

PostgreSQL 数据库开发规范

PostgreSQL 数据库开发规范

背景

PostgreSQL的功能非常强大，但是要把PostgreSQL用好，开发人员是非常关键的。

下面将针对PostgreSQL数据库原理与特性，输出一份开发规范，希望可以减少大家在使用PostgreSQL数据库过程中遇到的困惑。

目标是将PostgreSQL的功能、性能发挥好，她好我也好。

PostgreSQL 使用规范

命名规范

【强制】库名、表名限制命名长度，建议表名及字段名字符总长度小于等于63。

【强制】对象名（表名、列名、函数名、视图名、序列名、等对象名称）规范，对象名务必只使用小写字母，下划线，数字。不要以pg开头，不要以数字开头，不要使用保留字。

保留字参考

<https://www.postgresql.org/docs/9.5/static/sql-keywords-appendix.html>

【强制】query中的别名不要使用“小写字母，下划线，数字”以外的字符，例如中文。

【推荐】主键索引应以pk_开头，唯一索引要以uk_开头，普通索引要以idx_打头。

【推荐】临时表以tmp_开头，子表以规则结尾，例如按年分区的主表如果为tbl，则子表为tbl_2016, tbl_2017, ...。

【推荐】库名最好以部门名字开头 + 功能，如xxx_yyy, xxx_zzz，便于辨识，...。

【推荐】库名最好与应用名称一致，或便于辨识。

【推荐】不建议使用public schema(不同业务共享的对象可以使用public schema)，应该为每个应用分配对应的schema，schema_name最好与user name一致。

【推荐】comment不要使用中文，因为编码可能不一样，如果存进去和读取时的编码不一致，导致可读性不强。pg_dump时也必须与comment时的编码一致，否则可能乱码。

设计规范

【强制】多表中的相同列，必须保证列名一致，数据类型一致。

【强制】btree索引字段不建议超过2000字节，如果有超过2000字节的字段需要建索引，建议使用函数索引（例如哈希值索引），或者使用分词索引。

【强制】使用外键时，如果你使用的PG版本没有自动建立fk的索引，则必须要对foreign key手工建立索引，否则可能影响references列的更新或删除性能。

例如

```
postgres=# create table tbl(id int primary key, info text);
CREATE TABLE
postgres=# create table tb11(id int references tbl(id), info text);
CREATE TABLE
postgres=# \d tbl
Table "public.tbl"
 Column | Type    | Modifiers
-----+---------+-----
 id   | integer | not null
 info | text    |
Indexes:
 "tbl_pkey" PRIMARY KEY, btree (id)
Referenced by:
 TABLE "tb11" CONSTRAINT "tb11_id_fkey" FOREIGN KEY (id) REFERENCES tbl(id)

postgres=# \d tb11
Table "public.tb11"
 Column | Type    | Modifiers
-----+---------+-----
 id   | integer |
 info | text    |
Foreign-key constraints:
 "tb11_id_fkey" FOREIGN KEY (id) REFERENCES tbl(id)

postgres=# \di
      List of relations
 Schema | Name | Type | Owner | Table
-----+-----+-----+-----+-----
```

```
public | tbl_pkey | index | postgres | tbl
(1 row)

postgres=# create index idx_tbll_id on tbll(id);
CREATE INDEX
```

【强制】使用外键时，一定要设置fk的action，例如cascade, set null, set default。

例如

```
postgres=# create table tbll2(id int references tbll(id) on delete cascade on update cascade, info text);
CREATE TABLE
postgres=# create index idx_tbll2_id on tbll2(id);
CREATE INDEX
postgres=# insert into tbll values (1,'test');
INSERT 0 1
postgres=# insert into tbll2 values (1,'test');
INSERT 0 1
postgres=# update tbll set id=2;
UPDATE 1
postgres=# select * from tbll2;
 id | info
----+-----
 2 | test
(1 row)
```

【强制】对于频繁更新的表，建议建表时指定表的fillfactor=85，每页预留15%的空间给HOT更新使用。

```
postgres=# create table test123(id int, info text) with(fillfactor=85);
CREATE TABLE
```

**【强制】表结构中字段定义的数据类型与应用程序中的定义保持一致，表之间字段校对规则一致，避免报错或无法使用索引的情况发生。
说明：**

(1).比如A表user_id字段数据类型定义为varchar，但是SQL语句查询为 where user_id=1234；

【推荐】如何保证分区表的主键序列全局唯一。

使用多个序列，每个序列的步调不一样，或者每个序列的范围不一样即可。

例如

```
postgres=# create sequence seq_tab1 increment by 10000 start with 1;
CREATE SEQUENCE
postgres=# create sequence seq_tab2 increment by 10000 start with 2;
CREATE SEQUENCE
postgres=# create sequence seq_tab3 increment by 10000 start with 3;
CREATE SEQUENCE
postgres=# create table tab1 (id int primary key default nextval('seq_tab1') check(mod(id,10000)=1), info text);
CREATE TABLE
postgres=# create table tab2 (id int primary key default nextval('seq_tab2') check(mod(id,10000)=2), info text);
CREATE TABLE
postgres=# create table tab3 (id int primary key default nextval('seq_tab3') check(mod(id,10000)=3), info text);
CREATE TABLE

postgres=# insert into tab1 (info) select generate_series(1,10);
INSERT 0 10
postgres=# insert into tab2 (info) select generate_series(1,10);
INSERT 0 10
postgres=# insert into tab3 (info) select generate_series(1,10);
INSERT 0 10
postgres=# select * from tab1;
 id | info
----+-----
 1  | 1
10001 | 2
20001 | 3
30001 | 4
40001 | 5
50001 | 6
60001 | 7
70001 | 8
80001 | 9
90001 | 10
(10 rows)

postgres=# select * from tab2;
 id | info
----+-----
 2  | 1
10002 | 2
20002 | 3
30002 | 4
40002 | 5
50002 | 6
60002 | 7
70002 | 8
80002 | 9
90002 | 10
(10 rows)

postgres=# select * from tab3;
 id | info
----+-----
 3  | 1
10003 | 2
20003 | 3
30003 | 4
40003 | 5
50003 | 6
60003 | 7
70003 | 8
80003 | 9
90003 | 10
(10 rows)
```

或

```

postgres=# create sequence seq_tb1 increment by 1 minvalue 1 maxvalue 100000000 start with 1 no cycle ;
CREATE SEQUENCE
postgres=# create sequence seq_tb2 increment by 1 minvalue 100000001 maxvalue 200000000 start with 100000001 no cycle ;
CREATE SEQUENCE
postgres=# create sequence seq_tb3 increment by 1 minvalue 200000001 maxvalue 300000000 start with 200000001 no cycle ;
CREATE SEQUENCE

postgres=# create table tb1(id int primary key default nextval('seq_tb1') check(id >=1 and id<=100000000), info text);
CREATE TABLE
postgres=# create table tb2(id int primary key default nextval('seq_tb2') check(id >=100000001 and id<=200000000), info text);
CREATE TABLE
postgres=# create table tb3(id int primary key default nextval('seq_tb3') check(id >=200000001 and id<=300000000), info text);
CREATE TABLE
postgres=# insert into tb1 (info) select * from generate_series(1,10);
INSERT 0 10
postgres=# insert into tb2 (info) select * from generate_series(1,10);
INSERT 0 10
postgres=# insert into tb3 (info) select * from generate_series(1,10);
INSERT 0 10
postgres=# select * from tb1;
 id | info
----+---
 1 | 1
 2 | 2
 3 | 3
 4 | 4
 5 | 5
 6 | 6
 7 | 7
 8 | 8
 9 | 9
10 | 10
(10 rows)

postgres=# select * from tb2;
 id | info
----+---
100000001 | 1
100000002 | 2
100000003 | 3
100000004 | 4
100000005 | 5
100000006 | 6
100000007 | 7
100000008 | 8
100000009 | 9
100000010 | 10
(10 rows)

postgres=# select * from tb3;
 id | info
----+---
200000001 | 1
200000002 | 2
200000003 | 3
200000004 | 4
200000005 | 5
200000006 | 6
200000007 | 7
200000008 | 8
200000009 | 9
200000010 | 10
(10 rows)

