量都设置为1.
- digoal=# set random\_page\_cost=1;
- SET
- digoal=# set cpu\_tuple\_cost=1;
- SET
- digoal=# set cpu\_index\_tuple\_cost=1;
- SET
- digoal=# set cpu\_operator\_cost=1;
- SET

# EXPLAIN成本因子校准

- digoal=# set enable\_seqscan=off; set enable\_bitmapscan=off; explain (analyze,verbose,costs,buffers,timing) select \* from tbl\_cost\_align where id>1998999963;
- QUERY PLAN
- 
- 
- -----
- Index Scan using idx\_tbl\_cost\_align\_id on postgres.tbl\_cost\_align (cost=174.00..20181.67 rows=5031 width=45) (actual time=0.029..17.773 rows=5037 loops=1)
- Output: id, info, crt\_time
- Index Cond: (tbl\_cost\_align.id > 1998999963)
- Buffers: shared hit=5054
- Total runtime: 18.477 ms
- (5 rows)
- 执行计划表明这是个索引扫描,至于扫了多少个数据块是未知的,索引的tuples也是未知的,已知的是cost和rows.
- $20181.67 = \text{blocks} * \text{random\_page\_cost} + \text{cpu\_tuple\_cost} * 5031 + \text{cpu\_index\_tuple\_cost} * ? + \text{cpu\_operator\_cost} * ?$

# EXPLAIN成本因子校准

- 求这个问号, 可以通过更改cpu\_operator\_cost来得到.
- digoal=# set cpu\_operator\_cost=2;
- SET
- digoal=# set enable\_seqscan=off; set enable\_bitmapscan=off; explain (analyze,verbose,costs,buffers,timing) select \* from tbl\_cost\_align where id>1998999963;
- SET
- SET

## QUERY PLAN

- Index Scan using idx\_tbl\_cost\_align\_id on postgres.tbl\_cost\_align (cost=348.00..25386.67 rows=5031 width=45) (actual time=0.013..5.785 rows=5037 loops=1)
  - Output: id, info, crt\_time
  - Index Cond: (tbl\_cost\_align.id > 1998999963)
  - Buffers: shared hit=5054
  - Total runtime: 6.336 ms
  - (5 rows)
- $25386.67 - 20181.67 = 5205$  得到本例通过索引扫描的条数. 等式就变成了
- $20181.67 = \text{blocks} * \text{random_page_cost} + \text{cpu_tuple_cost} * 5031 + \text{cpu_index_tuple_cost} * 5031 + \text{cpu_operator_cost} * 5205$

# EXPLAIN成本因子校准

- 接下来要求blocks, 也就是扫描的随机页数.
- 通过调整random\_page\_cost得到.
- digoal=# set random\_page\_cost = 2;
- SET
- digoal=# set enable\_seqscan=off; set enable\_bitmapscan=off; explain (analyze,verbose,costs,buffers,timing) select \* from tbl\_cost\_align where id>1998999963;
- SET
- SET

## QUERY PLAN

- -----
- Index Scan using idx\_tbl\_cost\_align\_id on postgres.tbl\_cost\_align (cost=348.00..30301.33 rows=5031 width=45) (actual time=0.013..5.778 rows=5037 loops=1)
  - Output: id, info, crt\_time
  - Index Cond: (tbl\_cost\_align.id > 1998999963)
  - Buffers: shared hit=5054
  - Total runtime: 6.331 ms
  - (5 rows)
- $30301.33 - 25386.67 = 4914.66$  --得到blocks = 4914.66.

# EXPLAIN成本因子校准

- 更新等式：  
$$20181.67 = 4914.66 * \text{random\_page\_cost} + \text{cpu\_tuple\_cost} * 5031 + \text{cpu\_index\_tuple\_cost} * 5031 + \text{cpu\_operator\_cost} * 5205$$
- 接下来要做的是通过stap统计出random\_page\_cost.  
pg93@db-172-16-3-150-> taskset -c 1 /home/pg93/pgsql9.3.1/bin/postgres >/dev/null 2>&1  
[root@db-172-16-3-150 ~]# sync; echo 3 > /proc/sys/vm/drop\_caches  
digoal=# select pg\_backend\_pid();  
pg\_backend\_pid  
-----  
10009  
(1 row)

# EXPLAIN成本因子校准

```
■ [root@db-172-16-3-150 ~]# taskset -c 2 stap -e '
■ global a
■
■ probe process("/home/pg93/pgsql9.3.1/bin/postgres").mark("query__start") {
■ delete a
■ println("query__start ", user_string($arg1), "pid:", pid())
■ }
■
■ probe vfs.read.return {
■ t = gettimeofday_ns() - @entry(gettimeofday_ns())
■ # if (execname() == "postgres" && devname != "N/A")
■ a[pid()] <<< t
■ }
■
■ probe process("/home/pg93/pgsql9.3.1/bin/postgres").mark("query__done") {
■ if (@count(a[pid()]))
■ printdln(" ** ", pid(), @count(a[pid()]), @avg(a[pid()]))
■ println("query__done ", user_string($arg1), "pid:", pid())
■ if (@count(a[pid()])) {
■ # 未完
```

# EXPLAIN成本因子校准

- ```
println(@hist_log(a[pid()]))  
#println(@hist_linear(a[pid()],1024,4096,100))  
}  
delete a  
}' -x 10009
```
- 执行SQL：

```
digoal=# set enable_seqscan=off; set enable_bitmapscan=off; explain (analyze,verbose,costs,buffers,timing) select * from tbl_cost_align where id>1998999963;
```
- **QUERY PLAN**

 - Index Scan using idx_tbl_cost_align_id on postgres.tbl_cost_align (cost=0.43..5003.15 rows=5031 width=45) (actual time=0.609..1844.415 rows=5037 loops=1)
 - Output: id, info, crt_time
 - Index Cond: (tbl_cost_align.id > 1998999963)
 - Buffers: shared hit=152 read=4902
 - Total runtime: 1846.683 ms
 - (5 rows)

EXPLAIN成本因子校准

query_start explain (analyze,verbose,costs,buffers,timing) select * from tbl_cost_align where id>1998999963;pid:10009
10009**4946**368362 -- 得到random_page_cost
query_done explain (analyze,verbose,costs,buffers,timing) select * from tbl_cost_align where id>1998999963;pid:10009
value ----- count
2048 0
4096 0
8192 33
16384 2
32768 6
65536 4
131072 @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ @ 1193
262144 @ 2971
524288 @ @ @ @ @ @ @ @ @ @ @ 729
1048576 2
2097152 5
4194304 0
8388608 1
16777216 0
33554432 0

EXPLAIN成本因子校准

- 更新等式, 使用时间等式:
- 等式1:
$$1844.415 = 4914.66 * 0.368362 + 0.00018884145574257426 * 5031 + \text{cpu_index_tuple_cost} * 5031 + \text{cpu_operator_cost} * 5205$$
- cpu_tuple_cost 用例子1中计算得到的 0.00018884145574257426
- $\text{cpu_index_tuple_cost}$ 和 cpu_operator_cost 的比例用系统默认的2 : 1.
- 等式2:
$$\text{cpu_index_tuple_cost}/\text{cpu_operator_cost} = 2$$
- 最终得到:
$$\text{cpu_index_tuple_cost} = 0.00433497085216479990$$

$$\text{cpu_operator_cost} = 0.00216748542608239995$$

EXPLAIN成本因子校准

- 结合例子1 得到的两个常量,所有的5个常量值就调整好了.
- digoal=# set cpu_tuple_cost=0.00018884145574257426;
- SET
- digoal=# set cpu_index_tuple_cost = 0.00433497085216479990;
- SET
- digoal=# set cpu_operator_cost = 0.00216748542608239995;
- SET
- digoal=# set seq_page_cost=0.014329;
- SET
- digoal=# set random_page_cost = 0.368362;
- SET

auto_explain插件的使用

- <http://blog.163.com/digoal@126/blog/static/16387704020115825612145/>
- auto_explain的目的是给数据库中执行的SQL语句一个执行时间阈值, 超过阈值的话, 记录下当时这个SQL的执行计划到日志中, 便于未来查看这个SQL执行计划有没有问题.
- 编译安装
 - [root@db-172-16-3-150 ~]# export PATH=/home/pg93/pgsql/bin:\$PATH
 - [root@db-172-16-3-150 ~]# which pg_config
 - /usr/bin/pg_config
 - [root@db-172-16-3-150 ~]# cd /opt/soft_bak/postgresql-9.3.4/contrib/auto_explain/
 - [root@db-172-16-3-150 auto_explain]# gmake clean
 - [root@db-172-16-3-150 auto_explain]# gmake
 - [root@db-172-16-3-150 auto_explain]# gmake install
- auto_explain 有两种使用方法
 - 会话级使用
 - 数据库级使用

auto_explain插件的使用

- 会话级使用举例
- digoal=# load 'auto_explain';
- LOAD
- digoal=# set auto_explain.log_min_duration=0; 设置SQL执行时间阈值
- SET
- digoal=# select * from t limit 1;
- 查看日志
- 2013-12-10 14:32:15.587 CST,"postgres","digoal",12933,"[local]",52a6b506.3285,11,"SELECT",2013-12-10 14:30:30 CST,2/180059,0,LOG,00000,"duration: 0.043 ms plan:
- Query Text: select * from t limit 1;
- Limit (cost=0.00..0.03 rows=1 width=108)
- -> Seq Scan on t (cost=0.00..1409091.04 rows=50000004 width=108)",,,,,,,,"explain_ExecutorEnd, auto_explain.c:320","psql"

auto_explain插件的使用

- 数据库级使用
- vi \$PGDATA/postgresql.conf
- shared_preload_libraries = 'pg_stat_statements, auto_explain'
- auto_explain.log_min_duration = 100ms
- 修改shared_preload_libraries需要重启数据库
- pg93@db-172-16-3-150-> pg_ctl restart -m fast

PostgreSQL plan cache management

- src/backend/utils/cache/plancache.c
- <http://www.postgresql.org/docs/9.3/static/spi-spi-prepare.html>
- <http://www.postgresql.org/docs/9.3/static/plpgsql-implementation.html>
- <http://www.postgresql.org/docs/9.3/static/libpq-exec.html>

PostgreSQL plan cache management

- PostgreSQL 根据统计信息和成本因子计算出各种组合的成本, 计算总成本最低的组合即得到的执行计划(plan).
- 如果每次都需要计算的话, 将耗费大量的CPU资源. 因此需要将计算好的plan缓存起来, 减少CPU开销.
- prepared statement支持plan cache.
- simple query 不支持plan cache.

- plan缓存需要考虑的几个问题.
 - 1. 当SQL语句有参数时, 参数不同, 最低总成本对应的执行计划可能不同. 选择使用plan cache还是重新计算plan?
 - 2. 当SQL语句涉及的数据对象发生变化时, 需要将plan cache失效.
 - 3. 目前PostgreSQL plan cache是基于会话的, 所以当使用连接池时, 必须是会话模式才可以配合plan cache使用.

- 因此PostgreSQL有一个专门的plan缓存管理机制.
- 选择使用plan cache(**generic plan**)还是重新计算plan(**custom plan**)?
 - [choose_custom_plan@ src/backend/utils/cache/plancache.c](#)

PostgreSQL plan cache management

```
■ choose_custom_plan函数.  
■ /*  
■ * choose_custom_plan: choose whether to use custom or generic plan  
■ *  
■ * This defines the policy followed by GetCachedPlan.  
■ */  
■ static bool  
■ choose_custom_plan(CachedPlanSource *plansource, ParamListInfo boundParams)  
■ {  
■     double      avg_custom_cost;  
  
■     /* One-shot plans will always be considered custom */ // 如果plansource->is_oneshot非0, 使用custom plan. 用于客户端强制custom plan  
■     if (plansource->is_oneshot)  
■         return true;  
  
■     /* Otherwise, never any point in a custom plan if there's no parameters */  
■     if (boundParams == NULL)          // 如果prepared statement的SQL没有参数, 直接使用generic plan.  
■         return false;  
■     /* ... nor for transaction control statements */  
■     if (IsTransactionStmtPlan(plansource)) // 事务控制语句使用generic plan , 如begin;end;savepoint;rollback; rollback to;  
■         return false;
```

PostgreSQL plan cache management

```
■ /* See if caller wants to force the decision */

■ if (plansource->cursor_options & CURSOR_OPT_GENERIC_PLAN) // 用于SPI_prepare_cursor 接口函数强制generic|custom plan的flags
■     return false;

■ if (plansource->cursor_options & CURSOR_OPT_CUSTOM_PLAN)
■     return true;

■ /* Generate custom plans until we have done at least 5 (arbitrary) */ // 累计custom plan次数小于5次则继续使用custom plan.
■ if (plansource->num_custom_plans < 5)
■     return true;

■ avg_custom_cost = plansource->total_custom_cost / plansource->num_custom_plans;
```

PostgreSQL plan cache management

```
■    /*
■    * Prefer generic plan if it's less expensive than the average custom
■    * plan. (Because we include a charge for cost of planning in the
■    * custom-plan costs, this means the generic plan only has to be less
■    * expensive than the execution cost plus replan cost of the custom
■    * plans.)
■    *
■    * Note that if generic_cost is -1 (indicating we've not yet determined
■    * the generic plan cost), we'll always prefer generic at this point.
■    */
■    if (plansource->generic_cost < avg_custom_cost) // generic_cost 小于平均custom cost使用generic cost.
■        return false;
■
■    return true;
■ }
```

PostgreSQL plan cache management

```
■ /* 获得执行计划的函数
■ * GetCachedPlan: get a cached plan from a CachedPlanSource.
■ *
■ * This function hides the logic that decides whether to use a generic
■ * plan or a custom plan for the given parameters: the caller does not know
■ * which it will get.
■ *
■ * On return, the plan is valid and we have sufficient locks to begin
■ * execution.
■ *
■ * On return, the refcount of the plan has been incremented; a later
■ * ReleaseCachedPlan() call is expected. The refcount has been reported
■ * to the CurrentResourceOwner if useResOwner is true (note that that must
■ * only be true if it's a "saved" CachedPlanSource).
■ *
■ * Note: if any replanning activity is required, the caller's memory context
■ * is used for that work.
■ */
■ CachedPlan *
■ GetCachedPlan(CachedPlanSource *plansource, ParamListInfo boundParams,
■           bool useResOwner)
```

PostgreSQL plan cache management

```
■ {  
■     CachedPlan *plan;  
■     List      *qlist;  
■     bool       customplan;  
  
■     /* Assert caller is doing things in a sane order */  
■     Assert(plansource->magic == CACHEDPLANSOURCE_MAGIC);  
■     Assert(plansource->is_complete);  
■     /* This seems worth a real test, though */  
■     if (useResOwner && !plansource->is_saved)  
■         elog(ERROR, "cannot apply ResourceOwner to non-saved cached plan");  
  
■     /* Make sure the querytree list is valid and we have parse-time locks */  
■     qlist = RevalidateCachedQuery(plansource);  
  
■     /* Decide whether to use a custom plan */  
■     customplan = choose_custom_plan(plansource, boundParams);
```

PostgreSQL plan cache management

```
■ if (!customplan)
■ {
■     if (CheckCachedPlan(plansource)) // 判断generic plan是否存在.
■     {
■         /* We want a generic plan, and we already have a valid one */
■         plan = plansource->gplan; // 如果generic plan存在, 则直接重复使用.
■         Assert(plan->magic == CACHEDPLAN_MAGIC);
■     }
■     else
■     {
■         /* Build a new generic plan */
■         plan = BuildCachedPlan(plansource, qlist, NULL); // 不存在则创建一个generic plan.
■         /* Just make real sure plansource->gplan is clear */
■         ReleaseGenericPlan(plansource);
■         /* Link the new generic plan into the plansource */
■         plansource->gplan = plan;
■         plan->refcount++;
■         /* Immediately reparent into appropriate context */
■         if (plansource->is_saved)
■         {
■             /* ... */
■         }
■     }
■ }
```

PostgreSQL plan cache management

```
■          /* saved plans all live under CacheMemoryContext */
■          MemoryContextSetParent(plan->context, CacheMemoryContext);
■          plan->is_saved = true;
■      }
■      else
■      {
■          /* otherwise, it should be a sibling of the plansource */
■          MemoryContextSetParent(plan->context,
■                               MemoryContextGetParent(plansource->context));
■      }
■      /* Update generic_cost whenever we make a new generic plan */ // 更新generic plan的成本.
■      plansource->generic_cost = cached_plan_cost(plan, false);
■
■      /*
■       * If, based on the now-known value of generic_cost, we'd not have
■       * chosen to use a generic plan, then forget it and make a custom
■       * plan. This is a bit of a wart but is necessary to avoid a
```

PostgreSQL plan cache management

```
■ * glitch in behavior when the custom plans are consistently big
■ * winners; at some point we'll experiment with a generic plan and
■ * find it's a loser, but we don't want to actually execute that
■ * plan.
■ */
■ customplan = choose_custom_plan(plansource, boundParams); // 把prepared statement参数值以及plan cache传入重新估算是否选择custom plan.

■ /*
■ * If we choose to plan again, we need to re-copy the query_list,
■ * since the planner probably scribbled on it. We can force
■ * BuildCachedPlan to do that by passing NIL.
■ */
■ qlist = NIL;
■ }
■ }
```

PostgreSQL plan cache management

```
■ if (customplan)
■ {
■     /* Build a custom plan */
■     plan = BuildCachedPlan(plansource, qlist, boundParams);
■     /* Accumulate total costs of custom plans, but 'ware overflow */
■     if (plansource->num_custom_plans < INT_MAX)
■     {
■         plansource->total_custom_cost += cached_plan_cost(plan, true); // 更新total custom plan cost以及custom plan次数.
■         plansource->num_custom_plans++;
■     }
■ }
```

PostgreSQL plan cache management

```
■     /* Flag the plan as in use by caller */
■     if (useResOwner)
■         ResourceOwnerEnlargePlanCacheRefs(CurrentResourceOwner);
■     plan->refcount++;
■     if (useResOwner)
■         ResourceOwnerRememberPlanCacheRef(CurrentResourceOwner, plan);
■
■     /*
■      * Saved plans should be under CacheMemoryContext so they will not go away
■      * until their reference count goes to zero. In the generic-plan cases we
■      * already took care of that, but for a custom plan, do it as soon as we
■      * have created a reference-counted link.
■      */
■     if (customplan && plansource->is_saved)
■     {
■         MemoryContextSetParent(plan->context, CacheMemoryContext);
■         plan->is_saved = true;
■     }
■
■     return plan;
■ }
```

PostgreSQL plan cache management

PostgreSQL plan cache management

```
■ MemoryContext query_context; /* context holding the above, or NULL */  
■ /* If we have a generic plan, this is a reference-counted link to it: */  
■ struct CachedPlan *gplan; /* generic plan, or NULL if not valid */  
■ /* Some state flags: */  
■     bool      is_oneshot;      /* is it a "oneshot" plan? */  
■     bool      is_complete;    /* has CompleteCachedPlan been done? */  
■     bool      is_saved;       /* has CachedPlanSource been "saved"? */  
■     bool      is_valid;       /* is the query_list currently valid? */  
■     int       generation;    /* increments each time we create a plan */  
■ /* If CachedPlanSource has been saved, it is a member of a global list */  
■ struct CachedPlanSource *next_saved; /* list link, if so */  
■ /* State kept to help decide whether to use custom or generic plans: */  
■     double    generic_cost;   /* cost of generic plan, or -1 if not known */  
■     double    total_custom_cost; /* total cost of custom plans so far */  
■     int      num_custom_plans; /* number of plans included in total */  
■ } CachedPlanSource;
```

PostgreSQL plan cache management

- prepared statement接口.
- <http://www.postgresql.org/docs/9.3/static/libpq-exec.html>
- 同步调用
- PGresult *PQprepare(PGconn *conn,
 const char *stmtName,
 const char *query,
 int nParams,
 const Oid *paramTypes);
- PGresult *PQexecPrepared(PGconn *conn,
 const char *stmtName,
 int nParams,
 const char * const *paramValues,
 const int *paramLengths,
 const int *paramFormats,
 int resultFormat);

PostgreSQL plan cache management

- 异步调用
- ```
int PQsendPrepare(PGconn *conn,
 const char *stmtName,
 const char *query,
 int nParams,
 const Oid *paramTypes);
```
- ```
int PQsendQueryPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```
- ```
PGresult *PQgetResult(PGconn *conn);
```

# PostgreSQL plan cache management

- 例子：
- 跟踪choose custom plan 以及 query parser, rewrite, plan , executor.
- stap -D MAXSTRINGLEN=99999 -e '
  - global cnt;
  - probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_start")
    - {
    - println(pn(), user\_string(\$arg1), pid())
    - }
  - probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")
    - {
    - println(pn(), user\_string(\$arg1), pid())
    - }
  - probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")
    - {
    - println(pn(), user\_string(\$arg1), pid())
    - }
  - probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")
    - {
    - println(pn(), pid())
    - }

# PostgreSQL plan cache management

