云数据库 RDS 版

最佳实践

最佳实践

MySQL

对比ECS自建数据库与RDS性能时的注意事项

适用场景

测试ECS自建数据库库与云数据库RDS的性能差异。

背景信息

很多用户都会对比ECS自建数据库与云数据库RDS的性能，但在测试两种数据库的性能时，不仅需要保证二者处于相同的环境下，如相同的网络、配置等，还需要综合考虑数据库的运行机制，否则测试结果会出现偏差且不具有说服力。

云数据库RDS作为一个公共的关系型数据库，高可用和高安全是其首要优势，其次才是高性能，因为没人会使用既不稳定又不安全的服务。RDS的优势主要体现在如下几点：

RDS提供了主备双节点的实例，双节点可以在同一地域的不同可用区，MySQL实例的双节点还可以在不同地域，当主实例出现故障时可快速切换到备实例，保障了RDS的稳定性。

RDS在数据的存取上加入了中间层，所有请求都会经过中间层，而且有SQL注入的请求都会被中间层拦截掉。在底层数据写入上，RDS采用了最高安全级别的写入，保证在主机异常掉电的情况下数据不会出现丢失。以此来保障数据库的高安全性。

RDS源码团队持续对MySQL进行源码优化，在标准的基准测试中性能和稳定性上都是高于社区版本的。

性能测试注意事项

当您进行ECS自建库与云数据库RDS的性能对比时，请注意如下事项。

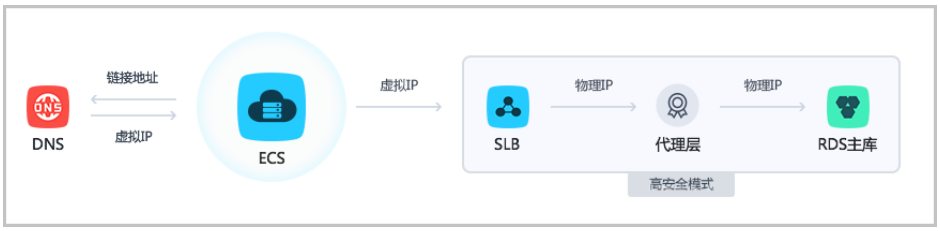
网络差异

可用区

RDS有单可用区和多可用区之分，单可用区是的数据库主备都在同一个机房，多可用区的数据库主备可能在不同机房，所以在测试RDS的过程中需要保证ECS和RDS在同一个可用区。

网络链路

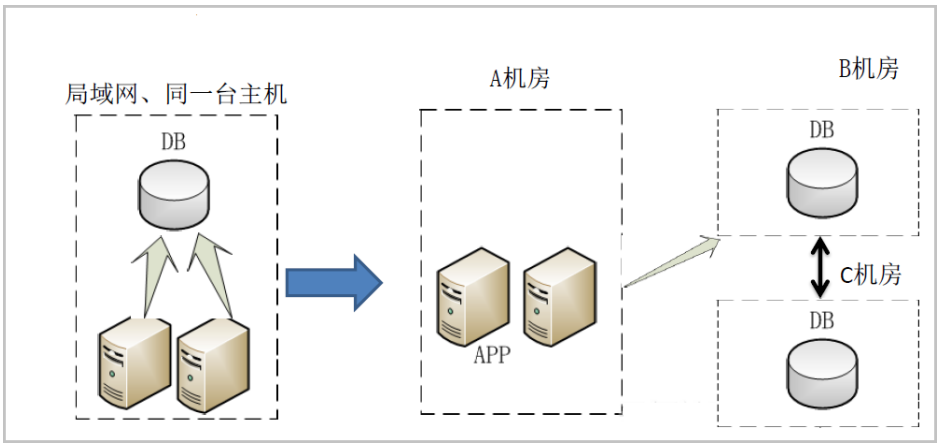
从ECS到RDS的网络链路中有多个环节，包括了ECS—>DNS—>SLB—>Proxy—>DB（如下图所示），而ECS自建数据库是ECS—>ECS，RDS的网络链路比ECS自建数据库多出了3个链路环节。



案例分析

场景：某电商要将系统迁移上云，在测试过程中发现，虽然自建库和RDS的应用代码和数据库配置完全相同，但云数据库性能较低。

原因：网络延迟放大，如下图所示。



配置差异

规格配置

RDS的规格配置主要包括了内存和CPU。在测试RDS的过程中，需要保证ECS的CPU核数与RDS的CPU核数一致。

参数配置

安全配置：RDS为了保证数据的安全性，在参数配置上采用了事务提交和binlog刷写的最高保护级别。

innodb_flush_log_at_trx_commit：该参数指定了InnoDB在事务提交后的日志写入频率。当取值为1时，每次事务提交时log buffer会被写入到日志文件并刷写到磁盘。1是默认值，这是最安全的配置，但由于每次事务都需要进行磁盘I/O，所以有一定的性能损耗。

sync_binlog：该参数是MySQL的binlog日志同步到磁盘的频率。MySQL在binlog每写入sync_binlog次后，刷写到磁盘。该参数值为1时最安全，在每个语句或事务后同步一次binary log，即使在崩溃时也最多丢失一个语句或事务的日志，但因此也最慢。

性能配置：RDS开放了除规格配置以外的参数供用户进行配置，绝大部分的参数都已经由官方团队优化过，用户不需要过多调整线上的参数就可以把数据库比较好的运行起来。但这些参数只是适合大多数的应用场景，针对一些特殊场景，需要您进行特殊定制。

tmp_table_size：该参数用于决定内部内存临时表的最大值，每个线程都要分配（实际起限制作用的是tmp_table_size和max_heap_table_size的最小值），如果内存临时表超出了限制，MySQL就会自动把它转化为基于磁盘的MyISAM表，优化查询语句的时候，要避免使用临时表，如果实在避免不了的话，要保证这些临时表是存在内存中的。

现象：如果复杂的SQL语句中包含了group by/distinct等不能通过索引进行优化而使用了临时表，则会导致SQL执行时间加长。

建议：如果应用中有很多group by/distinct等语句，同时数据库有足够的内存，可以增大tmp_table_size(max_heap_table_size)的值，以此来提升查询性能。

query_cache_size：该参数用于控制MySQL query cache的内存大小。如果MySQL开启query cache，在执行每一个query的时候会先锁住query cache，然后判断是否存在query cache中，如果存在直接返回结果，如果不存在，则再进行引擎查询等操作。同时，insert、update和delete这样的操作都会将query cahce失效掉，这种失效还包括结构或者索引的任何变化，cache失效的维护代价较高，会给MySQL带来较大的压力。所以，当我们的数据库不是特别频繁更新时，query cache是比较好用的；但如果写入非常频繁，并集中在某几张表上的时候，query cache lock的锁机制会造成很频繁的锁冲突，对于这一张表的写和读会互相等待query cache lock解锁，导致select的查询效率下降。

现象：数据库中有大量的连接状态为checking query cache for query、Waiting for query cache lock、storing result in query cache。

建议：RDS默认关闭query cache功能，如果您的实例打开了query cache，当出现上述情况后可以关闭query cache。当然有些情况也可以打开query cache，以解决数据库性能问题。

案例分析

场景：某客户正在将本地的业务系统迁移上云，在RDS上运行时间明显要比线下自建数据库运行时间要慢1倍。

原因：自建数据库与RDS的参数配置不同，如下所示：

用户本地参数配置：

join_buffer_size = 128M

read_rnd_buffer_size = 128M

tmp_table_size = 128M

RDS参数配置：

join_buffer_size = 1M

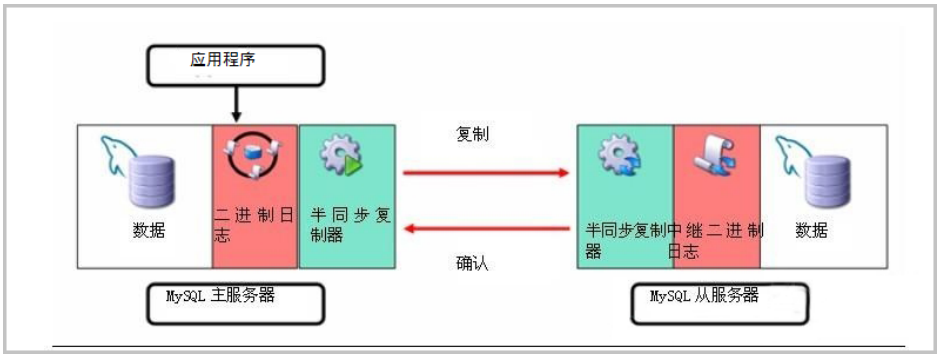
read_buffer_size = 1M

tmp_table_size =256K

架构差异

RDS采用了主从复制的高可用模式，同时打开了半同步复制，半同步复制是MySQL异步复制的改进，当主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待从库接收到并写到relay log中才返回给客户端。相对于异步复制，半同步复制提高了数据的安全性，同时它也造成了一定程度的延迟，这个延迟最少是一个TCP/IP往返的时间。所以半同步复制增加了事务的响应时间，详情如下图所示。

说明：SQL Server高可用模式采用mirror，同样存在上述的问题。



为应用选择和创建最佳索引，加速数据读取

在工作之中，由于SQL问题导致的数据库故障层出不穷，索引问题是SQL问题中出现频率最高的，常见的索引问题包括：无索引，隐式转换，索引创建不合理。

当数据库中出现访问表的SQL没创建索引导致全表扫描，如果表的数据量很大扫描大量的数据，执行效率过慢，占用数据库连接，连接数堆积很快达到数据库的最大连接数设置，新的应用请求将会被拒绝导致故障发生。

隐式转换是指SQL查询条件中的传入值与对应字段的数据定义不一致导致索引无法使用。常见隐式转换如字段的表结构定义为字符类型，但SQL传入值为数字；或者是字段定义collation为区分大小写，在多表关联的场景下，其表的关联字段大小写敏感定义各不相同。隐式转换会导致索引无法使用，进而出现上述慢SQL堆积数据库连接数跑满的情况。

索引使用策略及优化

创建索引

- 在经常查询而不经常增删改操作的字段加索引。
- order by与group by后应直接使用字段，而且字段应该是索引字段。
- 一个表上的索引不应该超过6个。
- 索引字段的长度固定，且长度较短。
- 索引字段重复不能过多。
- 在过滤性高的字段上加索引。

使用索引注意事项

- 使用like关键字时，前置%会导致索引失效。
- 使用null值会被自动从索引中排除，索引一般不会建立在有空值的列上。
- 使用or关键字时，or左右字段如果存在一个没有索引，有索引字段也会失效。
- 使用!=操作符时，将放弃使用索引。因为范围不确定，使用索引效率不高，会被引擎自动改为全表扫描。
- 不要在索引字段进行运算。
- 在使用复合索引时，最左前缀原则，查询时必须使用索引的第一个字段，否则索引失效；并且应尽量让字段顺序与索引顺序一致。
- 避免隐式转换，定义的数据类型与传入的数据类型保持一致。

无索引案例

无索引案例一

查看表结构。

```
mysql> show create table customers;


CREATE TABLE `customers` (
`cust_id` int(11) NOT NULL AUTO_INCREMENT,
`cust_name` char(50) NOT NULL,
`cust_address` char(50) DEFAULT NULL,
`cust_city` char(50) DEFAULT NULL,
`cust_state` char(5) DEFAULT NULL,
`cust_zip` char(10) DEFAULT NULL,
`cust_country` char(50) DEFAULT NULL,
`cust_contact` char(50) DEFAULT NULL,
`cust_email` char(255) DEFAULT NULL,
PRIMARY KEY (`cust_id`),
) ENGINE=InnoDB AUTO_INCREMENT=10006 DEFAULT CHARSET=utf8
```

执行语句。

```
mysql> select * from customers where cust_zip = '44444' limit 0,1 \G;
```

执行计划。

```
 mysql> explain select * from customers where cust_zip = '44444' limit 0,1 \G;


           id: 1
  select_type: SIMPLE
  table: customers
  type: ALL
  possible_keys: NULL
  key: NULL
  key_len: NULL
  ref: NULL
  rows: 505560
  Extra: Using where
```

执行计划看到type为ALL，是全表扫描，每次执行需要扫描505560行数据，这是非常消耗性能的，那么下面将介绍优化方式。

添加索引。

```
mysql> alter table customers add index idx_cus(cust_zip);
```

执行计划。

```
 mysql> explain select * from customers where cust_zip = '44444' limit 0,1 \G;


           id: 1
  select_type: SIMPLE
  table: customers
  type: ref
  possible_keys: idx_cus
  key: idx_cus
  key_len: 31
  ref: const
  rows: 4555
  Extra: Using index condition
```

执行计划看到type为ref，基于索引的等值查询，或者表间等值连接。

无索引案例二

表结构同上案例相同，执行语句。

```
 mysql> select cust_id,cust_name,cust_zip from customers where cust_zip = '42222'order by
cust_zip,cust_name;
```

执行计划。

```
 mysql> explain select cust_id,cust_name,cust_zip from customers where cust_zip = '42222'order by
cust_zip,cust_name\G;


           id: 1
select_type: SIMPLE
table: customers
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 505560
Extra: Using filesort
```

添加索引。

```
mysql> alter table customers add index idx_cu_zip_name(cust_zip,cust_name);
```

执行计划。

```
 mysql> explain select cust_id,cust_name,cust_zip from customers where cust_zip = '42222'order by
cust_zip,cust_name\G;


           id: 1
select_type: SIMPLE
table: customers
type: ref
possible_keys: idx_cu_zip_name
key: idx_cu_zip_name
key_len: 31
ref: const
rows: 4555
Extra: Using where; Using index
```

order by使用字段，而且字段应该是索引字段。

隐式转换案例

隐式转换案例一

```
mysql> explain select * from customers where cust_zip = 44444 limit 0,1 \G;


         id: 1
select_type: SIMPLE
table: customers
type: ALL
possible_keys: idx_cus
key: NULL
key_len: NULL
ref: NULL
rows: 505560
Extra: Using where


mysql> show warnings;
Warning： Cannot use range access on index 'idx_cus' due to type or collation conversion on field 'cust_zip'
```

上述案例中由于表结构定义cust_zip字段是字符串数据类型，而应用传入的是数字，导致了隐式转换，无法使用索引。

解决方案：

1. 将cust_zip字段修改为数字数据类型。
2. 将应用中传入的字符类型改为数据类型。

隐式转换案例二

查看表结构。

```
mysql> show create table customers1;


CREATE TABLE `customers1` (
`cust_id` varchar(10) CHARACTER SET latin1 COLLATE latin1_bin DEFAULT NULL,
`cust_name` char(50) NOT NULL,
KEY `idx_cu_id` (`cust_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
mysql> show create table customers2;
CREATE TABLE `customers2` (
`cust_id` varchar(10) CHARACTER SET utf8 COLLATE utf8_bin DEFAULT NULL,
`cust_name` char(50) NOT NULL,
KEY `idx_cu_id` (`cust_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

执行语句。

```
mysql> select customers1.* from  customers2 left  join  customers1 on
customers1.cust_id=customers2.cust_id where customers2.cust_id='x';
```

执行计划。

```
mysql> explain select customers1.* from  customers2 left  join  customers1 on
customers1.cust_id=customers2.cust_id where customers2.cust_id='x'\G;


*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: customers2
type: ref
possible_keys: idx_cu_id
key: idx_cu_id
key_len: 33
ref: const
rows: 1
Extra: Using where; Using index
```

```
*************************** 2. row ***************************
id: 1
select_type: SIMPLE
table: customers1
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 1
Extra: Using where; Using join buffer (Block Nested Loop)
```

修改COLLATE。

```
mysql> alter table customers1 modify column cust_id varchar(10) COLLATE utf8_bin ;
```

执行计划。

```
 mysql> explain select cust_id,cust_name,cust_zip from customers where cust_zip = '42222'order by
cust_zip,cust_name\G;


           id: 1
select_type: SIMPLE
table: customers2
type: ref
possible_keys: idx_cu_id
key: idx_cu_id
key_len: 33
ref: const
rows: 1
Extra: Using where; Using index


           id: 1
select_type: SIMPLE
table: customers1
type: ref
possible_keys: idx_cu_id
key: idx_cu_id
key_len: 33
ref: const
rows: 1
Extra: Using where
```

字段的COLLATE一致后执行计划使用到了索引，所以一定要注意表字段的collate属性的定义保持一致。

总结

在使用索引时，我们可以通过explain查看SQL的执行计划，判断是否使用了索引以及发生了隐式转换，创建合适的索引。索引太复杂，创建需谨慎。

MySQL实例参数调优参考

对于云数据库MySQL版的实例，您可以通过控制台上修改一些参数。对于某些重要参数而言，不恰当的参数值会导致实例性能问题或应用报错，所以本文将介绍一些重要参数的最优值建议以减少您在设置参数时的疑虑。

back_log

默认值：3000

修改完后是否需要重启：是

作用：MySQL每处理一个连接请求时都会创建一个新线程与之对应。在主线程创建新线程期间，如果前端应用有大量的短连接请求到达数据库，MySQL会限制这些新的连接进入请求队列，由参数back_log控制。如果等待的连接数量超过back_log的值，则将不会接受新的连接请求，所以如果需要MySQL能够处理大量的短连接，需要提高此参数的大小。

现象：如果参数过小，应用可能出现如下错误。

```
SQLSTATE[HY000] [2002] Connection timed out;
```

修改建议：提高此参数值的大小。

innodb_autoinc_lock_mode

默认值：1

修改完后是否需要重启：是

作用：在MySQL 5.1.22后，InnoDB为了解决自增主键锁表的问题，引入了参数

innodb_autoinc_lock_mode，用于控制自增主键的锁机制。该参数可以设置的值为0、1、2，RDS默认的参数值为1，表示InnoDB使用轻量级别的mutex锁来获取自增锁，替代最原始的表级锁。但是在load data（包括INSERT … SELECT和REPLACE … SELECT）场景下若使用自增表锁，则可能导致应用在并发导入数据时出现死锁。

现象：在load data（包括INSERT … SELECT和REPLACE … SELECT）场景下若使用自增表锁，在并发导入数据时出现如下死锁。