```

【推荐】建议有定期历史数据删除需求的业务，表按时间分区，删除时不要使用DELETE操作，而是DROP或者TRUNCATE对应的表。

【推荐】为了全球化的需求，所有的字符存储与表示，均以UTF-8编码，那么字符计数方法注意：

例如

计算字符串长度

```

postgres=# select length('阿里巴巴');
length
-----
        4
(1 row)

```

计算字节数

```

postgres=# select octet_length('阿里巴巴');
octet_length
-----
       12
(1 row)

```

其他长度相关接口

| Schema | Name | Result data type | Argument data types | Type |
|------------|--------------------|------------------|---------------------|--------|
| pg_catalog | array_length | integer | anyarray, integer | normal |
| pg_catalog | bit_length | integer | bit | normal |
| pg_catalog | bit_length | integer | bytea | normal |
| pg_catalog | bit_length | integer | text | normal |
| pg_catalog | char_length | integer | character | normal |
| pg_catalog | char_length | integer | text | normal |
| pg_catalog | character_length | integer | character | normal |
| pg_catalog | character_length | integer | text | normal |
| pg_catalog | json_array_length | integer | json | normal |
| pg_catalog | jsonb_array_length | integer | jsonb | normal |
| pg_catalog | length | integer | bit | normal |
| pg_catalog | length | integer | bytea | normal |
| pg_catalog | length | integer | bytea, name | normal |
| pg_catalog | length | integer | character | normal |
| pg_catalog | length | double precision | lseg | normal |

| | | | | |
|------------|--------------|------------------|-----------|--------|
| pg_catalog | length | double precision | path | normal |
| pg_catalog | length | integer | text | normal |
| pg_catalog | length | integer | tsvector | normal |
| pg_catalog | lseg_length | double precision | lseg | normal |
| pg_catalog | octet_length | integer | bit | normal |
| pg_catalog | octet_length | integer | bytea | normal |
| pg_catalog | octet_length | integer | character | normal |
| pg_catalog | octet_length | integer | text | normal |

【推荐】对于值与堆表的存储顺序线性相关的数据，如果通常的查询为范围查询，建议使用BRIN索引。例如流式数据，时间字段或自增字段，可以使用BRIN索引，减少索引的大小，加快数据插入速度。

例如

```
create index idx on tbl using brin(id);
```

【推荐】设计时应尽可能选择合适的数据类型，能用数字的坚决不用字符串，能用树类型的，坚决不用字符串。使用好的数据类型，可以使用数据库的索引，操作符，函数，提高数据的查询效率。

PostgreSQL支持的数据类型如下

精确的数字类型

浮点

货币

字符串

字符

字节流

日期

时间

布尔

枚举

几何

网络地址

比特流

文本

UUID

XML

JSON

数组

复合类型

范围类型

对象

行号

大对象

ltree 树结构类型

cube 多维类型

earth 地球类型

hstore KV类型

pg_trgm 相似类型

PostGIS (点、线段、面、路径、经纬度、raster、拓扑、.....)

【推荐】应该尽量避免全表扫描(除了大数据量扫描的数据分析)，PostgreSQL支持几乎所有数据类型的索引。

索引接口包括

btree

hash

gin

gist

sp-gist

brin

rum (扩展接口)

bloom (扩展接口)

【推荐】对于网络复杂并且RT要求很高的场景，如果业务逻辑冗长，应该尽量减少数据库和程序之间的交互次数，尽量使用数据库存储过程(如plpgsql)，或内置的函数。

PostgreSQL内置的plpgsql函数语言功能非常强大，可以处理复杂的业务逻辑。

PostgreSQL内置了非常多的函数，包括分析函数，聚合函数，窗口函数，普通类型函数，复杂类型函数，数学函数，几何函数，等。

【推荐】应用应该尽量避免使用数据库触发器，这会使得数据处理逻辑复杂，不便于调试。

【推荐】如果应用经常要访问较大结果集的数据（例如100条），可能造成大量的离散扫描。

建议想办法将数据聚合成1条，例如经常要按ID访问这个ID的数据，建议可以定期按ID聚合这些数据，查询时返回的记录数越少越快。如果无法聚合，建议使用IO较好的磁盘。

【推荐】流式的实时统计，为了防止并行事务导致的统计空洞，建议业务层按分表并行插入，单一分表串行插入。

例如

```
table1, table2, ...table100;
```

每个线程负责一张表的插入，统计时可以按时间或者表的自增ID进行统计。

```
select xxx from table1 where id>=上一次统计的截至ID group byyyy;
```

【推荐】范围查询，应该尽量使用范围类型，以及GIST索引，提高范围检索的查询性能。

例如

使用范围类型存储IP地址段，使用包含的GIST索引检索，性能比两个字段的between and提升20多倍。

```

CREATE TABLE ip_address_pool_3 (
    id serial8 primary key ,
    start_ip inet NOT NULL ,
    end_ip inet NOT NULL ,
    province varchar(128) NOT NULL ,
    city varchar(128) NOT NULL ,
    region_name varchar(128) NOT NULL ,
    company_name varchar(128) NOT NULL ,
    ip_decimal_segment int8range
) ;

CREATE INDEX ip_address_pool_3_range ON ip_address_pool_3 USING gist (ip_decimal_segment);

select province, ip_decimal_segment from ip_address_pool_3 where ip_decimal_segment @> :ip::int8;

```

【推荐】未使用的大对象，一定要同时删除数据部分，否则大对象数据会一直存在数据库中，与内存泄露类似。
vacuumlo可以用来清理未被引用的大对象数据。

【推荐】对于固定条件的查询，可以使用部分索引，减少索引的大小，同时提升查询效率。

例如

```

select * from tbl where id=1 and col=?; -- 其中id=1为固定的条件
create index idx on tbl (col) where id=1;

```

【推荐】对于经常使用表达式作为查询条件的语句，可以使用表达式或函数索引加速查询。

例如

```

select * from tbl where exp(xxx);
create index idx on tbl (exp);

```

【推荐】如果需要调试较为复杂的逻辑时，不建议写成函数进行调试，可以使用plpgsql的online code.

例如

```
do language plpgsql
```

```
$$
```

```

declare
begin
    -- logical code
end;

```

```
$$
```

【推荐】当业务有中文分词的查询需求时，建议使用PostgreSQL的分词插件zhparser或jieba，用户还可以通过接口自定义词组。
建议在分词字段使用gin索引，提升分词匹配的性能。

【推荐】当用户有规则表达式查询，或者文本近似度查询的需求时，建议对字段使用trgm的gin索引，提升近似度匹配或规则表达式匹配的查询效率，同时覆盖了前后模糊的查询需求。如果没有创建trgm gin索引，则不推荐使用前后模糊查询例如like %xxxx%。

【推荐】当用户有prefix或者suffix的模糊查询需求时，可以使用索引，或反转索引达到提速的需求。

如

```

select * from tbl where col ~ '^abc'; -- 前缀查询
select * from tbl where reverse(col) ~ '^def'; -- 后缀查询使用反转函数索引

```

【推荐】用户应该对频繁访问的大表（通常指超过8GB的表，或者超过1000万记录的表）进行分区，从而提升查询的效率、更新的效率、备份与恢复的效率、建索引的效率等等，（PostgreSQL支持多核创建索引后，可以适当将这个限制放大）。

【推荐】用户在设计表结构时，建议规划好，避免经常需要添加字段，或者修改字段类型或长度。某些操作可能触发表的重写，例如加字段并设置默认值，修改字段的类型。

如果用户确实不好规划结构，建议使用jsonb数据类型存储用户数据。

QUERY 规范

【强制】不要使用count(列名)或count(常量)来替代count()，count()就是SQL92定义的标准统计行数的语法，跟数据库无关，跟NULL和非NULL无关。

说明：count(*)会统计NULL值（真实行数），而count(列名)不会统计。

【强制】count(多列列名)时，多列列名必须使用括号，例如count((col1,col2,col3))。注意多列的count，即使所有列都为NULL，该行也被计数，所以效果与count(*)一致。

例如

```

postgres=# create table t123(c1 int,c2 int,c3 int);
CREATE TABLE
postgres=# insert into t123 values (null,null,null), (null,null,null), (1,null,null), (2,null,null), (null,1,null), (null,2,null);
INSERT 0 6
postgres=# select count((c1,c2)) from t123;
      count
-----
       6
(1 row)
postgres=# select count((c1)) from t123;
      count
-----
       2
(1 row)