```
■ probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query__execute__start")
■ {
■ println(pn(), pid())
■ }
■ probe process("/home/pg93/pgsql9.3.3/bin/postgres").function("choose_custom_plan@/opt/soft_bak/postgresql-9.3.3/src/backend/utils/cache/plancache.c")
■ {
■ cnt++;
■ printf("choose custom plan called %d\n %s\n", cnt, $$vars$$)
■ }
■ '
■
■ create table test(id int, info text);
■ insert into test select 1,repeat(random()::text,10) from generate_series(1,500000);
■ insert into test values (2,'test');
```

# PostgreSQL plan cache management

- psql始终使用exec\_simple\_query接口, 不会跟踪到choose custom plan, 因为psql没有使用prepared statement.

- digoal=# select count(\*) from test where id=1;

- count

- -----

- 500000

- (1 row)

- digoal=# select count(\*) from test where id=1;

- count

- -----

- 500000

- (1 row)

- digoal=# select count(\*) from test where id=2;

- count

- -----

- 1

- (1 row)

# PostgreSQL plan cache management

- 未跟踪到choose custom plan, 没有plan cache.
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_start")select count(\*) from test where id=1;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")select count(\*) from test where id=1;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")select count(\*) from test where id=1;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_start")select count(\*) from test where id=1;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")select count(\*) from test where id=1;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")select count(\*) from test where id=1;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_start")select count(\*) from test where id=2;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")select count(\*) from test where id=2;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")select count(\*) from test where id=2;2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")2624
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")2624

# PostgreSQL plan cache management

- 使用prepare语句, 跟踪到choose custom plan, 匹配前面的generic plan生成算法.

- digoal=# prepare p(int) as select count(\*) from test where id=\$1;

- PREPARE

- digoal=# execute p(2); -- execute p(2) 除了使用psql的简单调用外, 还会用到prepared statement接口

- count

- -----

- 1

- (1 row)

- digoal=# execute p(2);

- count

- -----

- 1

- (1 row)

- <http://www.postgresql.org/docs/9.3/static/sql-prepare.html>

- <http://www.postgresql.org/docs/9.3/static/sql-execute.html>

# PostgreSQL plan cache management

- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_start")prepare p(int) as select count(\*) from test where id=\$1;3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")prepare p(int) as select count(\*) from test where id=\$1;3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")prepare p(int) as select count(\*) from test where id=\$1;3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3622
  
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_start")execute p(2);3622 -- psql产生的简单调用.
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")execute p(2);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")execute p(2);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3622
- choose custom plan called 1
- plansource={.magic=195726186, .raw\_parse\_tree=0x2169950, .query\_string="prepare p(int) as select count(\*) from test where id=\$1;", .commandTag="SELECT", .param\_types=0x2169fa0, .num\_params=1, .parserSetup=0x0, .parserSetupArg=0x0, .cursor\_options=0, .fixed\_result='\001', .resultDes=0x2169fc0, .context=0x2159dc8, .query\_list=0x216a868, .relationOids=0x2174938, .invalItems=0x0, .search\_path=0x2174970, .query\_context=0x215a020, .gplan=0x0, .is\_oneshot='\000', .is\_complete='\001', .is\_saved='\001', .is\_valid='\001', .generation=0, .next\_saved=0x0, .generic\_cost=?, .total\_custom\_cost=?, .num\_custom\_plans=0} boundParams={.paramFetch=0x0, .paramFetchArg=0x0, .parserSetup=0x0, .parserSetupArg=0x0, .numParams=1, .params=[{.value=2, .isnull='\000', .pflags=1, .ptype=23}, ...]} avg\_custom\_cost=?
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3622
  
- 注意观察**num\_custom\_plans**的变化.

# PostgreSQL plan cache management

- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_\_start")execute p(2);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")execute p(2);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")execute p(2);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3622
- choose custom plan called 2
- plansource={ .magic=195726186, .raw\_parse\_tree=0x2169950, .query\_string="prepare p(int) as select count(\*) from test where id=\$1;", .commandTag="SELECT", .param\_types=0x2169fa0, .num\_params=1, .parserSetup=0x0, .parserSetupArg=0x0, .cursor\_options=0, .fixed\_result="\001", .resultDes=0x2169fc0, .context=0x2159dc8, .query\_list=0x216a868, .relationOids=0x2174938, .invalItems=0x0, .search\_path=0x2174970, .query\_context=0x215a020, .gplan=0x0, .is\_oneshot='\000', .is\_complete='\001', .is\_saved='\001', .is\_valid='\001', .generation=1, .next\_saved=0x0, .generic\_cost=?, .total\_custom\_cost=?, .num\_custom\_plans=1 } boundParams={.paramFetch=0x0, .paramFetchArg=0x0, .parserSetup=0x0, .parserSetupArg=0x0, .numParams=1, .params=[{.value=2, .isNull='\000', .pflags=1, .ptype=23}, ...] } avg\_custom\_cost=?
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3622

# PostgreSQL plan cache management

- 第5次开始生成generic plan cache. 后面可以看出query plan cache节点没有了.
- 但是execute 这个SQL因为是psql使用exec\_simple\_query调用的, 所以还需要parse, rewrite阶段.
  
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_\_start")execute p(1);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")execute p(1);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")execute p(1);3622
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3622
- choose custom plan called 16
- plansource={.magic=195726186, .raw\_parse\_tree=0x2169950, .query\_string="prepare p(int) as select count(\*) from test where id=\$1;", .commandTag="SELECT", .param\_types=0x2169fa0, .num\_params=1, .parserSetup=0x0, .parserSetupArg=0x0, .cursor\_options=0, .fixed\_result='\001', .resultDesc=0x2169fc0, .context=0x2159dc8, .query\_list=0x216a868, .relationOids=0x2174938, .invalidItems=0x0, .search\_path=0x2174970, .query\_context=0x215a020, .gplan=0x21890a0, .is\_oneshot='\000', .is\_complete='\001', .is\_saved='\001', .is\_valid='\001', .generation=6, .next\_saved=0x0, .generic\_cost=?, .total\_custom\_cost=?, .num\_custom\_plans=5} boundParams={.paramFetch=0x0, .paramFetchArg=0x0, .parserSetup=0x0, .parserSetupArg=0x0, .numParams=1, .params=[{.value=1, .isnull='\000', .pflags=1, .ptype=23}, ...]} avg\_custom\_cost=?
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3622
  
- 这里观察choose custom plan函数的结果, 有多少次选择了custom plan, 目前num\_custom\_plans=5

# PostgreSQL plan cache management

- 使用pgbench测试libpq prepared statement 接口. (隐藏choose custom plan的输出)
- pg93@db-172-16-3-150-> cat test.sql
- \setrandom id 1 2
- select count(\*) from test where id=:id;
  
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")select count(\*) from test where id=\$1;3765
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")3765
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3765
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")3765 // 生成generic plan
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3765 // 开始使用plan cache.
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3765
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")3765

# PostgreSQL plan cache management

- 跟踪函数的plan cache (plpgsql函数调用spi\_prepare接口)
- <http://www.postgresql.org/docs/9.3/static/plpgsql-implementation.html>
  
- CREATE OR REPLACE FUNCTION public.f\_test\_cnt(i\_id integer)
  - RETURNS bigint
  - LANGUAGE plpgsql
  - STRICT
  - AS \$function\$
  - declare
  - res int8;
  - begin
  - select count(\*) into res from test where id=i\_id; -- 调用spi prepare
  - return res; -- 调用spi prepare
  - end;
  - \$function\$;

# PostgreSQL plan cache management

- digoal=# explain analyze select \* from test where id=1;
- QUERY PLAN
- -----
- Seq Scan on test (cost=0.00..37948.05 rows=2000004 width=177) (actual time=0.042..710.060 rows=2000000 loops=1)
- Filter: (id = 1)
- Rows Removed by Filter: 4
- Total runtime: 865.982 ms
- (4 rows)
  
- digoal=# explain analyze select \* from test where id=2;
- QUERY PLAN
- -----
- Index Scan using idx\_test\_1 on test (cost=0.43..4.45 rows=1 width=177) (actual time=0.043..0.053 rows=4 loops=1)
- Index Cond: (id = 2)
- Total runtime: 0.177 ms
- (3 rows)
  
- digoal=# select f\_test\_cnt(2); // 由于数据分布不均匀, 如果在生成generic plan前有id=2的查询, 将造成choose custom plan得出结果持续使用custom plan.
- f\_test\_cnt
- -----
- 1
- (1 row)

# PostgreSQL plan cache management

- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_start")select f\_test\_cnt(2);4287
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")select f\_test\_cnt(2);4287
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")select f\_test\_cnt(2);4287
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")4287
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_execute\_start")4287
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")select count(\*) from test where id=i\_id4287
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")select count(\*) from test where id=i\_id4287
- choose custom plan called 213
- plansource={.magic=195726186, .raw\_parse\_tree=0x2165200, .query\_string="select count(\*) from test where id=i\_id", .commandTag="SELECT", .param\_types=0x0, .num\_params=0, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217df38, .cursor\_options=0, .fixed\_result='\000', .resultDesc=0x2165908, .context=0x2177f80, .query\_list=0x2166518, .relationOids=0x218fe98, .invalidItems=0x0, .search\_path=0x218fed0, .query\_context=0x2178248, .gplan=0x0, .is\_oneshot='\000', .is\_complete='\001', .is\_saved='\001', .is\_valid='\001', .generation=0, .next\_saved=0x0, .generic\_cost=?, .total\_custom\_cost=?, .num\_custom\_plans=0}  
boundParams={.paramFetch=0x7f3ddc31d9e0, .paramFetchArg=0x7fff8e199150, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217df38, .numParams=4, .params=[{.value=2, .isnull='\000', .pflags=1, .ptype=23}, ...]} avg\_custom\_cost=?
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")4287 对应上面这个SQL的plan
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_parse\_start")SELECT res4287
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_rewrite\_start")SELECT res4287
- choose custom plan called 214

# PostgreSQL plan cache management

- plansource={.magic=195726186, .raw\_parse\_tree=0x2190bb0, .query\_string="**SELECT**  
**res**", .commandTag="SELECT", .param\_types=0x0, .num\_params=0, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217e138, .cursor\_options=0, .fixed\_result='\000', .resultDesc=0x2190ed0, .context=0x2178f80, .query\_list=0x21916b8, .relationOids=0x0, .invalidItems=0x0, .search\_path=0x2191978, .query\_context=0x21788f0, .gplan=0x0, .is\_oneshot='\000', .is\_complete='\001', .is\_saved='\001', .is\_valid='\001', .generation=0, .next\_saved=0x21650e8, .generic\_cost=?, .total\_custom\_cost=?, .num\_custom\_plans=0} boundParams=ERROR avg\_custom\_cost=?
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query\_plan\_start")4287 // 对应上面这个SQL的plan
- choose custom plan called 215
- plansource={.magic=195726186, .raw\_parse\_tree=0x2190bb0, .query\_string="**SELECT**  
**res**", .commandTag="SELECT", .param\_types=0x0, .num\_params=0, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217e138, .cursor\_options=0, .fixed\_result='\000', .resultDesc=0x2190ed0, .context=0x2178f80, .query\_list=0x21916b8, .relationOids=0x0, .invalidItems=0x0, .search\_path=0x2191978, .query\_context=0x21788f0, .gplan=0x2192178, .is\_oneshot='\000', .is\_complete='\001', .is\_saved='\001', .is\_valid='\001', .generation=1, .next\_saved=0x21650e8, .generic\_cost=?, .total\_custom\_cost=?, .num\_custom\_plans=0} boundParams=ERROR avg\_custom\_cost=?
- choose custom plan called 216
- plansource={.magic=195726186, .raw\_parse\_tree=0x2190bb0, .query\_string="**SELECT**  
**res**", .commandTag="SELECT", .param\_types=0x0, .num\_params=0, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217e138, .cursor\_options=0, .fixed\_result='\000', .resultDesc=0x2190ed0, .context=0x2178f80, .query\_list=0x21916b8, .relationOids=0x0, .invalidItems=0x0, .search\_path=0x2191978, .query\_context=0x21788f0, .gplan=0x2192178, .is\_oneshot='\000', .is\_complete='\001', .is\_saved='\001', .is\_valid='\001', .generation=1, .next\_saved=0x21650e8, .generic\_cost=?, .total\_custom\_cost=?, .num\_custom\_plans=0} boundParams=ERROR avg\_custom\_cost=?

# PostgreSQL plan cache management

- n次调用后, 不管使用id=1还是id=2查询都会继续使用custom plan.

- choose custom plan called 375

- ```
plansource={.magic=195726186, .raw_parse_tree=0x2165200, .query_string="select count(*)      from test where
id=i_id", .commandTag="SELECT", .param_types=0x0, .num_params=0, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217df38, .cursor_options=0, .fixed_result='\00
0', .resultDesc=0x2165908, .context=0x2177f80, .query_list=0x2166518, .relationOids=0x218fe98, .invalidItems=0x0, .search_path=0x218fed0, .query_context=0x2178248,
.gplan=0x20b32a0, .is_oneshot='\000', .is_complete='\001', .is_saved='\001', .is_valid='\001', .generation=14, .next_saved=0x0, .generic_cost=?, .total_custom_cost=?, .num
_custom_plans=13}

boundParams={.paramFetch=0x7f3ddc31d9e0, .paramFetchArg=0x7fff8e199150, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217df38, .numParams=4, .params=[{.
value=1, .isnull='\000', .pflags=1, .ptype=23}, ...]} avg_custom_cost=?
```

PostgreSQL plan cache management

■ n次调用后, 不管使用id=1还是id=2查询都会继续使用custom plan. 原因 :

■ digoal=# explain select count(*) from test where id=2;

■ QUERY PLAN

■ Aggregate (cost=4.45..4.46 rows=1 width=0)

■ -> Index Scan using idx_test_1 on test (cost=0.43..4.45 rows=1 width=0)

■ Index Cond: (id = 2)

■ (3 rows)

■ digoal=# explain select count(*) from test where id=1;

■ QUERY PLAN

■ Aggregate (cost=42948.06..42948.07 rows=1 width=0)

■ -> Seq Scan on test (cost=0.00..37948.05 rows=2000004 width=0)

■ Filter: (id = 1)

■ (3 rows)

■ 一次id=2的调用加4次id=1的调用得到的avg custom plan cost如下 :

■ digoal=# select (4.45+42948*4)/5;

■ ?column?

■ -----

■ 34359.290000000000

■ (1 row)

PostgreSQL plan cache management

- generic plan = seq scan
- 全表扫描(generic plan)id=2的 cost显然大于 avg custom cost 34359.
- 所以会选择custom plan.
- digoal=# explain select count(*) from test where id=2;
 QUERY PLAN

■ Aggregate (cost=37948.05..37948.06 rows=1 width=0)
 -> Seq Scan on test (cost=0.00..37948.05 rows=1 width=0)
 Filter: (id = 2)
■ (3 rows)
- 使用custom plan后, avg custom cost将进一步降低.
- digoal=# select (4.45*2+42948*4)/6;
 ?column?

■ 28633.483333333333
■ (1 row)
- 所以后面会一直使用custom plan.

PostgreSQL plan cache management

- 场景2
- 前5次调用id=1的查询, 得到generic plan, 并可持续使用. (并且能观察到使用id=2时将选择custom plan)
- 前面讲了generic plan cache 基于会话, 所以请退出会话后测试.
- psql
- digoal=# select f_test_cnt(1);
- f_test_cnt
- -----
- 2000000
- (1 row)
- 5次后, select count(*) from test where id=i_id; 不再需要custom plan.

- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_start")select f_test_cnt(1);4522
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_parse_start")select f_test_cnt(1);4522
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_rewrite_start")select f_test_cnt(1);4522
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_plan_start")4522
- process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_execute_start")4522
- choose custom plan called 396

PostgreSQL plan cache management

- plansource={.magic=195726186, .raw_parse_tree=0x2165200, .query_string="select count(*) from test where id=i_id", .commandTag="SELECT", .param_types=0x0, .num_params=0, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217df38, .cursor_options=0, .fixed_result='\000', .resultDesc=0x2165908, .context=0x2177f80, .query_list=0x2166518, .relationOids=0x218fe98, .invalidItems=0x0, .search_path=0x218fed0, .query_context=0x2178248, .gplan=0x20b32a0, .is_oneshot='\000', .is_complete='\001', .is_saved='\001', .is_valid='\001', .generation=6, .next_saved=0x0, .generic_cost=?, .total_custom_cost=?, .num_custom_plans=5}
boundParams={.paramFetch=0x7f3ddc31d9e0, .paramFetchArg=0x7fff8e199150, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217df38, .numParams=4, .params=[{.value=1, .isnull='\000', .pflags=1, .ptype=23}, ...]} avg_custom_cost=?
- choose custom plan called 397
- plansource={.magic=195726186, .raw_parse_tree=0x2190bb0, .query_string="SELECT res", .commandTag="SELECT", .param_types=0x0, .num_params=0, .parserSetup=0x7f3ddc318af0, .parserSetupArg=0x217e138, .cursor_options=0, .fixed_result='\000', .resultDesc=0x2190ed0, .context=0x2179018, .query_list=0x21916b8, .relationOids=0x0, .invalidItems=0x0, .search_path=0x2191978, .query_context=0x21788f0, .gplan=0x2192178, .is_oneshot='\000', .is_complete='\001', .is_saved='\001', .is_valid='\001', .generation=1, .next_saved=0x21650e8, .generic_cost=?, .total_custom_cost=?, .num_custom_plans=0} boundParams=ERROR avg_custom_cost=?

PostgreSQL plan cache management

■ 5次调用id=1的查询后, 生成generic plan=seq scan, 并且num custom plan=5, total custom plan= 42948.07 * 5 ; avg custom plan = 42948.07

■ 后续将持续使用 generic plan, 的原因 :

■ id=2的generic plan cost < avg custom plan cost.

■ digoal=# set enable_indexscan=off; set enable_bitmapscan=off;

■ digoal=# explain select count(*) from test where id=2; (如果generic plan为全表扫描的话, 使用全表扫描得到的cost)

■ QUERY PLAN

■ -----
■ Aggregate (cost=37948.05..37948.06 rows=1 width=0)

■ -> Seq Scan on test (cost=0.00..37948.05 rows=1 width=0)

■ Filter: (id = 2)

■ (3 rows)

■ 显然id=2的generic plan cost(37948.06) < avg customplan cost(42948.07) 所以会选择generic plan.

■ digoal=# explain select count(*) from test where id=1; (如果前5次都是使用id=1的调用, 那么custom plan 的平均cost)

■ QUERY PLAN

■ -----
■ Aggregate (cost=42948.06..42948.07 rows=1 width=0)

■ -> Seq Scan on test (cost=0.00..37948.05 rows=2000004 width=0)

■ Filter: (id = 1)

■ (3 rows)

PostgreSQL plan cache management

- 如果要跟踪整个cacheplan.c 的函数调用, 使用通配符*
- stap -D MAXSTRINGLEN=99999 -e '
probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_start") {
 println(pn(), user_string(\$arg1), pid())
}
probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_parse_start") {
 println(pn(), user_string(\$arg1), pid())
}
probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_rewrite_start") {
 println(pn(), user_string(\$arg1), pid())
}
probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_plan_start") {
 println(pn(), pid())
}
probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query_execute_start") {
 println(pn(), pid())
}
probe process("/home/pg93/pgsql9.3.3/bin/postgres").function("*@/opt/soft_bak/postgresql-9.3.3/src/backend/utils/cache/plancache.c") {
 printf("%s called\n %s\n", pp(), \$\$vars\$\$)
}
'

PostgreSQL trace和debug

- 跟踪的目的
- 消息等级分类
- 消息内容开关
- 消息输出到何处
- 消息输出等级配置(过滤)
- 例子

PostgreSQL trace和debug

■ trace和debug的目的

- 从数据库中得到更多的信息,
- 跟踪定位问题.

■ PostgreSQL 对信息等级的分类例子

- `src/include/utils/elog.h`
- `/* Error level codes */`
- `#define DEBUG5 10 /* Debugging messages, in categories of`
`* decreasing detail. */`
- `#define DEBUG4 11`
- `#define DEBUG3 12`
- `#define DEBUG2 13`
- `#define DEBUG1 14 /* used by GUC debug_* variables */`
- `#define LOG 15 /* Server operational messages; sent only to`
`* server log by default. */`
- `#define COMMERROR 16 /* Client communication problems; same as LOG`
`* for server reporting, but never sent to`
`* client. */`
- `#define INFO 17 /* Messages specifically requested by user (eg`
`* VACUUM VERBOSE output); always sent to`
`* client regardless of client_min_messages,`

PostgreSQL trace和debug

```
■ * but by default not sent to server log. */  
■ #define NOTICE      18      /* Helpful messages to users about query  
■                                     * operation; sent to client and server log by  
■                                     * default. */  
■ #define WARNING     19      /* Warnings. NOTICE is for expected messages  
■                                     * like implicit sequence creation by SERIAL.  
■                                     * WARNING is for unexpected messages. */  
■ #define ERROR       20      /* user error - abort transaction; return to  
■                                     * known state */  
■ #ifdef WIN32  
■ #define PGERROR     20  
■ #endif  
■ #define FATAL        21      /* fatal error - abort process */  
■ #define PANIC        22      /* take down the other backends with me */
```

PostgreSQL trace和debug

■ PostgreSQL 如何输出消息, (elog, ereport)

```
■ #ifdef HAVE__BUILTIN_CONSTANT_P
■     #define elog(level, ...) \
■         do { \
■             elog_start(__FILE__, __LINE__, PG_FUNCNAME_MACRO); \
■             elog_finish(level, __VA_ARGS__); \
■             if (__builtin_constant_p(level) && (level) >= ERROR) \
■                 pg_unreachable(); \
■         } while(0)
■     #else /* !HAVE__BUILTIN_CONSTANT_P */
■         #define elog(level, ...) \
■             do { \
■                 int      level_; \
■                 elog_start(__FILE__, __LINE__, PG_FUNCNAME_MACRO); \
■                 level_ = (level); \
■                 elog_finish(level_, __VA_ARGS__); \
■                 if (level_ >= ERROR) \
■                     pg_unreachable(); \
■             } while(0)
■     #endif /* HAVE__BUILTIN_CONSTANT_P */
■     #define ereport(level, rest) \
■         ereport_domain(level, TEXTDOMAIN, rest)
```

PostgreSQL trace和debug

■ PostgreSQL 定义的消息开关举例, 记录客户端的连接信息的开关

```
■ src/backend/postmaster/postmaster.c
■ if (Log_connections) // 对应参数, postgresql.conf. log_connections
■ {
■     if (remote_port[o])
■         ereport(LOG,
■                 (errmsg("connection received: host=%s port=%s",
■                         remote_host,
■                         remote_port)));
■     else
■         ereport(LOG,
■                 (errmsg("connection received: host=%s",
■                         remote_host)));
■ }
```

PostgreSQL trace和debug

■ PostgreSQL 的消息开关1 - 参数开关(通过postgresql.conf配置)

- application_name (string) 输出客户端指定的应用名
- debug_print_parse (boolean) 输出parse阶段的信息
- debug_print_rewritten (boolean) 输出rewritten阶段的信息
- debug_print_plan (boolean) 输出执行计划信息
- debug_pretty_print (boolean) 以比较易读的方式输出以上三个参数的信息
- log_checkpoints (boolean) 输出检查点信息
- log_connections (boolean) 输出连接信息
- log_disconnections (boolean) 输出断开连接信息
- log_duration (boolean) 输出SQL语句执行时间信息
- log_error_verbosity (enum) 输出详细级别, 例如函数以及代码文件名
- log_hostname (boolean) 主机名
- log_line_prefix (string) 输出位置为标准输出或者系统日志时, 指定输出格式
- log_lock_waits (boolean) 输出锁等待
- log_statement (enum) 按过滤条件输出SQL语句, 例如ddl, mod, all
- log_temp_files (integer) 按过滤条件输出临时文件相关信息
- log_timezone (string) 指定时区

PostgreSQL trace和debug

■ PostgreSQL 的消息开关1 - 参数开关(通过postgresql.conf配置)

- #track_activities = on # 跟踪SQL语句的状态以及SQL语句的内容. 输出到pg_stat_activity
- #track_counts = on # 计数器, 例如表被插入了多少次, 更新了多少次
- #track_io_timing = off # 跟踪IO操作的时间, 例如一个SQL语句带来的IO时间是多少.
- #track_functions = none # none, pl, all # 跟踪函数的调用次数, 时间.
- track_activity_query_size = 4096 # (change requires restart) # 输出SQL语句的最大长度, 超出截断
- #update_process_title = on # 更新进程状态信息, 例如从select到idle, 显示进程当前的状态.
- #stats_temp_directory = 'pg_stat_tmp'
- # - Statistics Monitoring - getrusage()
- #log_parser_stats = off
- #log_planner_stats = off
- #log_executor_stats = off
- #log_statement_stats = off

PostgreSQL trace和debug

- PostgreSQL 的消息开关1 - 参数开关(通过postgresql.conf配置)
- 以下为隐含参数, 需要配合宏的定义使用.
 - `debug_assertions` 需要开启宏`USE_ASSERT_CHECKING`
 - `trace_locks`, `trace_lwlocks`, `trace_userlocks`, `trace_lock_oidmin`, `trace_lock_table`, `debug_deadlocks` 需要开启宏`LOCK_DEBUG`
 - `log_btreet_build_stats` 需要开启宏`BTREE_BUILD_STATS` (`getusage()`)
 - `wal_debug` 需要开启宏`WAL_DEBUG`.

PostgreSQL trace和debug

- PostgreSQL 的消息开关2 - 宏开关(通过Makefile.custom, pg_config_manual.h,等配置文件进行配置)
- 需要在编译PostgreSQL软件时完成

```
■ src/include/pg_config_manual.h
■ /*
■   * Define this to force all parse and plan trees to be passed through
■   * copyObject(), to facilitate catching errors and omissions in
■   * copyObject().
■   */
■ /* #define COPY_PARSE_PLAN TREES */
■ /*
■   * Enable debugging print statements for lock-related operations.
■   */
■ /* #define LOCK_DEBUG */
■ /*
■   * Enable debugging print statements for WAL-related operations; see
■   * also the wal_debug GUC var.
■   */
■ /* #define WAL_DEBUG */
```

PostgreSQL trace和debug

- PostgreSQL 的消息开关2 - 宏开关(通过Makefile.custom, pg_config_manual.h,等配置文件进行配置)

```
■ /*  
■ * Enable tracing of resource consumption during sort operations;  
■ * see also the trace_sort GUC var. For 8.1 this is enabled by default.  
■ */  
■ #define TRACE_SORT 1  
■ /*  
■ * Enable tracing of syncscan operations (see also the trace_syncscan GUC var).  
■ */  
■ /* #define TRACE_SYNCSCAN */  
■ /*  
■ * Other debug #defines (documentation, anyone?)  
■ */  
■ /* #define HEAPDEBUGALL */  
■ /* #define ACLDEBUG */  
■ /* #define RTDEBUG */
```

PostgreSQL trace和debug

- PostgreSQL 的消息开关2 - 宏开关(通过Makefile.custom, pg_config_manual.h, 等配置文件进行配置)
- 只有定义了对应的宏才能开启这些跟踪的输出.

- src/backend/optimizer/path/allpaths.c
 - #ifdef OPTIMIZER_DEBUG
 - debug_print_rel(root, rel);
 - #endif
- src/backend/optimizer/geqo/geqo_main.c
 - #ifdef GEQO_DEBUG
 - elog(DEBUG1, "GEQO best is %.2f after %d generations",
 - pool->data[o].worth, number_generations);
 - #endif
- src/backend/regex/regcomp.c
 - #ifdef REG_DEBUG
 - if (debug != NULL)
 - {
 - fprintf(debug, "\n\n\n===== TREE FIXED =====\n");
 - dumpst(v->tree, debug, 1);
 - }
 - #endif

PostgreSQL trace和debug

- PostgreSQL 的消息开关2 - 宏开关(通过Makefile.custom, pg_config_manual.h, 等配置文件进行配置)

- src/backend/utils/adt/acl.c
 - **#ifdef ACLDEBUG**
 - elog(LOG, "aclparse: input = \"%s\"", s);
 - **#endif**

 - src/backend/utils/adt/arrayfuncs.c
 - **#ifdef ARRAYDEBUG**
 - printf("array_in- ndim %d (", ndim);
 - for (i = 0; i < ndim; i++)
 - {
 - printf(" %d", dim[i]);
 - };
 - printf(") for %s\n", string);
 - **#endif**
 - ...

PostgreSQL trace和debug

■ 消息可以输出到哪些地方

■ 日志文件

- log_destination = 'csvlog' # Valid values are combinations of
 - stderr, csvlog, syslog, and eventlog,
 - depending on platform. csvlog
 - requires logging_collector to be on.
- log_directory = '/mnt/csvlog' # directory where log files are written,
 - can be absolute or relative to PGDATA
- #log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log' # log file name pattern,
 - can include strftime() escapes

■ 客户端

PostgreSQL trace和debug

■ 日志文件以及客户端消息过滤级别

```
■ #client_min_messages = notice      # values in order of decreasing detail: 客户端消息过滤级别
■                 # debug5
■                 # debug4
■                 # debug3
■                 # debug2
■                 # debug1
■                 # log
■                 # notice
■                 # warning
■                 # error
■ #log_min_messages = warning        # values in order of decreasing detail: 日志文件消息过滤级别
■                 # debug5
■                 # debug4
■                 # debug3
■                 # debug2
■                 # debug1
■                 # info
■                 # notice
■                 # warning
```

PostgreSQL trace和debug