```
RECORD LOCKS space id xx page no xx n bits xx index PRIMARY of table xx.xx trx id xxx lock_mode X insert intention waiting. TABLE LOCK table xxx.xxx trx id xxxx lock mode AUTO-INC waiting；
```

修改建议：建议将该参数值改为2，表示所有情况插入都使用轻量级别的mutex锁（只针对row模式），这样就可以避免auto_inc的死锁，同时在INSERT … SELECT的场景下性能会有很大提升。

说明：当该参数值为2时，binlog的格式需要被设置为row。

query_cache_size

默认值：3145728

修改完后是否需要重启：否

作用：该参数用于控制MySQL query cache的内存大小。如果MySQL开启query cache，在执行每一个query的时候会先锁住query cache，然后判断是否存在于query cache中，如果存在则直接返回结果，如果不存在，则再进行引擎查询等操作。同时，insert、update和delete这样的操作都会将query cahce失效掉，这种失效还包括结构或者索引的任何变化。但是cache失效的维护代价较高，会给MySQL带来较大的压力。所以，当数据库不会频繁更新时，query cache是很有用的，但如果写入操作非常频繁并集中在某几张表上，那么query cache lock的锁机制就会造成很频繁的锁冲突，对于这一张表的写和读会互相等待query cache lock解锁，从而导致select的查询效率下降。

现象：数据库中有大量的连接状态为checking query cache for query、Waiting for query cache lock、storing result in query cache。

修改建议：RDS默认是关闭query cache功能的，如果您的实例打开了query cache，当出现上述情况后可以关闭query cache。

net_write_timeout

默认值：60

修改完后是否需要重启：否

作用：等待将一个block发送给客户端的超时时间。

现象：若参数设置过小，可能会导致客户端出现错误the last packet successfully received from the server was milliseconds ago或the last packet sent successfully to the server was milliseconds ago。

修改建议：该参数在RDS中默认设置为60秒，一般在网络条件比较差时或者客户端处理每个block耗时较长时，由于net_write_timeout设置过小导致的连接中断很容易发生，建议增加该参数的大小。

tmp_table_size

默认值：2097152

修改完后是否需要重启：否

作用：该参数用于决定内部内存临时表的最大值，每个线程都要分配，实际起限制作用的是tmp_table_size和max_heap_table_size的最小值。如果内存临时表超出了限制，MySQL就会自动地把它转化为基于磁盘的MyISAM表。优化查询语句的时候，要避免使用临时表，如果实在避免不了的话，要保证这些临时表是存在内存中的。

现象：如果复杂的SQL语句中包含了group by、distinct等不能通过索引进行优化而使用了临时表，则会导致SQL执行时间加长。

修改建议：如果应用中有很多group by、distinct等语句，同时数据库有足够的内存，可以增大tmp_table_size（max_heap_table_size）的值，以此来提升查询性能。

loose_rds_max_tmp_disk_space

默认值：10737418240

修改完后是否需要重启：否

作用：用于控制MySQL能够使用的临时文件的大小。

现象：如果临时文件超出loose_rds_max_tmp_disk_space的取值，则会导致应用出现如下错误。

```
The table ‘/home/mysql/dataxxx/tmp/#sql_2db3_1’ is full
```

修改建议：首先，需要分析一下导致临时文件增加的SQL语句是否能够通过索引或者其它方式进行优化。其次，如果确定实例的空间足够，则可以提升此参数的值，以保证SQL能够正常执行。

loose_tokudb_buffer_pool_ratio

默认值：0

修改完后是否需要重启：是

作用：用于控制TokuDB引擎能够使用的buffer内存大小，比如innodb_buffer_pool_size设置为1000MB，tokudb_buffer_pool_ratio设置为50（代表50%），那么TokuDB引擎的表能够使用的buffer内存大小则为500MB。

修改建议：如果RDS中使用TokuDB引擎，建议调大该参数，以此来提升TokuDB引擎表的访问性能。

loose_max_statement_time

默认值：0

修改完后是否需要重启：否

作用：用于控制查询在MySQL的最长执行时间。如果超过该参数设置的时间，查询将会自动失败，默认是不限制。

现象：若查询时间超过了该参数的值，则会出现如下错误。

```
ERROR 3006 (HY000): Query execution was interrupted, max_statement_time exceeded
```

修改建议：如果您想要控制数据库中SQL的执行时间，则可以开启该参数，单位是毫秒。

loose_rds_threads_running_high_watermark

默认值：50000

修改完后是否需要重启：否

作用：用于控制MySQL并发的查询数目，比如将rds_threads_running_high_watermark的值设置为100，则允许MySQL同时进行的并发查询为100个，超过限制数量的查询将会被拒绝掉。该参数与rds_threads_running_ctl_mode配合使用（默认值为select）。

修改建议：该参数常常在秒杀或者大并发的场景下使用，对数据库具有较好的保护作用。

RDS for MySQL 5.7如何创建用户

本文将介绍以下几个知识点：

如何在RDS上创建数据库用户？

如何通过创建的用户来访问数据库？

如何用SQL语句创建用户并给用户授权？

文中所述操作步骤为RDS for MySQL 5.7版本的示例。关于如何创建数据库，请参见创建数据库和账号MySQL 5.7版。

MySQL 5.7版本权限体系相比MySQL 5.5/5.6版本更加开放，只需一个初始账号就可以对实例进行管理，使用初始账号通过SQL可以创建其它数据库账号。

数据库账号没有个数限制。

注意事项：

分配数据库账号权限时，请按最小权限原则和业务角色创建账号，并合理分配只读和读写权限。必要时可以把数据库账号和数据库拆分成更小粒度，使每个数据库账号只能访问其业务之内的数据。如果不需要数据库写入操作，请分配只读权限。

请设置数据库账号的密码为强密码，并定期更换。

如何在RDS上创建数据库用户

当从阿里云上的RDS新建实例后，需要先创建一个拥有最高管理权限的初始账号来对数据库进行管理。

一旦这个账号创建完成，就可以通过它访问到数据库进行普通用户的创建和管理，以及对整个RDS数据库的管理。

操作步骤

登录RDS管理控制台。

选择目标实例所在地域。

单击目标实例的ID，进入**基本信息**页面。

选择左侧菜单栏中的**账号管理**，进入*账号管理*页面。

单击**创建初始账号**。

输入要创建的账号信息，单击**确定**，如下图所示。

参数说明：

- 数据库账号：由 2~16 个字符的小写字母，数字或下划线组成、开头需为字母，结尾需为字母或数字。
- 密码：该账号对应的密码，由 6~32 个字符的字母、数字、中划线或下划线组成。
- 确认密码：输入与密码一致的字段，以确保密码正确输入。

选择左侧菜单栏中的**数据安全性**，进入**数据安全性**页面。

8. 单击*白名单分组 default* 后面的**修改**，进入**修改白名单分组**页面。

9. 将IP段*10.143.32.0/24*、*10.143.34.0/24*和*112.124.140.0/24*添加到组内白名单栏中，然后单击**确定**，返回**数据安全性**页面。

注意：

- 本文以网络类型是经典网络的数据库为例，关于网络类型详情，请参见设置网络类型。若是VPC网络，则需将IP段*100.104.175.0/24*添加到白名单中。
- 关于白名单的设置详情，请参见设置白名单。
- 设置白名单后，约5分钟左右才能生效。为避免以下操作失败，请等待5分钟后再继续下面的操作。

如何通过创建的用户来访问数据库

一般，访问MySQL数据库是通过在终端输入`mysql -u root –p输入密码后进行访问的，在阿里云上是通过控制台进行远程登录的。

操作步骤

1. 单击页面右上角的登录数据库，进入数据管理控制台的快捷登录页面。

   在快捷登录页面，检查阿里云数据库标签页面显示的连接地址和端口信息。若正确，填写数据库用户名和密码，然后单击登录。

   说明：若是VPC网络，请在快捷页面选择自建库标签页面，然后根据提示选择VPC网络类型并填写相关信息。关于操作详情，请参见DMS相关文档。

   您可以在RDS管理控制台的实例基本信息页面查看该账号的连接地址和端口信息。

   参数说明：

   - 数据库用户名：要访问数据库的账户名称。
   - 密码：上述账户所对应的密码。

   填写验证码，然后单击登录。

   说明：若您希望浏览器记住该账号的密码，可以先勾选**记住密码**，然后再单击**登录**。

如何用SQL语句创建用户并给用户授权

除了创建数据库初始账号是在控制台进行的，其它数据库账号都是通过SQL命令（例如create user语句）创建的。MySQL创建用户的方法分成三种：

INSERT USER表的方法：

因为数据库的用户信息都是保存在mysql.user这张表的，所以直接对该表进行插入语句，即可完成用户的创建。

向MySQL数据库的user表插入一条数据。

```
mysql> insert into mysql.user(Host,User,authentication_string)
values('localhost','phplamp',password('1234'));
```

刷新系统权限表。

```
mysql>flush privileges;
```

这样就创建了一个名为：phplamp，密码为：1234的用户。

CREATE USER的方法

语法：

CREATE USER 'username@host' [IDENTIFIED BY 'PASSWORD']

其中密码是可选项。

例子：

```
mysql> CREATE USER 'john@192.168.189.71' IDENTIFIED BY "123";
mysql> CREATE USER 'john@192.168.189.%' IDENTIFIED BY "123";
mysql> CREATE USER 'john@';
```

说明：该方法创建出来的用户只有连接数据库的权限，需要后续继续授权；

注意：用户与@后主机地址是一体的，用一个分号连接，否则会报错：ERROR 1396 (HY000): Operation CREATE USER failed for ‘remote’ @’ %’

GRANT的方法

当数据库存在用户的时候GRANT会对用户进行授权，但当数据库不存在该用户的时候，就会创建相应的用户并进行授权。

语法：

```
GRANT <ALL|priv1,priv2,.....privn> ON
[object] [IDENTIFIED BY 'password']
```

```
[WITH GRANT OPTION];
MAX_QUERIES_PER_HOUR count
MAX_UPDATES_PER_HOUR count
MAX_CONNECTIONS_PER_HOUR count
MAX_USER_CONNECTIONS count
```

说明：priv代表权限
select，insert，update，delete，create,drop，index，alter，grant，references，reload，shutdown，process，file等14个权限。

使用示例：

```
mysql>grant select,insert,update,delete,create,drop on test.hr to john@192.168.10.1 identified by '123';
```

说明：给主机为192.168.10.1的用户john分配可对数据库test的hr表进行select,insert,update,delete,create,drop等操作的权限，并设定口令为123。

```
mysql>grant all privileges on test.* to joe@192.168.10.1 identified by '123';
```

说明：给主机为192.168.10.1的用户john分配可对数据库test所有表进行所有操作的权限，并设定口令为123。

```
mysql>grant all privileges on *.* to john@192.168.10.1 identified by '123';
```

说明：给主机为192.168.10.1的用户john分配可对所有数据库的所有表进行所有操作的权限，并设定口令为123。

```
mysql>grant all privileges on *.* to john@localhost identified by '123';
```

说明：用户john分配可对所有数据库的所有表进行所有操作的权限，并设定口令为123。

了解了MySQL创建用户的三种方式后，就可以通过阿里云的SQL执行窗口进行用户的创建，具体操作步骤如下：

通过数据管理控制台的快捷登录页面登录初始账号。

在页面上方的菜单栏中，选择**SQL操作** > **SQL窗口**。

在SQL窗口中输入创建用户的命令命令，例如：