```

【强制】count(distinct col) 计算该列的非NULL不重复数量，NULL不被计数。

例如

```

postgres=# select count(distinct (c1)) from t123;
      count
-----

```

```
2
(1 row)
```

【强制】 count(distinct (col1,col2,...)) 计算多列的唯一值时，NULL会被计数，同时NULL与NULL会被认为是想同的。

例如

```
postgres=# select count(distinct (c1,c2)) from t123;
count
-----
5
(1 row)

postgres=# select count(distinct (c1,c2,c3)) from t123;
count
-----
5
(1 row)
```

【强制】 count(col)对 "是NULL的col列" 返回为0，而sum(col)则为NULL。

例如

```
postgres=# select count(c1),sum(c1) from t123 where c1 is null;
count | sum
-----+-----
0    |
```

因此注意sum(col)的NPE问题，如果你的期望是当SUM返回NULL时要得到0，可以这样实现

```
SELECT coalesce( SUM(g), 0, SUM(g) ) FROM table;
```

【强制】 NULL是UNKNOWN的意思，也就是不知道是什么。因此NULL与任意值的逻辑判断都返回NULL。

例如

NULL<>NULL 的返回结果是NULL，不是false。

NULL=NULL的返回结果也是NULL，不是true。

NULL值与任何值的比较都为NULL，即NULL<>1，返回的是NULL，而不是true。

【强制】 除非是ETL程序，否则应该尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

【强制】 任何地方都不要使用 `select from t`，用具体的字段列表代替，不要返回用不到的任何字段。另外表结构发生变化也容易出现问题。

管理规范

【强制】 数据订正时，删除和修改记录时，要先select，避免出现误删除，确认无误才能提交执行。

【强制】 DDL操作必须设置锁等待，可以防止堵塞所有其他与该DDL锁对象相关的QUERY。

例如

```
begin;
set local lock_timeout = '10s';
-- DDL query;
end;
```

【强制】 用户可以使用explain analyze查看实际的执行计划，但是如果需要查看的执行计划设计数据的变更，必须在事务中执行explain analyze，然后回滚。

例如

```
begin;
explain analyze query;
rollback;
```

【强制】 如何并行创建索引，不堵塞表的DML，创建索引时加CONCURRENTLY关键字，就可以并行创建，不会堵塞DML操作，否则会堵塞DML操作。

例如

```
create index CONCURRENTLY idx on tbl(id);
```

【强制】 为数据库访问账号设置复杂密码。

说明：密码由小写字母，数字、下划线组成、字母开头，字母或数字结尾，禁止123456，hello123等简单密码。

【强制】 业务系统，开发测试账号，不要使用数据库超级用户。非常危险。

【推荐】 多个业务共用一个PG集群时，建议为每个业务创建一个数据库。如果业务之间有数据交集，或者事务相关的处理，强烈建议在程序层处理数据的交互。

不能在程序中处理时，可以将多个业务合并到一个库，但是使用不同的schema将多个业务的对象分开来。

【推荐】 应该为每个业务分配不同的数据库账号，禁止多个业务共用一个数据库账号。

【推荐】 在发生主备切换后，新的主库在开放给应用程序使用前，建议使用pg_prewarm预热之前的主库shared buffer里的热数据。

【推荐】 快速的装载数据的方法，关闭autovacuum，删除索引，数据导入后，对表进行analyze同时创建索引。

【推荐】 如何加快创建索引的速度，调大maintenance_work_mem，可以提升创建索引的速度，但是需要考虑实际的可用内存。

例如

```
begin;
set local maintenance_work_mem='2GB';
create index idx on tbl(id);
end;
```

【推荐】 如何防止长连接，占用过多的relcache, syscache。

当系统中有很多张表时，元数据会比较庞大，例如1万张表可能有上百MB的元数据，如果一个长连接的会话，访问到了所有的对象，则可能会

长期占用这些syscache和relcache。

建议遇到这种情况时，定期释放长连接，重新建立连接，例如每个小时释放一次长连接。

PS

阿里云的RDS PGSQL版本提供了主动释放syscache和 relcache的接口，不需要断开连接。

【推荐】大批量数据入库的优化，如果有大批量的数据入库，建议使用copy语法，或者 insert into table values (),(),...(); 的方式。提高写入速度。

稳定性与性能规范

【强制】在代码中写分页查询逻辑时，若count为0应直接返回，避免执行后面的分页语句。

【强制】游标使用后要及时关闭。

【强制】两阶段提交的事务，要及时提交或回滚，否则可能导致数据库膨胀。

【强制】不要使用delete 全表，性能很差，请使用truncate代替，（truncate是DDL语句，注意加锁等待超时）。

【强制】应用程序一定要开启autocommit，同时避免应用程序自动begin事务，并且不进行任何操作的情况发生，某些框架可能会有这样的问题。

【强制】高并发的应用场合，务必使用绑定变量(prepared statement)，防止数据库硬解析消耗过多的CPU资源。

【强制】不要使用hash index，目前hash index不写REDO，在备库只有结构，没有数据，并且数据库crash后无法恢复。

同时不建议使用unlogged table，道理同上，但是如果你的数据不需要持久化，则可以考虑使用unlogged table来提升数据的写入和修改性能。

【强制】秒杀场景，一定要使用 advisory_lock先对记录的唯一ID进行锁定，拿到AD锁再去对数据进行更新操作。拿不到锁时，可以尝试重试拿锁。

例如

```
CREATE OR REPLACE FUNCTION public.f(i_id integer)
RETURNS void
LANGUAGE plpgsql
AS $function$
declare
    a_lock boolean := false;
begin
    select pg_try_advisory_xact_lock(i_id) into a_lock;
    拿到锁，更新
    if a lock then
        update t1 set count=count-1 where id=i_id;
    end if;
    exception when others then
        return;
    end;
$function$;

select f(id) from tbl where id=? and count>0;
```

可以再根据实际情况设计，原理如上即可。

函数可以如返回布尔，或者唯一ID，或者数字等。

【强制】在函数中，或程序中，不要使用count(*)判断是否有数据，很慢。建议的方法是limit 1;

例如

```
select 1 from tbl where xxx limit 1;
if found -- 存在
else -- 不存在
```

【强制】对于高并发的应用场景，务必使用程序的连接池，否则性能会很低下。

如果程序没有连接池，建议在应用层和数据库之间架设连接池，例如使用pgbouncer或者pgpool-II作为连接池。

【强制】当业务有近邻查询的需求时，务必对字段建立GIST或SP-GIST索引，加速近邻查询的需求。

例如

```
create index idx on tbl using gist(col);
select * from tbl order by col <-> '(0,100)';
```

【强制】避免频繁创建和删除临时表，以减少系统表资源的消耗，因为创建临时表会产生元数据，频繁创建，元数据可能会出现碎片。

【强制】必须选择合适的事务隔离级别，不要使用越级的隔离级别，例如READ COMMITTED可以满足时，就不要使用repeatable read和serializable隔离级别。

【推荐】高峰期对大表添加包含默认值的字段，会导致表的rewrite，建议只添加不包含默认值的字段，业务逻辑层面后期处理默认值。

【推荐】分页评估，不需要精确分页数时，请使用快速评估分页数的方法。

<https://yq.aliyun.com/articles/39682>

例如

```
CREATE OR REPLACE FUNCTION countit(text)
RETURNS float4
LANGUAGE plpgsql AS
$$
DECLARE
    v_plan json;
BEGIN
```

```

EXECUTE 'EXPLAIN (FORMAT JSON) ' ||$1
    INTO v_plan;
    RETURN v_plan #>> '[0,Plan,"Plan Rows"]';
END;

$$
;
postgres=# create table t1234(id int, info text);
CREATE TABLE
postgres=# insert into t1234 select generate_series(1,1000000), 'test';
INSERT 0 1000000
postgres=# analyze t1234;
ANALYZE
postgres=# select count(*) from t1234 where id<1000';
count
-----
      954
(1 row)
postgres=# select count(*) from t1234 where id between 1 and 1000 or (id between 100000 and 101000)';
count
-----
     1931
(1 row)

```

【推荐】分页优化，建议通过游标返回分页结果，避免越后面的页返回越慢的情况。

例如

```

postgres=# declare curl cursor for select * from sbtest1 where id between 100 and 1000000 order by id;
DECLARE CURSOR
Time: 0.422 ms

```

获取数据

```

postgres=# fetch 100 from curl;
...