```
■          # error
■          # log
■          # fatal
■          # panic
■ #log_min_error_statement = error      # values in order of decreasing detail: // 针对引起错误的日志, 是否记录引起错误的SQL语句的过滤级别.
■          # debug5
■          # debug4
■          # debug3
■          # debug2
■          # debug1
■          # info
■          # notice
■          # warning
■          # error
■          # log
■          # fatal
■          # panic (effectively off)
```

PostgreSQL trace和debug

- log_min_error_statement测试：
 - digoal=# show log_min_error_statement;
 - log_min_error_statement
 - -----
 - error
 - (1 row)
- digoal=# show log_min_error_statements;
 - ERROR: unrecognized configuration parameter "log_min_error_statements"
- 查看日志文件postgresql-2014-04-01_000000.csv, 记录了SQL
 - 2014-04-01 18:39:35.758 CST,"postgres","digoal",16752,"[local]",533a975b.4170,3,"SHOW",2014-04-01 18:39:23 CST,1/207,0,ERROR,42704,"unrecognized configuration parameter ""log_min_error_statements""",,,,,"show log_min_error_statements;","","GetConfigOptionByName, guc.c:6946","psql"

PostgreSQL trace和debug

- log_min_error_statement测试：
 - 改成fatal
 - digoal=# set log_min_error_statement=fatal;
 - SET
 - digoal=# show log_min_error_statements;
 - ERROR: unrecognized configuration parameter "log_min_error_statements"
 - **查看csvlog, 该错误消息, 不记录SQL.**
 - 2014-04-01 18:40:20.440 CST,"postgres","digoal",16752,"[local]",533a975b.4170,4,"SHOW",2014-04-01 18:39:23 CST,1/210,0,ERROR,42704,"unrecognized configuration parameter ""log_min_error_statements""",,,,,,,,"GetConfigOptionByName, guc.c:6946","psql"

PostgreSQL trace和debug

- 把客户端的日志打印级别改为error, 那么notice级别的消息将不会被打印.

- digoal=# set client_min_messages=error;

- SET

- digoal=# do language plpgsql \$\$

- digoal\$# declare

- digoal\$# begin

- digoal\$# raise notice 'this is a test.';

- digoal\$# end;

- digoal\$# \$\$;

- DO

PostgreSQL trace和debug

- 把客户端的日志打印级别改为notice, 那么这个例子的notice级别的消息会被打印出来.

- digoal=# set client_min_messages=notice;

- SET

- digoal=# do language plpgsql \$\$

- declare

- begin

- raise notice 'this is a test.';

- end;

- \$\$;

- **NOTICE: this is a test.**

- DO

- 因此要输出最详细的全局信息, 把log_min_messages设置为debug5,

- 如果要得到当前会话的最详细的信息, 把client_min_messages设置为debug5即可.

PostgreSQL trace和debug

- 其他配置
 - 日志中包含代码信息
 - log_error_verbosity = verbose # terse, default, or verbose messages
 - 客户端消息包含代码信息
 - \set VERBOSITY verbose

- 例如：
 - digoal=# \set VERBOSITY verbose
 - digoal=# show log_min_error_statements;
 - ERROR: 42704: unrecognized configuration parameter "log_min_error_statements"
 - LOCATION: GetConfigOptionByName, guc.c:6946 -- 输出以上信息的代码位置

PostgreSQL trace和debug

- 不需要开启某些宏的trace 和 debug举例
- 1. 跟踪排序
- digoal=# set client_min_messages=log;
- digoal=# set trace_sort=on;
- digoal=# \set VERBOSITY verbose
- digoal=# select count(*) from (select * from pg_class order by relpages) t;
- LOG: 00000: begin tuple sort: nkeys = 1, workMem = 1024, randomAccess = f
- LOCATION: tuplesort_begin_heap, tuplesort.c:617
- LOG: 00000: performsort starting: CPU 0.00s/0.00u sec elapsed 0.00 sec
- LOCATION: tuplesort_performsort, tuplesort.c:1319
- LOG: 00000: performsort done: CPU 0.00s/0.00u sec elapsed 0.00 sec
- LOCATION: tuplesort_performsort, tuplesort.c:1394
- LOG: 00000: internal sort ended, 105 KB used: CPU 0.00s/0.00u sec elapsed 0.00 sec
- LOCATION: tuplesort_end, tuplesort.c:932
- count
- -----
- 296
- (1 row)

PostgreSQL trace和debug

- 2. 跟踪执行计划
- digoal=# set client_min_messages=log;
- digoal=# set debug_pretty_print = on;
- digoal=# \set VERBOSITY verbose
- **digoal=# set debug_print_parse=on;**
- digoal=# select count(*) from (select * from pg_class order by relpages) t;
- LOG: parse tree:
- DETAIL: {QUERY
■ ... 略
- **digoal=# set debug_print_rewritten = on;**
- LOG: rewritten parse tree:
- DETAIL: (
■ {QUERY
■ ... 略
- **digoal=# set debug_print_plan = on;**
- digoal=# select count(*) from (select * from pg_class order by relpages) t;
- LOG: plan:
- DETAIL: {PLANNEDSTMT
■ :commandType 1
- ... 略

PostgreSQL trace和debug

- 3. 跟踪死锁
- **SESSION A :**
 - digoal=# create table t(id int, info text);
 - digoal=# insert into t values (1,'test'),(2,'test');
 - INSERT 0 2
 - digoal=# begin;
 - BEGIN
 - digoal=# update t set info='new' where id=1;
 - UPDATE 1
- **SESSOIN B :**
 - digoal=# begin;
 - BEGIN
 - digoal=# update t set info='new' where id=2;
 - UPDATE 1
 - digoal=# update t set info='new' where id=1;
- **SESSION A :**
 - digoal=# update t set info='new' where id=2;
 - ERROR: deadlock detected
 - DETAILED: Process 6173 waits for ShareLock on transaction 3268512748; blocked by process 6214.
 - Process 6214 waits for ShareLock on transaction 3268512747; blocked by process 6173.
 - HINT: See server log for query details.

PostgreSQL trace和debug

- 4. 跟踪锁超时SQL
- log_lock_waits = on
- deadlock_timeout = 1s
- SESSION A :
 - digoal=# begin;
 - BEGIN
 - digoal=# update t set info='new' where id=1;
 - UPDATE 1
- SESSION B :
 - digoal=# set client_min_messages=log;
 - SET
 - digoal=# begin;
 - BEGIN
 - digoal=# update t set info='new' where id=1;
 - LOG: statement: update t set info='new' where id=1;
 - LOG: process 6499 still waiting for ShareLock on transaction 3268512749 after 1000.177 ms

PostgreSQL trace和debug

- 5. 跟踪超时SQL
- `log_min_duration_statement = 100ms # 记录执行时间超过100毫秒的SQL`

- 6. 跟踪检查点,连接和断开连接信息
- `log_checkpoints = on`
- `log_connections = on`
- `log_disconnections = on`

PostgreSQL trace和debug

- 通过宏开关, 开启某些信息的输出.
- [root@db-172-16-3-150 postgresql-9.3.3]# **vi src/Makefile.custom**
- **CFLAGS+=-DLOCK_DEBUG**
- **CFLAGS+=-DBTREE_BUILD_STATS**
- **CFLAGS+=-DWAL_DEBUG**
- **CFLAGS+=-DOPTIMIZER_DEBUG**
- **CFLAGS+=-DGEQO_DEBUG**
- **CFLAGS+=-DCOPY_PARSE_PLAN TREES**
- **CFLAGS+=-DTRACE_SYNCSCAN**

- 重新配置编译安装, 使用已有的配置选项(\$PGSRC/config.log)
- **configure**
- **make && make install**
- 确认已使用定义的宏.
- **pg_config**
- **CFLAGS = -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -fwrapv -g -DLOCK_DEBUG -DBTREE_BUILD_STATS -DWAL_DEBUG -DOPTIMIZER_DEBUG -DGEQO_DEBUG -DCOPY_PARSE_PLAN TREES -DTRACE_SYNCSCAN**

PostgreSQL trace和debug

- 1. 跟踪btree索引建立时的资源开销信息
- digoal=# create table test(id int, info text);
- digoal=# set client_min_messages=log;
- digoal=# insert into test select generate_series(1,100000),'test';
- LOG: statement: insert into test select generate_series(1,100000),'test';
- INSERT 0 100000
- digoal=# set log_btreet_build_stats=on; **-- 在当前会话打开隐含参数.**
- SET
- digoal=# \set VERBOSITY verbose
- digoal=# create index idx_test_1 on test(id);
- LOG: 00000: statement: create index idx_test_1 on test(id);
- LOCATION: exec_simple_query, postgres.c:890
- **LOG: 00000: BTREE BUILD (Spool) STATISTICS**
- DETAIL: ! system usage stats: **-- 输出getusage()的信息.**
- ! 0.048502 elapsed 0.040993 user 0.006999 system sec
- ! [0.328949 user 0.038994 sys total]
- ! 0/0 [0/13888] filesystem blocks in/out
- ! 0/1706 [0/8926] page faults/reclaims, 0 [0] swaps

PostgreSQL trace和debug

- ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
- ! 0/0 [74/1] voluntary/involuntary context switches
- LOCATION: ShowUsage, postgres.c:4400
- **LOG: 00000: BTREE BUILD STATS**
- DETAIL: ! system usage stats:
- ! 0.043534 elapsed 0.031996 user 0.002999 system sec
- ! [0.360945 user 0.041993 sys total]
- ! 0/4864 [0/18752] filesystem blocks in/out
- ! 0/741 [0/9667] page faults/reclaims, 0 [0] swaps
- ! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
- ! 9/1 [83/2] voluntary/involuntary context switches
- LOCATION: ShowUsage, postgres.c:4400
- CREATE INDEX

PostgreSQL trace和debug

- 2. 跟踪锁
- digoal=# set trace_locks=on;
- LOG: 00000: LockReleaseAll: lockmethod=1
- LOCATION: LockReleaseAll, lock.c:1954
- LOG: 00000: LockReleaseAll done
- LOCATION: LockReleaseAll, lock.c:2199
- SET
- digoal=# update t set info='test' where id=1;
- LOG: 00000: statement: update t set info='test' where id=1;
- LOCATION: exec_simple_query, postgres.c:890
- LOG: 00000: LockAcquire: lock [16384,26061] RowExclusiveLock // 输出锁对象(db_oid, rel_oid), 锁类别等信息. 根据给出的代码解读输出的内容.
- LINE 1: update t set info='test' where id=1;
- ^
- LOCATION: LockAcquireExtended, lock.c:729
- LOG: 00000: LockAcquire: lock [16384,26061] RowExclusiveLock
- LOCATION: LockAcquireExtended, lock.c:729
- LOG: 00000: LockAcquire: lock [16384,26061] ExclusiveLock
- LOCATION: LockAcquireExtended, lock.c:729
- LOG: 00000: LockAcquire: new: lock(0x7f6970d12b00) id(16384,26061,0,1,3,1) grantMask(0) req(0,0,0,0,0,0)=0 grant(0,0,0,0,0,0)=0 wait(0) type(ExclusiveLock)
- LOCATION: LOCK_PRINT, lock.c:318
- LOG: 00000: LockAcquire: new: proclock(0x7f6970d98440) lock(0x7f6970d12b00) method(1) proc(0x7f6971004860) hold(0)
- LOCATION: PROCLOCK_PRINT, lock.c:330

PostgreSQL trace和debug

- LOG: 00000: LockCheckConflicts: no conflict: proclock(0x7f6970d98440) lock(0x7f6970d12b00) method(1) proc(0x7f6971004860) hold(0)
- LOCATION: PROCLOCK_PRINT, lock.c:330
- LOG: 00000: GrantLock: lock(0x7f6970d12b00) id(16384,26061,0,1,3,1) grantMask(80) req(0,0,0,0,0,1)=1 grant(0,0,0,0,0,1)=1 wait(0) type(ExclusiveLock)
- LOCATION: LOCK_PRINT, lock.c:318
- LOG: 00000: process 6499 still waiting for ShareLock on transaction 3268512749 after 1000.118 ms
- LOCATION: ProcSleep, proc.c:1246
- LOG: 00000: process 6499 acquired ShareLock on transaction 3268512749 after 5021.627 ms
- LOCATION: ProcSleep, proc.c:1250
- LOG: 00000: LockRelease: lock [16384,26061] ExclusiveLock
- LOCATION: LockRelease, lock.c:1761
- LOG: 00000: LockRelease: found: lock(0x7f6970d12b00) id(16384,26061,0,1,3,1) grantMask(80) req(0,0,0,0,0,1)=1 grant(0,0,0,0,0,1)=1 wait(0) type(ExclusiveLock)
- LOCATION: LOCK_PRINT, lock.c:318
- LOG: 00000: LockRelease: found: proclock(0x7f6970d98440) lock(0x7f6970d12b00) method(1) proc(0x7f6971004860) hold(80)
- LOCATION: PROCLOCK_PRINT, lock.c:330
- LOG: 00000: UnGrantLock: updated: lock(0x7f6970d12b00) id(16384,26061,0,1,3,1) grantMask(0) req(0,0,0,0,0,0)=0 grant(0,0,0,0,0,0)=0 wait(0) type(ExclusiveLock)
- LOCATION: LOCK_PRINT, lock.c:318
- LOG: 00000: UnGrantLock: updated: proclock(0x7f6970d98440) lock(0x7f6970d12b00) method(1) proc(0x7f6971004860) hold(0)
- LOCATION: PROCLOCK_PRINT, lock.c:330
- LOG: 00000: CleanUpLock: deleting: proclock(0x7f6970d98440) lock(0x7f6970d12b00) method(1) proc(0x7f6971004860) hold(0)
- LOCATION: PROCLOCK_PRINT, lock.c:330

PostgreSQL trace和debug

- LOG: 00000: CleanUpLock: deleting: lock(0x7f6970d12b00) id(16384,26061,0,1,3,1) grantMask(0) req(0,0,0,0,0,0)=0 grant(0,0,0,0,0,0)=0 wait(0) type(INVALID)
 - LOCATION: LOCK_PRINT, lock.c:318
 - LOG: 00000: LockReleaseAll: lockmethod=1
 - LOCATION: LockReleaseAll, lock.c:1954
 - LOG: 00000: LockReleaseAll done
 - LOCATION: LockReleaseAll, lock.c:2199
 - UPDATE 1
-
- 根据给出的代码解读输出的内容：

PostgreSQL trace和debug

```
■ inline static void
■ LOCK_PRINT(const char *where, const LOCK *lock, LOCKMODE type)
■ {
■     if (LOCK_DEBUG_ENABLED(&lock->tag))
■         elog(LOG,
■             "%s: lock(%p) id(%u,%u,%u,%u,%u,%u) grantMask(%x) "
■             "req(%d,%d,%d,%d,%d,%d,%d)=%d "
■             "grant(%d,%d,%d,%d,%d,%d,%d)=%d wait(%d) type(%s)",
■             where, lock,
■             lock->tag.locktag_field1, lock->tag.locktag_field2,
■             lock->tag.locktag_field3, lock->tag.locktag_field4,
■             lock->tag.locktag_type, lock->tag.locktag_lockmethodid,
■             lock->grantMask,
■             lock->requested[1], lock->requested[2], lock->requested[3],
■             lock->requested[4], lock->requested[5], lock->requested[6],
■             lock->requested[7], lock->nRequested,
■             lock->granted[1], lock->granted[2], lock->granted[3],
■             lock->granted[4], lock->granted[5], lock->granted[6],
■             lock->granted[7], lock->nGranted,
■             lock->waitProcs.size,
■             LockMethods[LOCK_LOCKMETHOD(*lock)]->lockModeNames[type]);
■ }
```

PostgreSQL trace和debug

- 3. 跟踪wal
 - digoal=# set wal_debug=on;
 - digoal=# select pg_switch_xlog();
 - LOG: 00000: INSERT @ 20E/D4BB2878: prev 20E/D4BB2848; xid 0; len 0: XLOG
 - LOCATION: XLogInsert, xlog.c:1077
 - pg_switch_xlog
 - -----
 - 20E/D4BB2898
 - (1 row)

PostgreSQL trace和debug

```
■ #ifdef WAL_DEBUG
■ static void
■ xlog_outrec(StringInfo buf, XLogRecord *record)
■ {
■     int             i;
■     appendStringInfo(buf, "prev %X/%X; xid %u",
■                       (uint32) (record->xl_prev >> 32),
■                       (uint32) record->xl_prev,
■                       record->xl_xid);
■
■     appendStringInfo(buf, "; len %u",
■                       record->xl_len);
■     for (i = 0; i < XLR_MAX_BKP_BLOCKS; i++)
■     {
■         if (record->xl_info & XLR_BKP_BLOCK(i))
■             appendStringInfo(buf, "; bkp% d", i);
■     }
■     appendStringInfo(buf, "; %s", RmgrTable[record->xl_rmid].rm_name);
■ }
■ #endif /* WAL_DEBUG */
■ 封装后在xlog.c中输出.
```

PostgreSQL trace和debug

■ 其他跟踪方法

■ stap

- 适合Linux系统
- 编译时加上--enable-dtrace --enable-debug选项
- 需要systemtap环境

■ strace

- 跟踪系统调用

■ ltrace

- 跟踪库函数调用

■ gdb

- 调试

PostgreSQL trace和debug

- stap举例, 跟踪SQL, 会话的块设备读写统计, cache的读写统计.
- vi test.stp
- global io_stat1%[120000] // 非cache读写字节数(单SQL)
- global io_stat11%[120000] // 非cache读写耗时(单SQL)
- global io_stat2%[120000] // cache读写字节数(单SQL)
- global io_stat22%[120000] // cache读写耗时(单SQL)
- global io_stat3%[120000] // 非cache读取字节数(总,只关心设备号)
- global io_stat33%[120000] // 非cache读写耗时(总,只关心设备号)
- global io_stat4%[120000] // cache读写字节数(总,只关心设备号)
- global io_stat44%[120000] // cache读写耗时(总,只关心设备号)
- global del%[120000] // 因为foreach中不允许修改本数组, 所以需要使用另一个数组来存储索引, 方便删除

- probe vfs.read.return {
- try {
- if (\$return>0) {
- v_us=gettimeofday_us() - @entry(gettimeofday_us())
- if (devname!="N/A" && execname()=='postgres') { /*skip read from cache, filter postgres otherwise*/
- io_stat1[pid(),execname(),"R",devname] <<< \$return // 非cache读字节数(单SQL)
- io_stat11[pid(),execname(),"R",devname] <<< v_us // 非cache读耗时(单SQL)
- io_stat3["R",devname] <<< \$return // 非cache读字节数(总,只关心设备号)
- io_stat33["R",devname] <<< v_us // 非cache读耗时(总,只关心设备号)

PostgreSQL trace和debug

```
■    }
■    if (devname=="N/A" && execname()=="postgres") {
■        io_stat2[pid(),execname(),"R",devname] <<< $return // cache读字节数(单SQL)
■        io_stat22[pid(),execname(),"R",devname] <<< v_us    // cache读耗费时间(单SQL)
■        io_stat4["R",devname] <<< $return // cache读字节数(总,只关心设备号)
■        io_stat44["R",devname] <<< v_us    // cache读耗费时间(总,只关心设备号)
■    }
■    }
■    }
■    catch(msg) {
■        println("---", pn(), msg)
■    }
■ }
```

PostgreSQL trace和debug

```
■ probe vfs.write.return {
■   try {
■     if ($return>0) {
■       v_us=gettimeofday_us() - @entry(gettimeofday_us())
■       if (devname!="N/A" && execname()=="postgres") { /*skip read from cache, filter postgres otherwise*/
■         io_stat1[pid(),execname(),"W",devname] <<< $return
■         io_stat11[pid(),execname(),"W",devname] <<< v_us
■         io_stat3["W",devname] <<< $return
■         io_stat33["W",devname] <<< v_us
■       }
■       if (devname=="N/A" && execname()=="postgres") {
■         io_stat2[pid(),execname(),"W",devname] <<< $return
■         io_stat22[pid(),execname(),"W",devname] <<< v_us
■         io_stat4["W",devname] <<< $return
■         io_stat44["W",devname] <<< v_us
■       }
■     }
■   }
■   catch(msg) {
■     println("---", pn(), msg)
■   }
■ }
```

PostgreSQL trace和debug

```
■ probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query__start") {  
■   try {  
■     // SQL开始前, 先清除之前SQL的统计变量存储的信息.  
■     // 清除非CACHE读写统计变量的信息.  
■     // 因为foreach中不允许修改本数组, 所以需要使用另一个数组来存储索引, 方便删除, 这里就是del的用意.  
■     foreach([a,b,c,d] in io_stat1) {  
■       if (a==pid() && b==execname()) {  
■         del[a,b,c,d]=1 // 将a,b,c,d作为索引存储任意一个值到del数组. a,b,c,d就是一会需要清除的io_stat1,io_stat11的索引值.  
■       }  
■     }  
■     foreach([a,b,c,d] in del) {  
■       delete io_stat1[a,b,c,d]  
■       delete io_stat11[a,b,c,d]  
■     }  
■     delete del // 用完del后, 记得清除del的值.  
■ }
```

PostgreSQL trace和debug

```
■ // 清除CACHE读写统计变量的信息.  
■ foreach([a,b,c,d] in io_stat2) {  
■     if (a==pid() && b==execname()) {  
■         del[a,b,c,d]=1  
■     }  
■ }  
■ foreach([a,b,c,d] in del) {  
■     delete io_stat2[a,b,c,d]  
■     delete io_stat22[a,b,c,d]  
■ }  
■ delete del  
■ }  
■ catch(msg) {  
■     println("---", pn(), msg)  
■ }  
■ }
```

PostgreSQL trace和debug

```
■ probe process("/home/pg93/pgsql9.3.3/bin/postgres").mark("query__done") {  
■   try {  
■     // 输出SQL语句  
■     printf("query: %s\n", user_string($arg1))  
  
■     // 非cache统计  
■     println("非cache输出")  
■     foreach([a,b,c,d] in io_stat1 @sum -) {  
■       if (c == "R" && a==pid() && b==execname()) {  
■         var1 = @count(io_stat1[a,b,c,d]) // 请求次数  
■         var2 = @sum(io_stat1[a,b,c,d]) / 1024 // 请求K字节数  
■         var3 = @sum(io_stat11[a,b,c,d]) // 请求时间, us  
■         spvar1 = ((var3!=0) ? ((1000000*var1)/var3) : 0) // 请求次数每秒  
■         spvar2 = ((var3!=0) ? ((1000000*var2)/var3) : 0) // 请求K字节数每秒  
■         printf("-%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var1, var2, spvar1, spvar2)  
■     }  
■   }  
■ }
```

PostgreSQL trace和debug

```
■ if (c == "W" && a==pid() && b==execname()) {  
■     var4 = @count(io_stat1[a,b,c,d]) // 请求次数  
■     var5 = @sum(io_stat1[a,b,c,d]) / 1024 // 请求K字节数  
■     var6 = @sum(io_stat11[a,b,c,d]) // 请求时间  
■     spvar4 = ((var6!=0) ? ((1000000*var4)/var6) : 0) // 请求次数每秒  
■     spvar5 = ((var6!=0) ? ((1000000*var5)/var6) : 0) // 请求K字节数每秒  
■     printf("-%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var4, var5, spvar4, spvar5)  
■ }  
■ del[a,b,c,d]=1 // 使用a,b,c,d填充del数组, 用于清除io_stat1, io_stat11.  
■ }  
■ foreach([a,b,c,d] in del) {  
■     delete io_stat1[a,b,c,d]  
■     delete io_stat11[a,b,c,d]  
■ }  
■ delete del // 用完清除del  
  
■ // 清除非cache统计用过的本地变量, 后面的cache统计需要继续使用.  
■ delete var1  
■ delete var2  
■ delete var3  
■ delete var4
```

PostgreSQL trace和debug

```
■ delete var5
■ delete var6
■ delete spvar1
■ delete spvar2
■ delete spvar4
■ delete spvar5

■ // cache统计
■ println("cache输出")
■ foreach([a,b,c,d] in io_stat2 @sum -) {
■   if (c == "R" && a==pid() && b==execname()) {
■     var1 = @count(io_stat2[a,b,c,d]) // 请求次数
■     var2 = @sum(io_stat2[a,b,c,d]) / 1024 // 请求K字节数
■     var3 = @sum(io_stat22[a,b,c,d]) // 请求时间
■     spvar1 = ((var3!=0) ? ((1000000*var1)/var3) : 0) // 请求次数每秒
■     spvar2 = ((var3!=0) ? ((1000000*var2)/var3) : 0) // 请求K字节数每秒
■     printf("-%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var1, var2, spvar1, spvar2)
■   }
■   if (c == "W" && a==pid() && b==execname()) {
■     var4 = @count(io_stat2[a,b,c,d]) // 请求次数
■     var5 = @sum(io_stat2[a,b,c,d]) / 1024 // 请求K字节数
```

PostgreSQL trace和debug

```
■    var6 = @sum(io_stat22[a,b,c,d]) // 请求时间
■    spvar4 = ((var6!=0) ? ((1000000*var4)/var6) : 0) // 请求次数每秒
■    spvar5 = ((var6!=0) ? ((1000000*var5)/var6) : 0) // 请求K字节数每秒
■    printf("%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var4, var5, spvar4, spvar5)
■    }
■    del[a,b,c,d]=1
■    }
■    foreach([a,b,c,d] in del) {
■        delete io_stat1[a,b,c,d]
■        delete io_stat11[a,b,c,d]
■    }
■    delete del
■    }
■    catch(msg) {
■        println("---", pn(), msg)
■    }
■ }
```

PostgreSQL trace和debug

```
■ probe end{
■   try {
■     println("-----END-----")
■     // 非cache, 按设备的读写统计输出.
■     println("非cache输出")
■     foreach([c,d] in io_stat3 @sum -) {
■       if (c == "R") {
■         var1 = @count(io_stat3[c,d]) // 请求次数
■         var2 = @sum(io_stat3[c,d]) / 1024 // 请求K字节数
■         var3 = @sum(io_stat33[c,d]) // 请求时间
■         spvar1 = ((var3!=0) ? ((1000000*var1)/var3) : 0) // 请求次数每秒
■         spvar2 = ((var3!=0) ? ((1000000*var2)/var3) : 0) // 请求K字节数每秒
■         printf("-%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var1, var2, spvar1, spvar2)
■       }
■       if (c == "W") {
■         var4 = @count(io_stat3[c,d]) // 请求次数
■         var5 = @sum(io_stat3[c,d]) / 1024 // 请求K字节数
■         var6 = @sum(io_stat33[c,d]) // 请求时间
■         spvar4 = ((var6!=0) ? ((1000000*var4)/var6) : 0) // 请求次数每秒
■         spvar5 = ((var6!=0) ? ((1000000*var5)/var6) : 0) // 请求K字节数每秒
■         printf("-%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var4, var5, spvar4, spvar5)
```

PostgreSQL trace和debug

```
■    }
■    }

■ delete var1
■ delete var2
■ delete var3
■ delete var4
■ delete var5
■ delete var6
■ delete spvar1
■ delete spvar2
■ delete spvar4
■ delete spvar5

■ // cache, 按设备的读写统计输出.
■ println("cache输出")
■ foreach([c,d] in io_stat4 @sum -) {
■     if (c == "R") {
■         var1 = @count(io_stat4[c,d]) // 请求次数
■         var2 = @sum(io_stat4[c,d]) / 1024 // 请求K字节数
■         var3 = @sum(io_stat44[c,d]) // 请求时间
```

PostgreSQL trace和debug

```
■ spvar1 = ((var3!=0) ? ((1000000*var1)/var3) : 0) // 请求次数每秒
■ spvar2 = ((var3!=0) ? ((1000000*var2)/var3) : 0) // 请求K字节数每秒
■ printf("-%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var1, var2, spvar1, spvar2)
■ }
■ if (c == "W") {
■     var4 = @count(io_stat4[c,d]) // 请求次数
■     var5 = @sum(io_stat4[c,d]) / 1024 // 请求K字节数
■     var6 = @sum(io_stat44[c,d]) // 请求时间
■     spvar4 = ((var6!=0) ? ((1000000*var4)/var6) : 0) // 请求次数每秒
■     spvar5 = ((var6!=0) ? ((1000000*var5)/var6) : 0) // 请求K字节数每秒
■     printf("-%s-devname:%s, reqs:%d, reqKbytes:%d, reqs/s:%d, reqKbytes/s:%d\n", c, d, var4, var5, spvar4, spvar5)
■ }
■ }

■ // 结束后清除所有的全局变量的值.
■ delete io_stat1
■ delete io_stat11
■ delete io_stat2
■ delete io_stat22
■ delete io_stat3
■ delete io_stat33
```

PostgreSQL trace和debug

```
■ delete io_stat4
■ delete io_stat44
■ delete del
■ }
■ catch(msg) {
■   println("---", pn(), msg)
■ }
■ }
```

PostgreSQL trace和debug

■ 输出示例