```
CREATE USER 'john@192.168.189.71' IDENTIFIED BY "123";
```

单击**执行**，用户创建完成。

用户创建后，就可以在数据管理控制台的快捷登录页面输入用户名和密码访问数据库。

利用CloudDBA解决MySQL实例CPU使用率过高的问题

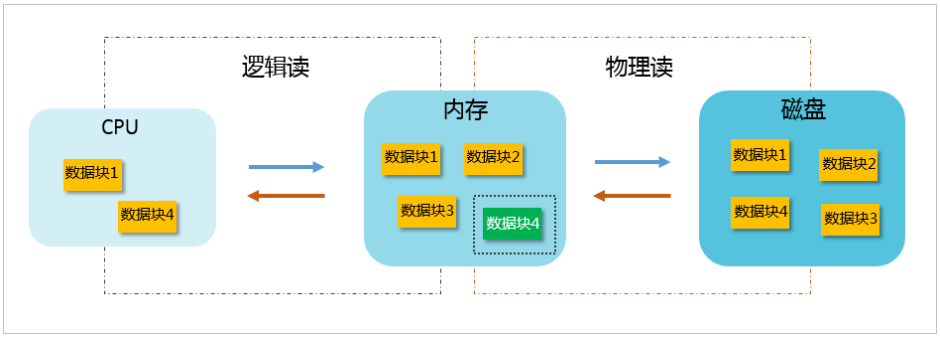
说明：本文暂不适用于MySQL 5.7版本的实例。关于MySQL 5.7实例的CPU使用率过高的解决方案，请参见MySQL CPU使用率高的原因和解决方法。

在使用MySQL数据库的过程中，经常会因CPU使用率过高而导致系统异常，如响应变慢、无法获取连接、出现报错等。在CPU使用率过高的场景中，有95%以上都是由异常SQL所致。另外，大量行锁冲突、行锁等待或后台任务也有可能导致实例CPU使用率过高，但这些状况出现的概率非常低，本文不做讨论。本文将介绍如何使用RDS的CloudDBA功能来定位系统中的慢SQL和其它异常SQL语句，然后您可通过CloudDBA提供的建议优化这些SQL语句来降低实例的CPU使用率以提升系统效率。

导致CPU使用率较高的原因

数据库中的数据以数据块为基本操作单位，一个MySQL数据块是8KB，一次逻辑读或物理读对应一个数据块的读取操作。

当数据库执行业务查询语句（包括数据修改操作）时，CPU会先从内存中请求数据块。如果内存中有对应的数据，CPU执行计算任务后会将结果返回给用户。如果内存中没有对应的数据，系统会触发从磁盘读取数据的操作。这两个数据获取过程分别是逻辑读和物理读，如下图所示。



当业务提交的SQL语句不够优化时，就会对数据库的性能产生如下影响：

使数据库产生大量的逻辑读，从而导致CPU使用率过高。

使数据库产生大量的物理读，从而导致IOPS和I/O延时过高。

解决方法

CloudDBA提供了如下两种SQL诊断功能：

诊断慢SQL：诊断当前实例最近1个月内的慢SQL，并给出慢SQL的优化建议。

SQL统计：利用云数据库SQL审计数据，从多个维度分析SQL语句并给出慢SQL的优化建议。

所以，您可以通过如下两种方法来排查和优化导致CPU使用率过高的异常SQL语句。

前提条件

实例是公共云华北1、华北2、华东1、华东2、华南1地域的MySQL 5.5或MySQL 5.6版本的实例。

使用SQL诊断功能排查异常SQL语句

登录RDS管理控制台。

选择目标实例所在地域。

单击目标实例ID，进入**基本信息**页面。

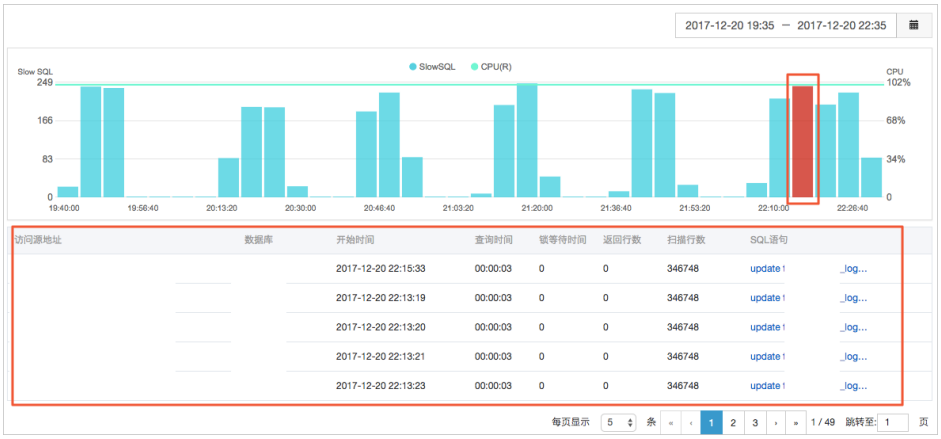
在左侧导航栏中，选择**CloudDBA** > **问题诊断**，进入**问题诊断**页面。

选择**慢SQL**标签页。

选择要查询的时间，然后单击**确定**。

说明：目前，系统只支持显示最近1个月内的慢SQL数据。

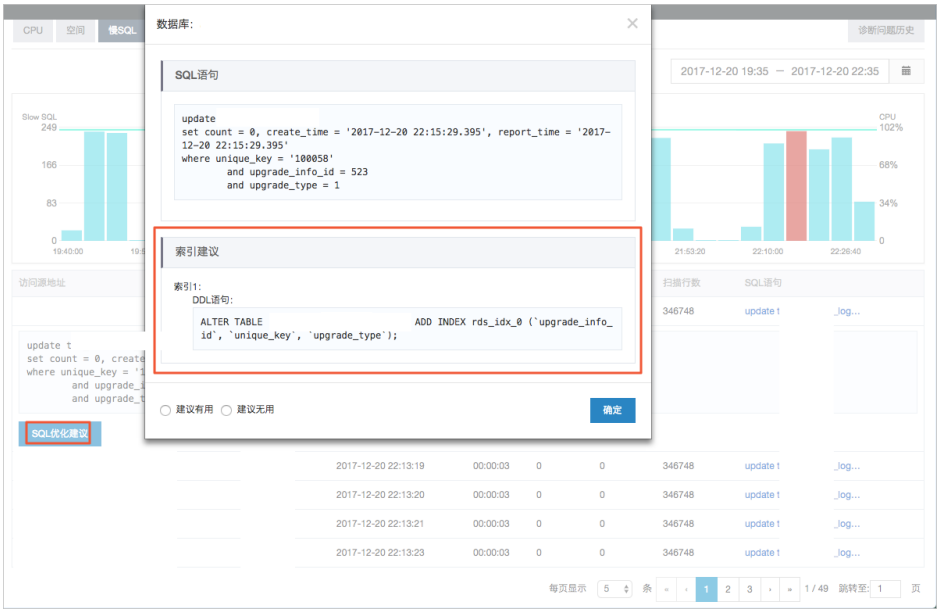
若实例中有慢SQL，图示中会以柱形图的方式显示慢SQL产生的时间点和个数。单击柱形图，下方的列表就会显示其对应的所有慢SQL信息，且柱形图会变成红色。



分析慢SQL，重点关注表格中**返回行数**和**扫描行数**的值。如下图中慢SQL，每条SQL语句都有很多扫描行数但返回行数都为0，说明系统产生了大量的逻辑读和物理读。产生物理读是因为内存大小有限，不可能缓存所有数据，当有大量数据请求时必然会产生大量物理I/O请求。大量的逻辑读会占用大量的CPU资源，导致CPU使用率上涨。

单击**SQL语句**栏中的SQL语句，查看该SQL语句的详情。

单击**SQL优化建议**，即可查看CloudDBA给该SQL语句提出的优化建议。



从上图的分析结果可以看出，该SQL语句执行时缺少对应的索引，导致执行该语句时要全表扫描。另外，根据MySQL的数据更新机制，每次执行该语句时整个表格都会被锁，进而使问题更加严重。

凡是执行该语句的session都会出现排队等待的状况，单次执行成本又极高，所以就很容易导致CPU使用率较高甚至达到100%的状况。

根据优化建议优化异常SQL语句，CPU使用率过高的问题就会随之而解。

使用SQL统计功能排查异常SQL语句

若使用SQL统计功能排查异常SQL语句，实例必须先要开通SQL审计，详情请参见SQL审计。

说明：SQL审计需另计费，为节约成本，您可以在查看数据库SQL语句前开通SQL审计，然后待问题排查完毕后再关闭该功能。

登录RDS管理控制台。

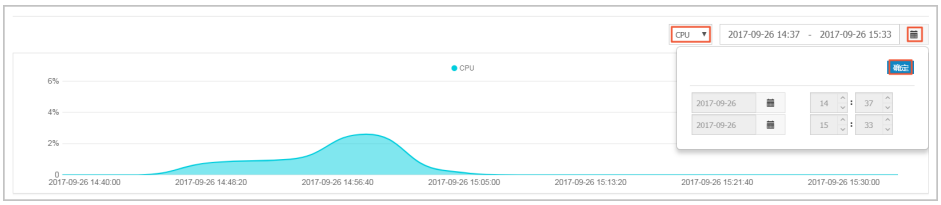
选择目标实例所在地域。

单击目标实例ID，进入**基本信息**页面。

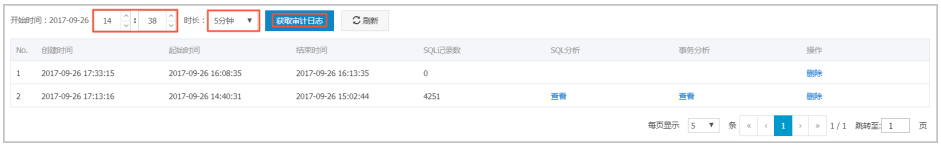
在左侧导航栏中，选择**CloudDBA** > **SQL统计**，进入**SQL统计**页面。

选择CPU，并选择要进行数据分析的时间范围，然后单击**确定**，状态图中即会显示当前实例的CPU在指定时间段内的使用率状况。

注意：您最多只能选择1天的时间段。



选择获取审计日志的起始时间（需在步骤5中所选择的时间范围内）以及时长，然后单击**获取审计日志**。



分析任务创建成功后，页面列表中会显示分析进度。

分析任务完成后，找到目标分析记录，并单击**SQL分析**栏下的**查看**，进入SQL分析详情页面。

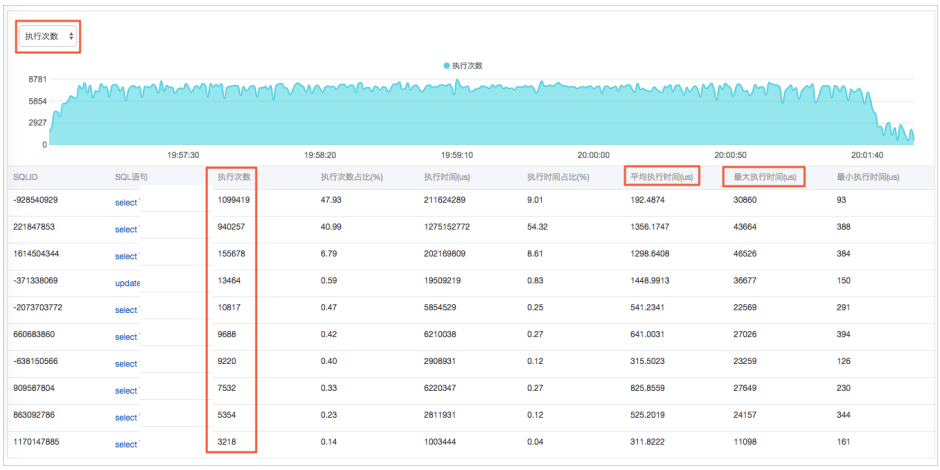
选择分析维度（执行次数、执行时间、扫描行数、返回行数、更新行数），并从不同维度分析SQL语句。

执行次数维度

在执行次数维度下，默认根据总执行次数倒序排序SQL语句，以定位执行量较大或者执行量有异常变化的语句，然后分析语句执行量的合理性并根据CloudDBA给出的建议优化SQL语句。

如果语句执行量相对比较合理，而且执行量大的SQL也是最优的，那么建议您升级实例规格，或者使用**读写分离**功能来分担执行量较大的SQL语句。您也可以通过优化程序来解决数据库的查询问题，例如采用Redis的缓存系统来缓存量大的最优SQL语句。缓存的成本相对较低，且响应速度更快，能够极大地优化系统的整体性能。

分析示例：



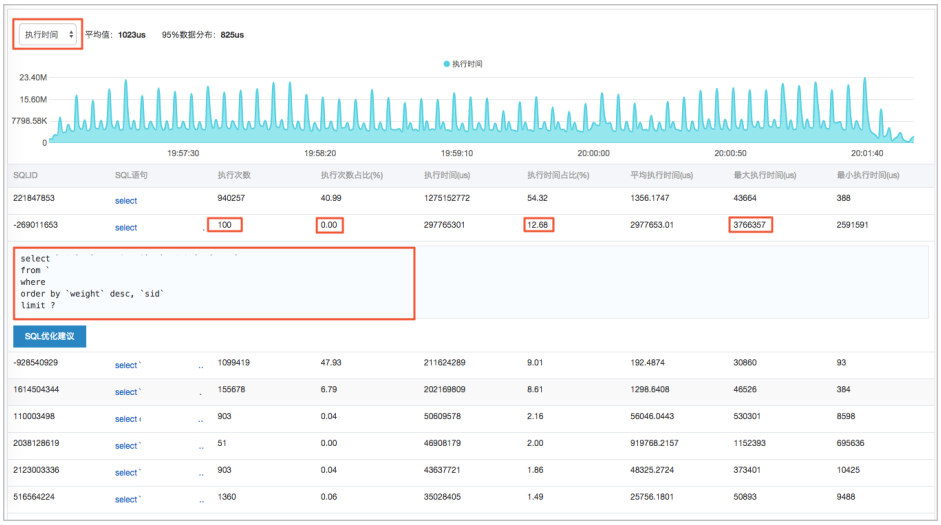
SQLID	SQL语句	执行次数	执行次数占比(%)	执行时间(us)	执行时间占比(%)	平均执行时间(us)	最大执行时间(us)	最小执行时间(us)
-928540929	select	1099419	47.93	211624289	9.01	192.4874	30860	93
221847853	select	940257	40.99	1275152772	54.32	1356.1747	43664	388
1614504344	select	155678	6.79	202169809	8.61	1298.6408	46526	384
-371338069	update	13464	0.59	19509219	0.83	1448.9913	36677	150
-2073703772	select	10817	0.47	5854529	0.25	541.2341	22569	291
660663860	select	9688	0.42	6210038	0.27	641.0031	27026	394
-638150566	select	9220	0.40	2908931	0.12	315.5023	23259	126
909587804	select	7532	0.33	6220347	0.27	825.8559	27649	230
863092786	select	5354	0.23	2811931	0.12	525.2019	24157	344
1170147685	select	3218	0.14	1003444	0.04	311.8222	11098	161

上图中，通过对比**平均执行时间**和**最大执行时间**，可以判断出执行次数最多的前两个SQL语句的平均执行时间很短，说明这两个SQL语句性能没有问题，您可以通过升级实例规格或开通读写分离功能来解决主库CPU压力大的问题。但这两个SQL语句的**最大执行时间**却很大，达到了30ms以上，说明整个系统存在波动，需要通过其它维度找出其它异常SQL语句。

执行时间维度

该维度默认按照执行时间倒序排列SQL语句，可以协助您筛选出执行总量不大但单次执行时间却很长的SQL语句。每次执行这类语句时，都会极大影响系统的响应时间和CPU使用率，当被大量执行时，CPU使用率就会较高甚至达到100%，系统卡死。

分析示例：

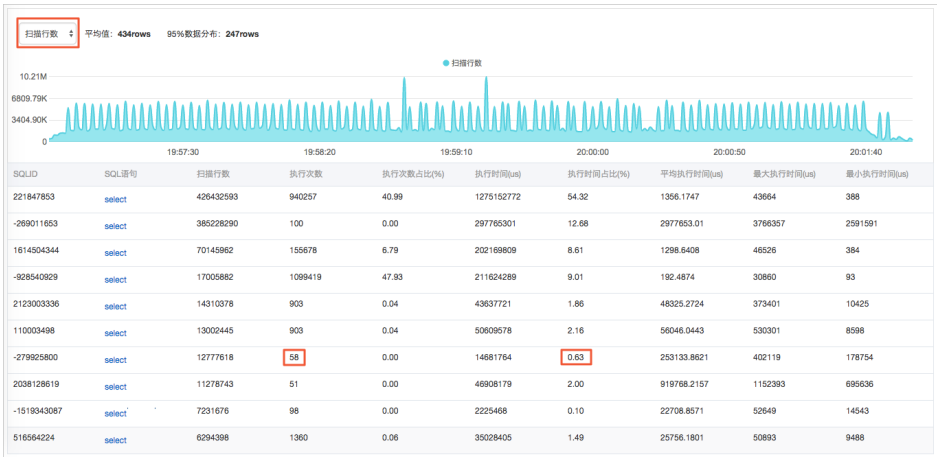


上图中，第二条SQL语句总共执行了100次，执行次数占比为0，但总执行时间占比却是12%，且单次最大执行时间约为300秒，说明了该语句的性能极差。单击该语句，通过语句详情可以看出，order by子句中既有asc又有desc排序，导致无法高效使用SQL排序，而where条件的过滤性很低，所以该SQL语句的性能很差。另外，由where条件过滤性能较低导致的SQL语句性能问题，还可以通过扫描行数维度排查。

扫描行数维度

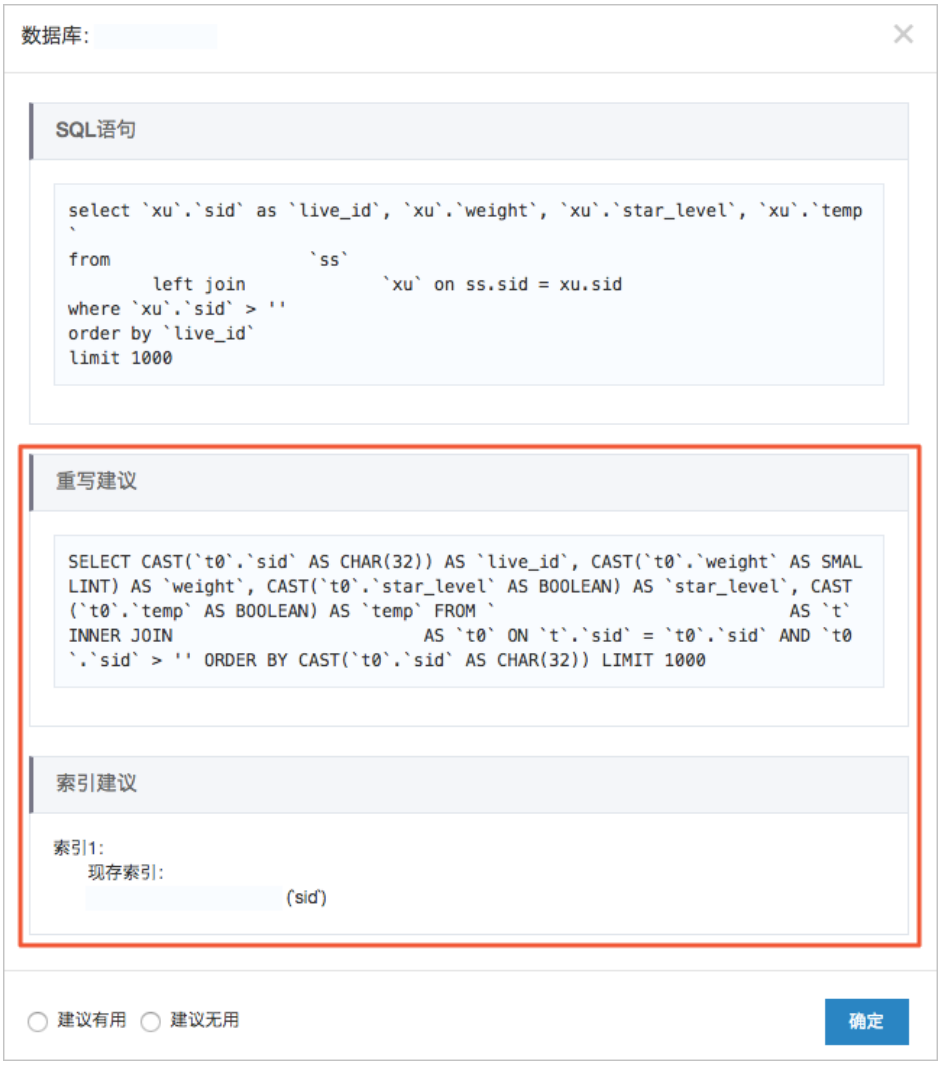
该维度默认按照扫描行数倒序排列SQL语句，可以协助您筛选出每次执行都会查询很多条记录的SQL语句，每次执行这类语句就会产生大量的逻辑读甚至物理读。针对这类SQL语句，如果语句已经足够优化，建议您优化程序，在where条件中传入更多的筛选条件，从而可以避免SQL语句单次查询时扫描很多行。

分析示例1：



SQLID	SQL语句	扫描行数	执行次数	执行次数占比(%)	执行时间(us)	执行时间占比(%)	平均执行时间(us)	最大执行时间(us)	最小执行时间(us)
221847853	select	426432593	940257	40.99	1275152772	54.32	1356.1747	43664	388
-269011653	select	385228290	100	0.00	297765301	12.68	2977653.01	3766357	2591591
1614504344	select	70145962	155678	6.79	202169809	8.61	1298.6408	46526	384
-928540929	select	17005882	1099419	47.93	211624289	9.01	192.4874	30860	93
2123003336	select	14310378	903	0.04	43637721	1.86	48325.2724	373401	10425
110003498	select	13002445	903	0.04	50609578	2.16	56046.0443	530301	8598
-279925800	select	12777618	58	0.00	14681764	0.63	253133.8621	402119	178754
2038128619	select	11278743	51	0.00	46908179	2.00	919768.2157	1152393	695636
-1519343087	select	7231676	98	0.00	2225468	0.10	22708.8571	52649	14543
516564224	select	6294398	1360	0.06	35028405	1.49	25756.1801	50893	9488

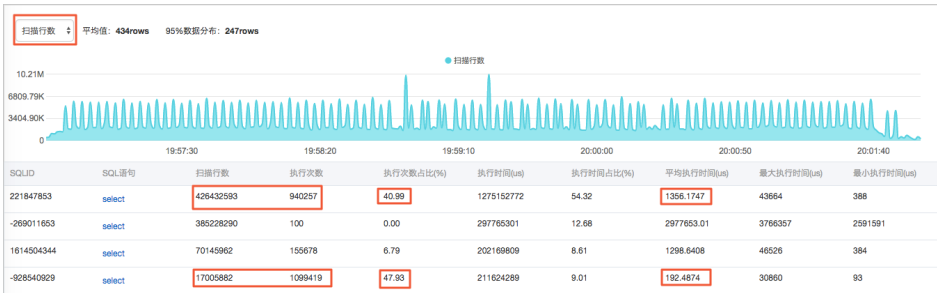
与执行时间维度相比，从扫描行数维度查看SQL语句性能时能筛选出更多异常SQL。例如，上图中执行次数为58次的语句占总执行时间的比例仅为0.63%，所以在执行时间维度的查询结果中很难搜索到该异常SQL。单击该SQL语句并查看CloudDBA给出的优化建议，如下图所示。



根据CloudDBA的分析可知，该SQL语句的写法存在很大的性能问题。因为数据库已经有了

本条语句所需要的索引，所以SQL语句中不需要额外创建索引来优化性能。此类写法导致无法使用正确的索引，所以执行该类SQL时会出现性能问题。根据CloudDBA给出的建议，针对此类异常SQL，您只需要修改SQL语句的写法即可降低CPU使用率，从而优化系统性能。

分析示例2：



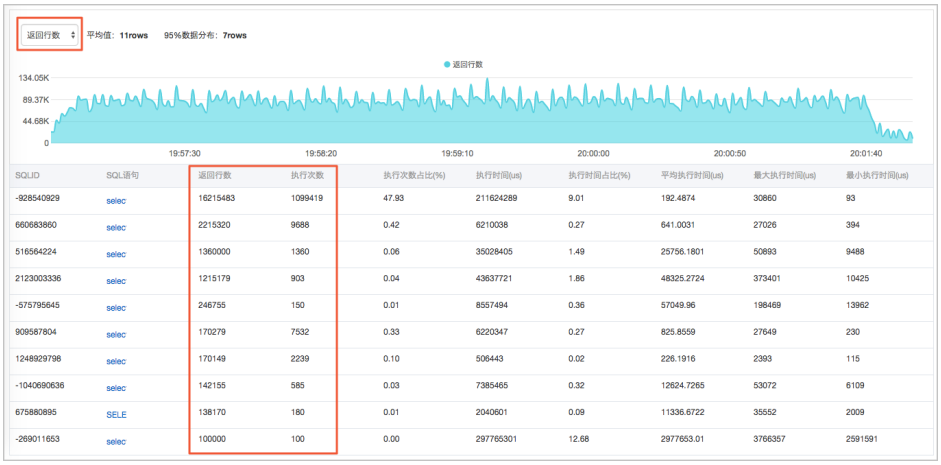
SQLID	SQL语句	扫描行数	执行次数	执行次数占比(%)	执行时间(us)	执行时间占比(%)	平均执行时间(us)	最大执行时间(us)	最小执行时间(us)
221847853	select	426432593	940257	40.99	1275152772	54.32	1356.1747	43664	388
-269011653	select	385228290	100	0.00	297765301	12.68	2977653.01	3766357	2591591
1614504344	select	70145962	155678	6.79	202169809	8.61	1298.6408	46526	384
-928540929	select	17005882	1099419	47.93	211624289	9.01	192.4874	30860	93

此外，扫描行数维度有时还能找出有性能隐患的SQL语句。例如，对比上图中第一条和第四条SQL语句，它们的执行次数占比分别是40.99%和47.93%，都占系统运行量的很大比重。同时，第一条和第四条语句的平均执行时间分别是1.35毫秒和0.19毫秒，虽然有些差距但性能都还不错。但若对比二者扫描行数和执行次数的比值，即平均单次扫描的行数，可看出单次执行第一条SQL时会扫描约500行数据，单次执行第二条SQL时会扫描约15行数据，说明第一条语句单次扫描行数过多，可能有异常。第一条SQL语句的执行次数占比很大，如果能将该语句的单次扫描行数从500行优化到10行以内，可以极大地提升系统的性能。

返回行数维度

该维度默认按照返回行数倒序排列SQL语句，会展示出单次执行返回很多行的SQL语句，一般这类SQL语句常见于导出数据的场景。其实，非正常业务的这类SQL语句破坏力很大，当大量返回数据库的数据时很容易撑爆程序的内存，会严重破坏业务系统整体的稳定性。

分析示例：



SQLID	SQL语句	返回行数	执行次数	执行次数占比(%)	执行时间(us)	执行时间占比(%)	平均执行时间(us)	最大执行时间(us)	最小执行时间(us)
-928540929	selec	16215483	1099419	47.93	211624289	9.01	192.4874	30860	93
660683860	selec	2215320	9688	0.42	6210038	0.27	641.0031	27026	394
516564224	selec	1360000	1360	0.06	35028405	1.49	25756.1801	50893	9488
2123003336	selec	1215179	903	0.04	43637721	1.86	48325.2724	373401	10425
-575795645	selec	246755	150	0.01	8557494	0.36	57049.96	198469	13962
909587804	selec	170279	7532	0.33	6220347	0.27	825.8559	27649	230
1248929798	selec	170149	2239	0.10	506443	0.02	226.1916	2393	115
-1040690636	selec	142155	585	0.03	7385465	0.32	12624.7265	53072	6109
675880895	SELE	138170	180	0.01	2040601	0.09	11336.6722	35552	2009
-269011653	selec	100000	100	0.00	297765301	12.68	2977653.01	3766357	2591591

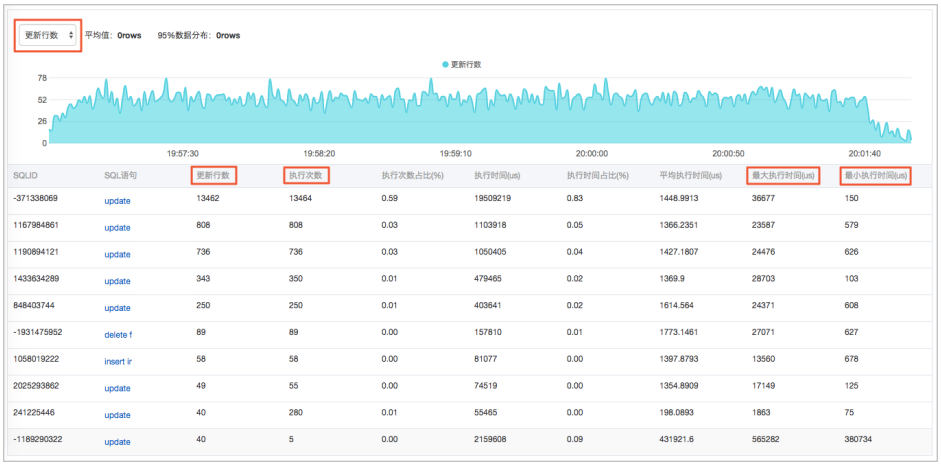
上图中，执行次数最多的SQL语句总的返回行数也较多是正常现象。但从第二条语句开始，语句的总执行量很小，但返回行数却很多，都需要优化。例如，第三条语句，每次执行

该语句时都会均等地返回1000条数据，业务场景应该是获取TOP 1000的数据的需求或某种数据导出需求。 除了优化该SQL语句外，还建议您采用缓存、利用只读实例、给应用程序做开关限流排队等优化手段降低数据库CPU的压力。

更新行数维度

该维度默认按照更新行数倒序排列SQL语句，可协助您找出单次执行写SQL语句时所影响的行数。另外，写SQL语句还会影响到锁、事务、连接数等问题，所以有性能问题的写SQL语句对数据库的破坏力极大，很容易导致系统卡住、大量锁争用甚至死锁、有很多执行和排队的连接等，最终使系统无法响应业务请求。

分析示例：



SQLID	SQL语句	更新行数	执行次数	执行次数占比(%)	执行时间(us)	执行时间占比(%)	平均执行时间(us)	最大执行时间(us)	最小执行时间(us)
-371338069	update	13462	13464	0.59	19509219	0.83	1448.9913	36677	150
1167984861	update	808	808	0.03	1103918	0.05	1366.2351	23587	579
1190894121	update	736	736	0.03	1050405	0.04	1427.1807	24476	626
1433634289	update	343	350	0.01	479465	0.02	1369.9	28703	103
848403744	update	250	250	0.01	403641	0.02	1614.564	24371	608
-1931475952	delete f	89	89	0.00	157810	0.01	1773.1461	27071	627
1058019222	insert ir	58	58	0.00	81077	0.00	1397.8793	13560	678
2025293862	update	49	55	0.00	74519	0.00	1354.8909	17149	125
241225446	update	40	280	0.01	55465	0.00	198.0893	1863	75
-1189290322	update	40	5	0.00	2159608	0.09	431921.6	565282	380734

上图中，更新行数和执行次数的比例基本为1：1，即平均执行一次写SQL时会更新一条记录，理论上性能应该很好。但根据**最大执行时间**可以看出，大部分写SQL的最大执行时间都超过了10ms，这说明整个数据库有很多异常时刻，极大地影响了更新性能。根据**最小执行时间**可以看出，大部分写SQL的性能都很好，但最后一条语句的性能极差，每次执行时都会影响系统的稳定性，例如CPU的使用率会出现一个尖刺，所以需要优化该语句。

单击最后一条SQL语句，并查看其SQL优化建议，系统会返回如下结果。经CloudDBA分析，该语句的写法和索引都没有问题，但由于存在where条件的字段隐式转换，使update语句不能正确使用索引，导致每次执行该语句时都是全表扫描并且锁全表，这就是该语句性能较差的原因。所以，只需要在where条件中显示转换sid字段类型即可优化该SQL语句。

SQL Server

解析SQL Server 2012常用的分析函数

分析函数CUME_DIST

微软的定义：

计算某个值在SQL Server 2012中的一组值内的累积分布。CUME_DIST计算某指定值在一组值中的相对位置。对于行r，假定采用升序，r的CUME_DIST是值低于或等于r的值的行数除以在分区或查询结果集中求出的行数。

函数解析：

执行如下代码，构造一组数据。