```

如果要前滚页，加SCROLL打开游标

```

declare curl SCROLL cursor for select * from sbtest1 where id between 100 and 1000000 order by id;

```

【推荐】可以预估SQL执行时间的操作，建议设置语句级别的超时，可以防止雪崩，也可以防止长时间持锁。

例如设置事务中执行的每条SQL超时时间为10秒

```

begin;
set local statement_timeout = '10s';
-- query;
end;

```

【推荐】TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少，但是TRUNCATE是DDL，锁粒度很大，故不建议在开发代码中使用DDL语句，除非加了lock_timeout锁超时的会话参数或事务参数。

【推荐】PostgreSQL支持DDL事务，支持回滚DDL，建议将DDL封装在事务中执行，必要时可以回滚，但是需要注意事务的长度，避免长时间堵塞DDL对象的读操作。

【推荐】如果用户需要在插入数据和，删除数据前，或者修改数据后马上拿到插入或被删除或修改后的数据，建议使用insert into .. returning ..; delete .. returning ..或update .. returning ..语法。减少数据库交互次数。

例如

```

postgres=# create table tb14(id serial, info text);
CREATE TABLE
postgres=# insert into tb14 (info) values ('test') returning *;
 id | info
----+-----
  1 | test
(1 row)

```

INSERT 0 1

```

postgres=# update tb14 set info='abc' returning *;
 id | info
----+-----
  1 | abc
(1 row)

```

UPDATE 1

```

postgres=# delete from tb14 returning *;
 id | info
----+-----
  1 | abc
(1 row)

```

DELETE 1

【推荐】自增字段建议使用序列，序列分为2字节，4字节，8字节几种(serial2,serial4,serial8)。按实际情况选择。禁止使用触发器产生序列值。

例如

```

postgres=# create table tb14(id serial, info text);
CREATE TABLE

```

【推荐】如果对全表的很多字段有任意字段匹配的查询需求，建议使用行级别全文索引，或行转数组的数组级别索引。

例如

```

select * from t where phonenum='digoal' or info ~ 'digoal' or cl='digoal' or ....;

```

更正为

```

postgres=# create or replace function f1(text) returns tsvector as
$$

```

```

select to_tsvector($1);

$$
language sql immutable strict;
CREATE FUNCTION

postgres=# alter function record_out(record) immutable;
ALTER FUNCTION
postgres=# alter function textin(cstring) immutable;
ALTER FUNCTION
postgres=# create index idx_t_1 on t using gin (f1('jiebacfg'::regconfig,t::text)) ;
CREATE INDEX

postgres=# select * from t where f1('jiebacfg'::regconfig,t::text) @@ to_tsquery('digoal & post') ;
phonenum | info | c1 | c2 | c3 | c4
-----+-----+-----+-----+
(0 rows)
postgres=# select * from t where f1('jiebacfg'::regconfig,t::text) @@ to_tsquery('digoal & china') ;
phonenum | info | c1 | c2 | c3 | c4
-----+-----+-----+-----+
13888888888 | i am digoal, a postgresqler | 123 | china | 中华人民共和国, 阿里巴巴, 阿 | 2016-04-19 11:15:55.208658
(1 row)

postgres=# select * from t where f1('jiebacfg'::regconfig,t::text) @@ to_tsquery('digoal & 阿里巴巴') ;
phonenum | info | c1 | c2 | c3 | c4
-----+-----+-----+-----+
13888888888 | i am digoal, a postgresqler | 123 | china | 中华人民共和国, 阿里巴巴, 阿 | 2016-04-19 11:15:55.208658
(1 row)

postgres=# explain select * from t where f1('jiebacfg'::regconfig,t::text) @@ to_tsquery('digoal & 阿里巴巴') ;
QUERY PLAN
-----
Seq Scan on t  (cost=0.00..1.52 rows=1 width=140)
  Filter: (to_tsvector('jiebacfg'::regconfig, (t.*)::text) @@ to_tsquery('digoal & 阿里巴巴'::text))
(2 rows)

```

【推荐】中文分词的token mapping一定要设置，否则对应的token没有词典进行处理。

例如

```
ALTER TEXT SEARCH CONFIGURATION testzhcfg ADD MAPPING FOR a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z WITH simple;
```

zhparser分词插件的其他配置

```

zhparser.punctuation_ignore = f
zhparser.seg_with_duality = f
zhparser.dict_in_memory = f
zhparser.multi_short = f
zhparser.multi_duality = f
zhparser.multi_zmain = f
zhparser.multi_zall = f

```

参考

<https://yq.aliyun.com/articles/7730>

<http://www.xunsearch.com/scws/docs.php#libscws>

【推荐】树形查询应该使用递归查询，尽量减少数据库的交互或JOIN。

例如

```

CREATE TABLE TBL_TEST
(
ID      numeric,
NAME    text,
PID     numeric          DEFAULT 0
);
INSERT INTO TBL_TEST(ID,NAME,PID) VALUES('1','10','0');
INSERT INTO TBL_TEST(ID,NAME,PID) VALUES('2','11','1');
INSERT INTO TBL_TEST(ID,NAME,PID) VALUES('3','20','0');
INSERT INTO TBL_TEST(ID,NAME,PID) VALUES('4','12','1');
INSERT INTO TBL_TEST(ID,NAME,PID) VALUES('5','121','2');

```

从Root往树末梢递归

```

with recursive t_result as (
  select * from tbl_test where id=1
  union all
  select t2.* from t_result t1 join tbl_test t2 on t1.id=t2.pid
)
select * from t_result;

id | name | pid
---+---+---
 1 | 10  | 0
 2 | 11  | 1
 4 | 12  | 1
 5 | 121 | 2
(4 rows)

```

从末梢往树ROOT递归

```

with recursive t_result as (
  select * from tbl_test where id=5
  union all
  select t2.* from t_result t1 join tbl_test t2 on t1.pid=t2.id
)
select * from t_result;

id | name | pid
---+---+---

```

| | | |
|---|-----|---|
| 5 | 121 | 2 |
| 2 | 11 | 1 |
| 1 | 10 | 0 |

(3 rows)

树形结构的注意事项

- 一定要能跳出循环，即循环子句查不到结果为止。
- 树形结构如果有多个值，则会出现查到的结果比实际的多的情况，这个业务上是需要保证不出现重复的。

【推荐】应尽量避免长事务，长事务可能造成垃圾膨胀。

【推荐】如果业务有多个维度的分析需求，应该尽量使用PostgreSQL的多维分析语法，减少数据的重复扫描。

支持的多维分析语法包括

GROUPING SETS, CUBE, ROLLUP

例如

假设有4个业务字段，一个时间字段。

```
postgres=# create table tab5(c1 int, c2 int, c3 int, c4 int, crt_time timestamp);
CREATE TABLE
```

生成一批测试数据

```
postgres=# insert into tab5 select
trunc(100*random()),
trunc(1000*random()),
trunc(10000*random()),
trunc(100000*random()),
clock_timestamp() + (trunc(10000*random())||' hour')::interval
from generate_series(1,1000000);
INSERT 0 1000000
```

```
postgres=# select * from tab5 limit 10;
c1 | c2 | c3 | c4 | crt_time
----+---+---+---+---
72 | 46 | 3479 | 20075 | 2017-02-02 14:56:36.854218
98 | 979 | 4491 | 83012 | 2017-06-13 08:56:36.854416
54 | 758 | 5838 | 45956 | 2016-09-18 02:56:36.854427
3 | 67 | 5148 | 74754 | 2017-01-01 01:56:36.854431
42 | 650 | 7681 | 36495 | 2017-06-20 15:56:36.854435
4 | 472 | 6454 | 19554 | 2016-06-18 19:56:36.854438
82 | 922 | 902 | 17435 | 2016-07-21 14:56:36.854441
68 | 156 | 1028 | 13275 | 2017-07-16 10:56:36.854444
0 | 674 | 7446 | 59386 | 2016-07-26 09:56:36.854447
0 | 629 | 2022 | 52285 | 2016-11-04 13:56:36.854445
(10 rows)
```

创建一个统计结果表，其中bitmap表示统计的字段组合，用位置符0,1表示是否统计了该维度

```
create table stat_tab5 (c1 int, c2 int, c3 int, c4 int, time1 text, time2 text, time3 text, time4 text, cnt int8, bitmap text);
```