```
[root@db-172-16-3-150 ~]# stap test.stp
query: explain (analyze,verbose,costs,buffers,timing) select count(*) from generate_series(1,1000000);
非cache输出
-R-devname:sdb1, reqs:428, reqKbytes:13671, reqs/s:70568, reqKbytes/s:2254080
-W-devname:sdb1, reqs:428, reqKbytes:13671, reqs/s:24126, reqKbytes/s:770631
cache输出
-W-devname:N/A, reqs:1, reqKbytes:0, reqs/s:71428, reqKbytes/s:0

diggoal=# explain (analyze,verbose,costs,buffers,timing) select count(*) from generate_series(1,1000000);
          QUERY PLAN
Aggregate (cost=12.50..12.51 rows=1 width=0) (actual time=610.733..610.733 rows=1 loops=1)
  Output: count(*)
  Buffers: temp read=429 written=428
-> Function Scan on pg_catalog.generate_series  (cost=0.00..10.00 rows=1000 width=0) (actual time=230.287..458.922 rows=1000000
loops=1)
    Output: generate_series
    Function Call: generate_series(1, 1000000)
    Buffers: temp read=429 written=428
    Total runtime: 615.404 ms
(8 rows)
```

PostgreSQL trace和debug

- strace帮助
 - pg93@db-172-16-3-150-> strace -h
 - usage: strace [-dDffhiqrTTvVxx] [-a column] [-e expr] ... [-o file]
 - [-p pid] ... [-s strsize] [-u username] [-E var=val] ...
 - [command [arg ...]]
 - or: strace -c [-D] [-e expr] ... [-O overhead] [-S sortby] [-E var=val] ...
 - [command [arg ...]]
 - -c -- count time, calls, and errors for each syscall and report summary
 - -f -- follow forks, -ff -- with output into separate files
 - -F -- attempt to follow vforks, -h -- print help message
 - -i -- print instruction pointer at time of syscall
 - -q -- suppress messages about attaching, detaching, etc.
 - -r -- print relative timestamp, -t -- absolute timestamp, -tt -- with usecs
 - -T -- print time spent in each syscall, -V -- print version
 - -v -- verbose mode: print unabbreviated argv, stat, termio[s], etc. args
 - -x -- print non-ascii strings in hex, -xx -- print all strings in hex
 - -a column -- alignment COLUMN for printing syscall results (default 40)
 - -e expr -- a qualifying expression: option=[!]all or option=[!]val1[,val2]...
 - options: trace, abbrev, verbose, raw, signal, read, or write

PostgreSQL trace和debug

- -o file -- send trace output to FILE instead of stderr
- -O overhead -- set overhead for tracing syscalls to OVERHEAD usecs
- -p pid -- trace process with process id PID, may be repeated
- -D -- run tracer process as a detached grandchild, not as parent
- -s strsize -- limit length of print strings to STRSIZE chars (default 32)
- -S sortby -- sort syscall counts by: time, calls, name, nothing (default time)
- -u username -- run command as username handling setuid and/or setgid
- -E var=val -- put var=val in the environment for command
- -E var -- remove var from the environment for command

PostgreSQL trace和debug

- strace举例
- digoal=# select pg_backend_pid();
- pg_backend_pid
- -----
- 16752
- (1 row)
- pg93@db-172-16-3-150-> strace -p 16752
- --一个打开了跟踪排序的客户端：
- digoal=# select count(*) from (select * from pg_class order by relpages) t;
- LOG: 00000: begin tuple sort: nkeys = 1, workMem = 1024, randomAccess = f
- LOCATION: tuplesort_begin_heap, tuplesort.c:617
- LOG: 00000: performsort starting: CPU 0.00s/0.00u sec elapsed 0.00 sec
- LOCATION: tuplesort_performsort, tuplesort.c:1319
- LOG: 00000: performsort done: CPU 0.00s/0.00u sec elapsed 0.00 sec
- LOCATION: tuplesort_performsort, tuplesort.c:1394
- LOG: 00000: internal sort ended, 105 KB used: CPU 0.00s/0.00u sec elapsed 0.00 sec
- LOCATION: tuplesort_end, tuplesort.c:932
- count
- -----
- 296
- (1 row)

PostgreSQL trace和debug

- Process 16752 attached - interrupt to quit
- recvfrom(10, "Q\0\0\0Gselect count(*) from (selec"..., 8192, 0, NULL, NULL) = 72
- lseek(7, 0, SEEK_END) = 65536
- lseek(8, 0, SEEK_END) = 65536
- lseek(11, 0, SEEK_END) = 65536
- write(1, "RELOPTINFO (1): rows=292 width=2"..., 207) = 207
- write(1, "RELOPTINFO (1): rows=292 width=0"..., 211) = 211
- lseek(7, 0, SEEK_END) = 65536
- **getrusage(RUSAGE_SELF, {ru_utime={0, 10998}, ru_stime={0, 1999}, ...}) = 0**
- write(2, "\0\0\372\0pA\0\0T2014-04-01 19:12:51.299"..., 259) = 259
- sendto(10, "T\0\0\36\0\1count\0\0\0\0\0\0\0\0\0\0\0\24\0\10\377\377\377\377\0\0N"..., 152, 0, NULL, 0) = 152
- brk(0x10a2000) = 0x10a2000
- **getrusage(RUSAGE_SELF, {ru_utime={0, 10998}, ru_stime={0, 1999}, ...}) = 0**
- write(2, "\0\0\371\0pA\0\0T2014-04-01 19:12:51.299"..., 258) = 258
- sendto(10, "N\0\0\0wSLOG\0C00000\0Mperformsort st"..., 120, 0, NULL, 0) = 120
- **getrusage(RUSAGE_SELF, {ru_utime={0, 10998}, ru_stime={0, 1999}, ...}) = 0**
- write(2, "\0\0\365\0pA\0\0T2014-04-01 19:12:51.299"..., 254) = 254
- sendto(10, "N\0\0\0sSLOG\0C00000\0Mperformsort do"..., 116, 0, NULL, 0) = 116
- **getrusage(RUSAGE_SELF, {ru_utime={0, 10998}, ru_stime={0, 1999}, ...}) = 0**



PostgreSQL trace和debug

PostgreSQL trace和debug

- ltrace帮助
- pg93@db-172-16-3-150-> ltrace --help
- Usage: ltrace [option ...] [command [arg ...]]
- Trace library calls of a given program.
- -a, --align=COLUMN align return values in a secific column.
- -c count time and calls, and report a summary on exit.
- -C, --demangle decode low-level symbol names into user-level names.
- -d, --debug print debugging info.
- --dl show calls to symbols in dlopened libraries.
- -e expr modify which events to trace.
- -f follow forks.
- -h, --help display this help and exit.
- -i print instruction pointer at time of library call.
- -l, --library=FILE print library calls from this library only.
- -L do NOT display library calls.
- -n, --indent=NR indent output by NR spaces for each call level nesting.
- -o, --output=FILE write the trace output to that file.

PostgreSQL trace和debug

- -p PID attach to the process with the process ID pid.
- -r print relative timestamps.
- -s STRLEN specify the maximum string size to print.
- -S display system calls.
- -t, -tt, -ttt print absolute timestamps.
- -T show the time spent inside each call.
- -u USERNAME run command with the userid, groupid of username.
- -V, --version output version information and exit.
- -x NAME treat the global NAME like a library subroutine.

PostgreSQL trace和debug

- ltrace举例
- pg93@db-172-16-3-150-> ltrace -p 16752
- memcpy(0x7ffffdc5de29c, "", 4) = 0x7ffffdc5de29c
- __sigsetjmp(0x7ffffdc5de1d0, 0, 67, 0, 0) = 0
- memcpy(0xfd1718, "select count(*) from (select * f"..., 67) = 0xfd1718
- gettimeofday(0x7ffffdc5de290, NULL) = 0
- strlen("select count(*) from (select * f"...) = 66
- strlen("select count(*) from (select * f"...) = 66
- gettimeofday(0x7ffffdc5de190, NULL) = 0
- memcpy(0x7fbe526a7320, "select count(*) from (select * f"..., 66) = 0x7fbe526a7320
- strlen("TopTransactionContext") = 21
- strcpy(0xfc6700, "TopTransactionContext") = 0xfc6700
- malloc(8192) = 0xf47630
- memset(0xf47758, '\000', 76) = 0xf47758
- strlen("select count(*) from (select * f"...) = 66
- memcpy(0xfd2060, "select count(*) from (select * f"..., 66) = 0xfd2060
- strlen("select") = 6
- strcmp("localtime", "select") = -7
- strcmp("search", "select") = -11
- strcmp("truncate", "select") = 1
- strcmp("stdout", "select") = 15
- ... 略

PostgreSQL trace和debug

- gdb举例
- 略



函数的三态

函数的三态分解-1

- volatile, stable, immutable
- VOLATILE
 - volatile函数没有限制, 可以修改数据(如执行delete, insert , update), 使用同样的参数调用可能返回不同的值.
- STABLE
 - 不允许修改数据, PG8.0以及以上版本不允许在volatile函数中使用非SELECT|PERFORM语句.
 - 使用同样的参数调用返回同样的结果, 在事务中有这个特性的也归属stable.
- IMMUTABLE
 - 不允许修改数据, 使用同样的参数调用返回同样的结果.

函数的三态分解-1(例子1)

- PostgreSQL 8.0以及以上版本不允许在stable或immutable函数中执行非select|perform语句.
- digoal=# create table tbl(id int primary key, info text, crt_time timestamp);
- CREATE TABLE
- digoal=# create or replace function f_tbl(i_id int) returns void as \$\$
- digoal\$# declare
- digoal\$# begin
- digoal\$# update tbl set crt_time=now() where id=i_id;
- digoal\$# end;
- digoal\$# \$\$ language plpgsql **stable**;

- digoal=# \set VERBOSITY verbose
- digoal=# select f_tbl(1);
- **ERROR: 0A000: UPDATE is not allowed in a non-volatile function**
- CONTEXT: SQL statement "update tbl set crt_time=now() where id=i_id"
- PL/pgSQL function f_tbl(integer) line 4 at SQL statement
- LOCATION: _SPI_execute_plan, spi.c:2127

函数的三态分解-1(例子1)

- 漏洞：**在stable或immutable函数中调用volatile函数是可以的.**

- digoal=# alter function f_tbl(int) **volatile**;
- digoal=# create or replace function f_tbl1(i_id int) returns void as \$\$
- declare
- begin
- perform f_tbl(i_id); **-- 在stable或immutable函数中调用volatile函数是可以的.**
- end;
- \$\$ language plpgsql **stable**;
- CREATE FUNCTION

- digoal=# insert into tbl values(1,'test',now());
- INSERT 0 1
- digoal=# select * from tbl;
- id | info | crt_time
- -----+-----+-----
- 1 | test | 2014-03-10 17:21:04.562394
- (1 row)

函数的三态分解-1(例子1)

- digoal=# select f_tbl1(1);
- f_tbl1
- -----
- (1 row)
- digoal=# select * from tbl;
- id | info | crt_time
- -----+-----+-----
- 1 | test | 2014-03-10 17:21:12.183329
- (1 row)

函数的三态分解-1(例子2)

- 同样的参数,多次调用.
- volatile 函数, 相同的参数, 多次调用返回结果可能不一样.
- digoal=# create table t2(id int);
- CREATE TABLE
- digoal=# select pg_relpages('t2');
- pg_relpages
- -----
- 0
- (1 row)
- digoal=# insert into t2 values (1);
- INSERT 0 1
- digoal=# select pg_relpages('t2'); -- 返回值变化
- pg_relpages
- -----
- 1
- (1 row)
- digoal=# select proname,provolatile from pg_proc where proname='pg_relpages';
- pg_relpages | v

函数的三态分解-1(例子2)

■ stable, immutable 函数同样的参数多次调用返回结果不变.

■ 在事务中多次调用返回结果一致的也可归属于stable.

■ digoal=# select now();

■ now

■ -----

■ 2014-03-11 02:49:12.198357+00

■ (1 row)

■ digoal=# select now();

■ now

■ -----

■ 2014-03-11 02:49:14.727296+00

■ (1 row)

■ 事务中now()函数结果一致.

■ digoal=# begin;

■ BEGIN

函数的三态分解-1(例子2)

- digoal=# select now();
 - now
 - -----
- 2014-03-11 02:49:16.706295+00
- digoal=# select now();
 - now
 - -----
- 2014-03-11 02:49:16.706295+00
- (1 row)
- digoal=# select provolatile,proname,proargtypes from pg_proc where proname='now';
 - provolatile | proname | proargtypes
 - -----+-----+
- s | now |
- (1 row)

函数的三态分解-1(例子2)

- immutable函数同stable, 同样的参数多次调用结果一致.
- digoal=# select proname,provolatile from pg_proc where proname='abs';
- proname | provolatile
 - -----+-----
 - abs | i
- digoal=# select abs(-10);
- abs
 - -----
 - 10
 - (1 row)
 -

函数的三态分解-2

■ VOLATILE

- volatile函数不能被优化器作为优化条件。
 - 例如单SQL处理多行时不能减少volatile函数的调用次数,
 - 不能使用volatile函数创建函数索引,
 - 在过滤条件中使用volatile函数时, 不能走索引扫描.
- 在同一个查询中, 同样参数的情况下可能被多次执行(QUERY有多行返回/扫描的情况下).

■ STABLE

- 优化器可根据实际场景优化stable函数的调用次数, 同样的参数多次调用可能减少成单次调用.
- stable函数可以用于优化器选择合适的索引扫描, 因为索引扫描仅评估被比较的值一次, 后多次比较.
- stable和volatile函数都不能用于创建函数索引, 只有immutable函数可以用于创建函数索引.

■ IMMUTABLE

- 优化器在处理immutable函数时, 先评估函数结果, 将结果替换为常量.

函数的三态分解-2(例子1, 影响优化器)

函数的三态分解-2(例子1, 影响优化器)

- 1
- 1
- 1
- (3 rows)
 - digoal=# select * from t3 where f_t3(1)=1; -- 这里使用常量调用f_t3()所以可以被优化器优化.
 - NOTICE: Called. -- 函数只被调用一次.
- id
- ----
- 1
- 1
- 1
- 2
- 2
- 2
- (6 rows)

函数的三态分解-2(例子1, 影响优化器)

- 把函数改成volatile后, 函数不能被优化.
- digoal=# alter function f_t3(int) **volatile**;

- digoal=# select * from t3 where f_t3(1)=1;
- NOTICE: Called.
- id
-

- 根据函数的实际情况设置稳定态, 可以达到优化效果.
- 例如f_t3(int)函数的一次调用耗时1秒, 并且是stable的状态, 那么以上例子可以减少5秒的查询时间. 使用volatile态则需要6秒.

函数的三态分解-2(例子2, 影响执行计划)

- 优化器在处理immutable函数时, 先评估函数结果, 将结果替换为常量.

- digoal=# explain select * from t2 where id>abs(-1);

- QUERY PLAN

- Seq Scan on t2 (cost=0.00..130.38 rows=3210 width=4)

- Filter: (id > 1) -- 因为abs(int)是immutable函数, 这里abs(-1) 替换成常量1.

- (2 rows)

- 如果把函数改成stable, 那么将不会替换成常量.

- digoal=# alter function abs(int) stable;

- ALTER FUNCTION

- digoal=# explain select * from t2 where id>abs(-1);

- QUERY PLAN

- Seq Scan on t2 (cost=0.00..154.45 rows=3210 width=4)

- Filter: (id > abs((-1))) -- 由于abs(int)被改成stable了, 将不会替换成常量

- (2 rows)

- 在prepared statement中使用需要注意区别. 后面会有例子.

函数的三态分解-2(例子3, 函数索引)

- 只有immutable函数可以创建函数索引.
- digoal=# create table t4(id int, info timestamp(0));
- CREATE TABLE
- digoal=# \set VERBOSITY verbose
- digoal=# create index idx_t4_1 on t4(to_char(info,'yyyymmdd'));
- ERROR: 42P17: functions in index expression **must be marked IMMUTABLE**
- LOCATION: ComputeIndexAttrs, indexcmds.c:1067

函数的三态分解-2(例子4, 函数值比较走索引)

- digoal=# create table t5(id int primary key, info text);
- digoal=# insert into t5 select generate_series(1,100000),md5(random()::text);
- digoal=# alter function abs(int) **volatile**;

- 索引扫描时, 用于过滤条件的表达式只被评估一次后, 再与索引值进行比较判断是否满足条件.
- digoal=# explain select * from t5 where id<abs(10);

QUERY PLAN

- -----
- **Seq Scan** on t5 (cost=0.00..1708.00 rows=33333 width=37)
- Filter: (id < abs(10))
- 只有stable函数和immutable函数符合索引扫描的刚性需求.
- digoal=# alter function abs(int) **stable**;
- digoal=# explain select * from t5 where id<abs(-100);

QUERY PLAN

- -----
- **Index Scan** using t5_pkey on t5 (cost=0.29..5.98 rows=96 width=37)
- Index Cond: (id < abs((-100)))
- (2 rows)

函数的三态分解-2(例子4, 函数值比较走索引)

- digoal=# alter function abs(int) **immutable**;
- digoal=# explain select * from t5 where id<abs(-100);
 - QUERY PLAN
- -----
- **Index Scan** using t5_pkey on t5 (cost=0.29..5.97 rows=96 width=37)
 - Index Cond: (id < 100)
 - (2 rows)
- volatile函数同样的参数输入可能返回不同值, 在一个查询中将被多次调用, 不符合索引扫描规则.
- 而stable和immutable同样的参数返回值不变, 因此可以作为索引扫描的比较值, 优化器允许走索引扫描.

函数的三态分解-3

- 函数内的每条query的数据可见性：
 - VOLATILE
 - snapshot为函数内的每个query开始时的snapshot. 因此对外部已提交的数据时可见的.
 - STABLE
 - snapshot为外部调用函数的QUERY的snapshot, 函数内部始终保持这个snapshot.
 - IMMUTABLE
 - 同stable

函数的三态分解-3(例子)

```
■ digoal=# create or replace function f_t6() returns void as $$  
■ declare  
■   r record;  
■ begin  
■   for i in 1..10 loop  
■     for r in select t.* from t6 t loop  
■       raise notice 'loop:%, t6:%.', i, r;  
■     end loop;  
■     perform pg_sleep(5);  
■   end loop;  
■ end;  
■ $$ language plpgsql volatile;  
■ CREATE FUNCTION
```

函数的三态分解-3(例子)

- digoal=# create table t6(id int, info text);
- digoal=# select f_t6();
- NOTICE: loop:1, t6:(1,test).
- NOTICE: loop:1, t6:(1,test1).
- NOTICE: loop:2, t6:(1,test).
- NOTICE: loop:2, t6:(1,test1).
- NOTICE: loop:3, t6:(1,test).
- NOTICE: loop:3, t6:(1,test1).
- NOTICE: loop:3, t6:(1,test3). -- 执行过程中, 往t6表插入新数据, volatile函数察觉新增行

函数的三态分解-3(例子)

- digoal=# alter function f_t6() stable; -- immutable同样
- digoal=# delete from t6;
- DELETE 4
- digoal=# select f_t6();
-- 没有输出
- digoal=# insert into t6 values (1,'test4'); -- f_t6()执行过程中对t6变更, f_t6()不感知.
- INSERT 0 1
- digoal=# insert into t6 values (1,'test2');
- INSERT 0 1

函数的三态分解-4

■ STABLE和IMMUTABLE的区别, 在select子句中, 优化器对stable和immutable区别对待.

```
digoal=# create table t7(id int);
digoal=# insert into t7 values (1),(2),(3);
digoal=# create or replace function f_t7(i int) returns int as $$ 
declare
begin
    raise notice 'called'; return i;
end;
$$ language plpgsql stable;
```

函数的三态分解-4

- digoal=# select f_t7(1),* from t7; -- SELECT子句中, stable函数不被优化器优化.
- NOTICE: called -- 多次调用
- NOTICE: called
- NOTICE: called
- f_t7 | id
- -----+-----
- 1 | 1
- 1 | 2
- 1 | 3

函数的三态分解-4

- digoal=# alter function f_t7(int) **immutable**;
- digoal=# select f_t7(1),* from t7; -- 因为immutable函数被替换成常量, 所以只执行一次.
- **NOTICE:** called -- 一次调用
- f_t7 | id
- -----+-----
- 1 | 1
- 1 | 2
- 1 | 3
- (3 rows)

函数的三态分解-4

- digoal=# alter function f_t7(int) **stable**;
- digoal=# select * from t7 where **id=f_t7(1)**; -- 当函数为**stable**时, 目前优化器没有处理这种过滤条件, 理论上是可以优化为一次调用f_t7(1)的. 目前仅仅**immutable**被优化.
- 仅仅当使用索引扫描时, **stable**在这里只会执行一次.
- 所以这里PostgreSQL的优化器是有改进之处的.
- NOTICE: called -- 其中一次为explain的评估输出.
- NOTICE: called
- NOTICE: called
- NOTICE: called
- id
- ----
- 1

函数的三态分解-4

- digoal=# select * from t7 where **f_t7(1)=1**; -- 将f_t7(1)=id替换成f_t7(1)=1
- NOTICE: called
- id
- ----
- 1
- 2
- 3
- (3 rows)
- digoal=# explain select * from t7 where f_t7(1)=1;
- QUERY PLAN
- -----
- Result (cost=0.25..106.55 rows=9630 width=4)
- One-Time Filter: (**f_t7(1) = 1**)
- -> Seq Scan on t7 (cost=0.25..106.55 rows=9630 width=4)
- (3 rows)

函数的三态分解-4

- digoal=# alter function f_t7(int) immutable;
- digoal=# explain select * from t7 where f_t7(1)=1;
- NOTICE: called
 - QUERY PLAN
- -----
- Seq Scan on t7 (cost=0.00..34.00 rows=2400 width=4)
- (1 row)

函数的三态分解-5

- STABLE和IMMUTABLE 在prepared statement中的使用区别.
- immutable函数在plan时以常量替代, stable函数在execute阶段被执行.
- 因此immutable函数参数为常量时, 在prepared statement场景只执行一次, 而stable函数被多次执行.
- digoal=# create table t6(id int, info text);
- digoal=# create or replace function f_pre(id int) returns int as \$\$
- declare
- cnt int8;
- begin
- select count(*) into cnt from t6;
- return cnt;
- end;
- \$\$ language plpgsql strict **immutable**;
- pg93@db-172-16-3-150-> lua
- Lua 5.2.3 Copyright (C) 1994-2013 Lua.org, PUC-Rio
- > pgsql = require "pgsql"
- > conn = pgsql.connectdb('host=/ssd2/pg93/pg_root port=5432 dbname=digoal user=postgres password=postgres')

函数的三态分解-5

- > conn:prepare('pre','select f_pre(1)')
- > print (conn:execPrepared('pre'):getValue(1,1))
- 0
- digoal=# insert into t6 values (1,'test');
- digoal=# insert into t6 values (2,'test');
- > print (conn:execPrepared('pre'):getValue(1,1))
- 0 -- 结果不变, 还是0. 只要prepared statement cache没有被清除, 返回结果都不会变化, 但是这里改变t6的表结构的话就会清除 prepared statement cache.

- digoal=# alter function f_pre(int) **stable**;
- > .. 重新prepare.
- > print (conn:execPrepared('pre'):getValue(1,1))
- 2
- digoal=# insert into t6 values (3,'test');
- digoal=# insert into t6 values (4,'test');
- > print (conn:execPrepared('pre'):getValue(1,1))
- 4 -- 结果变化. 因为stable函数在prepared statements中execute时被执行 .

函数的三态小结

■ VOLATILE

- volatile函数没有限制, 可以修改数据(如执行delete, insert , update).
- 使用同样的参数调用可能返回不同的值.
- volatile函数不能被优化器选择作为优化条件.(例如减少调用, 函数索引, 索引扫描不允许使用volatile函数)
- 在同一个查询中, 同样参数的情况下可能被多次执行(QUERY有多行返回/扫描的情况下).
- snapshot为函数内的每个query开始时的snapshot. 因此对在函数执行过程中, 外部已提交的数据可见.(仅仅限于调用函数的事务隔离级别为read committed)

■ STABLE

- stable和immutable函数, 函数内不允许修改数据.(如PGver>=8.0 函数内不可执行非SELECT|PERFORM语句.)
- 使用同样的参数调用返回同样的结果, 在事务中有这个特性的也归属stable.
- 优化器可根据实际场景优化stable函数的调用次数, 同样的参数多次调用可减少成单次调用.
- stable和immutable函数可用于优化器选择合适的索引扫描, 因为索引扫描仅评估被比较的表达式一次, 后多次与索引值进行比较.
- stable和volatile函数都不能用于创建函数索引, 只有immutable函数可以用于创建函数索引.
- stable和immutable函数, snapshot为外部调用函数的QUERY的snapshot, 函数内部始终保持这个snapshot, 外部会话带来的的数据变更不被反映到函数执行过程中.

函数的三态小结

■ IMMUTABLE

- 不允许修改数据, 使用同样的参数调用返回同样的结果.
- 优化器在处理immutable函数时, 先评估函数结果, 将结果替换为常量.
- 因此使用约束优化查询的场景中也只识别immutable函数.

■ STABLE和IMMUTABLE的区别

- stable函数在select和where子句中不被优化, 仅仅当使用索引扫描时where子句对应的stable函数才会被优化为1次调用.
- 在prepared statement中的使用区别:
- immutable函数在plan时以常量替代, stable函数在execute阶段被执行.
- 因此immutable函数参数为常量时, 在prepared statement场景只执行一次, 而stable函数被多次执行.

■ 函数稳定性通过查看pg_proc.provolatile得到

PostgreSQL MVCC

- 事务隔离级别,
- 并发控制,
- 锁的介绍,
- 死锁的发现和处理, 实际应用中如何避免死锁

PostgreSQL MVCC

- PostgreSQL的多版本并发控制
- 版本识别演示.(INSERT, UPDATE, DELETE, 使用ctid定位, 并查看该TUPLE xmin, xmax的变化)

- 关键词
- XID -- 数据库的事务ID
- Tuple head: xmin, xmax, 行头部的XID信息, xmin表示插入这条记录的事务XID, xmax表示删除(或锁)这条记录的事务XID
- Xid_snapshot : 当前集群中的未结束事务.
- Clog : 事务提交状态日志
- 事务隔离级别.

- 数据可见性条件：
 - 1. 记录的头部XID信息比当前事务更早. (repeatable read或ssi有这个要求, read committed没有这个要求)
 - 2. 记录的头部XID信息不在当前的XID_snapshot中. (即记录上的事务状态不是未提交的状态.)
 - 3. 记录头部的XID信息在CLOG中应该显示为已提交.

PostgreSQL MVCC

- 更新和删除数据时，并不是直接删除行的数据，而是更新行的头部信息中的xmax和infomask掩码.
- 事务提交后更新当前数据库集群的事务状态和pg_clog中的事务提交状态.
- Infomask和infomask2参看
- src/include/access/htup_details.h

- 例子：
- 会话1：

```
digoal=# truncate iso_test ;  
digoal# TRUNCATE TABLE  
digoal# insert into iso_test values (1,'test');  
digoal# INSERT 0 1  
digoal# begin;  
digoal# BEGIN  
digoal# update iso_test set info='new' where id=1;  
digoal# UPDATE 1
```

PostgreSQL MVCC

- 会话2：
 - digoal=# select ctid,xmin,xmax,* from iso_test where id=1;
 - ctid | xmin | xmax | id | info
 - -----+-----+-----+----+
 - (0,1) | 316732572 | 316732573 | 1 | test
 - (1 row)
- PostgreSQL多版本并发控制不需要UNDO表空间.

PostgreSQL MVCC

- **RepeatableRead1** tuple-v1 IDLE IN TRANSACTION;
- **ReadCommitted1** tuple-v1 IDLE IN TRANSACTION;
- **RC2** tuple-v1 UPDATE -> tuple-v2 COMMIT;
- **RR1** tuple-v1 IDLE IN TRANSACTION;
- **RC1** tuple-v2 IDLE IN TRANSACTION;
- RR2 tuple-v2 IDLE IN TRANSACTION;
- **RC3** tuple-v2 UPDATE -> tuple-v3 COMMIT;
- **RR1** tuple-v1 IDLE IN TRANSACTION;
- RR2 tuple-v2 IDLE IN TRANSACTION;
- **RC1** tuple-v3 IDLE IN TRANSACTION;

PostgreSQL 事务隔离级别

■ 脏读

- 在一个事务中可以读到其他未提交的事务产生或变更的数据.
- PostgreSQL不支持read uncommitted事务隔离级别, 无法测试.

■ 不可重复读

- 在一个事务中, 再次读取前面SQL读过的数据时, 可能出现读取到的数据和前面读取到的不一致的现象.(例如其他事务在此期间已提交的数据)
- 使用read committed事务隔离级别测试

■ 幻像读

- 在一个事务中, 再次执行同样的SQL, 得到的结果可能不一致.

■ 标准SQL事务隔离级别, (PostgreSQL的repeatable read隔离级别不会产生幻像读)

■ PostgreSQL不支持read uncommitted隔离级别.

Table 13-1. Standard SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

PostgreSQL 事务隔离级别测试1

■ 不可重复读测试

■ digoal=# create table iso_test(id int, info text);

■ digoal=# insert into iso_test values (1, 'test');

■ digoal=# begin isolation level read committed;

■ BEGIN

■ digoal=# select * from iso_test where id=1;

■ id | info

■ -----+-----

■ 1 | test

■ (1 row)

■ -- 其他会话更新这份数据, 并提交.

■ digoal=# update iso_test set info='new' where id=1;

■ -- 不可重复读出现.

■ digoal=# select * from iso_test where id=1;

■ id | info

■ -----+-----

■ 1 | new

■ (1 row)

PostgreSQL 事务隔离级别测试2

■ 幻象读测试

■ digoal=# begin isolation level read committed;

■ digoal=# select * from iso_test;

■ id | info

■ -----+-----

■ 1 | new

■ (1 row)

■ -- 其他会话新增数据

■ digoal=# insert into iso_test values (2, 'test');

■ -- 幻象读出现

■ digoal=# select * from iso_test;

■ id | info

■ -----+-----

■ 1 | new

■ 2 | test

■ (2 rows)

PostgreSQL 事务隔离级别测试3

- 使用repeatable read可避免不可重复读和幻象读.