```
DECLARE
@analytic TABLE(
```

```
name varchar(35) ,
dept varchar(35),
salary money
)
INSERT INTO @analytic
VALUES
--bd
('andy01','bd',15000),
('andy02','bd',12000),
('andy03','bd',12000),
('andy04','bd',10000),
('andy05','bd',8000),
--ca
('andy06','ca',20000),
('andy07','ca',18000),
('andy08','ca',18000),
('andy09','ca',15000),
('andy10','ca',12000),
('andy11','ca',12000),
('andy12','ca',10000),
('andy13','ca',8000),
('andy14','ca',8000),
('andy15','ca',8000)

SELECT
dept,name ,salary,
CUME_DIST() OVER(PARTITION BY dept ORDER BY salary) AS cume_dist_
FROM @analytic
ORDER BY dept,salary DESC
```

返回结果如下：

Results Messages

	dept	name	salary	cume_dist_
1	bd	andy01	15000.00	1
2	bd	andy02	12000.00	0.8
3	bd	andy03	12000.00	0.8
4	bd	andy04	10000.00	0.4
5	bd	andy05	8000.00	0.2
6	ca	andy06	20000.00	1
7	ca	andy07	18000.00	0.9
8	ca	andy08	18000.00	0.9
9	ca	andy09	15000.00	0.7
10	ca	andy10	12000.00	0.6
11	ca	andy11	12000.00	0.6
12	ca	andy12	10000.00	0.4
13	ca	andy13	8000.00	0.3
14	ca	andy14	8000.00	0.3
15	ca	andy15	8000.00	0.3

示例解析：

按照dept分组，根据salary逻辑排序，针对每一个分组里的每一个值，计算在该分组下等于或者小于自己的salary的分布的百分比。举个例子，bd部门的andy02，salary为12000，那么等于或者小于这个12000的有4条，总共5条记录，因此那么CUME_DIST()=4/5=0.8。 同理，其它也是这样计算。

分析函数LAST_VALUE

微软的定义：

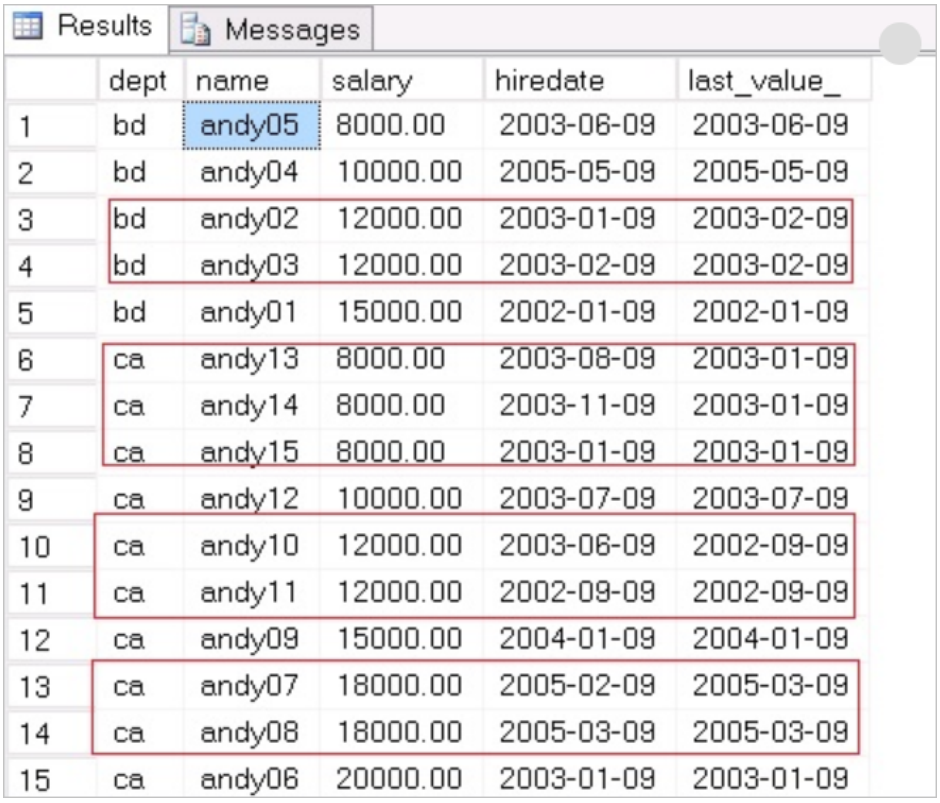
返回SQL Server 2012中有序值集中的最后一个值。

函数解析：

执行如下代码，构造一组数据。

```
DECLARE
@analytic TABLE(
name varchar(35) ,
dept varchar(35),
salary money ,
hiredate date
)
INSERT INTO @analytic
VALUES
--bd
('andy01','bd',15000,'2002-01-09'),
('andy02','bd',12000,'2003-01-09'),
('andy03','bd',12000,'2003-02-09'),
('andy04','bd',10000,'2005-05-09'),
('andy05','bd',8000,'2003-06-09'),
--ca
('andy06','ca',20000,'2003-01-09'),
('andy07','ca',18000,'2005-02-09'),
('andy08','ca',18000,'2005-03-09'),
('andy09','ca',15000,'2004-01-09'),
('andy10','ca',12000,'2003-06-09'),
('andy11','ca',12000,'2002-09-09'),
('andy12','ca',10000,'2003-07-09'),
('andy13','ca',8000,'2003-08-09'),
('andy14','ca',8000,'2003-11-09'),
('andy15','ca',8000,'2003-01-09')

SELECT
dept,name ,salary,hiredate,
LAST_VALUE(hiredate) OVER(PARTITION BY dept ORDER BY salary) AS last_value_
FROM @analytic
```

返回结果如下：



Results Messages

	dept	name	salary	hiredate	last_value_
1	bd	andy05	8000.00	2003-06-09	2003-06-09
2	bd	andy04	10000.00	2005-05-09	2005-05-09
3	bd	andy02	12000.00	2003-01-09	2003-02-09
4	bd	andy03	12000.00	2003-02-09	2003-02-09
5	bd	andy01	15000.00	2002-01-09	2002-01-09
6	ca	andy13	8000.00	2003-08-09	2003-01-09
7	ca	andy14	8000.00	2003-11-09	2003-01-09
8	ca	andy15	8000.00	2003-01-09	2003-01-09
9	ca	andy12	10000.00	2003-07-09	2003-07-09
10	ca	andy10	12000.00	2003-06-09	2002-09-09
11	ca	andy11	12000.00	2002-09-09	2002-09-09
12	ca	andy09	15000.00	2004-01-09	2004-01-09
13	ca	andy07	18000.00	2005-02-09	2005-03-09
14	ca	andy08	18000.00	2005-03-09	2005-03-09
15	ca	andy06	20000.00	2003-01-09	2003-01-09

示例解析：

按照OVER子句中ORDER BY根据salary排序，取salary最后行的hiredate值作为最后的LAST VALUE，重点在于当salary有相同的值时，需要取根据salary排序后的最后一条记录作为其他的LAST VALUE。

分析函数FIRST_VALUE

微软的定义：

返回SQL Server 2012中有序值集中的第一个值。

函数解析：

从微软的定义来看，FIRST_VALUE似乎跟LAST_VALUE是相反的含义，但实际并非如此。

执行如下代码，构造一组数据。

```
DECLARE
@analytic TABLE(
name varchar(35) ,
dept varchar(35),
salary money ,
hiredate date
)
INSERT INTO @analytic
VALUES
--bd
('andy01','bd',15000,'2002-01-09'),
```

```
('andy02','bd',12000,'2003-01-09'),
('andy03','bd',12000,'2003-02-09'),
('andy04','bd',10000,'2005-05-09'),
('andy05','bd',8000,'2003-06-09'),
--ca
('andy06','ca',20000,'2003-01-09'),
('andy07','ca',18000,'2005-02-09'),
('andy08','ca',18000,'2005-03-09'),
('andy09','ca',15000,'2004-01-09'),
('andy10','ca',12000,'2003-06-09'),
('andy11','ca',12000,'2002-09-09'),
('andy12','ca',10000,'2003-07-09'),
('andy13','ca',8000,'2003-08-09'),
('andy14','ca',8000,'2003-11-09'),
('andy15','ca',8000,'2003-01-09')

SELECT
dept,name ,salary,hiredate,
FIRST_VALUE(name) OVER(PARTITION BY dept ORDER BY salary) AS first_value_
FROM @analytic
```

返回结果如下：

	dept	name	salary	hiredate	first_value_
1	bd	andy05	8000.00	2003-06-09	andy05
2	bd	andy04	10000.00	2005-05-09	andy05
3	bd	andy02	12000.00	2003-01-09	andy05
4	bd	andy03	12000.00	2003-02-09	andy05
5	bd	andy01	15000.00	2002-01-09	andy05
6	ca	andy13	8000.00	2003-08-09	andy13
7	ca	andy14	8000.00	2003-11-09	andy13
8	ca	andy15	8000.00	2003-01-09	andy13
9	ca	andy12	10000.00	2003-07-09	andy13
10	ca	andy10	12000.00	2003-06-09	andy13
11	ca	andy11	12000.00	2002-09-09	andy13
12	ca	andy09	15000.00	2004-01-09	andy13
13	ca	andy07	18000.00	2005-02-09	andy13
14	ca	andy08	18000.00	2005-03-09	andy13
15	ca	andy06	20000.00	2003-01-09	andy13

示例分析：

显然，这个与LAST_VALUE并不是相反的含义。OVER子句根据ORDER BY来排序，按dept分组来确定这个分组的第一个值，而不是根据salary的值来确定的，所以与LAST_VALUE是不一样的。将FIRST_VALUE(name)修改为FIRST_VALUE(hiredate)后，对比看得更清楚，这个很有蒙蔽性。

分析函数LEAD

微软的定义：

访问相同结果集的后续行中的数据，而不使用SQL Server 2012中的自联接。LEAD以当前行之后的给定物理偏移量来提供对行的访问。在SELECT语句中使用此分析函数可将当前行中的值与后续行中的值进行比较。

函数解析：

执行如下代码，构造一组数据。

```
DECLARE
@analytic TABLE(
name varchar(35) ,
dept varchar(35),
salary money ,
hiredate date
)
INSERT INTO @analytic
VALUES
--bd
('andy01','bd',15000,'2002-01-09'),
('andy02','bd',12000,'2003-01-09'),
('andy03','bd',12000,'2003-02-09'),
('andy04','bd',10000,'2005-05-09'),
('andy05','bd',8000,'2003-06-09'),
--ca
('andy06','ca',20000,'2003-01-09'),
('andy07','ca',18000,'2005-02-09'),
('andy08','ca',18000,'2005-03-09'),
('andy09','ca',15000,'2004-01-09'),
('andy10','ca',12000,'2003-06-09'),
('andy11','ca',12000,'2002-09-09'),
('andy12','ca',10000,'2003-07-09'),
('andy13','ca',8000,'2003-08-09'),
('andy14','ca',8000,'2003-11-09'),
('andy15','ca',8000,'2003-01-09')