生成业务字段任意维度组合+4组时间任选一组的组合统计

PS (如果业务字段有空的情况，建议统计时用coalesce转一下，确保不会统计到空的情况)

```
insert into stat_tab5
select c1,c2,c3,c4,t1,t2,t3,t4,cnt,
' '
case when c1 is null then 0 else 1 end ||
case when c2 is null then 0 else 1 end ||
case when c3 is null then 0 else 1 end ||
case when c4 is null then 0 else 1 end ||
case when t1 is null then 0 else 1 end ||
case when t2 is null then 0 else 1 end ||
case when t3 is null then 0 else 1 end ||
case when t4 is null then 0 else 1 end
from (
select c1,c2,c3,c4,
to_char(crt_time, 'yyyy') t1,
to_char(crt_time, 'yyyy-mm') t2,
to_char(crt_time, 'yyyy-mm-dd') t3,
to_char(crt_time, 'yyyy-mm-dd hh24') t4,
count(*) cnt
from tab5
group by
cube(c1,c2,c3,c4),
grouping sets(to_char(crt_time, 'yyyy'), to_char(crt_time, 'yyyy-mm'), to_char(crt_time, 'yyyy-mm-dd'), to_char(crt_time, 'yyyy-mm-dd hh24'))
)
t;
```

INSERT 0 49570486

Time: 172373.714 ms

在bitmap上创建索引方便取数据

```
create index idx_stat_tab5_bitmap on stat_tab5 (bitmap);
```

用户勾选几个维度，取出数据

c1,c3,c4,t3 = bitmap(10110010)

```
postgres=# select c1,c3,c4,time3,cnt from stat_tab5 where bitmap='10110010' limit 10;
c1 | c3 | c4 | time3 | cnt
----+---+---+---+---
41 | 0 | 30748 | 2016-06-04 | 1
69 | 0 | 87786 | 2016-06-04 | 1
70 | 0 | 38805 | 2016-06-04 | 1
79 | 0 | 65892 | 2016-06-08 | 1
51 | 0 | 13615 | 2016-06-11 | 1
47 | 0 | 42196 | 2016-06-28 | 1
45 | 0 | 54736 | 2016-07-01 | 1
50 | 0 | 21605 | 2016-07-02 | 1
46 | 0 | 40888 | 2016-07-16 | 1
```

```

41 | 0 | 90258 | 2016-07-17 | 1
(10 rows)
Time: 0.528 ms

postgres=# select * from stat_tab5 where bitmap='00001000' limit 10;
 c1 | c2 | c3 | c4 | timel | time2 | time3 | time4 | cnt | bitmap
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
      |   |   | 2016 |       |       |       |       | 514580 | 00001000
      |   |   | 2017 |       |       |       |       | 485420 | 00001000
(2 rows)
Time: 0.542 ms

```

【推荐】对于有UV查询需求的场景（例如count(distinct xx) where time between xx and xx），如果要求非常快的响应速度，但是对精确度要求不高时，建议可以使用PostgreSQL的估值数据类型HLL。

例如

```
create table access_date (acc_date date unique, userids hll);
```

```

insert into access_date select current_date, hll_add_agg(hll_hash_integer(user_id)) from generate_series(1,10000) t(user_id);

select *, total_users=coalesce(lag(total_users,1) over (order by rn),0) AS new_users
FROM
(
  SELECT acc_date, row_number() over date as rn, #hll_union_agg(userids) OVER date as total_users
  FROM access_date
  WINDOW date AS (ORDER BY acc_date ASC ROWS UNBOUNDED PRECEDING)
) t;

```

【推荐】PostgreSQL 的insert on conflict语法如下

```

INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
  { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
  [ ON CONFLICT [ conflict_target ] conflict_action ]

```

where conflict_target can be one of:

```
( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] ) [ WHERE index_predicate ]
ON CONSTRAINT constraint_name
```

and conflict_action is one of:

```

DO NOTHING
DO UPDATE SET { column_name = { expression | DEFAULT } |
    ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) |
    ( column_name [, ...] ) = ( sub-SELECT )
} [, ...]
[ WHERE condition ]

```

例如

```
postgres=# insert into tbl values (1,'info') on conflict on constraint tbl_pkey do update set info=excluded.info;
INSERT 0 1
```

【推荐】如果用户经常需要访问一张大表的某些数据，为了提升效率可以使用索引，但是如果这个数据还需要被用于更复杂的与其他表的JOIN操作，则可以使用物化视图来提升性能。

同时物化视图还可以被用于OLAP场景，例如统计后的数据可以固化到物化视图中，以便快速的检索。

例如

```
CREATE MATERIALIZED VIEW mv_tbl as select xx,xx,xx from tbl where xxx with data;
```

增量刷新物化视图

```
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_tbl with data;
```

【推荐】不建议对宽表频繁的更新，原因是PG目前的引擎是多版本的，更新后会产生新的版本，如果对宽表的某几个少量的字段频繁更新，其实是存在写放大的。

建议将此类宽表的不更新或更新不频繁的列与频繁更新的列拆分成两张表，通过PK进行关联。

查询是通过PK关联查询出结果即可。

【推荐】使用窗口查询减少数据库和应用的交互次数。

例如

有一个这样的表，记录如下：

```

id | company | product
---+-----+-----
1 | c1      | p1
1 | c1      | p2
1 | b1      | p2
1 | c2      | p2
1 | c1      | p1
2 | c3      | p3

```

需要找出某个产品，这个产品只有一个公司生产。

```
select distinct product from (select min(company) over(partition by product) m1,max(company) over(partition by product) m2, product from tbl) t where
```

又如，根据指定窗口，查询当前行与以窗口为范围取其avg,max,min,sum,count,offset,rank,dist等，同时输出当前行。例如与第一名的差距，与前一名的差距，与全国第一名的差距，与全班第一名的差距，同时还输出当前记录的详情。

【推荐】应该尽量在业务层面避免死锁的产生，例如一个用户的读操作，尽量在一个线程内处理，而不要跨线程（即跨数据库会话处理）。

【推荐】OLTP系统不要频繁的使用聚合操作，聚合操作消耗较大的CPU与IO资源。例如实时的COUNT操作，如果并发很高，可能导致CPU资源撑爆。

对于实时性要求不高的场景，可以使用定期操作COUNT，并将COUNT数据缓存在缓存系统中的方式。

【推荐】数据去重的方法，当没有UK或PK时，如果数据出现了重复，有什么好的方法去重。或者某个列没有加唯一约束，但是业务层没有保证唯一，如何去重？

行级别去重

```
delete from tbl where ctid not in (select min(ctid) from tbl group by tbl::text);
```

带PK的列级别去重

```
delete from tbl where pk in (select pk from (select pk, row_number() over(partition by col order by pk) rn from tbl) t where t.rn>1);
```

不带PK的列级别去重(以业务逻辑为准, 可以选择其他的条件删除)

```
delete from tbl where ctid not in (select min(ctid) from tbl group by col);
```

【推荐】快速读取随机记录的方法**利用索引列进行优化的方法。****方法 1. 随机取出n条记录,以下取出5条随机记录**

```
digoal=> select * from tbl_user
digoal=> where id in
digoal=>     (select floor(random() * (max_id - min_id))::int
digoal=>         + min_id
digoal=>     from generate_series(1, 5),
digoal=>         (select max(id) as max_id,
digoal=>             min(id) as min_id
digoal=>         from tbl_user) s1
digoal=>     )
digoal=> limit 5;
      id | firstname | lastname | corp   | age
-----+-----+-----+-----+-----+
 965638 | zhou      | digoal   | sky-mobi | 27
 193491 | zhou      | digoal   | sky-mobi | 27
 294286 | zhou      | digoal   | sky-mobi | 27
 726263 | zhou      | digoal   | sky-mobi | 27
 470713 | zhou      | digoal   | sky-mobi | 27
(5 rows)
Time: 0.670 ms
```

方法 2. 取出N条连续的随机记录(此处用到函数)

```
digoal=> create or replace function f_get_random (i_range int) returns setof record as $BODY$
digoal$> declare
digoal$> v_result record;
digoal$> v_max_id int;
digoal$> v_min_id int;
digoal$> v_random numeric;
digoal$> begin
digoal$> select random() into v_random;
digoal$> select max(id),min(id) into v_max_id,v_min_id from tbl_user;
digoal$> for v_result in select * from tbl_user where id between (v_min_id+(v_random*(v_max_id-v_min_id))::int) and (v_min_id+(v_random*(v_max_id-v_min_id))::int)
digoal$> loop
digoal$> return next v_result;
digoal$> end loop;
digoal$> return;
digoal$> end
digoal$> $BODY$ language plpgsql;
CREATE FUNCTION
```