```
digoal=# delete from iso_test;
digoal=# insert into iso_test values (1, 'test');
digoal=# begin isolation level repeatable read;
digoal=# select * from iso_test where id=1;
id | info
----+-----
 1 | test
(1 row)

-- 其他会话修改数据, 并提交
digoal=# update iso_test set info='new' where id=1;
digoal# select * from iso_test where id=1;
id | info
----+-----
 1 | test
(1 row)
```

PostgreSQL 事务隔离级别测试3

- -- 其他会话新增数据.
- digoal=# insert into iso_test values (2, 'test');
- INSERT 0 1

- -- 未出现幻象读
- digoal=# select * from iso_test ;
- id | info
- -----+-----
- 1 | test
- (1 row)

PostgreSQL 事务隔离级别测试4

- PostgreSQL repeatable read情景案例
- 当repeatable read的事务去更新或删除在事务过程中被其他事务已经变更过的数据时, 将报错等待回滚.
- digoal=# truncate iso_test ;
- digoal=# insert into iso_test values (1,'test');
- digoal=# begin isolation level repeatable read;
- digoal=# select * from iso_test ;
 - id | info
 - -----+-----
 - 1 | test
 - (1 row)
- -- 其他事务更新或者删除这条记录, 并提交.
- digoal=# update iso_test set info='new' where id=1;
- UPDATE 1
 - -- 在repeatable read的事务中更新或者删除这条记录. 会报错回滚
- digoal=# update iso_test set info='tt' where id=1;
- ERROR: could not serialize access due to concurrent update
- digoal=# rollback;
- ROLLBACK

PostgreSQL 事务隔离级别测试4

- 先获取锁, 再处理行上的数据(例如做条件判断.)
- 所以会有这种现象.
- -- 会话1
- digoal=# truncate iso_test ;
- TRUNCATE TABLE
- digoal=# insert into iso_test values (1,'test');
- INSERT 0 1
- digoal=# begin;
- BEGIN
- digoal=# update iso_test set id=id+1 returning id;
- id
- ----
- 2
- (1 row)
- UPDATE 1

PostgreSQL 事务隔离级别测试4

- -- 会话2
- digoal=# select * from iso_test ;
- id | info
- -----
- 1 | test
- (1 row)
- digoal=# delete from iso_test where id=1; -- 等待ctid=(0,1)的行exclusive锁

- -- 会话1, 提交事务
- digoal=# end;
- COMMIT

- -- 会话2, 此时会话2等待的这条ctid(0,1)已经被会话1删除了(如果会话2是repeatable read模式的话这里会报错).
- DELETE 0
- digoal=# select * from iso_test;
- id | info
- -----
- 2 | test
- (1 row)

PostgreSQL 事务隔离级别测试5

- **Serializable** 隔离级别
- 目标是模拟serializable的隔离级别事务的提交顺序转换为串行的执行顺序.
- 例如 :
- Start transaction a serializable
- Start transaction b serializable
- Session a -> SQL ...
- Session b -> SQL ...
- Session a|b -> SQL ...
- Session a|b -> SQL ...
- Commit b
- Commit a
- 这个场景模拟成 :
- Start transaction b ssi
- Sql ...
- Commit b
- Start transaction a ssi
- Sql ... 如果会话a扫描过的数据在B中被加ssi锁, 那么a会话将提交失败.
- Commit a

PostgreSQL 事务隔离级别测试5

- PostgreSQL 串行事务隔离级别的实现, 通过对扫描过的数据加载预锁来实现(内存中的一种弱冲突锁, 只在事务结束时判断是否有数据依赖性的冲突)
- 因为涉及到扫描的数据, 所以这种锁和**执行计划**有关.

- 例如
- `Select * from tbl where a=1;`
- 如果没有索引, 那么是全表扫描, 需要扫描所有的数据块.
- 加载的预锁是表级别的预锁. (那么期间如果其他串行事务对这个表有任何变更, 包括插入, 删除, 更新等. 并且先提交的话.)
- 这个会话结束的时候会发现预加锁的数据被其他串行事务变更了, 所以会提交失败.

- 如果a上有索引的话, 执行计划走索引的情况下, 扫描的数据包括行和索引页.
- 那么加载的预锁包含行和索引页.
- 这种情况仅当其他串行事务在此期间变更了相对应的行或者是索引页才会在结束时发生冲突.

PostgreSQL 事务隔离级别测试5

- 例子：
- 会话A：

```
digoal=# select pg_backend_pid();
-[ RECORD 1 ]---+
pg_backend_pid | 12186
```
- 会话B：

```
digoal=# select pg_backend_pid();
-[ RECORD 1 ]---+
pg_backend_pid | 12222
```
- 会话A：

```
digoal=# truncate iso_test ;
TRUNCATE TABLE
digoal=# insert into iso_test select generate_series(1,100000);
INSERT 0 100000
digoal=# begin ISOLATION LEVEL SERIALIZABLE;
BEGIN
digoal=# select sum(id) from iso_test where id=100;
-[ RECORD 1 ]
sum | 100
```

PostgreSQL 事务隔离级别测试5

■ 会话 C :

```
digoal=# select relation::regclass,* from pg_locks where pid in (12186,12222);
relation | locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtrans
action | pid | mode | granted | fastpath
-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+
iso_test | relation | 16384 | 92992 | | | | | | | 1/157993
| 12186 | AccessShareLock | t | t
| virtualxid | | | | 1/157993 | | | | | 1/157993
| 12186 | ExclusiveLock | t | t
iso_test | relation | 16384 | 92992 | | | | | | | 1/157993
| 12186 | SIReadLock | t | f
(3 rows)
```

■ 会话 B :

```
digoal=# begin ISOLATION LEVEL SERIALIZABLE;
BEGIN
digoal=# select sum(id) from iso_test where id=10;
-[ RECORD 1 ]
sum | 10
```

PostgreSQL 事务隔离级别测试5

■ 会话 C :

```
digoal=# select relation::regclass,* from pg_locks where pid in (12186,12222);
relation | locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtrans
action | pid | mode | granted | fastpath
-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+
iso_test | relation | 16384 | 92992 | | | | | | | 1/157993
| 12186 | AccessShareLock | t | t
| virtualxid | | | | 1/157993 | | | | | 1/157993
| 12186 | ExclusiveLock | t | t
iso_test | relation | 16384 | 92992 | | | | | | | 2/6433312
| 12222 | AccessShareLock | t | t
| virtualxid | | | | 2/6433312 | | | | | 2/6433312
| 12222 | ExclusiveLock | t | t
iso_test | relation | 16384 | 92992 | | | | | | | 1/157993
| 12186 | SIReadLock | t | f
iso_test | relation | 16384 | 92992 | | | | | | | 2/6433312
| 12222 | SIReadLock | t | f
(6 rows)
```

PostgreSQL 事务隔离级别测试5

- 会话 A :
 - digoal=# insert into iso_test values (1,'test');
 - INSERT 0 1
- 会话 B :
 - digoal=# insert into iso_test values (2,'test');
 - INSERT 0 1

PostgreSQL 事务隔离级别测试5

■ 会话 C :

```
digoal=# select relation::regclass,* from pg_locks where pid in (12186,12222);
relation | locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtransaction | pid | mode | granted | fastpath
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
iso_test | relation | 16384 | 92992 |  |  |  |  |  |  |  |  |  | 1/157993
| 12186 | AccessShareLock | t | t
iso_test | relation | 16384 | 92992 |  |  |  |  |  |  |  |  |  | 1/157993
| 12186 | RowExclusiveLock | t | t
| virtualxid |  |  |  |  | 1/157993 |  |  |  |  | 1/157993
| 12186 | ExclusiveLock | t | t
iso_test | relation | 16384 | 92992 |  |  |  |  |  |  |  |  | 2/6433312
| 12222 | AccessShareLock | t | t
```



PostgreSQL 事务隔离级别测试5

PostgreSQL 事务隔离级别测试5



PostgreSQL 事务隔离级别测试5

PostgreSQL 事务隔离级别测试5

- 会话 B：
 - digoal=# commit;
 - ERROR: could not serialize access due to read/write dependencies among transactions
 - DETAILED: Reason code: Canceled on identification as a pivot, during commit attempt.
 - HINT: The transaction might succeed if retried.

- 会话 C：
 - digoal=# select relation::regclass,* from pg_locks where pid in (12186,12222);
 - relation | locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtransaction | pid | mode | granted | fastpath
 - -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 - -----+-----+-----+-----+
 - (0 rows)

PostgreSQL 事务隔离级别测试6

- 同样的场景, 加索引测试 :
 - digoal=# create index idx_iso_test_1 on iso_test (id);
 - CREATE INDEX

- digoal=# begin ISOLATION LEVEL SERIALIZABLE;
- BEGIN
- digoal=# select sum(id) from iso_test where id=100;
- -[RECORD 1]
- sum | 100

PostgreSQL 事务隔离级别测试6

PostgreSQL 事务隔离级别测试6

- digoal=# begin ISOLATION LEVEL SERIALIZABLE;
- BEGIN
- digoal=# select sum(id) from iso_test where id=10;
- -[RECORD 1]
- sum | 10

- digoal=# select relation::regclass,* from pg_locks where pid in (12186,12222);

relation	locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualxid	transactionid	pid	mode	granted	fastpath	
idx_iso_test_1	relation	16384	93017										1/1579	96	12186 AccessShareLock t t		
iso_test	relation	16384	92992										1/1579	96	12186 AccessShareLock t t		
													1/157996	96	12186 ExclusiveLock t t		

PostgreSQL 事务隔离级别测试6

PostgreSQL 事务隔离级别测试6

- digoal=# insert into iso_test values (1,'test');
- INSERT 0 1

- digoal=# select relation::regclass,* from pg_locks where pid in (12186,12222);
 - relation | locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtransaction | pid | mode | granted | fastpath
 - -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 - -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 - idx_iso_test_1 | relation | 16384 | 93017 | | | | | | | | | | | | | | | | 1/1
 - 57996 | 12186 | AccessShareLock | t | t
 - iso_test | relation | 16384 | 92992 | | | | | | | | | | | | | | | | 1/1
 - 57996 | 12186 | AccessShareLock | t | t
 - iso_test | relation | 16384 | 92992 | | | | | | | | | | | | | | | | 1/1
 - 57996 | 12186 | RowExclusiveLock | t | t
 - | virtualxid | | | | | 1/157996 | | | | | | | | | | | | | | | | 1/1

PostgreSQL 事务隔离级别测试6

PostgreSQL 事务隔离级别测试6

- digoal=# insert into iso_test values (2,'test');
- INSERT 0 1

- digoal=# select relation::regclass,* from pg_locks where pid in (12186,12222);
 - relation | locktype | database | relation | page | tuple | virtualxid | transactionid | classid | objid | objsubid | virtualtransaction | pid | mode | granted | fastpath
 - -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 - -----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
 - idx_iso_test_1 | relation | 16384 | 93017 | | | | | | | | | | | | | | | | 1/1
 - 57996 | 12186 | AccessShareLock | t | t
 - iso_test | relation | 16384 | 92992 | | | | | | | | | | | | | | | | 1/1
 - 57996 | 12186 | AccessShareLock | t | t
 - iso_test | relation | 16384 | 92992 | | | | | | | | | | | | | | | | 1/1
 - 57996 | 12186 | RowExclusiveLock | t | t
 - virtualxid | | | | | 1/157996 | | | | | | | | | | | | 1/1
 - 57996 | 12186 | ExclusiveLock | t | t
 - idx_iso_test_1 | relation | 16384 | 93017 | | | | | | | | | | | | | | | | 2/6
 - 433314 | 12222 | AccessShareLock | t | t

PostgreSQL 事务隔离级别测试6

PostgreSQL 事务隔离级别测试6

- digoal=# commit;
- COMMIT

- digoal=# commit;
- ERROR: could not serialize access due to read/write dependencies among transactions
- DETAIL: Reason code: Canceled on identification as a pivot, during commit attempt.
- HINT: The transaction might succeed if retried.
- 索引页用了同一个, 并且被插入语句更新了. 所以发生了冲突

- 如果其中一个插入的值不在1号索引页则没有问题, 例如
- digoal=# begin ISOLATION LEVEL SERIALIZABLE;
- BEGIN
- digoal=# select sum(id) from iso_test where id=100;
- -[RECORD 1]
- sum | 100

PostgreSQL 事务隔离级别测试6

- digoal=# insert into iso_test values (1,'test');
- INSERT 0 1
- digoal=# commit;
- COMMIT

- digoal=# begin ISOLATION LEVEL SERIALIZABLE;
- BEGIN
- digoal=# select sum(id) from iso_test where id=10;
- -[RECORD 1]
- sum | 10

- digoal=# insert into iso_test values (200000,'test');
- INSERT 0 1
- digoal=# commit;
- COMMIT

- (200000,'test') 这个索引页不在1号, 在275
- idx_iso_test_1 | page | 16384 | 93017 | 275 | | | | | | 2/6433
- 316 | 12222 | SIReadLock | t | f

PostgreSQL 事务隔离级别测试6

- 注意事项
- PostgreSQL 的 hot_standby 节点不支持串行事务隔离级别, 只能支持read committed和repeatable read隔离级别.

PostgreSQL锁的介绍

- ```
■ 锁对象的类型
■ relation, extend, page, tuple, transactionid, virtualxid, object, userlock, or advisory
■ src/include/storage/lock.h
■ /*
■ * LOCKTAG is the key information needed to look up a LOCK item in the
■ * lock hashtable. A LOCKTAG value uniquely identifies a lockable object.
■ *
■ * The LockTagType enum defines the different kinds of objects we can lock.
■ * We can handle up to 256 different LockTagTypes.
■ */
■ typedef enum LockTagType
■ {
■ LOCKTAG_RELATION, /* whole relation */
■ /* ID info for a relation is DB OID + REL OID; DB OID = 0 if shared */
■ LOCKTAG_RELATION_EXTEND, /* the right to extend a relation */
■ /* same ID info as RELATION */
■ LOCKTAG_PAGE, /* one page of a relation */
■ /* ID info for a page is RELATION info + BlockNumber */
■ }
```

# PostgreSQL锁的介绍

```
■ LOCKTAG_TUPLE, /* one physical tuple */
■ /* ID info for a tuple is PAGE info + OffsetNumber */
■ LOCKTAG_TRANSACTION, /* transaction (for waiting for xact done) */
■ /* ID info for a transaction is its TransactionId */
■ LOCKTAG_VIRTUALTRANSACTION, /* virtual transaction (ditto) */
■ /* ID info for a virtual transaction is its VirtualTransactionId */
■ LOCKTAG_OBJECT, /* non-relation database object */
■ /* ID info for an object is DB OID + CLASS OID + OBJECT OID + SUBID */

■ /*
■ * Note: object ID has same representation as in pg_depend and
■ * pg_description, but notice that we are constraining SUBID to 16 bits.
■ * Also, we use DB OID = 0 for shared objects such as tablespaces.
■ */
■ LOCKTAG_USERLOCK, /* reserved for old contrib/userlock code */
■ LOCKTAG_ADVISORY, /* advisory user locks */
■ } LockTagType;
```

# PostgreSQL锁的介绍

- #### ■ 锁模式1(标准锁和用户锁方法支持的模式都在这里)

# PostgreSQL锁的介绍

## ■ 锁模式冲突表

**Table 13-2. Conflicting Lock Modes**

| Requested Lock Mode    | Current Lock Mode |           |               |                        |       |                     |           |                  |   |
|------------------------|-------------------|-----------|---------------|------------------------|-------|---------------------|-----------|------------------|---|
|                        | ACCESS SHARE      | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE |   |
| ACCESS SHARE           |                   |           |               |                        |       |                     |           |                  | X |
| ROW SHARE              |                   |           |               |                        |       |                     | X         |                  | X |
| ROW EXCLUSIVE          |                   |           |               |                        | X     | X                   |           | X                | X |
| SHARE UPDATE EXCLUSIVE |                   |           |               | X                      | X     | X                   |           | X                | X |
| SHARE                  |                   |           | X             | X                      |       | X                   |           | X                | X |
| SHARE ROW EXCLUSIVE    |                   |           | X             | X                      | X     | X                   |           | X                | X |
| EXCLUSIVE              |                   | X         | X             | X                      | X     | X                   |           | X                | X |
| ACCESS EXCLUSIVE       | X                 | X         | X             | X                      | X     | X                   |           | X                | X |

# PostgreSQL锁的介绍

- 行锁模式
- src/include/access/heapam.h

```
■ /*
■ * Possible lock modes for a tuple.
■ */
■ typedef enum LockTupleMode
■ {
■ /* SELECT FOR KEY SHARE */
■ LockTupleKeyShare,
■ /* SELECT FOR SHARE */
■ LockTupleShare,
■ /* SELECT FOR NO KEY UPDATE, and UPDATEs that don't modify key columns */
■ LockTupleNoKeyExclusive,
■ /* SELECT FOR UPDATE, UPDATEs that modify key columns, and DELETE */
■ LockTupleExclusive
■ } LockTupleMode;
```

# PostgreSQL锁的介绍

- 行锁模式冲突表
- [src/backend/access/heap/README.tuplock](#)

|            | KEY UPDATE | UPDATE   | SHARE    | KEY SHARE |
|------------|------------|----------|----------|-----------|
| KEY UPDATE | conflict   | conflict | conflict | conflict  |
| UPDATE     | conflict   | conflict | conflict |           |
| SHARE      | conflict   | conflict |          |           |
| KEY SHARE  | conflict   |          |          |           |

- 例子
- <http://blog.163.com/digoal@126/blog/static/16387704020130305109687/>

# PostgreSQL锁的介绍

- 串行锁模式
- SIReadLock
- src/backend/storage/lmgr/README-SSI
- <http://www.postgresql.org/docs/9.3/static/transaction-iso.html#XACT-SERIALIZABLE>

# PostgreSQL锁的介绍

- PostgreSQL获取锁的宏定义

```
■ /*
■ * These macros define how we map logical IDs of lockable objects into
■ * the physical fields of LOCKTAG. Use these to set up LOCKTAG values,
■ * rather than accessing the fields directly. Note multiple eval of target!
■ */
■ #define SET_LOCKTAG_RELATION(locktag,dboid,reloid) \
■ ((locktag).locktag_field1 = (dboid), \
■ (locktag).locktag_field2 = (reloid), \
■ (locktag).locktag_field3 = 0, \
■ (locktag).locktag_field4 = 0, \
■ (locktag).locktag_type = LOCKTAG_RELATION, \
■ (locktag).locktag_lockmethodid = DEFAULT_LOCKMETHOD)
```

# PostgreSQL锁的介绍

```
■ #define SET_LOCKTAG_RELATION_EXTEND(locktag,dboid,reloid) \
■ ((locktag).locktag_field1 = (dboid), \
■ (locktag).locktag_field2 = (reloid), \
■ (locktag).locktag_field3 = 0, \
■ (locktag).locktag_field4 = 0, \
■ (locktag).locktag_type = LOCKTAG_RELATION_EXTEND, \
■ (locktag).locktag_lockmethodid = DEFAULT_LOCKMETHOD)

■ #define SET_LOCKTAG_PAGE(locktag,dboid,reloid,blocknum) \
■ ((locktag).locktag_field1 = (dboid), \
■ (locktag).locktag_field2 = (reloid), \
■ (locktag).locktag_field3 = (blocknum), \
■ (locktag).locktag_field4 = 0, \
■ (locktag).locktag_type = LOCKTAG_PAGE, \
■ (locktag).locktag_lockmethodid = DEFAULT_LOCKMETHOD)
```

# PostgreSQL锁的介绍

```
■ #define SET_LOCKTAG_TUPLE(locktag,dboid,relloid,blocknum,offnum) \
■ ((locktag).locktag_field1 = (dboid), \
■ (locktag).locktag_field2 = (relloid), \
■ (locktag).locktag_field3 = (blocknum), \
■ (locktag).locktag_field4 = (offnum), \
■ (locktag).locktag_type = LOCKTAG_TUPLE, \
■ (locktag).locktag_lockmethodid = DEFAULT_LOCKMETHOD)

■ #define SET_LOCKTAG_TRANSACTION(locktag,xid) \
■ ((locktag).locktag_field1 = (xid), \
■ (locktag).locktag_field2 = 0, \
■ (locktag).locktag_field3 = 0, \
■ (locktag).locktag_field4 = 0, \
■ (locktag).locktag_type = LOCKTAG_TRANSACTION, \
■ (locktag).locktag_lockmethodid = DEFAULT_LOCKMETHOD)
```

# PostgreSQL锁的介绍

- #define SET\_LOCKTAG\_VIRTUALTRANSACTION(locktag,vxid) \  
    ((locktag).locktag\_field1 = (vxid).backendId, \  
    (locktag).locktag\_field2 = (vxid).localTransactionId, \  
    (locktag).locktag\_field3 = 0, \  
    (locktag).locktag\_field4 = 0, \  
    (locktag).locktag\_type = LOCKTAG\_VIRTUALTRANSACTION, \  
    (locktag).locktag\_lockmethodid = DEFAULT\_LOCKMETHOD)
  
- #define SET\_LOCKTAG\_OBJECT(locktag,dboid,classoid,objoid,objsubid) \  
    ((locktag).locktag\_field1 = (dboid), \  
    (locktag).locktag\_field2 = (classoid), \  
    (locktag).locktag\_field3 = (objoid), \  
    (locktag).locktag\_field4 = (objsubid), \  
    (locktag).locktag\_type = LOCKTAG\_OBJECT, \  
    (locktag).locktag\_lockmethodid = DEFAULT\_LOCKMETHOD)

# PostgreSQL锁的介绍

- #define SET\_LOCKTAG\_ADVISORY(locktag,id1,id2,id3,id4) \
- ((locktag).locktag\_field1 = (id1), \
- (locktag).locktag\_field2 = (id2), \
- (locktag).locktag\_field3 = (id3), \
- (locktag).locktag\_field4 = (id4), \
- (locktag).locktag\_type = LOCKTAG\_ADVISORY, \
- (locktag).locktag\_lockmethodid = USER\_LOCKMETHOD)

# PostgreSQL advisory锁介绍

- repeatable read及以上级别长事务带来的问题举例

- 会话A：

- digoal=# begin isolation level repeatable read;

- BEGIN

- digoal=# select 1;

- ?column?

- -----

- 1

- (1 row)

- 假设这是个长事务.

# PostgreSQL advisory锁介绍

- 会话B：
  - digoal=# delete from iso\_test;
  - DELETE 10000
  - digoal=# vacuum verbose iso\_test ;
  - INFO: vacuuming "postgres.iso\_test"
  - INFO: "iso\_test": found 0 removable, 10000 nonremovable row versions in 55 out of 55 pages
  - DETAIL: 10000 dead row versions cannot be removed yet.
  - There were 0 unused item pointers.
  - 0 pages are entirely empty.
  - CPU 0.00s/0.00u sec elapsed 0.00 sec.
  - INFO: vacuuming "pg\_toast.pg\_toast\_93022"
  - INFO: index "pg\_toast\_93022\_index" now contains 0 row versions in 1 pages

# PostgreSQL advisory锁介绍

- DETAIL: 0 index row versions were removed.
- 0 index pages have been deleted, 0 are currently reusable.
- CPU 0.00s/0.00u sec elapsed 0.00 sec.
- INFO: "pg\_toast\_93022": found 0 removable, 0 nonremovable row versions in 0 out of 0 pages
- DETAIL: 0 dead row versions cannot be removed yet.
- There were 0 unused item pointers.
- 0 pages are entirely empty.
- CPU 0.00s/0.00u sec elapsed 0.00 sec.
- VACUUM
  
- 会话A结束后, 这部分数据才可以被回收掉
- End;
  
- digoal=# vacuum verbose iso\_test ;
- INFO: vacuuming "postgres.iso\_test"
- INFO: "iso\_test": removed 10000 row versions in 55 pages
- INFO: "iso\_test": found 10000 removable, 0 nonremovable row versions in 55 out of 55 pages

# PostgreSQL advisory锁介绍

- DETAIL: 0 dead row versions cannot be removed yet.
- There were 0 unused item pointers.
- 0 pages are entirely empty.
- CPU 0.00s/0.00u sec elapsed 0.00 sec.
- INFO: "iso\_test": truncated 55 to 0 pages
- DETAIL: CPU 0.00s/0.00u sec elapsed 0.00 sec.
- INFO: vacuuming "pg\_toast.pg\_toast\_93022"
- INFO: index "pg\_toast\_93022\_index" now contains 0 row versions in 1 pages
- DETAIL: 0 index row versions were removed.
- 0 index pages have been deleted, 0 are currently reusable.
- CPU 0.00s/0.00u sec elapsed 0.00 sec.
- INFO: "pg\_toast\_93022": found 0 removable, 0 nonremovable row versions in 0 out of 0 pages
- DETAIL: 0 dead row versions cannot be removed yet.
- There were 0 unused item pointers.
- 0 pages are entirely empty.
- CPU 0.00s/0.00u sec elapsed 0.00 sec.
- VACUUM

# PostgreSQL advisory锁介绍

- advisory会话锁解决的问题
  - <http://blog.163.com/digoal@126/blog/static/163877040201172492217830/>
  - <http://blog.163.com/digoal@126/blog/static/1638770402013518111043463/>
- advisory lock的应用场景举例(应用控制的锁):
  - 比如数据库里面存储了文件和ID的对应关系，应用程序需要长时间得获得一个锁，然后对文件进行修改，再释放锁。
  - 测试数据:
    - digoal=> create table tbl\_file\_info (id int primary key,file\_path text);
    - NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "tbl\_file\_info\_pkey" for table "tbl\_file\_info"
    - CREATE TABLE
    - digoal=> insert into tbl\_file\_info values (1,'/home/postgres/advisory\_lock\_1.txt');
    - INSERT 0 1
    - digoal=> insert into tbl\_file\_info values (2,'/home/postgres/advisory\_lock\_2.txt');
    - INSERT 0 1
    - digoal=> insert into tbl\_file\_info values (3,'/home/postgres/advisory\_lock\_3.txt');
    - INSERT 0 1

# PostgreSQL advisory锁介绍

- SESSION A:
  - digoal=> select pg\_advisory\_lock(id),file\_path from tbl\_file\_info where id=1;
  - pg\_advisory\_lock | file\_path
  - -----+-----
  - | /home/postgres/advisory\_lock\_1.txt
  - (1 row)
  - 应用程序对/home/postgres/advisory\_lock\_1.txt文件进行编辑之后，再释放这个advisory锁。
  
- SESSION B:
  - 当SESSIONA在编辑/home/postgres/advisory\_lock\_1.txt这个文件的时候，无法获得这个锁，所以可以确保不会同时编辑这个文件。

# PostgreSQL 死锁介绍

- SESSION A:
  - Lock tuple 1;
- SESSION B:
  - Lock tuple 2;
- SESSION A:
  - Lock tuple 2 waiting;
- SESSION B:
  - Lock tuple 1 waiting;
- A,B相互等待.
  
- 死锁检测算法介绍
- src/backend/storage/lmgr/README
  
- 死锁检测的时间间隔配置, deadlock\_timeout 默认为1秒.
- 锁等待超过这个配置后, 触发死锁检测算法.
- 因为死锁检测比较耗资源, 所以这个时间视情况而定.
- PostgreSQL和Oracle死锁检测的区别例子.
- <http://blog.163.com/digoal@126/blog/static/16387704020113811711716/>
- 规避死锁需要从业务逻辑的角度去规避, 避免发生这种交错持锁和交错等待的情况.

# 目录

- 授课环境
- SQL优化基础
- 如何让数据库输出好的执行计划
- 压力测试工具的使用和建模
- 性能分析工具的使用
- 综合优化案例

# 如何让数据库输出好的执行计划

- 成本因子校准(前面已讲述)
- 执行计划节点开关(数据访问, 数据关联开关)
- 子查询表关联提升开关
- 强制表关联顺序开关
- GEQO(表关联-遗传算法)

# PostgreSQL 数据访问方法开关

- 某些开关并不是强制执行的, 例如只有全表扫描这种方法可选的时候, 即使关闭全表扫描也会走全表扫描, 但是这种情况下cost值会不准确.
- enable\_seqscan -- 全表扫描开关
- enable\_indexscan -- 索引扫描开关
- enable\_indexonlyscan -- indexonly索引扫描开关(仅扫描索引和VM, 以及vm以外的heap page)
- enable\_bitmapscan -- 位图扫描开关 (先扫描索引, 然后按ctid排序扫描heap)
  
- enable\_nestloop -- 嵌套循环连接
- enable\_hashjoin -- hash连接
- enable\_mergejoin -- 合并连接
- enable\_material -- 物化开关, 例如嵌套循环的内部表是否采用物化方式减少扫描成本.(例子见嵌套循环的例子)
  
- enable\_hashagg -- hash聚合
- enable\_sort -- 排序(例子见合并连接的例子)
- enable\_tidscan -- tid扫描开关

# PostgreSQL 数据访问方法开关

例子1：