SELECT
dept,name,hiredate,salary,
LEAD(salary,1,0) OVER(PARTITION BY dept ORDER BY salary) AS lead_,
(LEAD(salary,1,0) OVER(PARTITION BY dept ORDER BY salary)-salary) AS diff_salary
FROM @analytic
```

返回结果如下：

Results Messages

	dept	name	hiredate	salary	lead_	diff_salary
1	bd	andy05	2003-06-09	8000.00	10000.00	2000.00
2	bd	andy04	2005-05-09	10000.00	12000.00	2000.00
3	bd	andy02	2003-01-09	12000.00	12000.00	0.00
4	bd	andy03	2003-02-09	12000.00	15000.00	3000.00
5	bd	andy01	2002-01-09	15000.00	0.00	-15000.00
6	ca	andy13	2003-08-09	8000.00	8000.00	0.00
7	ca	andy14	2003-11-09	8000.00	8000.00	0.00
8	ca	andy15	2003-01-09	8000.00	10000.00	2000.00
9	ca	andy12	2003-07-09	10000.00	12000.00	2000.00
10	ca	andy10	2003-06-09	12000.00	12000.00	0.00
11	ca	andy11	2002-09-09	12000.00	15000.00	3000.00
12	ca	andy09	2004-01-09	15000.00	18000.00	3000.00
13	ca	andy07	2005-02-09	18000.00	18000.00	0.00
14	ca	andy08	2005-03-09	18000.00	20000.00	2000.00
15	ca	andy06	2003-01-09	20000.00	0.00	-20000.00

示例分析：

按照dept分区，根据salary排序，比较当前记录和后一条记录（偏移量为1）的salary值的差值，这个非常实用。

通过Linked Server访问云下自建SQL Server

使用场景

RDS for SQL Server 2012和2016双机高可用版开放了Linked Server功能，不仅可以在RDS之间建立Linked Server，在网络连通的前提下，也支持和云下自建SQL Server建立Linked Server。

本文将介绍如何利用VPN在RDS for SQL Server上建立Linked Server连接到云下自建SQL Server。

> **注意**：和云下自建数据库链接的前提是网络连通，若有问题请咨询相关团队协助解决。

前提条件

要求RDS for SQL Server为以下版本：

- 2012 标准双机高可用版。
- 2016 标准双机高可用版。
- 2012 企业双机高可用版。
- 2016 企业双机高可用版。

RDS for SQL Server实例状态为**运行中**。

在部署VPN网关前，需要做好网络规划：

本地移动设备和云上VPC内需要访问的私网IP地址段不能相同，否则无法通信。

客户端必须能访问Internet。

操作步骤

利用VPN打通RDS for SQL Server所在VPC和云下机器间的网络连接通道

创建VPN网关

创建SSL服务端

创建客户端证书

客户端配置

连接测试

在RDS for SQL Server上创建Linked Server

利用VPN打通RDS for SQL Server所在VPC和云下机器的网络连接通道

创建VPN网关

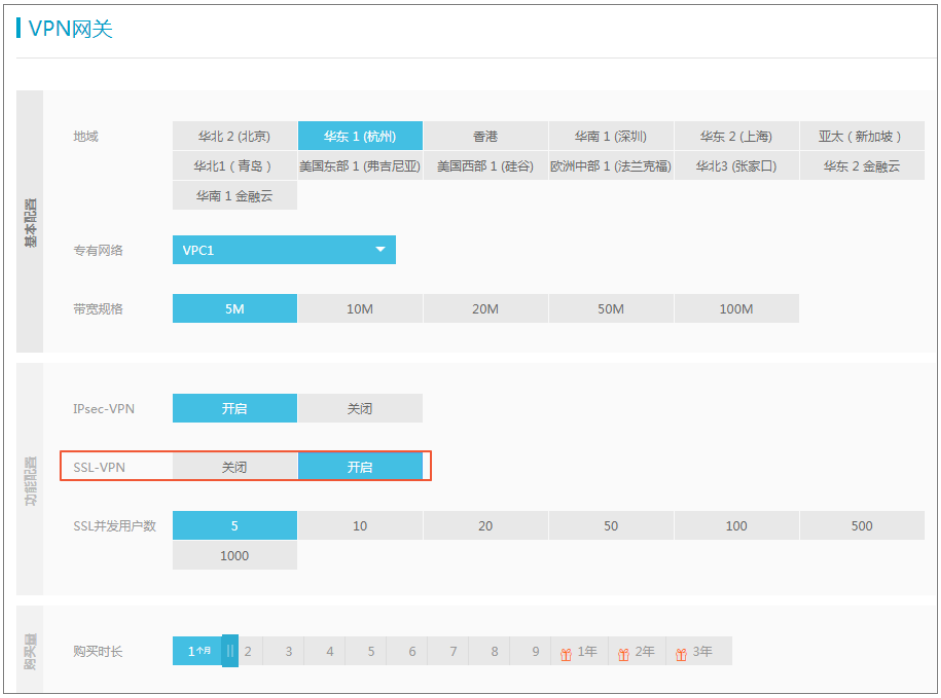
登录VPC管理控制台。

在左侧导航栏，单击**VPN** > **VPN网关**。

在VPN网关页面，单击**创建VPN网关**。

在购买页面，配置VPN网关，完成支付。本操作中VPN网关的配置如下：

配置项	说明
地域	- 选择VPN网关的地域。本操作中选择**华东1（杭州）**。 - 确保VPC的地域和VPN网关的地域相同。
专有网络	选择要连接的VPC。
带宽规格	选择一个带宽规格。带宽规格是VPN网关所具备的公网带宽。
IPsec-VPN	选择是否开启IPsec-VPN功能，IPsec-VPN功能适用于站点到站点的连接，可以根据您的实际需要选择开启。
SSL-VPN	选择是否开启SSL-VPN功能。本操作选择**开启**。
SSL并发连接数	选择您需要同时连接的客户端最大规格。



返回VPN网关页面，单击**华东1**地域，查看创建的VPN网关。

> **说明**：VPN网关的创建一般需要1-5分钟。

刚创建好的VPN网关的状态是**准备中**，约两分钟左右会变成**正常**状态。**正常**状态就表明VPN网关完成了初始化，可以正常使用了。

创建SSL服务端

在专有网络的左侧导航栏，单击**VPN** > **SSL服务端**。

单击**创建SSL服务端**。本操作中SSL服务端的配置如下：

配置项	说明
名称	输入SSL服务端的名称。
VPN网关	选择上面创建的VPN网关。
本端网段	以CIDR地址块的形式输入要连接的网络。单击**添加本端网段**添加多个本端网段，本端网段可以是任何VPC或交换机的网段，也可以是本地网络的网段。
客户端网段	以CIDR地址块的形式输入客户端连接服务端时使用的IP地址。
高级配置	使用默认高级配置。

创建客户端证书

在专有网络的左侧导航栏，单击**VPN** > **SSL客户端**。

单击**创建SSL客户端证书**。

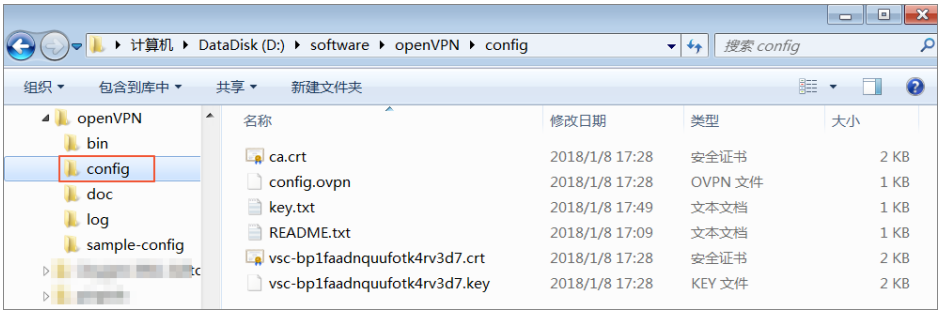
在**创建客户端证书**对话框，输入客户端证书名称并选择对应的SSL服务端，单击**确定**。

在**SSL客户端**页面，找到已创建的客户端证书，单击**下载**下载生成的客户端证书。

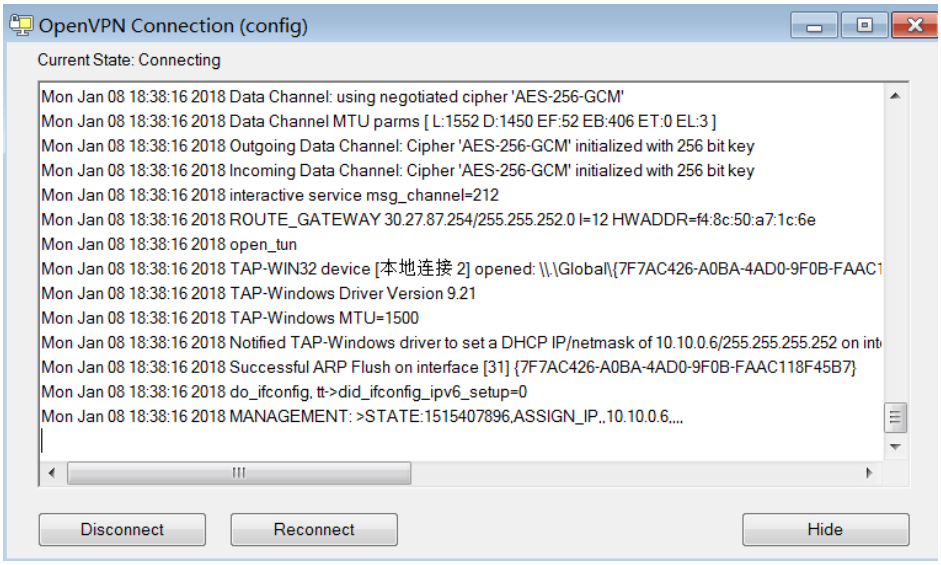
客户端配置

下载并安装OpenVPN客户端。

将**创建客户端证书**中下载的证书解压后复制到OpenVPN安装目录中的config文件夹中。



单击**Connect**发起连接。



连接测试

在客户端ping已连接的VPC内的一台ECS实例，测试连通性。

> **说明**：确保测试的ECS实例的安全组规则允许客户端远程连接，授权对象为SSL服务端配置中指定的客户端网段。详情请参考**安全组配置案例**。

注意：若连接不通，可能是设置了本地机器防火墙，需要将防火墙设置为允许远端连接。

在RDS for SQL Server上创建Linked Server

在RDS for SQL Server上创建Linked Server：

```
 USE master
GO

DECLARE
@linked_server_name sysname = N'myTestLinkedServer',
@data_source sysname = N'30.40.13.6,1433',
@user_name sysname = N'TestDbo' ,
@password nvarchar(128) = N'TestDbo@123'
;

EXEC [dbo].[sp_rds_add_linked_server_ha]
@linked_server_name,
@data_source,
```

```
@user_name,
@password
;
GO
```

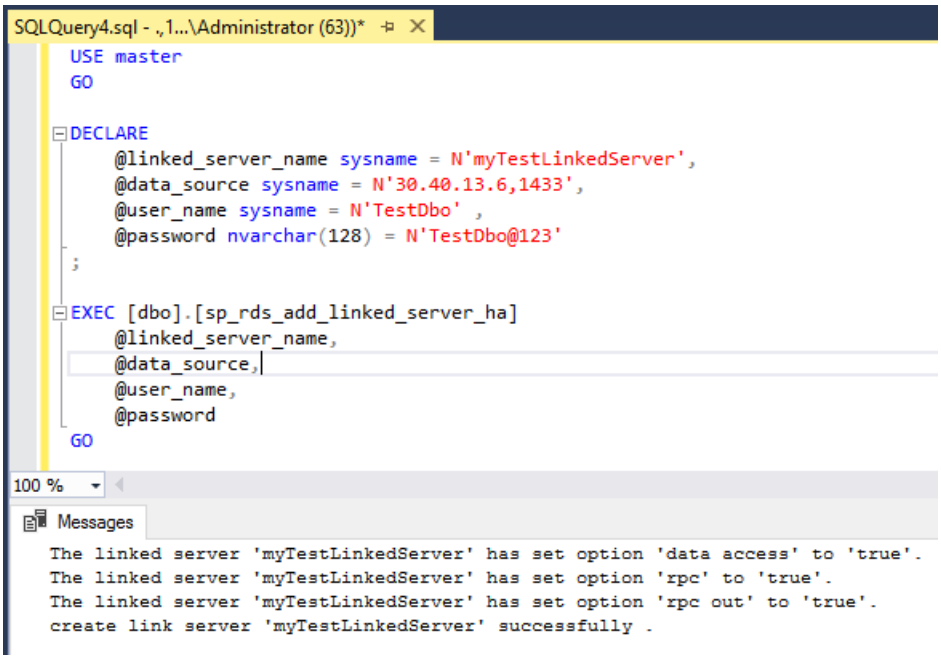
参数说明：

@linked_server_name：Linked Server的名称。

@data_source：用户线下SQL Server的IP和端口号，格式为：IP,Port。

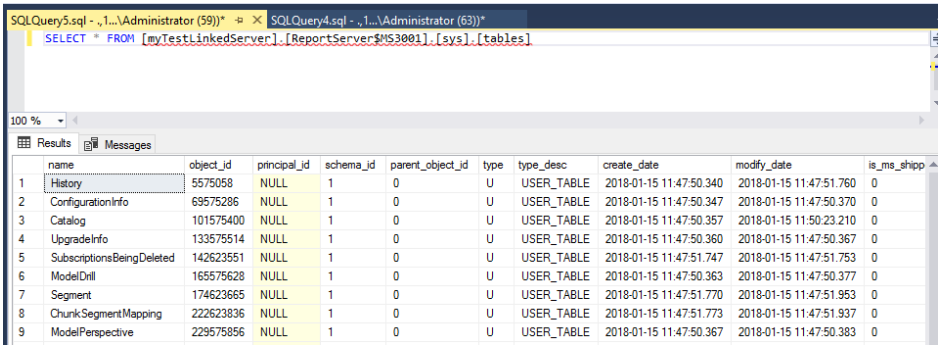
@user_name：用户线下SQL Server的登录用户名。

@password：用户线下SQL Server登录名对应的密码。



执行以下命令测试Linked Server：

SELECT * FROM [myTestLinkedServer].[ReportServer$MS3001].[sys].[tables]



	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date	is_ms_shipp
1	History	5575058	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:50.340	2018-01-15 11:47:51.760	0
2	ConfigurationInfo	69575286	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:50.347	2018-01-15 11:47:50.370	0
3	Catalog	101575400	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:50.357	2018-01-15 11:50:23.210	0
4	UpgradeInfo	133575514	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:50.360	2018-01-15 11:47:50.367	0
5	SubscriptionsBeingDeleted	142623551	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:51.747	2018-01-15 11:47:51.753	0
6	ModelDrill	165575628	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:50.363	2018-01-15 11:47:50.377	0
7	Segment	174623665	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:51.770	2018-01-15 11:47:51.953	0
8	ChunkSegmentMapping	222623836	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:51.773	2018-01-15 11:47:51.937	0
9	ModelPerspective	229575856	NULL	1	0	U	USER_TABLE	2018-01-15 11:47:50.367	2018-01-15 11:47:50.383	0

解决 SQL Server 表中的中文乱码问题

背景信息

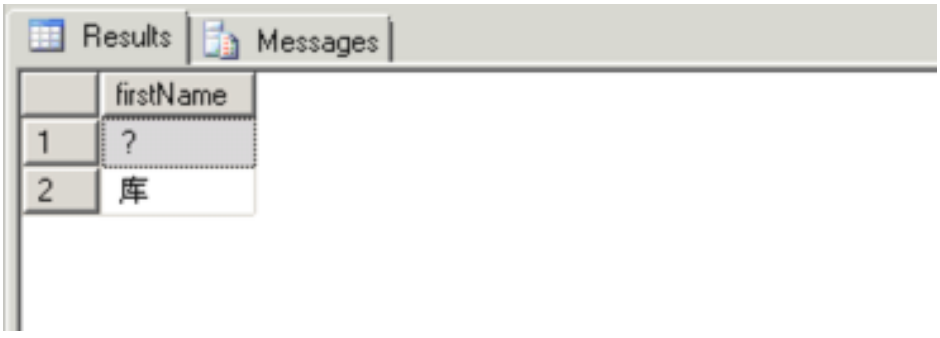
用户在查询 SQL Server 表中的生僻字时，查询结果出现乱码。本文将介绍该问题的原因以及解决方法。

问题复现示例

执行如下代码，查询 SQL Server 表中的生僻字“粟 (su)”。

```
use tempdb
go
IF OBJECT_ID('#temp', 'U') IS NOT NULL
DROP TABLE #temp
GO

create table #temp(
firstName varchar(10)
)

insert into #temp
select '粟'
union all
select '库'
;

select * from #temp
```

显示结果如下，“粟 (su)”字并未正确显示，而是出现了问号“?”。



原因分析

SQL Server 使用 Unicode 编码格式的数据类型（例如 NCHAR、NVARCHAR）来支持包含中文在内的亚洲语

言。在查询代码中，数据类型必须是 Unicode 编码的数据类型。但在上述示例代码中使用的数据类型是 VARCHAR，所以导致查询结果出现乱码。

解决方法

要解决在 SQL Server 的表中查询生僻字出现乱码的问题，只需要将上述示例代码中的数据类型改为 Unicode 编码格式的数据类型即可（下述示例中使用的是 NVARCHAR）。

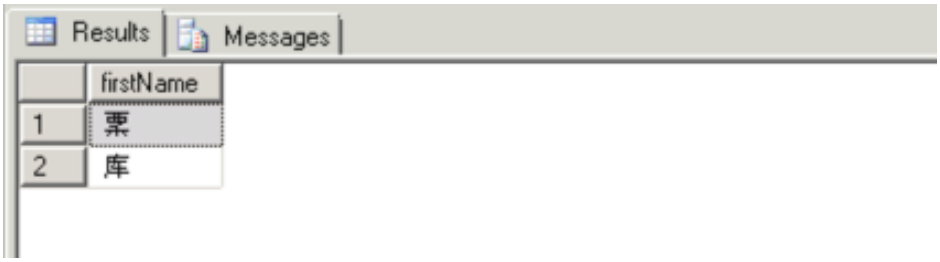
另外，为避免乱码问题，在向 Unicode 编码格式的数据类型插入数据时，需要使用前置词 N。前置词 N 代表的是 SQL-92 标准中的国家语言，且 N 必须大写。若您没有在 Unicode 字符串的常数前加 N 做为前置词，则 SQL Server 会在使用字符串之前将其转换成目前资料库的非 Unicode 字码页。

操作步骤

将上述示例中的数据类型 VARCHAR 改为 NVARCHAR，执行如下代码，查询 SQL Server 表中的生僻字“粟 (su)”。

```
use tempdb
go
IF OBJECT_ID('#temp', 'U') IS NOT NULL
DROP TABLE #temp
GO

create table #temp(
firstName nvarchar(10)
)

insert into #temp
select N'粟'
union all
select N'库'
;

select * from #temp
```

显示结果如下，“粟 (su)”字正确显示出来了。



使用SSMS和BCP迁移SQL Server数据库

本文以本地SQL Server数据库到阿里云云数据库SQL Server 2012的数据全量迁移为例，介绍了如何通过使用SQL Server Management Studio（SSMS）和大容量复制程序实用工具（BCP）来迁移SQL Server数据库。

适用场景

SQL Server数据库的结构迁移。

数据的全量迁移，不支持数据的增量迁移。

本地数据库到本地数据库、本地数据库到阿里云云数据库SQL Server和阿里云云数据库SQL Server间的数据全量迁移。

背景信息

SSMS 是用于管理SQL Server基础架构的集成环境，提供用于配置、监视和管理SQL Server实例的工具。 此外，它还提供了用于部署、监视和升级数据层组件（如应用程序使用的数据库和数据仓库）的工具以生成查询和脚本。

BCP 可以在SQL Server实例和用户指定格式的数据文件间大容量复制数据，您可以通过BCP实用工具将大量新行导入SQL Server表，或将表数据导出到数据文件。

所以，本文直接使用SSMS的功能来生成源端数据库对象结构的创建脚本，然后在目标数据库中去执行，进行数据库结构的迁移；再配合使用BCP命令行来进行数据库数据的导出和导入操作，进行数据的全量迁移。下面将介绍如何将本地SQL Server 2012数据库AdventureWorks2012的数据全量迁移到阿里云云数据库SQL Server 2012。

前提条件

目标数据库主机需要有充足的存储空间来存放导入的数据和因此而带来的日志文件增长，两者加起来的空间增长大概是源端数据库大小的2-3倍（如果数据库是Full模式）。如果目标数据库是在本地自建环境，请确保宿主机有足够的存储空间；如果是阿里云云数据库SQL Server，请确保已经购买了充足的存储空间。

注意事项

在创建目标数据库时，要确保目标数据库和源端数据库排序规则的一致性，否则很可能会导致全量数

据迁移失败。

为防止数据全量迁移过程报错，需要在目标数据库中禁用外键、索引和触发器，然后再启用，以此来避免错误发生和提高数据导入效率。

BCP导入计算列或时间戳（timestamp）列时，会忽略它们的列值，SQL Server将自动分配该列的值。

操作步骤

视频介绍

文本介绍

打开SSMS客户端。

分别连接源数据库AdventureWorks2012和阿里云云数据库SQL Server。

执行如下代码，在源数据库创建具有读写权限的用户。代码中的testdbo是指用户名称，XXXXXXXX是该用户的登录密码，在执行代码前请将这两个参数改成您想要的用户名和密码。若您的数据库中已存在具有读写权限的用户，请跳过此步骤。

```
USE MASTER
GO
CREATE LOGIN testdbo
WITH PASSWORD = N'XXXXXXXX',CHECK_POLICY = OFF
GO
USE AdventureWorks2012
GO