以下举例取出10条连续的随机记录

```
digoal=> select * from f_get_random(9) as (id bigint,firstname varchar(32),lastname varchar(32),corp varchar(32),age smallint);
      id | firstname | lastname | corp   | age
-----+-----+-----+-----+-----+
 694686 | zhou      | digoal   | sky-mobi | 27
 694687 | zhou      | digoal   | sky-mobi | 27
 694688 | zhou      | digoal   | sky-mobi | 27
 694689 | zhou      | digoal   | sky-mobi | 27
 694690 | zhou      | digoal   | sky-mobi | 27
 694691 | zhou      | digoal   | sky-mobi | 27
 694692 | zhou      | digoal   | sky-mobi | 27
 694693 | zhou      | digoal   | sky-mobi | 27
 694694 | zhou      | digoal   | sky-mobi | 27
 694695 | zhou      | digoal   | sky-mobi | 27
(10 rows)
Time: 0.418 ms
```

【推荐】线上表结构的变更包括添加字段, 索引操作在业务低高峰期进行。**【推荐】OLTP系统, 在高峰期或高并发期间 拒绝长SQL, 大事务, 大批量。****说明:**

- (1). 长SQL占用大量的数据库时间和资源, 占用连接, 可能影响正常业务运行。
- (2). 大事务, 或长事务, 可能导致长时间持锁, 与其他事务产生锁冲突。
- (3). 大批量, 大批量在并发事务中增加锁等待的几率。

【推荐】查询条件要和索引匹配, 例如查询条件是表达式时, 索引也要是表达式索引, 查询条件为列时, 索引就是列索引。**【推荐】如何判断两个值是不是不一样 (并且将NULL视为一样的值), 使用col1 IS DISTINCT FROM col2****例如**

```
postgres=# select null is distinct from null;
?column?
-----
f
(1 row)

postgres=# select null is distinct from 1;
?column?
-----
t
(1 row)
```

另外还有IS NOT DISTINCT FROM的用法。**【推荐】如果在UDF或online code逻辑中有数据的处理需求时, 建议使用游标进行处理。****例如**

```

do language plpgsql
$$

declare
    cur refcursor;
    rec record;
begin
    open cur for select * from tbl where id>1;
    loop
        fetch cur into rec;
        if found then
            raise notice '%', rec;
            update tbl set info='ab' where current of cur;
            -- other query
        else
            close cur;
            exit;
        end if;
    end loop;
end;

$$
;

```

【推荐】应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

如果业务确实有这种需求的查询，可以有几种优化方法

1. partial index

这个是最有效的方法，可以使用到索引扫描，如果有其他条件，也可以在其他条件的索引上建立partial index.

```
create index idx1 on tbl (id) where cond1 <> xx;
```

2. 分区表

使用分区表，如果有!=的查询条件，PostgreSQL会根据分区约束，避免扫描不需要扫描的表。

3. 约束

```
set constraint_exclusion=on;
exec query;
```

在查询列上有约束的情况下，如果!=或<>与约束违背，则可以提前返回查询，不会扫描表。

【推荐】对于经常变更，或者新增，删除记录的表，应该尽量加快这种表的统计信息采样频率，获得较实时的采样，输出较好的执行计划。
例如

当垃圾达到表的千分之五时，自动触发垃圾回收。

当数据变化达到表的百分之一时，自动触发统计信息的采集。

当执行垃圾回收时，不等待，当IOPS较好时可以这么设置。

```
postgres=# create table t2l(id int, info text) with (
autovacuum_enabled=on, toast.autovacuum_enabled=on,
autovacuum_vacuum_scale_factor=0.005, toast.autovacuum_vacuum_scale_factor=0.005,
autovacuum_analyze_scale_factor=0.01, autovacuum_vacuum_cost_delay=0,
toast.autovacuum_vacuum_cost_delay=0);
CREATE TABLE
```

【推荐】PostgreSQL 对or的查询条件，会使用bitmap or进行索引的过滤，所以不需要改SQL语句，可以直接使用。

例如

以下查询都可以走索引

```
select * from tbl where col1 =1 or col1=2 or col2=1 or ... ;
select * from tbl where col1 in (1,2);
```

【推荐】很多时候用 exists 代替 in 是一个好的选择:

```
select num from a where num in (select num from b);
```

用下面的语句替换:

```
select num from a where exists(select 1 from b where num=a.num)
```

【推荐】尽量使用数组变量来代替临时表。如果临时表有非常庞大的数据时，才考虑使用临时表。

【推荐】对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

使用explain可以查看执行计划，如果发现执行计划不优，可以通过索引或者调整QUERY的写法解决。

例如

```
begin;
explain (verbose,costs,timing,buffers,analyze) query;
rollback;
.....
```

【推荐】PG优化器可以动态调整JOIN的顺序，获取更好的执行计划，但是如何强制优化器的显示JOIN顺序呢？

首先PG根据join_collapse_limit的设置，当需要关联的表的个数超过这个设置时，超出的JOIN部分不会继续动态调整JOIN顺序。

另外需要注意，如果开启了GEQO，当JOIN的表(含隐式JOIN,以及子查询) (full outer join 只算1)数量超过了geqo_threshold设置的值，则会触发遗传算法，可能无法得到最佳的JOIN顺序。

要让优化器固定JOIN顺序，首先必须使用显示的JOIN，其次将join_collapse_limit设置为1，显示的JOIN顺序将被固定，固定JOIN顺序可以减少优化器的编排时间，降低频繁执行多表JOIN带来的优化阶段的CPU开销。

显示的JOIN例子

```
t1 join t2 on (xxx)
```

隐式的JOIN例子

```
t1, t2 where xxx
```

例如

```
begin;
set local join_collapse_limit=1;
set local geqo=off;
```

```

postgres=# create table t1(id int, info text);
CREATE TABLE
postgres=# create table t2(id int, info text);
CREATE TABLE
postgres=# create table t3(id int, info text);
CREATE TABLE
postgres=# create table t4(id int, info text);
CREATE TABLE
postgres=# create table t5(id int, info text);
CREATE TABLE
postgres=# create table t6(id int, info text);
CREATE TABLE
postgres=# create table t7(id int, info text);
CREATE TABLE
JOIN顺序固定为如下

```

```
postgres=# explain select * from t2 join t1 using (id) join t3 using (id) join t4 using (id) join t7 using (id) join t6 using (id) join t5 using (id)
QUERY PLAN
```

```

Merge Join  (cost=617.21..1482900.86 rows=83256006 width=228)
  Merge Cond: (t5.id = t2.id)
    -> Sort  (cost=88.17..91.35 rows=1270 width=36)
      Sort Key: t5.id
        -> Seq Scan on t5  (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize  (cost=529.03..266744.20 rows=13111182 width=216)
      -> Merge Join  (cost=529.03..233966.24 rows=13111182 width=216)
        Merge Cond: (t6.id = t2.id)
          -> Sort  (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t6.id
              -> Seq Scan on t6  (cost=0.00..22.70 rows=1270 width=36)
        -> Materialize  (cost=440.86..42365.87 rows=2064753 width=180)
          -> Merge Join  (cost=440.86..37203.99 rows=2064753 width=180)
            Merge Cond: (t7.id = t2.id)
              -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                Sort Key: t7.id
                  -> Seq Scan on t7  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=352.69..6951.07 rows=325158 width=144)
              -> Merge Join  (cost=352.69..6138.17 rows=325158 width=144)
                Merge Cond: (t4.id = t2.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t4.id
                      -> Seq Scan on t4  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=264.52..1294.30 rows=51206 width=108)
              -> Merge Join  (cost=264.52..1166.28 rows=51206 width=108)
                Merge Cond: (t3.id = t2.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t3.id
                      -> Seq Scan on t3  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=176.34..323.83 rows=8064 width=72)
              -> Merge Join  (cost=176.34..303.67 rows=8064 width=72)
                Merge Cond: (t2.id = t1.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t2.id
                      -> Seq Scan on t2  (cost=0.00..22.70 rows=1270 width=36)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t1.id
                -> Seq Scan on t1  (cost=0.00..22.70 rows=1270 width=36)

(38 rows)
end;