```
digoal=# create table tbl(id int, info text, crt_time timestamp);
CREATE TABLE
digoal=# insert into tbl select generate_series(1,10000),md5(random()::text),clock_timestamp();
INSERT 0 10000
digoal=# create index idx_tbl_id on tbl(id);
CREATE INDEX
digoal=# analyze tbl;
ANALYZE
digoal=# explain select * from tbl where id=1;
 QUERY PLAN
```

---

Index Scan using idx\_tbl\_id on tbl (cost=0.29..8.30 rows=1 width=45)

  Index Cond: (id = 1)  
(2 rows)

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_indexscan=off;
```

SET

```
digoal=# explain select * from tbl where id=1;
```

QUERY PLAN

---

```
Bitmap Heap Scan on tbl (cost=4.29..8.31 rows=1 width=45)
```

```
 Recheck Cond: (id = 1)
```

```
 -> Bitmap Index Scan on idx_tbl_id (cost=0.00..4.29 rows=1 width=0)
```

```
 Index Cond: (id = 1)
```

```
(4 rows)
```

```
digoal=# set enable_bitmapscan=off;
```

SET

```
digoal=# explain select * from tbl where id=1;
```

QUERY PLAN

---

```
Seq Scan on tbl (cost=0.00..149.00 rows=1 width=45)
```

```
 Filter: (id = 1)
```

```
(2 rows)
```

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_seqscan=off;
```

SET

```
digoal=# explain select * from tbl where id=1;
```

QUERY PLAN

---

```
Index Scan using idx_tbl_id on tbl (cost=1000000000.28..1000000008.30 rows=1 width=45)
```

Index Cond: (id = 1)

(2 rows)

全表扫描开关关闭后, 没有其他可选访问方法, 选择全表扫描.

成本计算如下 :

```
src/backend/optimizer/path/costsize.c
```

```
Cost disable_cost = 1.0e10;
```

```
cost_seqscan
```

...

```
if (!enable_seqscan)
```

```
 startup_cost += disable_cost;
```

...

# PostgreSQL 数据访问方法开关

例子2：

```
digoal=# create table tbl1(id int primary key, info text, crt_time timestamp);
CREATE TABLE
digoal=# insert into tbl1 select generate_series(1,20000),md5(random()::text),clock_timestamp();
INSERT 0 20000
digoal=# explain select tbl.* ,tbl1.* from tbl, tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
 QUERY PLAN
```

---

Merge Join (cost=0.57..420.07 rows=5000 width=90)

Merge Cond: (tbl.id = tbl1.id)

-> Index Scan using idx\_tbl\_id on tbl (cost=0.29..213.28 rows=10000 width=45)

-> Index Scan using tbl1\_pkey on tbl1 (cost=0.29..238.30 rows=10001 width=45)

Index Cond: (id > 9999)

(5 rows)

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_indexscan=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl, tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
 QUERY PLAN
```

---

```
Hash Join (cost=362.80..734.82 rows=5000 width=90)
 Hash Cond: (tbl1.id = tbl.id)
 -> Bitmap Heap Scan on tbl1 (cost=113.80..285.81 rows=10001 width=45)
 Recheck Cond: (id > 9999)
 -> Bitmap Index Scan on tbl1_pkey (cost=0.00..111.29 rows=10001 width=0)
 Index Cond: (id > 9999)
 -> Hash (cost=124.00..124.00 rows=10000 width=45)
 -> Seq Scan on tbl (cost=0.00..124.00 rows=10000 width=45)
```

(8 rows)

关闭索引扫描后, 选择了hash连接

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_hashjoin=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl, tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
 QUERY PLAN
```

---

```
Merge Join (cost=1738.65..1863.65 rows=5000 width=90)
 Merge Cond: (tbl.id = tbl1.id)
 -> Sort (cost=788.39..813.39 rows=10000 width=45)
 Sort Key: tbl.id
 -> Seq Scan on tbl (cost=0.00..124.00 rows=10000 width=45)
 -> Sort (cost=950.27..975.27 rows=10001 width=45)
 Sort Key: tbl1.id
 -> Bitmap Heap Scan on tbl1 (cost=113.80..285.81 rows=10001 width=45)
 Recheck Cond: (id > 9999)
 -> Bitmap Index Scan on tbl1_pkey (cost=0.00..111.29 rows=10001 width=0)
 Index Cond: (id > 9999)
(11 rows)
```

关闭hash连接后, 选择了合并连接. 合并连接的tbl1表选择了位图扫描和排序节点.

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_sort=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl, tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
 QUERY PLAN
```

---

```
Nested Loop (cost=0.30..43415.50 rows=5000 width=90)
 -> Seq Scan on tbl (cost=0.00..124.00 rows=10000 width=45)
 -> Bitmap Heap Scan on tbl1 (cost=0.30..4.32 rows=1 width=45)
 Recheck Cond: ((id = tbl.id) AND (id > 9999))
 -> Bitmap Index Scan on tbl1_pkey (cost=0.00..0.30 rows=1 width=0)
 Index Cond: ((id = tbl.id) AND (id > 9999))
```

(6 rows)

关闭排序后, 选择了嵌套循环.

# PostgreSQL 数据访问方法开关

```
digoal=# \q
digoal=# set enable_mergejoin=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl,tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
```

## QUERY PLAN

---

```
Hash Join (cost=249.29..687.32 rows=5000 width=90)
 Hash Cond: (tbl1.id = tbl.id)
 -> Index Scan using tbl1_pkey on tbl1 (cost=0.29..238.30 rows=10001 width=45)
 Index Cond: (id > 9999)
 -> Hash (cost=124.00..124.00 rows=10000 width=45)
 -> Seq Scan on tbl (cost=0.00..124.00 rows=10000 width=45)
(6 rows)
```

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_hashjoin=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl, tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
QUERY PLAN
```

---

```
Nested Loop (cost=0.57..3520.62 rows=5000 width=90)
-> Index Scan using tbl1_pkey on tbl1 (cost=0.29..238.30 rows=10001 width=45)
 Index Cond: (id > 9999)
-> Index Scan using idx_tbl_id on tbl (cost=0.29..0.32 rows=1 width=45)
 Index Cond: (id = tbl1.id)
(5 rows)
```

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_nestloop=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl,tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
QUERY PLAN
```

---

Nested Loop (cost=1000000000.57..10000003520.62 rows=5000 width=90)

```
-> Index Scan using tbl1_pkey on tbl1 (cost=0.29..238.30 rows=10001 width=45)
 Index Cond: (id > 9999)
-> Index Scan using idx_tbl_id on tbl (cost=0.29..0.32 rows=1 width=45)
 Index Cond: (id = tbl1.id)
(5 rows)
```

所有的连接方法都关闭后, 选择了nestloop关联.

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_indexscan=off;
SET
digoal=# set enable_bitmapscan=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl, tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
QUERY PLAN
```

---

Nested Loop (cost=10000000000.00..10001500596.00 rows=5000 width=90)

Join Filter: (tbl.id = tbl1.id)  
-> Seq Scan on tbl (cost=0.00..124.00 rows=10000 width=45)  
-> Materialize (cost=0.00..347.00 rows=10001 width=45)  
-> Seq Scan on tbl1 (cost=0.00..297.00 rows=10001 width=45)  
Filter: (id > 9999)

(6 rows)

关闭索引扫描和位图扫描后, 嵌套循环连接的内部节点选择了物化.

# PostgreSQL 数据访问方法开关

```
digoal=# set enable_material=off;
SET
digoal=# explain select tbl.* ,tbl1.* from tbl, tbl1 where tbl.id=tbl1.id and tbl1.id>9999;
QUERY PLAN
```

---

Nested Loop (cost=10000000000.00..10002490546.00 rows=5000 width=90)

Join Filter: (tbl.id = tbl1.id)  
-> Seq Scan on tbl1 (cost=0.00..297.00 rows=10001 width=45)  
    Filter: (id > 9999)  
-> Seq Scan on tbl (cost=0.00..124.00 rows=10000 width=45)  
(5 rows)

关闭物化, 嵌套循环连接的内部节点使用全表扫描.

# 提升子查询关联等级配置参数

- 参数 : from\_collapse\_limit 代码 : src/backend/optimizer/plan/initplan.c
- ```
foreach(l, f->fromlist)
{
    Relids      sub_qualscope;
    List       *sub_joinlist;
    int        sub_members;
    sub_joinlist = deconstruct_recurse(root, lfirst(l),
                                         below_outer_join,
                                         &sub_qualscope,
                                         inner_join_rels,
                                         &child_postponed_quals);
    *qualscope = bms_add_members(*qualscope, sub_qualscope);
    sub_members = list_length(sub_joinlist);
    remaining--;
    if (sub_members <= 1 || list_length(joinlist) + sub_members + remaining <= from_collapse_limit) // 此子查询只包含1个表 或者 提升此子查询关联等级后的总list长度小于等于from_collapse_limit时, 提升此子查询.
        joinlist = list_concat(joinlist, sub_joinlist);
    else
        joinlist = lappend(joinlist, sub_joinlist);
}
```

提升子查询关联等级配置参数

- 例子：
- SELECT *
- FROM x, y,
- (SELECT * FROM a, b, c WHERE something) AS ss
- WHERE somethingelse;
- ss子查询提升后, from list的长度变成5, 小于等于from_collapse_limit=8(default). 因此这个查询会提升子查询关联等级.
- 提升关联等级的**好处**, 关联顺序更多, 可以得到更优的执行计划(cost更低的执行计划).
- **坏处**, 因为子查询提升后, 关联顺序指数级提升, 执行计划耗费的时间会更长, 例如本例将从9种顺序提升到60种顺序.
- 默认的from_collapse_limit=8(default) 将达到20160种关联顺序.

- 如果不提升子查询SS的关联, 那么关联组合为a,b,c一组以及x,y,ss一组. 一共9种关联顺序.
 - a b c
 - a c b
 - b c a

- x y ss
- x ss y
- y ss x

- 如果提升子查询的关联等级, 那么关联组合将变成x,y,a,b,c一组. 一共60种关联顺序(5*4*3)
- 把from_collapse_limit设置为1, 则相当于关闭了提升子查询关联的功能. 相当于强制子查询内按照SQL写法关联.

指定表的关联顺序配置参数

- 参数 : join_collapse_limit=8 (default) 代码 : src/backend/optimizer/plan/initplan.c

```
■ if (j->jointype == JOIN_FULL)
■ {
■     /* force the join order exactly at this node */
■     joinlist = list_make1(list_make2(leftjoinlist, rightjoinlist));
■ }
■ else if (list_length(leftjoinlist) + list_length(rightjoinlist) <= join_collapse_limit) // full join因为语义的原因,无法修改它的关联顺序,
■ 所以join_collapse_limit只针对非full join的情况.
■ {
■     /* OK to combine subproblems */
■     joinlist = list_concat(leftjoinlist, rightjoinlist);
■ }
```

指定表的关联顺序配置参数

- 例子：
 - `SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);` -- 首先确保语义的正确性, 所以这个SQL不能改变关联顺序.
 - `SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);` -- 这个SQL 可以改变b和c的顺序. abc, acb.
 - `SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;` -- 以下SQL没有左连接,右连接,全连接. 可以任意改变关联顺序.
 - `SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;`
 - `SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);`
- 当`join_collapse_limit=1`时, 按SQL写法进行关联. 例如：
 - `SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;`
 - `SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);`

指定表的关联顺序配置参数

- digoal=# create table tbl_join_1(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_2(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_3(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_4(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_5(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_6(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_7(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_8(id int primary key,info text);
- CREATE TABLE
- digoal=# create table tbl_join_9(id int primary key,info text);
- CREATE TABLE

指定表的关联顺序配置参数

- digoal=# insert into tbl_join_1 select generate_series(1,10),md5(random()::text);
- digoal=# insert into tbl_join_2 select generate_series(1,100),md5(random()::text);
- digoal=# insert into tbl_join_3 select generate_series(1,1000),md5(random()::text);
- digoal=# insert into tbl_join_4 select generate_series(1,10000),md5(random()::text);
- digoal=# insert into tbl_join_5 select generate_series(1,100000),md5(random()::text);
- digoal=# insert into tbl_join_6 select generate_series(1,1000000),md5(random()::text);
- digoal=# insert into tbl_join_7 select generate_series(1,2000000),md5(random()::text);
- digoal=# insert into tbl_join_8 select generate_series(1,3000000),md5(random()::text);
- digoal=# insert into tbl_join_9 select generate_series(1,4000000),md5(random()::text);

指定表的关联顺序配置参数

```
■ digoal=# explain select t1.info, t5.info from tbl_join_1 t1,  
■     tbl_join_2 t2,  
■     tbl_join_3 t3,  
■     tbl_join_4 t4,  
■     tbl_join_5 t5,  
■     tbl_join_6 t6,  
■     tbl_join_7 t7,  
■     tbl_join_8 t8,  
■     tbl_join_9 t9  
■     where  
■     t1.id=t2.id and  
■     t2.id=t3.id and  
■     t3.id=t4.id and  
■     t4.id=t5.id and  
■     t5.id=t6.id and  
■     t6.id=t7.id and  
■     t7.id=t8.id and  
■     t8.id=t9.id and  
■     t9.id=10000;
```

指定表的关联顺序配置参数

- QUERY PLAN
-
-
- -----
- Nested Loop (cost=2.35..68.82 rows=1 width=65)
 - -> Nested Loop (cost=1.92..60.36 rows=1 width=69)
 - -> Nested Loop (cost=1.49..51.90 rows=1 width=69)
 - -> Nested Loop (cost=1.19..43.57 rows=1 width=69)
 - -> Nested Loop (cost=0.89..35.25 rows=1 width=69)
 - -> Nested Loop (cost=0.59..26.93 rows=1 width=36)
 - -> Nested Loop (cost=0.31..18.61 rows=1 width=36)
 - -> Nested Loop (cost=0.16..10.44 rows=1 width=36)
 - -> Index Scan using tbl_join_1_pkey on tbl_join_1 t1 (cost=0.16..8.18 rows=1 width=36)
 - Index Cond: (id = 10000)
 - -> Seq Scan on tbl_join_2 t2 (cost=0.00..2.25 rows=1 width=4)
 - Filter: (id = 10000)
 - -> Index Only Scan using tbl_join_3_pkey on tbl_join_3 t3 (cost=0.15..8.17 rows=1 width=4)
 - Index Cond: (id = 10000)

指定表的关联顺序配置参数

- -> Index Only Scan using tbl_join_4_pkey on tbl_join_4 t4 (cost=0.29..8.30 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Scan using tbl_join_5_pkey on tbl_join_5 t5 (cost=0.29..8.31 rows=1 width=37)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_6_pkey on tbl_join_6 t6 (cost=0.30..8.32 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_7_pkey on tbl_join_7 t7 (cost=0.30..8.32 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_8_pkey on tbl_join_8 t8 (cost=0.43..8.45 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_9_pkey on tbl_join_9 t9 (cost=0.43..8.45 rows=1 width=4)
- Index Cond: (id = 10000)
- (26 rows)
- Time: 48.680 ms

指定表的关联顺序配置参数

```
■ digoal=# explain select t1.info, t5.info from
■     tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
■     join tbl_join_3 t3 on (t2.id=t3.id)
■     join tbl_join_4 t4 on (t3.id=t4.id)
■     join tbl_join_5 t5 on (t4.id=t5.id)
■     join tbl_join_6 t6 on (t5.id=t6.id)
■     join tbl_join_7 t7 on (t6.id=t7.id)
■     join tbl_join_8 t8 on (t7.id=t8.id)
■     join tbl_join_9 t9 on (t8.id=t9.id)
■     where t9.id=10000;
```

QUERY PLAN

```
■ -----
■ Nested Loop (cost=2.35..68.82 rows=1 width=65)
■   -> Nested Loop (cost=1.92..60.36 rows=1 width=69)
■     -> Nested Loop (cost=1.49..51.90 rows=1 width=69)
■       -> Nested Loop (cost=1.19..43.57 rows=1 width=69)
■         -> Nested Loop (cost=0.89..35.25 rows=1 width=69)
■           -> Nested Loop (cost=0.59..26.93 rows=1 width=36)
■             -> Nested Loop (cost=0.31..18.61 rows=1 width=36)
```

指定表的关联顺序配置参数

- -> Nested Loop (cost=0.16..10.44 rows=1 width=36)
 - -> Index Scan using tbl_join_1_pkey on tbl_join_1 t1 (cost=0.16..8.18 rows=1 width=36)
 - Index Cond: (id = 10000)
 - -> Seq Scan on tbl_join_2 t2 (cost=0.00..2.25 rows=1 width=4)
 - Filter: (id = 10000)
 - -> Index Only Scan using tbl_join_3_pkey on tbl_join_3 t3 (cost=0.15..8.17 rows=1 width=4)
 - Index Cond: (id = 10000)
 - -> Index Only Scan using tbl_join_4_pkey on tbl_join_4 t4 (cost=0.29..8.30 rows=1 width=4)
 - Index Cond: (id = 10000)
 - -> Index Scan using tbl_join_5_pkey on tbl_join_5 t5 (cost=0.29..8.31 rows=1 width=37)
 - Index Cond: (id = 10000)
 - -> Index Only Scan using tbl_join_6_pkey on tbl_join_6 t6 (cost=0.30..8.32 rows=1 width=4)
 - Index Cond: (id = 10000)
 - -> Index Only Scan using tbl_join_7_pkey on tbl_join_7 t7 (cost=0.30..8.32 rows=1 width=4)
 - Index Cond: (id = 10000)
 - -> Index Only Scan using tbl_join_8_pkey on tbl_join_8 t8 (cost=0.43..8.45 rows=1 width=4)
 - Index Cond: (id = 10000)
 - -> Index Only Scan using tbl_join_9_pkey on tbl_join_9 t9 (cost=0.43..8.45 rows=1 width=4)
 - Index Cond: (id = 10000)
 - (26 rows)
 - Time: 23.650 ms

指定表的关联顺序配置参数

- digoal=# set join_collapse_limit=9;

- digoal=# explain select t1.info, t5.info from
■ tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
■ join tbl_join_3 t3 on (t2.id=t3.id)
■ join tbl_join_4 t4 on (t3.id=t4.id)
■ join tbl_join_5 t5 on (t4.id=t5.id)
■ join tbl_join_6 t6 on (t5.id=t6.id)
■ join tbl_join_7 t7 on (t6.id=t7.id)
■ join tbl_join_8 t8 on (t7.id=t8.id)
■ join tbl_join_9 t9 on (t8.id=t9.id)
■ where t9.id=10000;
■ Time: 51.591 ms

指定表的关联顺序配置参数

- digoal=# set geqo_threshold=9;

- digoal=# explain select t1.info, t5.info from
■ tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
■ join tbl_join_3 t3 on (t2.id=t3.id)
■ join tbl_join_4 t4 on (t3.id=t4.id)
■ join tbl_join_5 t5 on (t4.id=t5.id)
■ join tbl_join_6 t6 on (t5.id=t6.id)
■ join tbl_join_7 t7 on (t6.id=t7.id)
■ join tbl_join_8 t8 on (t7.id=t8.id)
■ join tbl_join_9 t9 on (t8.id=t9.id)
■ where t9.id=10000;

- Time: 18.359 ms

- 因为geqo_threshold和join_collapse_limit设置为相等, 所以这个SQL在生成join list时, 会达到geqo限制, 触发geqo plan.
- 因此只需计划的时间也缩短了.
- 后面会讲geqo.

指定表的关联顺序配置参数

- digoal=# set join_collapse_limit=1;

- digoal=# explain select t1.info, t5.info from
■ tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
■ join tbl_join_3 t3 on (t2.id=t3.id)
■ join tbl_join_4 t4 on (t3.id=t4.id)
■ join tbl_join_5 t5 on (t4.id=t5.id)
■ join tbl_join_6 t6 on (t5.id=t6.id)
■ join tbl_join_7 t7 on (t6.id=t7.id)
■ join tbl_join_8 t8 on (t7.id=t8.id)
■ join tbl_join_9 t9 on (t8.id=t9.id)
■ where t9.id=10000;

- Time: 5.605 ms

- 设置join_collapse_limit=1后, 按照SQL写法进行关联. 执行计划的时间也缩短了.

- 表关联是执行计划最耗时的部分,特别是表多的情况下,关联顺序指数增长.
- 例如9个表关联最多有 $9*8*7*6*5*4*3 = 181440$ 种关联顺序,
- 如果再乘上连接方法(nestloop, hashjoin, mergejoin)和扫描方法(indexscan, seqscan, indexonlyscan, bitmapscan等)则最多有 $181440*(3^{(9-1)})*(n^{9-1})$ 种组合.
- 所以执行计划的开销会随着关联表的数量而指数级的增加.

- GEQO是一种利用遗传算法解决穷举法随着关联表的增加而带来的执行计划耗费暴增的问题.
- src/backend/optimizer/geqo
- geqo得到的执行计划也许不是最优的,但是可以降低执行计划耗费的时间.
- 相关参数
- geqo (boolean) -- geqo开关,默认打开
- geqo_threshold (integer) -- JOIN表数量,大于或等于这个值时,将启用GEQO. 默认12.
- geqo_effort (integer) -- 优化倾向,值越大,得出的路径越多,越有可能得到更优的执行计划,但是带来更多的开销. 默认5. (1-10)
- geqo_pool_size (integer) -- 默认0,从geqo_effort得出合适的值,含义同上.
- geqo_generations (integer) -- 默认0,从geqo_pool_size得出合适的值,含义同上.
- geqo_selection_bias (floating point) -- GEQO选择性偏差,默认2.0 (1.5-2.0)
- geqo_seed (floating point) -- 随机数初始值,默认0.

- 例子：
 - digoal=# set geqo_threshold=8;
 - digoal=# explain select t1.info, t5.info from
 - tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
 - join tbl_join_3 t3 on (t2.id=t3.id)
 - join tbl_join_4 t4 on (t3.id=t4.id)
 - join tbl_join_5 t5 on (t4.id=t5.id)
 - join tbl_join_6 t6 on (t5.id=t6.id)
 - join tbl_join_7 t7 on (t6.id=t7.id)
 - join tbl_join_8 t8 on (t7.id=t8.id)
 - join tbl_join_9 t9 on (t8.id=t9.id)
 - where t9.id=10000;

QUERY PLAN

- -> Index Only Scan using tbl_join_4_pkey on tbl_join_4 t4 (cost=0.29..8.30 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Scan using tbl_join_1_pkey on tbl_join_1 t1 (cost=0.16..8.18 rows=1 width=36)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_6_pkey on tbl_join_6 t6 (cost=0.30..8.32 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Seq Scan on tbl_join_2 t2 (cost=0.00..2.25 rows=1 width=4)
- Filter: (id = 10000)
- -> Index Scan using tbl_join_5_pkey on tbl_join_5 t5 (cost=0.29..8.31 rows=1 width=37)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_8_pkey on tbl_join_8 t8 (cost=0.43..8.45 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_3_pkey on tbl_join_3 t3 (cost=0.15..8.17 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_7_pkey on tbl_join_7 t7 (cost=0.30..8.32 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_9_pkey on tbl_join_9 t9 (cost=0.43..8.45 rows=1 width=4)
- Index Cond: (id = 10000)
- (26 rows)
- Time: 16.694 ms -- 因为join_collapse_limit=8, 所以join list长度为8, 达到使用geqo的阈值, 执行计划耗时16.694毫秒

- digoal=# set geqo_threshold=9;
- digoal=# explain select t1.info, t5.info from
- tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
- join tbl_join_3 t3 on (t2.id=t3.id)
- join tbl_join_4 t4 on (t3.id=t4.id)
- join tbl_join_5 t5 on (t4.id=t5.id)
- join tbl_join_6 t6 on (t5.id=t6.id)
- join tbl_join_7 t7 on (t6.id=t7.id)
- join tbl_join_8 t8 on (t7.id=t8.id)
- join tbl_join_9 t9 on (t8.id=t9.id)
- where t9.id=10000;

QUERY PLAN

- Nested Loop (cost=2.35..**68.82** rows=1 width=65)
 - -> Nested Loop (cost=1.92..60.36 rows=1 width=69)
 - -> Nested Loop (cost=1.49..51.90 rows=1 width=69)
 - -> Nested Loop (cost=1.19..43.57 rows=1 width=69)
 - -> Nested Loop (cost=0.89..35.25 rows=1 width=69)
 - -> Nested Loop (cost=0.59..26.93 rows=1 width=36)
 - -> Nested Loop (cost=0.31..18.61 rows=1 width=36)
 - -> Nested Loop (cost=0.16..10.44 rows=1 width=36)

- -> Index Scan using tbl_join_1_pkey on tbl_join_1 t1 (cost=0.16..8.18 rows=1 width=36)
- Index Cond: (id = 10000)
- -> Seq Scan on tbl_join_2 t2 (cost=0.00..2.25 rows=1 width=4)
- Filter: (id = 10000)
- -> Index Only Scan using tbl_join_3_pkey on tbl_join_3 t3 (cost=0.15..8.17 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_4_pkey on tbl_join_4 t4 (cost=0.29..8.30 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Scan using tbl_join_5_pkey on tbl_join_5 t5 (cost=0.29..8.31 rows=1 width=37)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_6_pkey on tbl_join_6 t6 (cost=0.30..8.32 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_7_pkey on tbl_join_7 t7 (cost=0.30..8.32 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_8_pkey on tbl_join_8 t8 (cost=0.43..8.45 rows=1 width=4)
- Index Cond: (id = 10000)
- -> Index Only Scan using tbl_join_9_pkey on tbl_join_9 t9 (cost=0.43..8.45 rows=1 width=4)
- Index Cond: (id = 10000)
- (26 rows)
- Time: 23.407 ms -- joinCollapse_limit=8时, join list长度为8, 小于geqo阈值9, 采用穷举法, 执行计划耗时23.407毫秒.

- digoal=# set geqo_threshold=12;
- digoal=# set join_collapse_limit=9; -- join list长度到9.
- digoal=# explain select t1.info, t5.info from
- tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
- join tbl_join_3 t3 on (t2.id=t3.id)
- join tbl_join_4 t4 on (t3.id=t4.id)
- join tbl_join_5 t5 on (t4.id=t5.id)
- join tbl_join_6 t6 on (t5.id=t6.id)
- join tbl_join_7 t7 on (t6.id=t7.id)
- join tbl_join_8 t8 on (t7.id=t8.id)
- join tbl_join_9 t9 on (t8.id=t9.id)
- where t9.id=10000;
- ...
- Time: 54.376 ms -- 采用穷举法

- digoal=# set geqo_threshold=9; -- geqo阈值也调到9
- digoal=# explain select t1.info, t5.info from
- tbl_join_1 t1 join tbl_join_2 t2 on (t1.id=t2.id)
- join tbl_join_3 t3 on (t2.id=t3.id)
- join tbl_join_4 t4 on (t3.id=t4.id)
- join tbl_join_5 t5 on (t4.id=t5.id)
- join tbl_join_6 t6 on (t5.id=t6.id)
- join tbl_join_7 t7 on (t6.id=t7.id)
- join tbl_join_8 t8 on (t7.id=t8.id)
- join tbl_join_9 t9 on (t8.id=t9.id)
- where t9.id=10000;
- ...
- Time: 18.506 ms -- 采用geqo.

- digoal=# set geqo_effort =10; -- 把effort改到10, 则geqo的耗时增加. 执行计划有所变化.
- Time: 32.341 ms

目录

- 授课环境
- SQL优化基础
- 如何让数据库输出好的执行计划
- 压力测试工具的使用和建模
- 性能分析工具的使用
- 综合优化案例



压力测试工具

压力测试工具

- <http://www.postgresql.org/docs/9.3/static/pgbench.html>
- contrib/pgbench/pgbench.c

- pgbench是PostgreSQL自带的一个数据库压力测试工具,
- 支持TPC-B测试模型, 或自定义测试模型.
- 自定义测试模型 支持元命令, 调用shell脚本, 设置随机数, 变量等等.
- 支持3种异步接口.
- ```
int PQsendQuery(PGconn *conn, const char *command); // 简单调用, -M simple
```
- ```
int PQsendQueryParams(PGconn *conn, // 带参数的扩展调用, -M extended
```

 - const char *command,
 - int nParams,
 - const Oid *paramTypes,
 - const char * const *paramValues,
 - const int *paramLengths,
 - const int *paramFormats,
 - int resultFormat);
- ```
PGresult *PQprepare(PGconn *conn, // 生成prepare使用同步接口.
```

  - const char \*stmtName,
  - const char \*query,
  - int nParams,
  - const Oid \*paramTypes);
- ```
int PQsendQueryPrepared ( ... ) // prepared 调用, -M prepared
```

压力测试工具

- -M simple
 - if (querymode == QUERY_SIMPLE)
 - {
 - char *sql;
 -
 - sql = pg_strdup(command->argv[0]);
 - sql = assignVariables(st, sql);
 -
 - if (debug)
 - fprintf(stderr, "client %d sending %s\n", st->id, sql);
 - r = PQsendQuery(st->con, sql);
 - free(sql);
 - }

压力测试工具

- -M extended
- else if (querymode == QUERY_EXTENDED)
- {
- const char *sql = command->argv[0];
- const char *params[MAX_ARGS];
-
- getQueryParams(st, command, params);
-
- if (debug)
- fprintf(stderr, "client %d sending %s\n", st->id, sql);
- r = **PQsendQueryParams**(st->con, sql, command->argc - 1,
■ NULL, params, NULL, NULL, 0);
- }

压力测试工具

- -M prepared
- else if (querymode == QUERY_PREPARED)
- {
- char name[MAX_PREPARE_NAME];
- const char *params[MAX_ARGS];
-
- if (!st->prepared[st->use_file])
- {
- int j;
-
- for (j = 0; commands[j] != NULL; j++)
- {
- PGresult *res;
- char name[MAX_PREPARE_NAME];
-
- if (commands[j]->type != SQL_COMMAND)
- continue;
- preparedStatementName(name, st->use_file, j);
- res = PQprepare(st->con, name,
- commands[j]->argv[0], commands[j]->argc - 1, NULL);

压力测试工具

- 默认的测试模型： TPC-B测试模型：