CREATE USER testdbo FOR LOGIN testdbo;

EXEC sys.sp_addrolemember 'db_owner','testdbo'
GO
```

禁用掉TCP/IP协议，以断开源数据库的所有客户端连接，以确保源端数据迁移前后的一致性。

重启SQL Server服务。

> **注意：**禁用掉TCP/IP协议后，远端应用程序将无法通过TCP/IP端口来访问本地实例，在使用BCP进行数据导出时，必须在该实例所在的物理机上进行。

执行如下命令，在阿里云云数据库SQL Server 2012的实例上创建数据库。

```
create database db01 DEFAULT CHARACTER SET gbk COLLATE gbk_chinese_ci;
```

执行如下代码，分别查看源端数据库和目标数据库的排序规则。

```
-- Check Collation name
SELECT name,collation_name
FROM sys.databases
WHERE name = 'adventureworks2012'
```

若结果显示的排序规则不同，在目标数据库执行如下代码，将SQL_Latin1_General_CP1_CI_AS替换成跟跟源端数据库一致的排序规则。若排序规则一致，请跳过此步骤。

```
-- change the collate if need.
USE master;
GO
ALTER DATABASE adventureworks2012
COLLATE SQL_Latin1_General_CP1_CI_AS;
GO
```

在目标数据库中执行如下代码，创建与步骤3中一致的源端数据库用户。

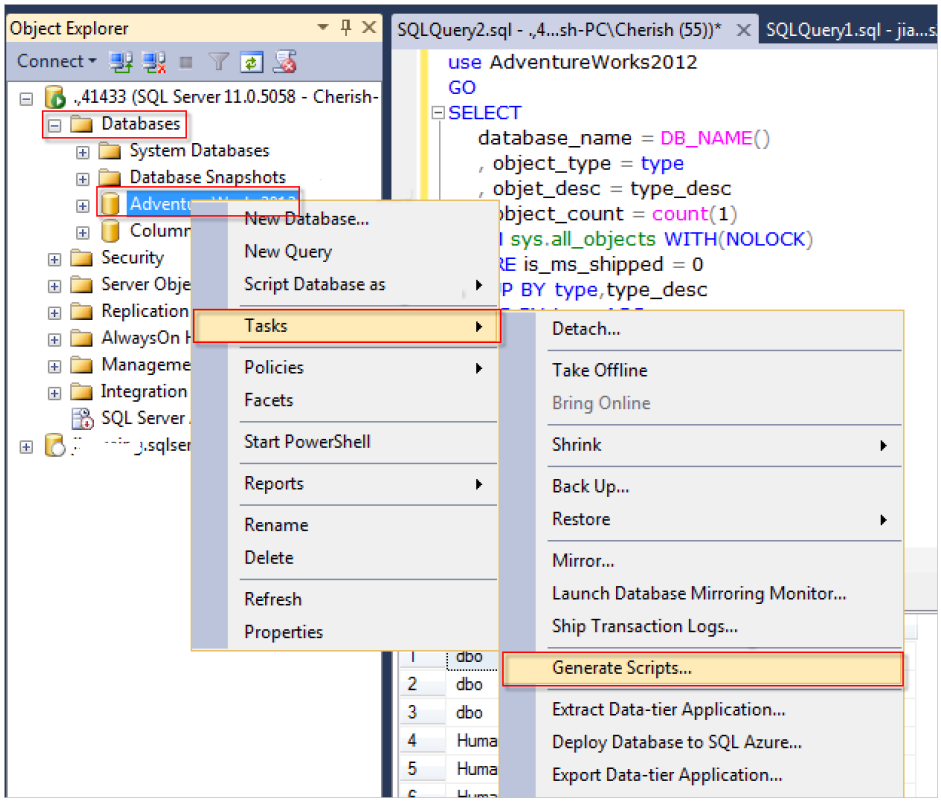
```
 USE MASTER
GO
CREATE LOGIN testdbo
WITH PASSWORD = N'XXXXXXXX',CHECK_POLICY = OFF
GO
USE AdventureWorks2012
GO

CREATE USER testdbo FOR LOGIN testdbo;

EXEC sys.sp_addrolemember 'db_owner','testdbo'
GO
```

在源端数据库生成对象信息创建脚本。操作步骤如下：

单击源数据库中的**Databases**。

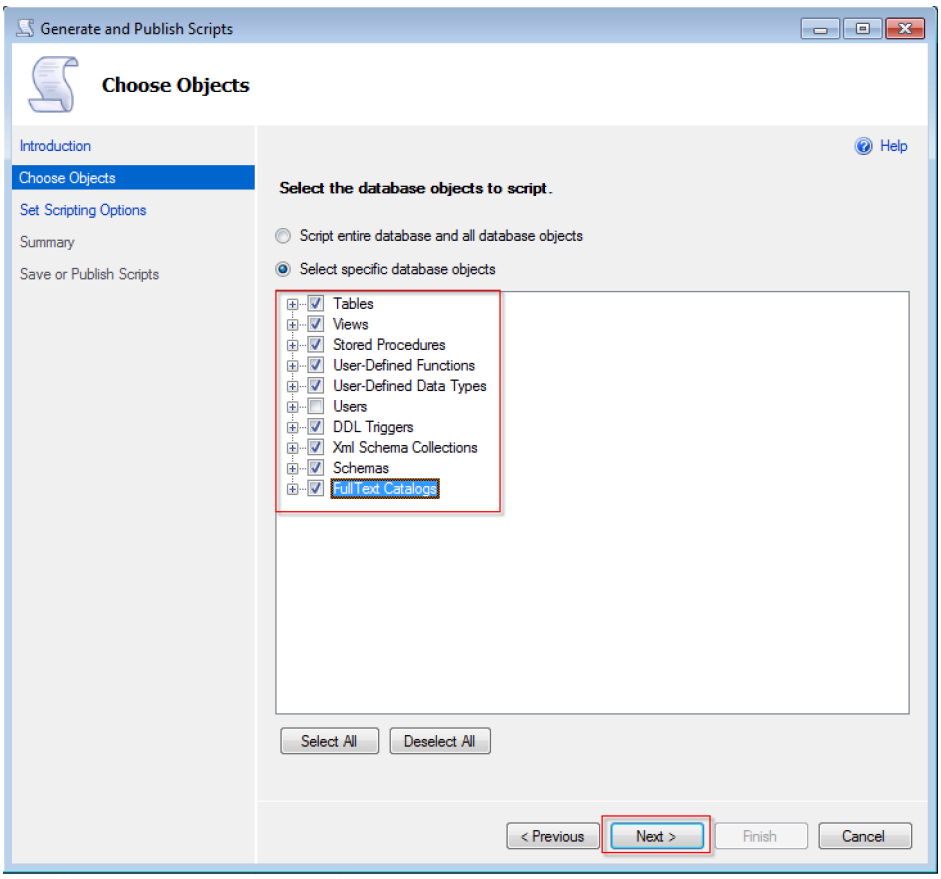
右击**AdventureWorks2012**。

选择**Tasks** > **Generate Scripts**，如下图所示。



进入生成脚本的界面后，单击 **Next**。

选中**Select specific database objects**，选择除**Users**以外所有的迁移对象，然后单击**Next**。如下图所示。



在**Set Scripting Options**页面，单击**Advanced**，然后进行如下设置：

单击**Script for Server Version**，将其值设成与目标数据库一致的数据库版本，本示例中使用的是SQL Server 2012.

单击**Script for the database engine edition**，将其值设成 Microsoft SQL Server Enterprise Edition。

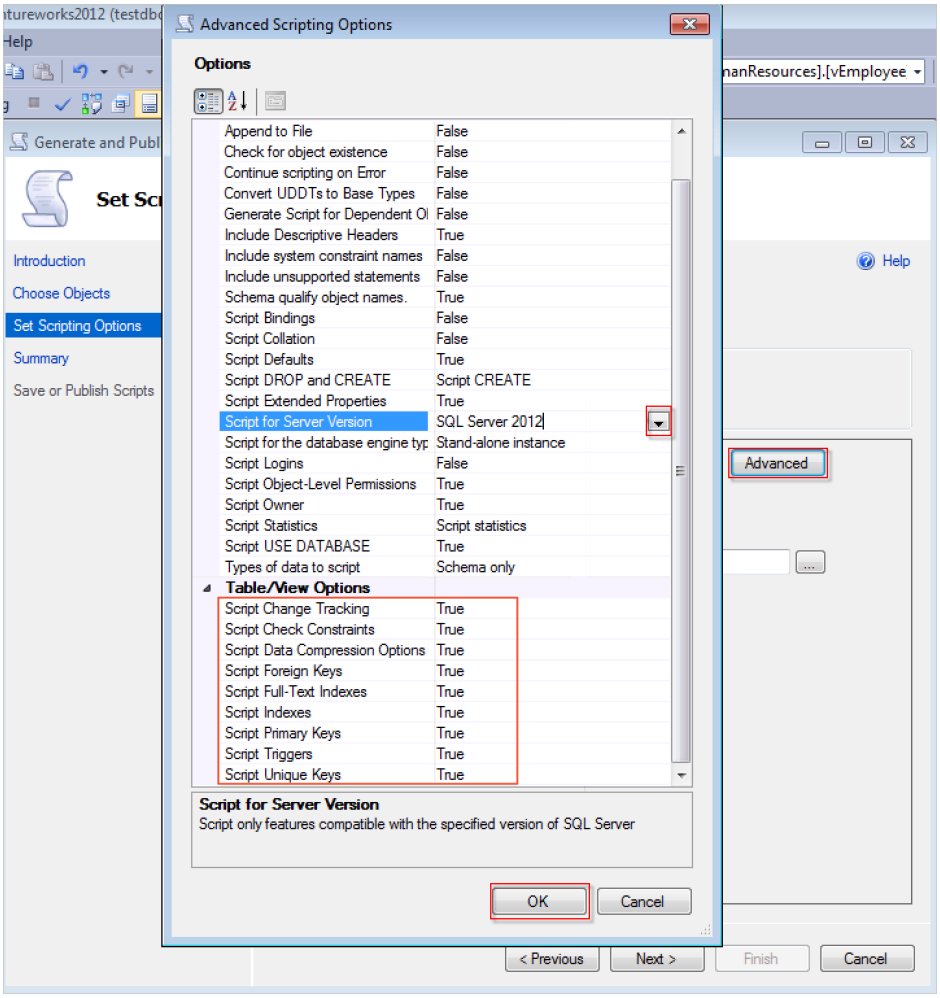
将**Script Object-Level Permissions**、**Script Owner** 和 **Script USE DATAABASE**的值设成True。

单击**Types of data to script**，将其值设成Schema only。

> **注意**：该选项不能设成Schema and data，否则会生成schema文件和INSERT语句文件，造成效率低下。

建议将**Table/View Options**中的所有选项值都设置成True。

详情如下图所示：



单击**OK**。导出的脚本文件会放在页面上显示的**File name**中所示的位置。

单击**Next**、**Next**、**Finish**，完成脚本生成工作。

在目标数据库中执行在上一步中创建的脚本文件。

当创建对象信息的脚本文件执行完毕后，执行如下代码，并将参数@is_disable BIT的值设成1，以禁用外键约束、索引和触发器。

> **说明**：表外键约束的存在会导致数据导入失败，而表索引和触发器的存在会导致数据导入效率降低，所以需要将其禁用。

```
 USE [adventureworks2012]
GO

--public variables: need init by users.
DECLARE
@is_disable BIT = 1 -- 1: disalbe indexes, foreign keys and triggers;
```

```
-- 0: enable indexes, foreign keys and triggers;
;

---------------- Private variables
DECLARE
@sql NVARCHAR(MAX)
, @sql_index NVARCHAR(MAX)
, @tb_schema SYSNAME
, @tb_object_name SYSNAME
, @tr_schema SYSNAME
, @tr_object_name SYSNAME
, @ix_name SYSNAME
;

----------------- Disable/Enable indexes on all tables
DECLARE
cur_indexes CURSOR LOCAL STATIC FORWARD_ONLY READ_ONLY
FOR
SELECT
ix_name = ix.name
, tb_schema = SCHEMA_NAME(obj.schema_id)
, tb_object_name = obj.name
FROM sys.indexes as ix
INNER JOIN sys.objects as obj
ON ix.object_id = obj.object_id
WHERE ix.type >= 2
AND obj.is_ms_shipped = 0
AND ix.is_disabled = (1 - @is_disable)

OPEN cur_indexes;
FETCH NEXT FROM cur_indexes INTO @ix_name, @tb_schema, @tb_object_name;

WHILE @@FETCH_STATUS = 0
BEGIN
SET
@sql_index = N'ALTER INDEX ' + QUOTENAME(@ix_name)
+ N' ON ' + QUOTENAME(@tb_schema) + N'.' + QUOTENAME(@tb_object_name)
+ CASE @is_disable
WHEN 1 THEN N' DISABLE;'
WHEN 0 THEN N' REBUILD; '
ELSE N''
END;
RAISERROR(N'%s', 10, 1, @sql_index) WITH NOWAIT;
EXEC sys.sp_executesql @sql_index
FETCH NEXT FROM cur_indexes INTO @ix_name, @tb_schema, @tb_object_name;
END

CLOSE cur_indexes;
DEALLOCATE cur_indexes;

----------------- Disable/Enable foreign keys on all tables
--disable
IF @is_disable = 1
BEGIN
SELECT
@sql = N'
```

```
RAISERROR(N''ALTER TABLE ? NOCHECK CONSTRAINT ALL;'', 10, 1) WITH NOWAIT
ALTER TABLE ? NOCHECK CONSTRAINT ALL;'
;
END
ELSE --enable
BEGIN
SELECT
@sql = N'
RAISERROR(N''ALTER TABLE ? WITH CHECK CHECK CONSTRAINT ALL;'', 10, 1) WITH NOWAIT
ALTER TABLE ? WITH CHECK CHECK CONSTRAINT ALL;'
;
END

EXEC sys.sp_MSforeachtable @sql

------------------ Disable/Enable triggers on all tables

DECLARE
cur_triggers CURSOR LOCAL STATIC FORWARD_ONLY READ_ONLY
FOR
SELECT
tb_schema = SCHEMA_NAME(tb.schema_id)
,tb_object_name = tb.name
,tr_schema = SCHEMA_NAME(obj.schema_id)
,tr_object_name = obj.name
FROM sys.objects as obj
INNER JOIN sys.tables as tb
ON obj.parent_object_id = tb.object_id
INNER JOIN sys.triggers as tr
ON obj.object_id = tr.object_id
WHERE obj.type = 'TR'
AND obj.is_ms_shipped = 0
AND tr.is_disabled = (1 - @is_disable)
ORDER BY tb_schema, tb_object_name

OPEN cur_triggers;
FETCH NEXT FROM cur_triggers INTO @tb_schema, @tb_object_name, @tr_schema, @tr_object_name;

WHILE @@FETCH_STATUS = 0
BEGIN
SET @sql = CASE @is_disable
WHEN 1 THEN N'DISABLE TRIGGER '
WHEN 0 THEN N'ENABLE TRIGGER '
ELSE N''
END
+ QUOTENAME(@tr_schema) + N'.' + QUOTENAME(@tr_object_name)
+ N' ON '
+ QUOTENAME(@tb_schema) + N'.' + QUOTENAME(@tb_object_name)
;
RAISERROR(N'%s', 10, 1, @sql) WITH NOWAIT;
EXEC sys.sp_executesql @sql;
FETCH NEXT FROM cur_triggers INTO @tb_schema, @tb_object_name, @tr_schema, @tr_object_name;
END

CLOSE cur_triggers;
DEALLOCATE cur_triggers;
```

```
GO
```

在源端数据库和目标数据库中分别执行如下代码，查询数据库的对象汇总信息。

```
 USE AdventureWorks2012
GO
;WITH objs
AS(
-- check objects
SELECT
database_name = LOWER(DB_NAME())
, object_type = type
, objet_desc = type_desc
, object_count = COUNT(1)
FROM sys.all_objects WITH(NOLOCK)
WHERE is_ms_shipped = 0
GROUP BY type,type_desc
UNION ALL

--check indexes
SELECT
database_name = LOWER(DB_NAME())
, object_type = CAST(ix.type AS VARCHAR)
, objet_desc = ix.type_desc
, object_count = COUNT(1)
FROM sys.indexes as ix
INNER JOIN sys.objects as obj
ON ix.object_id = obj.object_id
WHERE obj.is_ms_shipped = 0
GROUP BY ix.type,ix.type_desc
)
SELECT * FROM objs
ORDER BY object_type
```

查询结果返回后，对比源端数据库和目标数据库的对象信息。若结果显示一致，则说明所有对象信息已经从源数据库迁移到了目标数据库。

注意：

为方便对比和避免人眼观察带来的人为错误，建议您使用对比工具来对比对象汇总信息。本文推荐您使用Araxis Merge 2001 v6.0 Professional。

阿里云云数据库SQL Server数据库的名称仅支持小写字母，而源数据库名称可能含有大写字母，在使用工具进行对比时，请将其设置为忽略字母大小写的检查。若您使用的是Araxis Merge 2001 v6.0 Professional，可以通过选择**View** > **Options**，然后再选中**Ignore differences in character case**来设置。

在源端数据库上执行如下脚本，执行前请按实际情况修改如下参数：

source_User：源数据库中的登录用户名。

source_Password：登录源数据库的用户名所对应的密码。

destination_Instance：将 XXXX 改成目标数据库的实例名称。

destination_Database：目标端的数据库名称。若为空，则表示与源端数控保持一致。

destination_User：目标数据库中登录的用户名，与源端数据库一致。

destination_Password：登录目标数据库中的用户名所对应的密码。

```
USE AdventureWorks2012
GO

-- declare public variables, need to init by user
DECLARE
@source_Instance sysname
, @source_Database sysname
, @source_User sysname
, @source_Passwd sysname

, @destination_Instance sysname
, @destination_Database sysname
, @destination_User sysname
, @destination_Passwd sysname

, @batch_Size int

, @transfer_table_list nvarchar(max)
;

-- Public variables init.
SELECT
@source_Instance = @@SERVERNAME -- Source Instance Name
, @source_Database = DB_NAME() -- Source Database is current database.
, @source_User = 'XXX' -- Source Instance Connect User Name
, @source_Passwd = N'XXX' -- Source Instance User Password

, @destination_Instance = N'XXXX.sqlserver.rds.aliyuncs.com,3433' -- Destination Instance Name
, @destination_Database = N'' -- Destination Database name: NULL/empty: Keep the same as source db
, @destination_User = 'XXX' -- Destination Instance User Name
, @destination_Passwd = N'XXX' -- Destination Instance User Password

, @transfer_table_list = N'' --NULL/empty: ALL Tables are needed to be transfered.
, @batch_Size = 50000 -- BCP IN Batch Size, by default, it is 50000. Must between 1 and 50000.
;
```

```
-- Private variables, there is no need to init.
DECLARE
@transfer_table_list_xml xml
, @timestamp char(14)
;