```

或者设置会话级别的join_collapse_limit=1;

```

set join_collapse_limit=1;
set geooff;
postgres=# explain select * from t2 join t1 using (id) join t3 using (id) join t4 using (id) join t7 using (id) join t6 using (id) join t5 using (id)
QUERY PLAN
```

```

Merge Join  (cost=617.21..1482900.86 rows=83256006 width=228)
  Merge Cond: (t5.id = t2.id)
    -> Sort  (cost=88.17..91.35 rows=1270 width=36)
      Sort Key: t5.id
        -> Seq Scan on t5  (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize  (cost=529.03..266744.20 rows=13111182 width=216)
      -> Merge Join  (cost=529.03..233966.24 rows=13111182 width=216)
        Merge Cond: (t6.id = t2.id)
          -> Sort  (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t6.id
              -> Seq Scan on t6  (cost=0.00..22.70 rows=1270 width=36)
        -> Materialize  (cost=440.86..42365.87 rows=2064753 width=180)
          -> Merge Join  (cost=440.86..37203.99 rows=2064753 width=180)
            Merge Cond: (t7.id = t2.id)
              -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                Sort Key: t7.id
                  -> Seq Scan on t7  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=352.69..6951.07 rows=325158 width=144)
              -> Merge Join  (cost=352.69..6138.17 rows=325158 width=144)
                Merge Cond: (t4.id = t2.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t4.id
                      -> Seq Scan on t4  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=264.52..1294.30 rows=51206 width=108)
              -> Merge Join  (cost=264.52..1166.28 rows=51206 width=108)
                Merge Cond: (t3.id = t2.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t3.id
                      -> Seq Scan on t3  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=176.34..323.83 rows=8064 width=72)
              -> Merge Join  (cost=176.34..303.67 rows=8064 width=72)
                Merge Cond: (t2.id = t1.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t2.id
                      -> Seq Scan on t2  (cost=0.00..22.70 rows=1270 width=36)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t1.id
                -> Seq Scan on t1  (cost=0.00..22.70 rows=1270 width=36)

(38 rows)
end;

```

```
-> Seq Scan on t1 (cost=0.00..22.70 rows=1270 width=36)
```

(38 rows)

如何通过优化器获得最好的JOIN顺序?

通常可以将join_collapse_limit设置为一个很大的值, 然后查看执行计划, 根据JOIN顺序修改SQL语句。

例如

```
postgres=# set join_collapse_limit=100;
SET
postgres=# set geqo=off;
SET
postgres=# explain select * from t2 join t1 using (id) join t3 using (id) join t4 using (id) join t7 using (id) join t6 using (id) join t5 using (id)
QUERY PLAN
```

```
Merge Join (cost=617.21..1255551.94 rows=83256006 width=228)
  Merge Cond: (t2.id = t4.id)
    -> Merge Join (cost=264.52..1166.28 rows=51206 width=108)
      Merge Cond: (t3.id = t2.id)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t3.id
            -> Seq Scan on t3 (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize (cost=176.34..323.83 rows=8064 width=72)
      -> Merge Join (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t2.id = t1.id)
          -> Sort (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t2.id
              -> Seq Scan on t2 (cost=0.00..22.70 rows=1270 width=36)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t1.id
            -> Seq Scan on t1 (cost=0.00..22.70 rows=1270 width=36)
  -> Materialize (cost=352.69..6317.49 rows=325158 width=144)
    -> Merge Join (cost=352.69..5504.60 rows=325158 width=144)
      Merge Cond: (t4.id = t6.id)
        -> Merge Join (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t4.id = t7.id)
          -> Sort (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t4.id
              -> Seq Scan on t4 (cost=0.00..22.70 rows=1270 width=36)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t7.id
            -> Seq Scan on t7 (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize (cost=176.34..323.83 rows=8064 width=72)
      -> Merge Join (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t6.id = t5.id)
          -> Sort (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t6.id
              -> Seq Scan on t6 (cost=0.00..22.70 rows=1270 width=36)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t5.id
            -> Seq Scan on t5 (cost=0.00..22.70 rows=1270 width=36)
```

(36 rows)

修改SQL, 固定为最佳JOIN顺序。

```
postgres=# set join_collapse_limit=1;
SET
postgres=# set geqo=off;
SET

explain select * from ((t4 join t7 using (id)) join (t6 join t5 using (id)) using (id)) join (t3 join (t2 join t1 using (id)) using (id)) using (id)
postgres=# explain select * from ((t4 join t7 using (id)) join (t6 join t5 using (id)) using (id)) join (t3 join (t2 join t1 using (id)) using (id))
QUERY PLAN
```

```
Merge Join (cost=617.21..1255482.81 rows=83245594 width=228)
  Merge Cond: (t2.id = t4.id)
    -> Merge Join (cost=264.52..1166.28 rows=51206 width=108)
      Merge Cond: (t3.id = t2.id)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t3.id
            -> Seq Scan on t3 (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize (cost=176.34..323.83 rows=8064 width=72)
      -> Merge Join (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t2.id = t1.id)
          -> Sort (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t2.id
              -> Seq Scan on t2 (cost=0.00..22.70 rows=1270 width=36)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t1.id
            -> Seq Scan on t1 (cost=0.00..22.70 rows=1270 width=36)
  -> Materialize (cost=352.69..6317.45 rows=325140 width=144)
    -> Merge Join (cost=352.69..5504.60 rows=325140 width=144)
      Merge Cond: (t4.id = t6.id)
        -> Merge Join (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t4.id = t7.id)
          -> Sort (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t4.id
              -> Seq Scan on t4 (cost=0.00..22.70 rows=1270 width=36)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t7.id
            -> Seq Scan on t7 (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize (cost=176.34..323.83 rows=8064 width=72)
      -> Merge Join (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t6.id = t5.id)
          -> Sort (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t6.id
              -> Seq Scan on t6 (cost=0.00..22.70 rows=1270 width=36)
        -> Sort (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t5.id
            -> Seq Scan on t5 (cost=0.00..22.70 rows=1270 width=36)
```

(36 rows)

【推荐】PG优化器可以提升子查询，转换为JOIN，以获得更好的执行计划，但是如何强制优化器使用子查询呢？
仅当子查询的数量小于from_collapse_limit时，这些子查询才会被提升为JOIN子句，超过的部分不会被提升为JOIN子句。

同样需要考虑GEQO的设置，如果你不想使用遗传算法，可以设置geqo=off；

要固定FROM子查询，两个设置即可from_collapse_limit=1, geqo=off；

例如

```
postgres=# set from_collapse_limit=1; -- 这一不会提升子查询了，但是JOIN顺序还是可能变化的，需要通过join_collapse_limit=1来设置
SET
postgres=# set geqo=off;
SET
postgres=# explain select * from t1 join t2 using (id) join (select * from t4) t4 using (id) join (select * from t6) t6 using (id) join (select * fr
QUERY PLAN
```

```
Merge Join  (cost=529.03..233966.24 rows=13111182 width=196)
  Merge Cond: (t3.id = t1.id)
    -> Sort  (cost=88.17..91.35 rows=1270 width=36)
      Sort Key: t3.id
        -> Seq Scan on t3  (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize  (cost=440.86..42365.87 rows=2064753 width=180)
      -> Merge Join  (cost=440.86..37203.99 rows=2064753 width=180)
        Merge Cond: (t5.id = t1.id)
          -> Sort  (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t5.id
              -> Seq Scan on t5  (cost=0.00..22.70 rows=1270 width=36)
        -> Materialize  (cost=352.69..6951.07 rows=325158 width=144)
          -> Merge Join  (cost=352.69..6138.17 rows=325158 width=144)
            Merge Cond: (t6.id = t1.id)
              -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                Sort Key: t6.id
                  -> Seq Scan on t6  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=264.52..1294.30 rows=51206 width=108)
              -> Merge Join  (cost=264.52..1166.28 rows=51206 width=108)
                Merge Cond: (t4.id = t1.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t4.id
                      -> Seq Scan on t4  (cost=0.00..22.70 rows=1270 width=36)
            -> Materialize  (cost=176.34..323.83 rows=8064 width=72)
              -> Merge Join  (cost=176.34..303.67 rows=8064 width=72)
                Merge Cond: (t1.id = t2.id)
                  -> Sort  (cost=88.17..91.35 rows=1270 width=36)
                    Sort Key: t1.id
                      -> Seq Scan on t1  (cost=0.00..22.70 rows=1270 width=36)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t2.id
                -> Seq Scan on t2  (cost=0.00..22.70 rows=1270 width=36)

(32 rows)
```