```
/* default scenario */
static char *tpc_b = {
    "\set nbranches " CppAsString2(nbranches) " * :scale\n"
    "\set ntellers " CppAsString2(ntellers) " * :scale\n"
    "\set naccounts " CppAsString2(naccounts) " * :scale\n"
    "\setrandom aid 1 :naccounts\n" // 每个会话独立, 所以每个会话将得到不一样的随机数.
    "\setrandom bid 1 :nbranches\n"
    "\setrandom tid 1 :ntellers\n"
    "\setrandom delta -5000 5000\n"
    "BEGIN;\n"
    "UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;\n"
    "SELECT abalance FROM pgbench_accounts WHERE aid = :aid;\n"
    "UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;\n"
    "UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;\n"
    "INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);\n"
    "END;\n";
};
```

压力测试工具

- TPC-B测试模型减掉两条更新SQL. 使用-N 参数

```
/* -N case */

static char *simple_update = {

    "\\set nbranches " CppAsString2(nbranches) " * :scale\n"
    "\\set ntellers " CppAsString2(ntellers) " * :scale\n"
    "\\set naccounts " CppAsString2(naccounts) " * :scale\n"
    "\\setrandom aid 1 :naccounts\n"
    "\\setrandom bid 1 :nbranches\n"
    "\\setrandom tid 1 :ntellers\n"
    "\\setrandom delta -5000 5000\n"
    "BEGIN;\n"
    "UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;\n"
    "SELECT abalance FROM pgbench_accounts WHERE aid = :aid;\n"
    "INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);\n"
    "END;\n";
}
```

压力测试工具

- 仅仅包含查询的SQL. 使用-S 参数

```
/* -S case */  
static char *select_only = {  
    "\\\set naccounts " CppAsString2(naccounts) " * :scale\n"  
    "\\\setrandom aid 1 :naccounts\n"  
    "SELECT abalance FROM pgbench_accounts WHERE aid = :aid;\n"  
};
```

压力测试工具

■ 使用TPC-B 进行测试

■ 首先要初始化数据.

■ int scale = 1; // 通过-s 指定 scale的值, 产生3个表的测试数据(以下的倍数, 例如-s 10 则产生10条branches记录.).

■ #define nbranches 1 /* Makes little sense to change this. Change -s instead */ // 1条记录.

■ #define ntellers 10 // 10条记录.

■ #define naccounts 100000 // 10万条记录.

■ for (i = 0; i < nbranches * scale; i++)

■ {

■ snprintf(sql, 256, "insert into pgbench_branches(bid,bbalance) values(%d,0)", i + 1);

■ executeStatement(con, sql);

■ }

■ for (i = 0; i < ntellers * scale; i++)

■ {

■ snprintf(sql, 256, "insert into pgbench_tellers(tid,bid,tbalance) values (%d,%d,0)",

■ i + 1, i / ntellers + 1);

■ executeStatement(con, sql);

■ }

压力测试工具

- pg93@db-172-16-3-150-> pgbench -i --foreign-keys --unlogged-tables -s 32 -h \$PGDATA -p \$PGPORT -U \$PGUSER \$PGDATABASE
 - // 指定scale=32, 产生32条branch记录.
 - // 还可以指定表空间以及索引表空间. --tablespace=, --index-tablespace=
 - creating tables...
 - 100000 of 3200000 tuples (3%) done (elapsed 0.15 s, remaining 4.74 s).
 - ...
 - 3200000 of 3200000 tuples (100%) done (elapsed 5.59 s, remaining 0.00 s).
 - vacuum...
 - set primary keys...
 - set foreign keys...
 - done.

- public | pgbench_accounts | table | postgres | 403 MB |
- public | pgbench_branches | table | postgres | 160 kB |
- public | pgbench_history | table | postgres | 0 bytes |
- public | pgbench_tellers | table | postgres | 160 kB |
- // 另外两个初始化参数
- -F NUM fill factor // 指定建表的fill_factor.(heap page的保留空间, 对于更新频繁的表, 可以产生HOT, 有利于降低索引膨胀以及索引更新的可能性)
- -n do not run VACUUM after initialization // 数据初始化后不执行vacuum

压力测试工具

- TPC-B测试, 使用8个数据库长连接, 2个工作线程, prepared接口, 报告每条语句的平均执行延迟, 测试时长10秒.
- pg93@db-172-16-3-150-> pgbench -M prepared -r -c 8 -j 2 -T 10 -h \$PGDATA -p \$PGPORT -U \$PGUSER \$PGDATABASE
- starting vacuum...end.
- transaction type: TPC-B (sort of)
- scaling factor: 32
- query mode: prepared
- number of clients: 8
- number of threads: 2
- duration: 10 s
- number of transactions actually processed: 39448
- tps = 3943.580251 (including connections establishing) // TPS指整个测试文本的统计, 例如这里的测试文本包含以下所有的元命令和SQL. 即使这里包含了多个begin和end, 也统计为一个"transaction".
- tps = 3949.696241 (excluding connections establishing)
- statement latencies in milliseconds:
 - 0.003896 \set nbranches 1 * :scale
 - 0.001069 \set ntellers 10 * :scale
 - 0.001076 \set naccounts 100000 * :scale
 - 0.001242 \setrandom aid 1 :naccounts
 - 0.000907 \setrandom bid 1 :nbranches

压力测试工具

- 0.000975 \setrandom tid 1 :ntellers
- 0.000911 \setrandom delta -5000 5000
- 0.128683 BEGIN;
- 0.288973 UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
- 0.178117 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
- 0.222753 UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
- 0.271114 UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
- 0.367600 INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
- 0.541556 END;

压力测试工具

- pgbench用法以及所有参数
- Usage:
- pgbench [OPTION]... [DBNAME]

- // TPC-B测试模型的初始化数据参数.
- Initialization options:
 - -i invokes initialization mode
 - -F NUM fill factor
 - -n do not run VACUUM after initialization // 使用自定义测试模型时, 请带上这个参数.
 - -q quiet logging (one message each 5 seconds)
 - -s NUM scaling factor
 - --foreign-keys
 - create foreign key constraints between tables
 - --index-tablespace=TABLESPACE
 - create indexes in the specified tablespace
 - --tablespace=TABLESPACE
 - create tables in the specified tablespace
 - --unlogged-tables
 - create tables as unlogged tables

压力测试工具

- // 压力测试相关参数
- Benchmarking options:
- -c NUM number of concurrent database clients (default: 1) // 指定pgbench连接到数据库的连接数
- -C establish new connection for each transaction // 是否使用短连接
- -D VARNAME=VALUE // 设置变量, 在自定义脚本中使用 :varname 引用. 可使用多个-D设置多个变量.
 define variable for use by custom script
- -f FILENAME read transaction script from FILENAME // 指定自定义的测试文件 (由元命令和SQL组成), 可使用多个-f 指定多个文件, 每个文件作为一个事务, 每次执行事务时随机选择一个文件执行.
- -j NUM number of threads (default: 1) // pgbench的工作线程.
- -l write transaction times to log file // 开启事务统计, 输出文件名格式 pgbench_log.\$PID.\$threadID ,(当-j >= 2时 , threadID从1开始)
- -M simple|extended|prepared // libpq接口
 protocol for submitting queries to server (default: simple)
- -n do not run VACUUM before tests // vacuum开关, 使用自定义文件时, 使用-n关闭vacuum.
- -N do not update tables "pgbench_tellers" and "pgbench_branches" // TPC-B 非默认测试模式, 少两个表的更新.
- -r report average latency per command // 报告测试文件中每条命令(包括元命令和SQL)的平均执行延迟.
- -s NUM report this scale factor in output // 使用自定义脚本测试时, 指定scale的输出. 没有实质意义.
- -S perform SELECT-only transactions // TPC-B 非默认测试模式, 只查询.

压力测试工具

- -t NUM number of transactions each client runs (default: 10) // 指定每个连接的执行事务数.
- -T NUM duration of benchmark test in seconds // 指定总的压力测试时间. 与-t不能同时使用.
- -v vacuum all four standard tables before tests // 测试前先vacuum 4个和tpc-b相关的表.
- --aggregate-interval=NUM // 输出聚合后的事务统计信息.
 aggregate data over NUM seconds
- --sampling-rate=NUM // 指定采样百分比, 得出的TPS将只有正常TPS*rate
 fraction of transactions to log (e.g. 0.01 for 1% sample)

- Common options:
- -d print debugging output
- -h HOSTNAME database server host or socket directory
- -p PORT database server port number
- -U USERNAME connect as specified database user
- -V, --version output version information, then exit
- -?, --help show this help, then exit

压力测试工具

- 事务统计输出信息解读(使用-l 输出)
- 非聚合模式输出的格式, 每个事务对应1条记录.
- client_id transaction_no time file_no time_epoch time_us
- client_id // 连接ID, 如使用-c 8, 则client_id范围0..7
- transaction_no // 事务号, 每个连接独立从0开始计数.
- time // 事务耗时, 微秒
- file_no // 文件号, 从0开始. 当使用了多个-f参数时用于辨认对应哪个文件.
- time_epoch // 当前的epoch时间
- time_us // 当前的epoch时间的偏移量, 微秒(千分之一毫秒).

- // 使用--sampling-rate减少采样.

压力测试工具

- 事务统计输出信息解读
- 聚合模式输出的格式(--aggregate-interval), 每隔一段时间输出事务时间段的事务统计.
- `interval_start num_of_transactions latency_sum latency_2_sum min_latency max_latency`
- `interval_start` // epoch时间, 指这个统计段的开始时间.
- `num_of_transactions` // 这个统计段运行了多少个"事务", 指独立的文件运行次数.
- `latency_sum` // 这个统计段的事务执行总耗时, 单位微秒.
- `latency_2_sum` // 这个统计段的事务执行耗时平方的总和, 单位微秒.
- `min_latency` // 这个统计段内, 单个事务的最小耗时.
- `max_latency` // 这个统计段内, 单个事务的最大耗时.

- `INSTR_TIME_SET_CURRENT(now);`
`diff = now;`
`INSTR_TIME_SUBTRACT(diff, st->txn_begin);`
`usec = (double) INSTR_TIME_GET_MICROSEC(diff);`
- ...
- `agg->sum += usec; // latency_sum`
`agg->sum2 += usec * usec; // latency_2_sum`

压力测试工具

- 自定义测试文件, 元命令
- `\set varname operand1 [operator operand2]`
- Sets variable varname to a calculated integer value. Each operand is either an integer constant or a :variablename reference to a variable having an integer value. The operator can be +, -, *, or /.
- Example:
- `\set ntellers 10 * :scale`

- `\setrandom varname min max`
- Sets variable varname to a random integer value between the limits min and max inclusive. Each limit can be either an integer constant or a :variablename reference to a variable having an integer value.
- Example:
- `\setrandom aid 1 :naccounts`

- `\sleep number [us | ms | s]`
- Causes script execution to sleep for the specified duration in microseconds (us), milliseconds (ms) or seconds (s). If the unit is omitted then seconds are the default. number can be either an integer constant or a :variablename reference to a variable having an integer value.
- Example:
- `\sleep 10 ms`

压力测试工具

- \setshell varname command [argument ...] // 调用SHELL, 并把SHELL执行返回值赋予给变量
- Sets variable varname to the result of the shell command command. The command must return an integer value through its standard output.
- argument can be either a text constant or a :variablename reference to a variable of any types. If you want to use argument starting with colons, you need to add an additional colon at the beginning of argument.
- Example:
- \setshell variable_to_be_assigned command literal_argument :variable ::literal_starting_with_colon

- \shell command [argument ...] // 调用SHELL, 忽略SHELL执行返回值.
- Same as \setshell, but the result is ignored.
- Example:
- \shell command literal_argument :variable ::literal_starting_with_colon

压力测试工具

- 自定义测试模型例子
- ```
digoal=# create table tbl_userinfo(id int primary key, info text, crt_time timestamp, mod_time timestamp);
digoal=# create table tbl_userinfo_audit(id int, info text, crt_time timestamp, mod_time timestamp, dml char(1));
digoal= create or replace function f_reg(i_id int) returns tbl_userinfo as $$
declare
 res tbl_userinfo;
begin
 update tbl_userinfo set mod_time=now() where id=i_id returning * into res;
 if not found then
 insert into tbl_userinfo(id,info,crt_time) values(i_id,md5(random()::text),now()) returning * into res;
 insert into tbl_userinfo_audit(id,info,crt_time,dml) values (res.id,res.info,res.crt_time,'i');
 return res;
 end if;
 insert into tbl_userinfo_audit(id,info,crt_time,mod_time,dml) values (res.id,res.info,res.crt_time,res.mod_time,'u');
 return res;
exception
 WHEN SQLSTATE '23505' THEN -- 防止违反唯一约束时调用函数报错. 以免pgbench输出一堆错误.
 return null;
end;
$$ language plpgsql strict;
```

# 压力测试工具

- 创建测试文件,如果有多个测试逻辑,可以创建多个测试文件,使用pgbench时,给出多个-f参数即可. 每个"事务"随机选择一个文件执行.
- vi test.sql
- \setrandom id :minid :maxid
- select \* from f\_reg(:id);
  
- 测试 :
- pgbench -M prepared -n -r -f ./test.sql -D minid=1 -D maxid=1000000 -c 8 -j 2 -T 10 -h \$PGDATA -p \$PGPORT -U \$PGUSER \$PGDATABASE
- transaction type: Custom query
- scaling factor: 1
- query mode: prepared
- number of clients: 8
- number of threads: 2
- duration: 10 s
- number of transactions actually processed: 91263
- tps = 9125.210450 (including connections establishing)
- tps = 9139.109833 (excluding connections establishing)
- statement latencies in milliseconds:
- 0.003303    \setrandom id :minid :maxid
- 0.869977    select \* from f\_reg(:id);

# 压力测试工具

- digoal=# select count(\*) from tbl\_userinfo\_audit ;
  - count
  - -----
  - 91263
  - (1 row)
  
- digoal=# select count(\*) from tbl\_userinfo;
  - count
  - -----
  - 87262
  - (1 row)

# 压力测试工具

- pgbench使用建议：
  - 1. 测试时间尽量延长(例如至少跨越2个checkpoint), 更能暴露真实存在的问题.
  - 2. 如果使用TPC-B测试模型, 那么初始化数据时指定的-s scale必须大于测试阶段指定的连接数-c NUM. 因为涉及更新branchs表的操作, 如果连接数大于这个表记录数, 那么势必带来更新锁等待. 因此如果测试连接数比较多的情况下, 尽量使用大的scale. 当然, 你可以选择-N参数, 不更新branchs表.
  - 3. 如果获取系统时间带来的额外开销比较大的话, 那么统计单条命令(-r)以及事务级别的统计(-l) 会导致测试的TPS与真实的TPS偏差较大.
  - 使用pg\_test\_timing测试获取系统时间带来的额外开销. 小于1微秒的越多越好(>90%).
  - 4. 被测试的数据库配置尽量和生产配置一致, 包括表结构, 索引, 参数, 表空间, IO能力, CPU能力等.
  - 5. pgbench测试时尽量模拟实际的客户端请求, 例如可以使用\sleep设置延迟, 使用-C 控制是否使用短连接. -M 指定是否使用绑定变量等.
  - 6. 测试数据在多次测试后, 可能导致表或索引的膨胀, 可以使用vacuum清理垃圾数据, 或者使用vacuum full重建测试数据.

# 目录

- 授课环境
- SQL优化基础
- 如何让数据库输出好的执行计划
- 压力测试工具的使用和建模
- 性能分析工具的使用
- 综合优化案例



# 性能分析工具

# 性能分析工具

## ■ 系统层面

- sar // Linux操作系统自带的统计信息收集. sysstat包.
- iostat // 跟踪块设备的IO读写请求次数, 字节数, 使用率, 等待队列, 平均等待时间等.
- vmstat // 跟踪虚拟内存的统计信息.
- stap // 内核动态跟踪.
- [http://blog.163.com/digoal@126/blog/#m=o&t=1&c=fks\\_084068084086080075085082085095085080082075083081086071084](http://blog.163.com/digoal@126/blog/#m=o&t=1&c=fks_084068084086080075085082085095085080082075083081086071084)

## ■ 数据库层面

- pg\_stat\_statements // 统计SQL的执行次数, CPU时间, IO时间, 命中数, 未命中数, 写buffer, 产生脏数据等统计信息.
- auto\_explain // 记录超过执行时间的SQL当时的执行计划.
- log\_min\_duration\_statement // 记录超过执行时间的SQL.

## ■ 其他第三方工具

- pg\_statsinfo // 可视化工具. 支持统计信息快照, 时间段统计报告输出等.
- <http://blog.163.com/digoal@126/blog/static/16387704020142585616183/>

# pg\_stat\_statements

- pg\_stat\_statements 利用\_PG\_init 接口创建钩子程序, 统计SQL信息. 在加载pg\_stat\_statements.so时, \_PG\_init被触发创建钩子, 初始化内存区域等.
- pg\_stat\_statements统计项, 视图pg\_stat\_statements :

| Name               | Type             | References                      | Description                                                                                                                    |
|--------------------|------------------|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| userid             | oid              | <a href="#">pg_authid.oid</a>   | OID of user who executed the statement                                                                                         |
| dbid               | oid              | <a href="#">pg_database.oid</a> | OID of database in which the statement was executed                                                                            |
| query              | text             | <a href="#">SQL</a>             | Text of a representative statement (up to <a href="#">track_activity_query_size</a> bytes)                                     |
| calls              | bigint           | 调用次数                            | Number of times executed                                                                                                       |
| total_time         | double precision | 毫秒                              | Total time spent in the statement, in milliseconds                                                                             |
| rows               | bigint           |                                 | Total number of rows retrieved or affected by the statement                                                                    |
| shared_blk_hit     | bigint           | 命中读                             | Total number of shared block cache hits by the statement                                                                       |
| shared_blk_read    | bigint           | 未命中读                            | Total number of shared blocks read by the statement                                                                            |
| shared_blk_dirtied | bigint           | 产生脏块                            | Total number of shared blocks dirtied by the statement                                                                         |
| shared_blk_written | bigint           | 写出脏块                            | Total number of shared blocks written by the statement                                                                         |
| local_blk_hit      | bigint           |                                 | Total number of local block cache hits by the statement                                                                        |
| local_blk_read     | bigint           |                                 | Total number of local blocks read by the statement                                                                             |
| local_blk_dirtied  | bigint           |                                 | Total number of local blocks dirtied by the statement                                                                          |
| local_blk_written  | bigint           |                                 | Total number of local blocks written by the statement                                                                          |
| temp_blk_read      | bigint           |                                 | Total number of temp blocks read by the statement                                                                              |
| temp_blk_written   | bigint           |                                 | Total number of temp blocks written by the statement                                                                           |
| blk_read_time      | double precision |                                 | Total time the statement spent reading blocks, in milliseconds (if <a href="#">track_io_timing</a> is enabled, otherwise zero) |
| blk_write_time     | double precision |                                 | Total time the statement spent writing blocks, in milliseconds (if <a href="#">track_io_timing</a> is enabled, otherwise zero) |

# pg\_stat\_statements

- pg\_stat\_statements安装和配置
  - export PATH=/opt/pgsql/bin:\$PATH
  - cd \$PGSRC/contrib/pg\_stat\_statements
  - make
  - make install
  
- vi \$PGDATA/postgresql.conf
  - shared\_preload\_libraries = 'pg\_stat\_statements' # 必要配置项
  - pg\_stat\_statements.max = 1024 # 最多存储的SQL数量, 如果条目占满的话, 使用最不频繁的SQL被覆盖掉.
  - pg\_stat\_statements.track = all # 跟踪哪些SQL, all包含嵌套的SQL, 例如函数中的SQL独立跟踪. top则表示只跟踪最顶层的SQL, 不会跟踪到函数内部的SQL. none表示不跟踪.
  - pg\_stat\_statements.track\_utility = on # 是否跟踪insert, update, delete, select以外的SQL.
  - pg\_stat\_statements.save = on # 关闭数据库后, 共享内存区的信息保存到\$PGDATA/global/pg\_stat\_statements.stat
  - track\_activity\_query\_size = 1024 # 存储的sql长度, 超出长度的部分被截断.
  
- 重启数据库, 在需要查看pg\_stat\_statements的库中安装extension. 因为pg\_stat\_statements中存储的是整个集群的统计信息, 建议只在需要用到的库安装.
  - psql -d postgres
  - create extension pg\_stat\_statements;

# pg\_stat\_statements

- 相关的视图和函数.
- 视图 :
  - pg\_stat\_statements
- 函数 :
  - pg\_stat\_statements\_reset() -- 用于重置pg\_stat\_statements的共享内存区的信息, 清除统计信息 .

# pg\_stat\_statements

- 注意1：
- 自从9.2版本开始, pg\_stat\_statements 支持将SQL里的某些常量替换为?, 减少存储条目. 类似绑定变量效果.
- 例如：
  - `select * from tbl_userinfo where id= 1;`
  - `select * from tbl_userinfo where id=2;`
  - `select id,info,crt_time,mod_time from tbl_userinfo where id=100; -- 包含表的所有列`
  - 末尾的常量都会替换成?, 具体存储的query为第一次跟踪到的query text. 例如：
    - `select * from tbl_userinfo where id= ?;`
    - 如果第一次执行的是`select id,info,crt_time,mod_time from tbl_userinfo where id=100;` 那么存储的query就是：
      - `select id,info,crt_time,mod_time from tbl_userinfo where id=?;`
  - <http://blog.163.com/digoal@126/blog/static/1638770402014130252595>

# pg\_stat\_statements

- 注意2：
  - search\_path不同的情况下,两条相同的SQL在pg\_stat\_statements中会存储2条记录.
  - 例如：
    - digoal=# \dn
    - List of schemas
    - Name | Owner
    - -----+-----
    - digoal | postgres
    - public | postgres
    - (2 rows)
  - digoal=# create table digoal.ttt(id int);
    - CREATE TABLE
  - digoal=# create table public.ttt(id int);
    - CREATE TABLE
  - digoal=# set search\_path='digoal';
    - SET
  - digoal=# select \* from ttt;
    - id
    - ----
  - (0 rows)

# pg\_stat\_statements

- digoal=# set search\_path='public';
- SET
- digoal=# select \* from ttt;
- id
- ----
- (0 rows)
  
- 存储2条. 实质上分别表示两个SQL. digoal.ttt和public.ttt
- digoal=# select calls,query from pg\_stat\_statements ;
- 1 | select \* from ttt;
- 1 | select \* from ttt;

# pg\_stat\_statements

- 注意3：
  - 如果配置项 pg\_stat\_statements.track\_utility = on
  - 那么除了select, insert, update, delete以外的其他SQL都会被跟踪, 包括修改用户密码的SQL. 创建用户的SQL等等.
  - 所以在执行敏感SQL时注意关闭这个开关.
  - digoal=# set pg\_stat\_statements.track\_utility=off;
  - SET
  - digoal=# set log\_statement=none; -- 执行以上两条后, 以下SQL就不会记录到pg\_stat\_statements的共享内存区以及数据库日志中.
  - digoal=# create role test login encrypted password 'ttt';
  - CREATE ROLE
  
- 如果只需要跟踪业务SQL性能的话, 推荐关闭pg\_stat\_statements.track\_utility.
- 除非业务中经常要用到select, update, insert, delete以外的SQL.

# pg\_stat\_statements

- 注意4：
  - 因为pg\_stat\_statements存储了SQL的信息, 所以只有超级用户可以查看所有用户的query信息, 普通用户只能看到自己执行的SQL.
  - digoal=# \c digoal test
    - You are now connected to database "digoal" as user "test".
    - digoal=> select query from pg\_stat\_statements;
      - query
      - -----
      - <insufficient privilege>
      - <insufficient privilege>
      - <insufficient privilege>
      - <insufficient privilege>
      - select ?; -- 自己执行的SQL可以看到.
      - 其他的只有超级用户能看到.
    - digoal=> \c digoal postgres
      - You are now connected to database "digoal" as user "postgres".
      - digoal=# select query from pg\_stat\_statements;
        - -----
        - delete from dbsize where dbname=?;
        - .....

# pg\_stat\_statements

- 用法举例：
- 定期统计CPU耗时排名前20的SQL, 以邮件形式发送出来. 统计结束后清除该时间段的pg\_stat\_statements共享内存区.
- ```
#!/bin/bash
export PGPORT=5432
export PGDATA=/data01/pgdata
export LANG=en_US.utf8
export PGHOME=/opt/pgsql
export LD_LIBRARY_PATH=$PGHOME/lib:/lib64:/usr/lib64:/usr/local/lib64:/lib:/usr/lib:/usr/local/lib
export DATE=`date +"%Y%m%d%H%M"`
export PATH=$PGHOME/bin:$PATH:.
export PGHOST=$PGDATA
export PGDATABASE=postgres
```
- ```
psql -A -x -c "select row_number() over() as rn, * from (select query,' calls:'||calls||' total_time_s:'||round(total_time::numeric,2)||'
avg_time_ms:'||round(1000*(total_time::numeric/calls),2) as stats from pg_stat_statements order by total_time desc limit 20) t;" >/tmp/stat_query.log 2>&1
echo -e "$DATE"|mutt -s "$DATE TOP20 query report" -a /tmp/stat_query.log digoal@126.com
psql -c "select pg_stat_statements_reset()"
```

# 目录

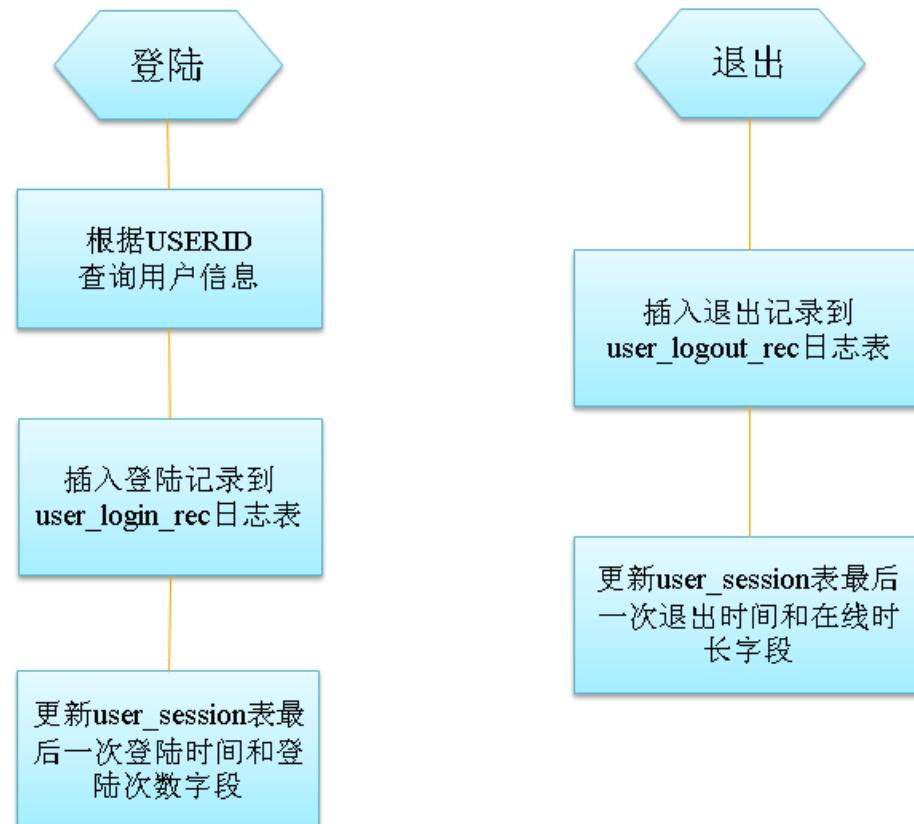
- 授课环境
- SQL优化基础
- 如何让数据库输出好的执行计划
- 压力测试工具的使用和建模
- 性能分析工具的使用
- 综合优化案例



# 综合优化案例

# 优化案例

- 测试场景建模
- (OLTP场景)



# 优化案例

- 创建用户, 库, schema
- create role digoal nosuperuser login encrypted password 'digoal';
- -- mkdir /home/postgres/tbs\_digoal
- create tablespace tbs\_digoal location '/home/postgres/tbs\_digoal';
- create database digoal with template template0 encoding 'UTF8' tablespace tbs\_digoal;
- grant all on database digoal to digoal;
- grant all on tablespace tbs\_digoal to digoal;
- \c digoal digoal
- create schema digoal;

# 优化案例

- create table user\_info -- 用户信息表
  - (userid int,
  - engname text,
  - cname text,
  - occupation text,
  - birthday date,
  - signname text,
  - email text,
  - qq numeric,
  - crt\_time timestamp without time zone,
  - mod\_time timestamp without time zone
  - );
  
- create table user\_session -- 用户会话表
  - (userid int,
  - logintime timestamp(0) without time zone,
  - login\_count bigint default 0,
  - logouttime timestamp(0) without time zone,
  - online\_interval interval default interval '0'
  - );

# 优化案例

- create table user\_login\_rec -- 用户登录记录
  - (userid int,
  - login\_time timestamp without time zone,
  - ip inet
  - );
  
- create table user\_logout\_rec -- 用户退出记录
  - (userid int,
  - logout\_time timestamp without time zone,
  - ip inet
  - );

# 优化案例

- -- 生成测试数据, 2000万条用户数据和会话数据.
- insert into user\_info (userid,engname,cnname,occupation,birthday,signname,email,qq,crt\_time,mod\_time)
- select generate\_series(1,4000000),
- 'digoal.zhou',
- '德哥',
- 'DBA',
- '1970-01-01'
- ,E'公益是一辈子的事, I'm Digoal.Zhou, Just do it!',
- 'digoal@126.com',
- 276732431,
- clock\_timestamp(),
- NULL;
  
- insert into user\_session (userid) select generate\_series(1,4000000);
  
- set work\_mem='2048MB';
- set maintenance\_work\_mem='2048MB';
- alter table user\_info add constraint pk\_user\_info primary key (userid);
- alter table user\_session add constraint pk\_user\_session primary key (userid);

# 优化案例