-- correct the variables init by user.
SELECT
@source_Instance = RTRIM( LTRIM(@source_Instance) )
, @source_User = RTRIM( LTRIM( @source_User ) )
, @source_Passwd = RTRIM( LTRIM( @source_Passwd ) )

, @destination_Instance = RTRIM( LTRIM( @destination_Instance ) )
, @destination_Database = CASE
WHEN ISNULL(@destination_Database, N'') = N'' THEN @source_Database
ELSE @destination_Database
END
, @destination_User = RTRIM( LTRIM( @destination_User ) )
, @destination_Passwd = RTRIM( LTRIM( @destination_Passwd ) )

, @batch_Size = CASE
WHEN (@batch_Size>0 AND @batch_Size<=50000) THEN @batch_Size
ELSE 50000
END
, @transfer_table_list_xml = '<V><![CDATA[' + REPLACE(
REPLACE(
REPLACE(
@transfer_table_list,CHAR(10),']]></V><V><![CDATA['
),',',']]></V><V><![CDATA['
),CHAR(13),']]></V><V><![CDATA['
) + ']]></V>'
, @timestamp =
REPLACE(
REPLACE(
REPLACE(
CONVERT(CHAR(19), GETDATE(), 120), N'-', '')
, N':', N'')
, CHAR(32), N'')
;

IF OBJECT_ID('tempdb..#tb_list', 'U') IS NOT NULL
DROP TABLE #tb_list
CREATE TABLE #tb_list(
RowID INT IDENTITY(1, 1) NOT NULL PRIMARY KEY
,Table_name SYSNAME NOT NULL
)

IF ISNULL(@transfer_table_list, '') = ''
BEGIN
INSERT INTO #tb_list
SELECT name
FROM sys.tables AS tb
WHERE tb.is_ms_shipped = 0
END
ELSE
BEGIN
```

```
INSERT INTO #tb_list
SELECT table_name = T.C.value('(./text())[1]','sysname')
FROM @transfer_table_list_xml.nodes('./V') AS T(C)
WHERE T.C.value('(./text())[1]','sysname') IS NOT NULL
END
;

SELECT
BCP_OUT = N'BCP ' + @source_Database + '.' + sch.name + '.' + tb.name
+ N' Out '
+ QUOTENAME( REPLACE(@source_Instance, N'\', N'_')+ '.' + @source_Database + '.' + sch.name + '.' +
tb.name, '"')
+ N' /N /U ' + @source_User +N' /P ' + @source_Passwd +N' /S ' + @source_Instance
+ N' >> BCPOUT_' + @timestamp +N'.txt'
,BCP_IN = N'BCP ' + @destination_Database + '.' + sch.name + '.' + tb.name
+ N' In '
+ QUOTENAME( REPLACE(@source_Instance, N'\', N'_')+ '.' + @source_Database + '.' + sch.name + '.' +
tb.name, '"')
+ N' /N /E /q /k /U ' + @destination_User + N' /P ' + @destination_Passwd + N' /b '
+ CAST(@batch_Size as varchar) + N' /S ' + @destination_Instance
+ N' >> BCPIN_' + @timestamp + N'.txt'
--,*
FROM sys.tables as tb
LEFT JOIN sys.schemas as sch
ON tb.schema_id = sch.schema_id
WHERE tb.is_ms_shipped = 0
AND tb.name IN (SELECT Table_name FROM #tb_list)
```

分别将生成的BCP_OUT和BCP_IN文件保存至本地。

运行本地保存的BCP_OUT.bat文件，生成源数据库的数据导出文件。

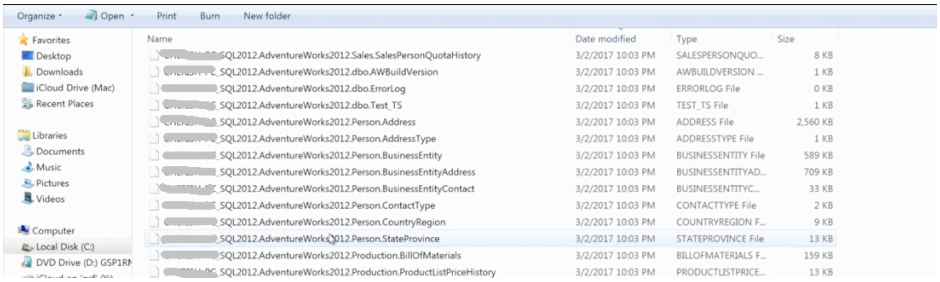
BCP_OUT.bat文件运行结束后，双击其日志文件**BCPOUT_YYYYMMDDHHMMSS.txt**（YYYYMMDDHHMMSS为文件生成时间），查看数据导出过程是否全部成功。

运行本地保存的BCP_IN.bat文件，将从源数据库中导出的文件从本地导入到远端目标数据库中。

> **说明**：由于目标数据库是在阿里云上，由于网络原因，BCP_IN.bat文件的运行时间会比BCP_OUT.bat文件的运行时间长。

BCP_IN.bat文件运行结束后，双击其日志文件**BCPOUT_YYYYMMDDHHMMSS.txt**，查看数据导入过程是否全部成功。

若确保数据已经全部从源端数据库导入到目标数据库中，删除磁盘上BCP在步骤 17中导出的中间临时文件，如下图所示。



分别在源端数据库和目标数据库中执行如下代码，查询源端数据库和目标数据库中的表记录总数。

```
USE AdventureWorks2012
GO
SELECT
schema_name = SCHEMA_NAME(tb.schema_id)
,table_name = OBJECT_NAME(tb.object_id)
,row_count = SUM(CASE WHEN ps.index_id < 2 THEN ps.row_count ELSE 0 END)
FROM sys.dm_db_partition_stats as ps WITH(NOLOCK)
INNER JOIN sys.tables as tb WITH(NOLOCK)
ON ps.object_id = tb.object_id
WHERE tb.is_ms_shipped = 0
GROUP BY tb.object_id,tb.schema_id
ORDER BY schema_name,table_name
```

使用对比工具对比源端数据库和目标数据库中的表记录总数。若数据一致，则说明源数据库的所有数据已经全部导入目标数据库中。

执行如下代码，并将参数@is_disable BIT的值设成0，启用外键约束、索引和触发器，完成数据库迁移。

```
 USE [adventureworks2012]
GO

--public variables: need init by users.
DECLARE
@is_disable BIT = 1 -- 1: disalbe indexes, foreign keys and triggers;
-- 0: enable indexes, foreign keys and triggers;
;

---------------------- Private variables
DECLARE
@sql NVARCHAR(MAX)
, @sql_index NVARCHAR(MAX)
, @tb_schema SYSNAME
, @tb_object_name SYSNAME
, @tr_schema SYSNAME
, @tr_object_name SYSNAME
, @ix_name SYSNAME
;
```

```
------------------- Disable/Enable indexes on all tables
DECLARE
cur_indexes CURSOR LOCAL STATIC FORWARD_ONLY READ_ONLY
FOR
SELECT
ix_name = ix.name
, tb_schema = SCHEMA_NAME(obj.schema_id)
, tb_object_name = obj.name
FROM sys.indexes as ix
INNER JOIN sys.objects as obj
ON ix.object_id = obj.object_id
WHERE ix.type >= 2
AND obj.is_ms_shipped = 0
AND ix.is_disabled = (1 - @is_disable)

OPEN cur_indexes;
FETCH NEXT FROM cur_indexes INTO @ix_name, @tb_schema, @tb_object_name;

WHILE @@FETCH_STATUS = 0
BEGIN
SET
@sql_index = N'ALTER INDEX ' + QUOTENAME(@ix_name)
+ N' ON ' + QUOTENAME(@tb_schema) + N'.' + QUOTENAME(@tb_object_name)
+ CASE @is_disable
WHEN 1 THEN N' DISABLE;'
WHEN 0 THEN N' REBUILD; '
ELSE N''
END;
RAISERROR(N'%s', 10, 1, @sql_index) WITH NOWAIT;
EXEC sys.sp_executesql @sql_index
FETCH NEXT FROM cur_indexes INTO @ix_name, @tb_schema, @tb_object_name;
END

CLOSE cur_indexes;
DEALLOCATE cur_indexes;

------------------- Disable/Enable foreign keys on all tables
--disable
IF @is_disable = 1
BEGIN
SELECT
@sql = N'
RAISERROR(N''ALTER TABLE ? NOCHECK CONSTRAINT ALL;'', 10, 1) WITH NOWAIT
ALTER TABLE ? NOCHECK CONSTRAINT ALL;'
;
END
ELSE --enable
BEGIN
SELECT
@sql = N'
RAISERROR(N''ALTER TABLE ? WITH CHECK CHECK CONSTRAINT ALL;'', 10, 1) WITH NOWAIT
ALTER TABLE ? WITH CHECK CHECK CONSTRAINT ALL;'
;
END

EXEC sys.sp_MSforeachtable @sql
```

```
--================== Disable/Enable triggers on all tables

DECLARE
cur_triggers CURSOR LOCAL STATIC FORWARD_ONLY READ_ONLY
FOR
SELECT
tb_schema = SCHEMA_NAME(tb.schema_id)
,tb_object_name = tb.name
,tr_schema = SCHEMA_NAME(obj.schema_id)
,tr_object_name = obj.name
FROM sys.objects as obj
INNER JOIN sys.tables as tb
ON obj.parent_object_id = tb.object_id
INNER JOIN sys.triggers as tr
ON obj.object_id = tr.object_id
WHERE obj.type = 'TR'
AND obj.is_ms_shipped = 0
AND tr.is_disabled = (1 - @is_disable)
ORDER BY tb_schema, tb_object_name

OPEN cur_triggers;
FETCH NEXT FROM cur_triggers INTO @tb_schema, @tb_object_name, @tr_schema, @tr_object_name;

WHILE @@FETCH_STATUS = 0
BEGIN
SET @sql = CASE @is_disable
WHEN 1 THEN N'DISABLE TRIGGER '
WHEN 0 THEN N'ENABLE TRIGGER '
ELSE N''
END
+ QUOTENAME(@tr_schema) + N'.' + QUOTENAME(@tr_object_name)
+ N' ON '
+ QUOTENAME(@tb_schema) + N'.' + QUOTENAME(@tb_object_name)
;
RAISERROR(N'%s', 10, 1, @sql) WITH NOWAIT;
EXEC sys.sp_executesql @sql;
FETCH NEXT FROM cur_triggers INTO @tb_schema, @tb_object_name, @tr_schema, @tr_object_name;
END

CLOSE cur_triggers;
DEALLOCATE cur_triggers;
GO
```

PostgreSQL

CREATE DATABASE 命令的具体使用方法

PostgreSQL 实例支持创建多个数据库。创建数据库时，您可以指定模板库，并为每个数据库设置不同的字符集、本地化 collate、货币等属性。

本文将通过使用示例来介绍如何通过CREATE DATABASE命令来设置模板库、字符集、LC_COLLATE 及 LC_CTYPE 的信息。

CREATE DATABASE 命令的语法

```
Command:     CREATE DATABASE
Description: create a new database
Syntax:
CREATE DATABASE name
[ [ WITH ] [ OWNER [=] user_name ]
[ TEMPLATE [=] template ]
[ ENCODING [=] encoding ]
[ LC_COLLATE [=] lc_collate ]
[ LC_CTYPE [=] lc_ctype ]
[ TABLESPACE [=] tablespace_name ]
[ CONNECTION LIMIT [=] connlimit ] ]
```

克隆模板库

克隆模板库是指从指定模板创建数据库。如下示例将介绍如何通过CREATE DATABASE命令从指定模板创建数据库。

使用示例

以 test 数据库为模板，克隆一个名为 test01 的数据库，其命令如下所示：

```
test=> create database test01 with template test;
CREATE DATABASE
```

注意：

若不指定模板，则默认的模板为 template1。

从指定模板库创建数据库时，除了当前连接，不能有其他用户连在对应的模板库上面。例如，若有其他用户连在 test 数据库上，可能会报类似如下的错误：

```
 test=> create database test01 with template test;
ERROR: source database "test" is being accessed by other users
DETAIL: There is 1 other session using the database.
```

指定字符集类型

创建数据库时，您可以指定 encoding。如下示例将介绍如何通过CREATE DATABASE命令创建指定字符集类型的数据库。

PostgreSQL 支持的字符集类型

您可以通过 PostgreSQL 的官方文档或下表查看对应的字符集支持列表，Server=Yes 表示该字符集支持用于CREATE DATABASE命令，Server=No 表示只支持作为客户端字符集。

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	-
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	-
EUC_JIS_2004	Extended UNIX Code-JP, JIS X 0213	Japanese	Yes	1-3	-
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	-
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	-
GB18030	National Standard	Chinese	No	1-4	-
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	-
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	-
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	-

ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	-
JOHAB	JOHAB	Korean (Hangul)	No	1-3	-
KOI8R	KOI8-R	Cyrillic (Russian)	Yes	1	KOI8
KOI8U	KOI8-U	Cyrillic (Ukrainian)	Yes	1	-
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	-
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	Japanese	No	1-2	-
SQL_ASCII	unspecified (see text)	any	Yes	1	-
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949

UTF8	Unicode, 8-bit	all	Yes	1-4	Unicode
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	-
WIN1250	Windows CP1250	Central European	Yes	1	-
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	-
WIN1253	Windows CP1253	Greek	Yes	1	-
WIN1254	Windows CP1254	Turkish	Yes	1	-
WIN1255	Windows CP1255	Hebrew	Yes	1	-
WIN1256	Windows CP1256	Arabic	Yes	1	-
WIN1257	Windows CP1257	Baltic	Yes	1	-
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

使用示例

创建一个字符集为 UTF-8 的数据库，其命令如下所示：

```
test=> create database test02 with encoding 'UTF-8';
CREATE DATABASE
```

注意：

指定的字符集必须是模板库字符集的超集，否则会报错。

指定的 LC_COLLATE 和 LC_CTYPE 必须与目标字符集兼容，否则会报错。

报错示例：template1 是默认模板库，它的字符集为 UTF8，如下所示。

```
test=> \l template1
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges
-----------+----------+----------+-------------+-------------+-----------------------
template1 | xxxxxxxx | UTF8 | zh_CN.UTF-8 | zh_CN.UTF-8 | =c/xxxxxxxx +
| | | | | xxxxxxxx=CTc/xxxxxxxx
(1 row)
```

当使用上述 template1 作为模板创建一个字符集为 EUC_CN 的数据库时，会出现如下错误：

EUC_CN 字符集与模板库的 LC_COLLATE 和 LC_CTYPE 不兼容。

```
test=> create database test03 with encoding 'EUC_CN';
ERROR: encoding "EUC_CN" does not match locale "zh_CN.UTF-8"
DETAIL: The chosen LC_CTYPE setting requires encoding "UTF8".
```

EUC_CN 字符集与模板库的字符集 UTF-8 不兼容。

```
test=> create database test03 with encoding 'EUC_CN' lc_collate='C' lc_ctype='C';
ERROR: new encoding (EUC_CN) is incompatible with the encoding of the template database (UTF8)
HINT: Use the same encoding as in the template database, or use template0 as template.
```

解决方法示例：使用 template0 作为模板库，即可解决上述报错的问题。

```
create database test03 with encoding 'EUC_CN' template template0;
```

设置 LC_COLLATE 和 LC_CTYPE 的信息

本示例将介绍如何查询字符集支持的 LC_COLLATE 和 LC_CTYPE、如何通过CREATE DATABASE命令指定 LC_COLLATE 和 LC_CTYPE 以及如何修改已有数据库的 LC_COLLATE 和 LC_CTYPE。

查询字符集支持的 LC_COLLATE 和 LC_CTYPE 信息

您可以使用如下 SQL 查询系统表 pg_collation，来获取字符集支持的LC_COLLATE 和 LC_CTYPE 信息。

```
test=> select pg_encoding_to_char(collencoding) as encoding,collname,collcollate,collctype from pg_collation ;
```

返回结果如下所示，encoding 为空时，表示这个 collation 支持所有的字符集。

```
 encoding |      collname       |     collcollate     |      collctype
------------+---------------------+---------------------+---------------------
| default | |
```

```
| C | C | C
| POSIX | POSIX | POSIX
UTF8 | aa_DJ | aa_DJ.utf8 | aa_DJ.utf8
LATIN1 | aa_DJ | aa_DJ | aa_DJ
LATIN1 | aa_DJ.iso88591 | aa_DJ.iso88591 | aa_DJ.iso88591
UTF8 | aa_DJ.utf8 | aa_DJ.utf8 | aa_DJ.utf8
UTF8 | aa_ER | aa_ER | aa_ER
UTF8 | aa_ER.utf8 | aa_ER.utf8 | aa_ER.utf8
.......
EUC_CN | zh_CN | zh_CN | zh_CN
UTF8 | zh_CN | zh_CN.utf8 | zh_CN.utf8
EUC_CN | zh_CN.gb2312 | zh_CN.gb2312 | zh_CN.gb2312
UTF8 | zh_CN.utf8 | zh_CN.utf8 | zh_CN.utf8
UTF8 | zh_HK | zh_HK.utf8 | zh_HK.utf8
UTF8 | zh_HK.utf8 | zh_HK.utf8 | zh_HK.utf8
EUC_CN | zh_SG | zh_SG | zh_SG
UTF8 | zh_SG | zh_SG.utf8 | zh_SG.utf8
EUC_CN | zh_SG.gb2312 | zh_SG.gb2312 | zh_SG.gb2312
UTF8 | zh_SG.utf8 | zh_SG.utf8 | zh_SG.utf8
EUC_TW | zh_TW | zh_TW.euctw | zh_TW.euctw
UTF8 | zh_TW | zh_TW.utf8 | zh_TW.utf8
EUC_TW | zh_TW.euctw | zh_TW.euctw | zh_TW.euctw
UTF8 | zh_TW.utf8 | zh_TW.utf8 | zh_TW.utf8
UTF8 | zu_ZA | zu_ZA.utf8 | zu_ZA.utf8
LATIN1 | zu_ZA | zu_ZA | zu_ZA
LATIN1 | zu_ZA.iso88591 | zu_ZA.iso88591 | zu_ZA.iso88591
UTF8 | zu_ZA.utf8 | zu_ZA.utf8 | zu_ZA.utf8
(869 rows)
```

创建数据库时指定 LC_COLLATE 和 LC_CTYPE

使用示例

创建一个 LC_COLLATE 和 LC_CTYPE 分别为 zh_CN.utf8 的数据库，其命令如下所示：

```
test=> create database test05 with encoding 'UTF-8' template template0 lc_collate='zh_CN.utf8'
lc_ctype='zh_CN.utf8';
CREATE DATABASE
```

注意:

若指定的 LC_COLLATE 和 LC_CTYPE 与模板库的 LC_COLLATE 和 LC_CTYPE 不兼容，会出现如下错误：

```
test=> create database test04 with encoding 'UTF-8' lc_collate='zh_CN.utf8' lc_ctype='zh_CN.utf8';
ERROR: new collation (zh_CN.utf8) is incompatible with the collation of the template database (zh_CN.UTF-8)
HINT: Use the same collation as in the template database, or use template0 as template.
```

出现上述错误时，有如下两种解决方法：

使用兼容的 LC_COLLATE 和 LC_CTYPE，其命令如下所示：