如何通过优化器判断FROM子句是否需要提升以得到好的执行计划？

通过优化器的指导，调整SQL即可

例如

```
postgres=# set join_collapse_limit=100;
SET
postgres=# set from_collapse_limit=100;
SET
postgres=# set geqo=off;
SET
postgres=# explain select * from t1 join t2 using (id) join (select * from t4) t4 using (id) join (select * from t6) t6 using (id) join (select * fr
QUERY PLAN
```

```
Merge Join  (cost=529.03..199114.66 rows=13111182 width=196)
  Merge Cond: (t1.id = t6.id)
    -> Merge Join  (cost=264.52..1166.28 rows=51206 width=108)
      Merge Cond: (t4.id = t1.id)
        -> Sort  (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t4.id
            -> Seq Scan on t4  (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize  (cost=176.34..323.83 rows=8064 width=72)
      -> Merge Join  (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t1.id = t2.id)
          -> Sort  (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t1.id
              -> Seq Scan on t1  (cost=0.00..22.70 rows=1270 width=36)
        -> Sort  (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t2.id
            -> Seq Scan on t2  (cost=0.00..22.70 rows=1270 width=36)

  -> Materialize  (cost=264.52..1294.30 rows=51206 width=108)
    -> Merge Join  (cost=264.52..1166.28 rows=51206 width=108)
      Merge Cond: (t3.id = t6.id)
        -> Sort  (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t3.id
            -> Seq Scan on t3  (cost=0.00..22.70 rows=1270 width=36)
    -> Materialize  (cost=176.34..323.83 rows=8064 width=72)
      -> Merge Join  (cost=176.34..303.67 rows=8064 width=72)
        Merge Cond: (t6.id = t5.id)
          -> Sort  (cost=88.17..91.35 rows=1270 width=36)
            Sort Key: t6.id
              -> Seq Scan on t6  (cost=0.00..22.70 rows=1270 width=36)
        -> Sort  (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t5.id
            -> Seq Scan on t5  (cost=0.00..22.70 rows=1270 width=36)

(31 rows)
```

调整SQL如下

```
explain select * from ((select * from t4) t4 join (t1 join t2 using (id)) using (id)) join ((select * from t3) t3 join ((select * from t6) t6 join (
postgres=# set join_collapse_limit=1;
SET
postgres=# set from_collapse_limit=1;
SET
postgres=# set geqo=off;
SET
```

```
postgres=# explain select * from ((select * from t4) t4 join (t1 join t2 using (id)) using (id)) join ((select * from t3) t3 join ((select * from t6
          QUERY PLAN
-----
```

```
Merge Join  (cost=529.03..199114.66 rows=13110272 width=196)
  Merge Cond: (t1.id = t6.id)
    -> Merge Join  (cost=264.52..1166.28 rows=51206 width=108)
      Merge Cond: (t4.id = t1.id)
        -> Sort  (cost=88.17..91.35 rows=1270 width=36)
          Sort Key: t4.id
            -> Seq Scan on t4  (cost=0.00..22.70 rows=1270 width=36)
      Materialize  (cost=176.34..323.83 rows=8064 width=72)
        -> Merge Join  (cost=176.34..303.67 rows=8064 width=72)
          Merge Cond: (t1.id = t2.id)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t1.id
                -> Seq Scan on t1  (cost=0.00..22.70 rows=1270 width=36)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t2.id
                -> Seq Scan on t2  (cost=0.00..22.70 rows=1270 width=36)
      -> Materialize  (cost=264.52..1294.30 rows=51206 width=108)
        -> Merge Join  (cost=264.52..1166.28 rows=51206 width=108)
          Merge Cond: (t3.id = t6.id)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t3.id
                -> Seq Scan on t3  (cost=0.00..22.70 rows=1270 width=36)
      Materialize  (cost=176.34..323.83 rows=8064 width=72)
        -> Merge Join  (cost=176.34..303.67 rows=8064 width=72)
          Merge Cond: (t6.id = t5.id)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t6.id
                -> Seq Scan on t6  (cost=0.00..22.70 rows=1270 width=36)
            -> Sort  (cost=88.17..91.35 rows=1270 width=36)
              Sort Key: t5.id
                -> Seq Scan on t5  (cost=0.00..22.70 rows=1270 width=36)
(31 rows)
```

【推荐】 GIN索引的写优化，因为GIN的索引列通常是多值列，所以一条记录可能影响GIN索引的多个页，为了加快数据插入和更新删除的速度，建议打开fastupdate，同时设置合适的gin_pending_list_limit(单位KB)。

这么做的原理是，当变更GIN索引时，先记录在PENDING列表，而不是立即合并GIN索引。从而提升性能。

例如

```
create index idx_1 on tbl using gin (tsvector) with (fastupdate=on, gin_pending_list_limit=10240)
```

【推荐】 b-tree索引优化，不建议对频繁访问的数据上使用非常离散的数据，例如UUID作为索引，索引页会频繁的分裂，重锁，重IO和CPU开销都比较高。

如何降低频繁更新索引字段的索引页IO，设置fillfactor为一个合适的值，默认90已经适合大部分场景。

【推荐】 BRIN索引优化，根据数据的相关性，以及用户需求的查询的范围，设置合适的pages_per_range=n。

例如用户经常需要按范围查询10万条记录，通过评估，发现10万条记录通常分布在100个数据页中，那么可以设置pages_per_range=100。

评估方法

如何获取平均每个页存了多少条记录。

```
analyze tbl;
select reltuples/relpages from tbl;
```

阿里云RDS PostgreSQL 使用规范

如果你是阿里云RDS PGSQL的用户，推荐你参考一下规范，阿里云RDS PGSQL提供了很多有趣的特性帮助用户解决社区版本不能解决的问题。

【推荐】 冷热数据分离

当数据库非常庞大（例如超过2TB）时，建议使用阿里云PGSQL的OSS_EXT外部表插件，将冷数据存入OSS。

通过建立OSS外部表，实现对OSS数据的透明访问。

参考

https://help.aliyun.com/document_detail/35457.html

【推荐】 对RT要求高的业务，请使用SLB链路 或 PROXY透传模式连接数据库。

【推荐】 RDS的地域选择与应用保持一致。

说明：比如应用上海环境，数据库选择上海region，避免应用和数据库出现跨区域访问。

【推荐】 为RDS报警设置多位接收人，并设置合适的报警阀值。

【推荐】 为RDS设置合适的白名单，加固数据访问的安全性。

【推荐】 尽量禁止数据库被公网访问，如果真的要访问，一定要设置白名单。

【推荐】 如果数据用户的查询中，使用索引的列，数据倾斜较为严重，即某些值很多记录，某些值很少记录，则查询某些列时可能不走索引，而查询另外一些列可能走索引。

特别是这种情况，可能造成绑定变量执行计划倾斜的问题，如果用户使用了绑定变量，同时出现了执行计划的倾斜，建议使用pg_hint_plan绑定执行计划，避免倾斜。

例如

```
test=> create extension pg_hint_plan;
CREATE EXTENSION
```

```
test=> alter role all set session_preload_libraries='pg_hint_plan';
ALTER ROLE
```

```
test=> create table test(id int primary key, info text);
CREATE TABLE
```

```
test=> insert into test select generate_series(1,100000);
INSERT 0 100000
test=> explain select * from test where id=1;
QUERY PLAN
-----
Index Scan using test_pkey on test  (cost=0.29..8.31 rows=1 width=36)
  Index Cond: (id = 1)
(2 rows)

test=> /*+ seqscan(test) */ explain select * from test where id=1;
QUERY PLAN
-----
Seq Scan on test  (cost=0.00..1124.11 rows=272 width=36)
  Filter: (id = 1)
(2 rows)

test=> /*+ bitmapscan(test) */ explain select * from test where id=1;
QUERY PLAN
-----
Bitmap Heap Scan on test  (cost=4.30..8.31 rows=1 width=36)
  Recheck Cond: (id = 1)
    -> Bitmap Index Scan on test_pkey  (cost=0.00..4.30 rows=1 width=0)
      Index Cond: (id = 1)
(4 rows)
```