```
■ -- 模拟用户登录的函数
■ create or replace function f_user_login
■ (i_userid int,
■ OUT o_userid int,
■ OUT o_engname text,
■ OUT o_cname text,
■ OUT o_occupation text,
■ OUT o_birthday date,
■ OUT o_signname text,
■ OUT o_email text,
■ OUT o_qq numeric
■)
■ as $BODY$
■ declare
■ begin
■ select userid,engname,cname,occupation,birthday,signname,email,qq
■ into o_userid,o_engname,o_cname,o_occupation,o_birthday,o_signname,o_email,o_qq
■ from user_info where userid=i_userid;
```

# 优化案例

```
■ if FOUND then
■ insert into user_login_rec (userid,login_time,ip) values (i_userid,now(),inet_client_addr());
■ update user_session set logintime=now(),login_count=login_count+1 where userid=i_userid;
■ else
■ insert into user_info (userid) values (i_userid);
■ insert into user_session (userid, login_count) values (i_userid, 1);
■ end if;
■ return;
■ end;
■ $BODY$
■ language plpgsql;
```

# 优化案例

```
■ -- 模拟用户退出的函数
■ create or replace function f_user_logout
■ (i_userid int,
■ OUT o_result int
■)
■ as $BODY$
■ declare
■ begin
■ update user_session set logouttime=now(),online_interval=online_interval+(now()-logintime) where userid=i_userid;
■ if not found then
■ insert into user_session (userid) values (i_userid);
■ end if;
```

# 优化案例

```
■ insert into user_logout_rec (userid,logout_time,ip) values (i_userid,now(),inet_client_addr());
■ o_result := 0;
■ return;
■ exception
■ when others then
■ o_result := 1;
■ return;
■ end;
■ $BODY$
■ language plpgsql;
```

# 优化案例

- [优化阶段1]
  - 使用pgbench进行压力测试,发现瓶颈并合理优化.
  - 模拟场景我们假设有400万的活跃用户.大概占总用户的1/5.
    - 1. pgbench用到的登陆脚本
    - cat login.sql
      - \setrandom userid 1 4000000
      - select userid,engname,cnname,occupation,birthday,signname,email,qq from user\_info where userid=:userid;
      - insert into user\_login\_rec (userid,login\_time,ip) values (:userid,now(),inet\_client\_addr());
      - update user\_session set logintime=now(),login\_count=login\_count+1 where userid=:userid;
    - 2. pgbench用到的退出脚本
    - cat logout.sql
      - \setrandom userid 1 4000000
      - insert into user\_logout\_rec (userid,logout\_time,ip) values (:userid,now(),inet\_client\_addr());
      - update user\_session set logouttime=now(),online\_interval=online\_interval+(now()-logintime) where userid=:userid;
    - 3. 压力测试
    - 简单调用协议,长连接,8个连接,8个工作线程,模拟登陆和退出,测试180秒.最好测试时间跨越两次checkpoint.
    - pgbench -M simple -n -r -f ./login.sql -f ./logout.sql -c 8 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal

# 优化案例

- 4. 压力测试结果
- 略

# 优化案例

- 5. 瓶颈分析与优化
- 压力测试中查看数据库服务器的iostat -x
- avg-cpu: %user %nice %system %iowait %steal %idle
  - 0.69 0.00 0.25 24.11 0.00 74.95
- Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
  - cciss/c0d0 0.00 6.00 0.00 1.50 0.00 60.00 40.00 0.01 6.67 6.67 1.00
  - cciss/c0d0p1 0.00 6.00 0.00 1.50 0.00 60.00 40.00 0.01 6.67 6.67 1.00
  - ...
  - cciss/c0d2 0.00 638.50 10.00 217.50 160.00 6444.00 29.03 152.58 707.89 4.40 100.10
  - ...
  - dm-1 0.00 0.00 10.00 866.50 160.00 6932.00 8.09 446.26 510.49 1.14 100.10
  - ...
- 平均IO请求等待700多毫秒, PostgreSQL数据文件所处的块设备使用率100%.
- 每秒合并的IO写请求638, 读请求没有.
- 存在严重的写IO性能瓶颈.
- 另外我们还可以观察csvlog的连接记录, 从而得到是否使用短连接.
- 如果没有开启连接和断开连接的审计, 因为PostgreSQL是进程模式的, 那么我们可以通过查看操作系统的每秒创建进程数来反映是否存在短连接的情况.

# 优化案例

- 使用pg\_stat\_statements查看total\_time TOP SQL
- (实际生产场景中, 通过pg\_stat\_statements可以定位到哪些SQL是最耗CPU或者IO的)

# 优化案例

- 6. 优化手段
  - 使用异步提交, 或合并wal flush写请求降低写请求数 .
  - 两种优化手段选一种即可.
  
- 异步提交, 不需要等待wal buffer flush.
- `synchronous_commit = off`
- `wal_writer_delay = 10ms`
- `pg_ctl reload`
- 当数据库异常DOWN 机或数据库所在服务器异常DOWN机时, 最多丢失 $2 * \text{wal\_writer\_delay}$ 时间段内的wal信息.
- 即数据库恢复时最坏可能只能恢复到DOWN机发生的前 $2 * \text{wal\_writer\_delay}$ 毫秒, 但是可以保证数据一致性.
  
- 合并wal flush写请求. 9.3开始对高并发的短事务wal合并非常有效. 以前的版本没有什么效果.
- `commit_delay = 10ms`
- `commit_siblings = 5`
- `pg_ctl reload`

# 优化案例

- 调整后重新进行压力测试
- pgbench -M simple -n -r -f ./login.sql -f ./logout.sql -c 8 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal
- 性能提升百分比
- 优化后结果分析
- 对比每条SQL的耗时, 是否达到预期.

# 优化案例

- [优化阶段2]
- 瓶颈分析与优化
- 通过pg\_stat\_statements查看调用次数较多的SQL, 分析执行计划.
- 对于读SQL, 把相关的对象尽量的缓存到内存.
- 对于写SQL, 把相关的对象尽量的缓存到内存, 并使用更好的IOPS的块设备(即使用不同的表空间来区分存储).
- 同时还需要考虑到大表的年龄, 年龄老化后需要freeze, 会带来大量的读写, 以及wal的写操作. 这些自动或手动的维护性操作要和业务峰值错开.

# 优化案例

- 优化手段
- 首先使用pgfincore降低TOP SQL的读物理IO请求数.
- pgfincore的相关文章可参考如下, 利用posix\_fadvise接口改变文件访问策略. 可以使文件持久化到内存.
  - 《use posix\_fadvise pre-cache frequency data》
  - <http://blog.163.com/digoal@126/blog/static/163877040201062944945126/>
  - 《a powerful upgrade from pgfincore 1.0》
  - <http://blog.163.com/digoal@126/blog/static/1638770402011630102117658/>
  - 《TOAST table with pgfincore》
  - <http://blog.163.com/digoal@126/blog/static/16387704020120524144140/>
- -- 将TOP SQL涉及的对象相对应的活跃数据载入os cache , 前提是内存足够大.
- 例如：
  - select pgfadvise\_loader('user\_info', segment, true, false, databit) from pgfincore('user\_info',true);
  - 以及用到的索引 , toast 对象等.
- 如果本地内存不够的话, 可以考虑使用外部缓存, 例如redis, memcache.
- 如果想减少缓存和数据库之间的交互, 甚至可以使用PostgreSQL的插件pgmemcache. 已经将memcache的API封装到PostgreSQL的函数中.
  - <http://blog.163.com/digoal@126/blog/static/163877040201210172341257/>

# 优化案例

- -- 如果内存足够大的话, 也可以将整个对象加载到内存, 例如
- digoal=> select reltoastrelid from pg\_class where relname='user\_info';
- reltoastrelid
- -----
- 16424
- (1 row)
- digoal=> select relname from pg\_class where oid=16424;
- relname
- -----
- pg\_toast\_16421
- (1 row)
  
- digoal=> \c digoal postgres
- seYou are now connected to database "digoal" as user "postgres".
- digoal=# select \* from pgfadvise\_willneed('pg\_toast.pg\_toast\_16421');
- | relpath                                      | os_page_size | rel_os_pages | os_pages_free |
|----------------------------------------------|--------------|--------------|---------------|
| pg_tblspc/16385/PG_9.1_201105231/16386/16424 | 4096         | 0            | 243865        |
- (1 row)

# 优化案例

- digoal=# select \* from pgfadvise\_willneed('digoal.user\_info');
- relpath | os\_page\_size | rel\_os\_pages | os\_pages\_free
- +-----+-----+-----+
- pg\_tblspc/16385/PG\_9.1\_201105231/16386/16421 | 4096 | 262144 | 243834
- pg\_tblspc/16385/PG\_9.1\_201105231/16386/16421.1 | 4096 | 262144 | 243834
- pg\_tblspc/16385/PG\_9.1\_201105231/16386/16421.2 | 4096 | 244944 | 243834
- (3 rows)
  
- digoal=# select \* from pgfadvise\_willneed('digoal.user\_session');
- relpath | os\_page\_size | rel\_os\_pages | os\_pages\_free
- +-----+-----+-----+
- pg\_tblspc/16385/PG\_9.1\_201105231/16386/16431 | 4096 | 262144 | 243834
- pg\_tblspc/16385/PG\_9.1\_201105231/16386/16431.1 | 4096 | 33640 | 243834
- (2 rows)

# 优化案例

- digoal=# select reltoastrelid from pg\_class where relname='user\_session';  
■ reltoastrelid  
■ -----  
■ 0  
■ (1 row)
  
- digoal=# select \* from pgfadvise\_willneed('digoal.pk\_user\_session');  
■ relpath | os\_page\_size | rel\_os\_pages | os\_pages\_free  
■ -----+-----+-----+-----  
■ pg\_tblspc/16385/PG\_9.1\_201105231/16386/16438 | 4096 | 109680 | 243865  
■ (1 row)
  
- digoal=# select \* from pgfadvise\_willneed('digoal.pk\_user\_info');  
■ relpath | os\_page\_size | rel\_os\_pages | os\_pages\_free  
■ -----+-----+-----+-----  
■ pg\_tblspc/16385/PG\_9.1\_201105231/16386/16436 | 4096 | 109680 | 235567  
■ (1 row)

# 优化案例

- 调整后重新进行压力测试
- pgbench -M simple -n -r -f ./login.sql -f ./logout.sql -c 8 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal
- 性能提升百分比
- 优化后结果分析
- 对比每条SQL的耗时, 是否达到预期.

# 优化案例

- [优化阶段3]
- 瓶颈分析与优化
- 分析程序代码或数据库csvlog查看是否存在使用简单调用(非绑定变量)的情形.
- 本场景客户端连接使用simple协议, 存在一定的可优化空间.
- 修改协议为extended, 以及prepared, 查看性能提升多少.
  
- 调整后重新进行压力测试
- pgbench -M extended -n -r -f ./login.sql -f ./logout.sql -c 8 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal
- pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 8 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal

# 优化案例

- [优化阶段4]
- 瓶颈分析与优化
- 程序与数据库目前使用的是SQL交互, 使用数据库函数可以减少客户端和数据库的交互次数, 性能还会有一定提升.
  
- 调整后重新进行压力测试
- 1. 登陆脚本
- cat login.sql
- \setrandom userid 1 2000000
- SELECT f\_user\_login(:userid);
- 2. 退出脚本
- cat logout.sql
- \setrandom userid 1 2000000
- SELECT f\_user\_logout(:userid);
  
- pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 8 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal

# 优化案例

- [优化阶段5]
- 瓶颈分析与优化
- 根据经验, 数据库所在服务器的物理CPU核数与活动连接数的比例为2时, 可以发挥CPU的最大性能, 因此如果服务器为8核的话, 压力测试使用16个连接是能发挥最大效能的.
- 调整后重新进行压力测试
- `pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 16 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal`
- 使用更大的连接数(以增加活跃连接)不会再有性能提升.

# 优化案例

- [优化阶段6]
- 瓶颈分析与优化
- IO分布的调整, 适用于物理块设备足够多的硬件环境.
- 将xlog, pgdata, 活跃表, 活跃索引的物理块设备隔开, 减少IO争抢.
- 通过在不同的块设备上创建表空间, 将对象分布存放到不同的表空间里面.
- 块设备遇到IO瓶颈时, 可以考虑使用SSD, 或高端存储.
  
- 分配原则 :
- 读活跃索引和读活跃表分开存放, 在内存足够的情况下, 对IOPS能力可以降低要求.
- 写活跃索引和写活跃表分开存放, 同时要求IOPS能力足够强大.
- 在没有使用异步提交的情况下, xlog要求放在IOPS能力最好的块设备上, 如果开启了异步提交, 可以适当降低IOPS的要求.
- pgdata目录, 在不使用默认表空间存储数据的情况下, 一般对IOPS没有太高的要求.
- stats\_temp\_directory 对应的统计信息临时存放目录, 选择IOPS能力足够强大的块设备存储
- log\_directory和存放数据的块设备隔开存储.
  
- 调整后重新进行压力测试
  
- `pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 16 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal`

# 优化案例

- [优化阶段7]
- 瓶颈分析与优化
- 当活跃表的SIZE达到一定大小(例如2GB)后, 建议分表.
- 例如我们这里把user\_info和user\_session 这两个频繁更新和读取的表分表后进行测试.
- 为了方便测试, 我们直接把表压缩到200万进行测试.
- 测试前同样先加载到os cache.
  
- 调整后重新进行压力测试
  
- `pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 16 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal`

# 优化案例

- [优化阶段8]
- 瓶颈分析与优化
- SQL语句层面的优化, 本案例不涉及可优化项, 仅仅针对实际的生产场景.
- 通过pg\_stat\_statements找到CPU耗时排前的SQL, 各个击破.
- 索引, 改变SQL写法, 改变逻辑, 使用一些插件等.
- 例如
- 中文分词, 创建分词的gin/gist索引加速检索.
- <http://blog.163.com/digoal@126/blog/static/163877040201422410175698/>
- 数组或多值变量的空间索引, gin索引
- 近似度查询
- <http://blog.163.com/digoal@126/blog/static/1638770402013416102141801/>
- 分区优化
- <http://blog.163.com/digoal@126/blog/static/163877040201422293824929/>
- 减少数据库运算开销
- count优化, 随机访问优化, ...
  
- 更多参考
- [http://blog.163.com/digoal@126/blog/#m=0&t=1&c=fks\\_084071080084080064085080095095085080082075083081086071084](http://blog.163.com/digoal@126/blog/#m=0&t=1&c=fks_084071080084080064085080095095085080082075083081086071084)

# 优化案例

- [优化阶段9]
- 瓶颈分析与优化
- 横向扩展1, 日志表和业务表分开到独立的数据库集群.
- 降低CPU争用, 解决CPU瓶颈.
  
- 调整后重新进行压力测试
  
- `pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 16 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal`

# 优化案例

- [优化阶段10]
- 瓶颈分析与优化
- 横向扩展2, 读写分离
- 适用于业务库CPU瓶颈, 使用流复制或第三方插件如(londiste3)将可以进行读写分离的表复制到额外的数据库集群.
- 读负载均衡分发到slave节点, 写还在master节点.
- 注意slave节点的延迟, 流复制带来的延迟最小, londiste3对DML频繁的场景延时比流复制大.
  
- 读写分离的实现 :
- 不推荐使用pgpool-II, pgpool本身比较容易产生瓶颈, 建议使用pgbouncer或者jdbc-HA或者程序内分配多数据源.
- <http://blog.163.com/digoal@126/blog/static/1638770402014413104753331/>
  
- 调整后重新进行压力测试
  
- `pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 16 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal`

# 优化案例

- [优化阶段11]
- 瓶颈分析与优化
- 横向扩展3, shared nothing
  
- 业务库推荐使用plproxy来做shared nothing.  
《A Smart PostgreSQL extension plproxy 2.2 practices》
- <http://blog.163.com/digoal@126/blog/static/163877040201192535630895/>
- <http://blog.163.com/digoal@126/blog/static/1638770402013102242543765/>
- 注意跨库事务的问题, plproxy不支持跨库事务. 需要程序在逻辑层面解决.
  
- 还需要考虑事务一致性备份和还原. 目前plproxy不能简单的实现整个shared nothing集群的一致性备份和还原, pg-xc可以.
  
- 调整后重新进行压力测试
  
- pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 16 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal

# 优化案例

- [优化阶段12]
- 瓶颈分析与优化
- 本优化案例的未来扩展方向
- select能力可以通过数据库流复制扩展, 9.2以后可以级联复制因此基本上可以做到不影响主库性能的情况下无限扩展.
- insert能力可以通过增加logdb服务器扩展, 无限扩展.
- update能力可以通过将表拆分到多个服务器上, 无限扩展.
  
- 调整后重新进行压力测试
  
- `pgbench -M prepared -n -r -f ./login.sql -f ./logout.sql -c 16 -j 8 -T 180 -h 172.16.3.33 -p 5432 -U digoal digoal`



# Thanks

- Q&A