```
test=> create database test04 with encoding 'UTF-8' lc_collate='zh_CN.UTF-8' lc_ctype='zh_CN.UTF-8';
CREATE DATABASE
```

使用 template0 作为模板库，其命令如下所示：

```
test=> create database test05 with encoding 'UTF-8' template template0 lc_collate='zh_CN.utf8'
lc_ctype='zh_CN.utf8';
CREATE DATABASE
```

修改已有数据库的 LC_COLLATE 和 LC_CTYPE 信息

目前，您无法直接通过 alter database 命令修改已有数据库的 LC_COLLATE 和 LC_CTYPE 信息，但可以通过创建新的数据库，然后导出再导入数据的方式进行修改。

操作步骤

创建新数据库，指定目标 LC_COLLATE 和 LC_CTYPE。

使用 pg_dump 或其它客户端工具逻辑导出源数据库的数据。

使用 pg_restore 或其它客户端工具，将第 2 步导出的数据导入新数据库。

参考文档

PostgreSQL 9.6.2 Documentation — CREATE DATABASE

设置数据库的本土化信息及字符串排序规则

初始化数据库集群时，可以设置如下参数，用于设置数据库的字符串排序、字符归类方法、数值\日期\时间\货币的格式等。另外，为了支持国际化，数据库通常会涉及到 LC_COLLATE 和 LC_CTYPE 的概念。

LC_COLLATE	String sort order
LC_CTYPE	Character classification (What is a letter? Its upper-case equivalent?)
LC_MESSAGES	Language of messages

LC_MONETARY	Formatting of currency amounts
LC_NUMERIC	Formatting of numbers
LC_TIME	Formatting of dates and times

您可以利用这些特性，按本土化需求，输出对应的顺序或者格式。本文将通过示例介绍如何设置数据库的本土化信息以及如何设置输出结果按中文的拼音顺序进行排序。

PostgreSQL 支持的字符集类型

您可以通过 PostgreSQL 的官方文档或下表查看对应的字符集支持列表，Server=Yes 表示该字符集支持用于CREATE DATABASE命令，Server=No 表示只支持作为客户端字符集。

Name	Description	Language	Server?	Bytes/Char	Aliases
BIG5	Big Five	Traditional Chinese	No	1-2	WIN950, Windows950
EUC_CN	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	-
EUC_JP	Extended UNIX Code-JP	Japanese	Yes	1-3	-
EUC_JIS_2004	Extended UNIX Code-JP, JIS X 0213	Japanese	Yes	1-3	-
EUC_KR	Extended UNIX Code-KR	Korean	Yes	1-3	-
EUC_TW	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	-
GB18030	National Standard	Chinese	No	1-4	-
GBK	Extended National Standard	Simplified Chinese	No	1-2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	-
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	-
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	-
ISO_8859_8	ISO 8859-8,	Latin/Hebre	Yes	1	-

	ECMA 121	w			
JOHAB	JOHAB	Korean (Hangul)	No	1-3	-
KOI8R	KOI8-R	Cyrillic (Russian)	Yes	1	KOI8
KOI8U	KOI8-U	Cyrillic (Ukrainian)	Yes	1	-
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	-
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	Japanese	No	1-2	-
SQL_ASCII	unspecified (see text)	any	Yes	1	-
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-	all	Yes	1-4	Unicode

	bit				
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	-
WIN1250	Windows CP1250	Central European	Yes	1	-
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	-
WIN1253	Windows CP1253	Greek	Yes	1	-
WIN1254	Windows CP1254	Turkish	Yes	1	-
WIN1255	Windows CP1255	Hebrew	Yes	1	-
WIN1256	Windows CP1256	Arabic	Yes	1	-
WIN1257	Windows CP1257	Baltic	Yes	1	-
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

查询字符集支持的 LC_COLLATE 和 LC_CTYPE 信息

您可以使用如下 SQL 查询系统表 pg_collation，来获取字符集支持的 LC_COLLATE 和 LC_CTYPE 信息。

```
test=> select pg_encoding_to_char(collencoding) as encoding,collname,collcollate,collctype from pg_collation ;
```

返回结果如下所示，encoding 为空时，表示这个 collation 支持所有的字符集。

```
 encoding |      collname       |     collcollate     |      collctype
------------+---------------------+---------------------+---------------------
| default | |
| C | C | C
| POSIX | POSIX | POSIX
UTF8 | aa_DJ | aa_DJ.utf8 | aa_DJ.utf8
LATIN1 | aa_DJ | aa_DJ | aa_DJ
LATIN1 | aa_DJ.iso88591 | aa_DJ.iso88591 | aa_DJ.iso88591
UTF8 | aa_DJ.utf8 | aa_DJ.utf8 | aa_DJ.utf8
UTF8 | aa_ER | aa_ER | aa_ER
UTF8 | aa_ER.utf8 | aa_ER.utf8 | aa_ER.utf8
```

```
.....
EUC_CN | zh_CN | zh_CN | zh_CN
UTF8 | zh_CN | zh_CN.utf8 | zh_CN.utf8
EUC_CN | zh_CN.gb2312 | zh_CN.gb2312 | zh_CN.gb2312
UTF8 | zh_CN.utf8 | zh_CN.utf8 | zh_CN.utf8
UTF8 | zh_HK | zh_HK.utf8 | zh_HK.utf8
UTF8 | zh_HK.utf8 | zh_HK.utf8 | zh_HK.utf8
EUC_CN | zh_SG | zh_SG | zh_SG
UTF8 | zh_SG | zh_SG.utf8 | zh_SG.utf8
EUC_CN | zh_SG.gb2312 | zh_SG.gb2312 | zh_SG.gb2312
UTF8 | zh_SG.utf8 | zh_SG.utf8 | zh_SG.utf8
EUC_TW | zh_TW | zh_TW.euctw | zh_TW.euctw
UTF8 | zh_TW | zh_TW.utf8 | zh_TW.utf8
EUC_TW | zh_TW.euctw | zh_TW.euctw | zh_TW.euctw
UTF8 | zh_TW.utf8 | zh_TW.utf8 | zh_TW.utf8
UTF8 | zu_ZA | zu_ZA.utf8 | zu_ZA.utf8
LATIN1 | zu_ZA | zu_ZA | zu_ZA
LATIN1 | zu_ZA.iso88591 | zu_ZA.iso88591 | zu_ZA.iso88591
UTF8 | zu_ZA.utf8 | zu_ZA.utf8 | zu_ZA.utf8
(869 rows)
```

设置数据库的本土化（collate）信息

关于如何设置字符集、LC_COLLATE 及 LC_CTYPE 的信息，请参见文档 CREATE DATABASE 命令的具体使用方法。

设置字段的本土化

前提条件

执行如下 SQL 命令，查询当前数据库的字符集（encoding）类型，并了解清楚与您当前数据库字符集兼容的 collate。

```
postgres=# select datname,pg_encoding_to_char(encoding) as encoding from pg_database;
```

返回结果如下所示：

```
    datname      | encoding
--------------------+-----------
template1 | UTF8
template0 | UTF8
db | SQL_ASCII
db1 | EUC_CN
contrib_regression | UTF8
test01 | UTF8
test02 | UTF8
postgres | UTF8
(8 rows)
```

操作步骤

在创建表时，执行如下命令，指定兼容当前字符集的 collate。

```
CREATE TABLE test1 (
a text COLLATE "de_DE",
b text COLLATE "es_ES",
...
);
```

执行如下命令，修改列 collate。

注意：修改列 collate 时，会导致 rewrite table，大表请谨慎操作。

```
alter table a alter c1 type text COLLATE "zh_CN";
```

在 SQL 使用本土化

使用本土化，改变 order by 输出排序。

```
 test=# select * from a order by c1 collate "C";
c1
--------
刘少奇
刘德华
(2 rows)

test=# select * from a order by c1 collate "zh_CN";
c1
--------
刘德华
刘少奇
(2 rows)
```

使用本土化，改变操作符的结果。

```
  test=# select * from a where c1 > '刘少奇' collate "C";
c1
--------
刘德华
(1 row)

test=# select * from a where c1 > '刘少奇' collate "zh_CN";
```

```
c1
----
(0 rows)
```

使用本土化索引进行排序

排序语句中的 collate 与索引的 collate 保持一致，才能使用这个索引进行排序。

```
postgres=# create index idxa on a(c1 collate "zh_CN");
CREATE INDEX

postgres=# explain select * from a order by c1 collate "zh_CN";
QUERY PLAN
------------------------------------------------------------------------
Index Only Scan using idxa on a (cost=0.15..31.55 rows=1360 width=64)
(1 row)
```

设置输出结果按拼音排序

您可以通过如下四种方法来设置按拼音排序：

使用本土化 SQL。该方法不修改原有数据。

```
 test=# select * from a order by c1 collate "zh_CN";
c1
--------
刘德华
刘少奇
(2 rows)
```

使用本土化字段。若已有数据，使用该方法时需要调整原有数据。

```
alter table a alter c1 type text COLLATE "zh_CN";
```

使用本土化索引以及本土化 SQL。该方法不修改原有数据。

```
 postgres=# create index idxa on a(c1 collate "zh_CN");
CREATE INDEX

postgres=# explain select * from a order by c1 collate "zh_CN";
QUERY PLAN
------------------------------------------------------------------------
Index Only Scan using idxa on a (cost=0.15..31.55 rows=1360 width=64)
(1 row)
```

将数据库的 collate 设置为 zh_CN，数据会将默认使用这个 collate 按拼音排序。

```
 test02=# create database test03 encoding 'UTF8' lc_collate 'zh_CN.utf8' lc_ctype 'zh_CN.utf8'    template
template0;
CREATE DATABASE

test02=# \c test03
You are now connected to database "test03" as user "postgres".

test03=# select * from (values ('刘德华'),('刘少奇')) as a(c1) order by c1 ;
c1
--------
刘德华
刘少奇
(2 rows)
```

注意：在设置按拼音排序时，要注意多音字。例如重庆（chongqing），在编码时，”重”可能会按照 zhong 编码，影响输出，如下面的例子所示：

```
test03=# select * from (values ('中山'),('重庆')) as a(c1) order by c1 collate "zh_CN";
c1
------
中山
重庆
(2 rows)
```

在 Greenplum 中设置输出结果按拼音排序

Greenplum 不支持单列设置 collate，按拼音排序有些许不同。

在 Greenplum 中，可以使用字符集转换，按对应二进制排序，得到拼音排序的效果，如下面的例子所示：

```
postgres=# select * from (values ('刘德华'), ('刘少奇')) t(id) order by byteain(textout(convert(id,'UTF8','EUC_CN')));
id
--------
刘德华
刘少奇
(2 rows)
```

参考文档

CREATE DATABASE 命令的具体使用方法

PostgreSQL 9.6.2 Documentation — Chapter 23. Localization

PostgreSQL UPSERT 的功能与用法

PostgreSQL 9.5 引入了一项新功能，即 UPSERT（insert on conflict do）。当插入遇到约束错误时，直接返回或者改为执行 UPDATE。

UPSERT 语法

UPSERT 的语法如下所示。PostgreSQL 9.5 以前的版本，可以通过函数或者 with 语法来实现与 UPSERT 类似的功能。

```
Command:     INSERT
Description: create new rows in a table
Syntax:
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
{ DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
[ ON CONFLICT [ conflict_target ] conflict_action ]
[ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]

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 ]
```

PostgreSQL 9.5 及以上版本的 UPSERT 用法示例

执行如下命令，创建一张测试表，其中一个字段为唯一键或者主键。

```
create table test(id int primary key, info text, crt_time timestamp);
```

执行如下任一命令，选择插入数据时，若数据存在是进行更新还是直接返回。

不存在则插入，存在则更新，其命令如下所示：

```
test03=# insert into test values (1,'test',now()) on conflict (id) do update set
info=excluded.info,crt_time=excluded.crt_time;
INSERT 0 1

test03=# select * from test;
id | info | crt_time
----+------+----------------------------
1 | test | 2017-04-24 15:27:25.393948
(1 row)

test03=# insert into test values (1,'hello digoal',now()) on conflict (id) do update set
info=excluded.info,crt_time=excluded.crt_time;
INSERT 0 1

test03=# select * from test;
id | info | crt_time
----+--------------+----------------------------
1 | hello digoal | 2017-04-24 15:27:39.140877
(1 row)
```

不存在则插入，存在则直接返回，即不做任何处理，其命令如下所示：

```
test03=# insert into test values (1,'hello digoal',now()) on conflict (id) do nothing;
INSERT 0 0
test03=# insert into test values (1,'pu',now()) on conflict (id) do nothing;
INSERT 0 0
test03=# insert into test values (2,'pu',now()) on conflict (id) do nothing;
INSERT 0 1
test03=# select * from test;
id | info | crt_time
----+--------------+----------------------------
1 | hello digoal | 2017-04-24 15:27:39.140877
2 | pu | 2017-04-24 15:28:20.37392
(2 rows)
```

PostgreSQL 9.5 以下版本的 UPSERT 用法示例

您可以根据需求，通过如下三种方法实现类似 UPSERT 的功能。

通过函数实现。

```
 test03=# create or replace function f_upsert(int,text,timestamp) returns void as $$
declare
res int;
begin
update test set info=$2,crt_time=$3 where id=$1;
```

```
if not found then
insert into test (id,info,crt_time) values ($1,$2,$3);
end if;
exception when others then
return;
end;
$$ language plpgsql strict;
CREATE FUNCTION

test03=# select f_upsert(1,'digoal',now()::timestamp);
f_upsert
----------

(1 row)

test03=# select * from test;
id | info | crt_time
----+--------+----------------------------
2 | pu | 2017-04-24 15:28:20.37392
1 | digoal | 2017-04-24 15:31:29.254325
(2 rows)

test03=# select f_upsert(1,'digoal001',now()::timestamp);
f_upsert
----------

(1 row)

test03=# select * from test;
id | info | crt_time
----+-----------+----------------------------
2 | pu | 2017-04-24 15:28:20.37392
1 | digoal001 | 2017-04-24 15:31:38.0529
(2 rows)

test03=# select f_upsert(3,'hello',now()::timestamp);
f_upsert
----------

(1 row)

test03=# select * from test;
id | info | crt_time
----+-----------+----------------------------
2 | pu | 2017-04-24 15:28:20.37392
1 | digoal001 | 2017-04-24 15:31:38.0529
3 | hello | 2017-04-24 15:31:49.14291
(3 rows)
```

通过 WITH 语法，用法 1，操作步骤如下。

执行如下命令，创建一张测试表，其中一个字段为唯一键或者主键。

```
create table test(id int primary key, info text, crt_time timestamp);
```

执行如下命令，若数据存在则更新，不存在则插入。

```
with upsert as (update test set info=$info,crt_time=$crt_time where id=$id returning *) insert
into test select $id,$info,$crt_time where not exists (select 1 from upsert where id=$id);
```

执行如下命令，替换变量，进行测试。同时插入一条不存在的值，只有一个会话成功，另一个会话会报主键约束错误。

```
with upsert as (update test set info='test',crt_time=now() where id=1 returning *) insert into test select
1,'test',now() where not exists (select 1 from upsert where id=1);
```

通过 WITH 语法，用法 2，操作步骤如下。

执行如下命令，创建一张数据表，即使表没有主键或者唯一约束，也能保证并发。

```
create table test(id int, info text, crt_time timestamp);
```

进行如下任一操作步骤，选择记录不存在时，对于同一条数据更新的处理结果。

对于记录不存在，可以保证只有一个 session 插入数据，对于同一条数据更新，先来的 session 会 lock着记录，后来的 session 会 wait。操作步骤如下所示：

执行如下命令，确定对数据更新的处理结果。

```
with
w1 as(select ('x'||substr(md5('$id'),1,16))::bit(64)::bigint as tra_id),
upsert as (update test set info=$info,crt_time=$crt_time where id=$id
returning *)
insert into test select $id, $info, $crt_time from w1
where pg_try_advisory_xact_lock(tra_id) and not exists (select 1 from upsert
where id=$id);
```

替换变量，进行测试。

```
with
w1 as(select ('x'||substr(md5('1'),1,16))::bit(64)::bigint as tra_id),
upsert as (update test set info='digoal0123',crt_time=now() where id=1 returning *)
insert into test select 1, 'digoal0123', now() from w1
where pg_try_advisory_xact_lock(tra_id) and not exists (select 1 from upsert where
id=1);

INSERT 0 0
```

```
test03=# select * from test;
id | info | crt_time
----+------------+----------------------------
2 | pu | 2017-04-24 15:28:20.37392
3 | hello | 2017-04-24 15:31:49.14291
1 | digoal0123 | 2017-04-24 15:31:38.0529
(3 rows)

with
w1 as(select ('x'||substr(md5('4'),1,16))::bit(64)::bigint as tra_id),
upsert as (update test set info='digoal0123',crt_time=now() where id=4 returning *)
insert into test select 4, 'digoal0123', now() from w1
where pg_try_advisory_xact_lock(tra_id) and not exists (select 1 from upsert where
id=4);

INSERT 0 1

test03=# select * from test;
id | info | crt_time
----+------------+----------------------------
2 | pu | 2017-04-24 15:28:20.37392
3 | hello | 2017-04-24 15:31:49.14291
1 | digoal0123 | 2017-04-24 15:31:38.0529
4 | digoal0123 | 2017-04-24 15:38:39.801908
(4 rows)
```

对于记录不存在，可以保证只有一个 session 插入数据，对于同一条数据更新，先来的 session 会更新数据，后来的 session 不等待，直接失败。操作步骤如下所示：

执行如下命令，确定对数据更新的处理结果。

```
with w1 as(select ('x'||substr(md5('$id'),1,16))::bit(64)::bigint as tra_id),
upsert as (update test set info=$info,crt_time=$crt_time from w1 where
pg_try_advisory_xact_lock(tra_id) and id=$id returning *)
insert into test select $id,$info,$crt_time from w1
where pg_try_advisory_xact_lock(tra_id) and not exists (select 1 from upsert
where id=$id);
```

替换变量，进行测试。

```
with w1 as(select ('x'||substr(md5('1'),1,16))::bit(64)::bigint as tra_id),
upsert as (update test set info='test',crt_time=now() from w1 where pg_try_advisory_xact_lock(tra_id) and
id=1 returning *)
insert into test select 1,'test',now() from w1
where pg_try_advisory_xact_lock(tra_id) and not exists (select 1 from upsert where id=1);

INSERT 0 0

test03=# select * from test;
id | info | crt_time
```

```
----+------------+----------------------------
2 | pu | 2017-04-24 15:28:20.37392
3 | hello | 2017-04-24 15:31:49.14291
4 | digoal0123 | 2017-04-24 15:42:50.912887
1 | test | 2017-04-24 15:44:44.245167
(4 rows)
```

批量更新、删除或插入数据

批量操作可以减少数据库与应用程序的交互次数，提高数据处理的吞吐量。本文将通过示例介绍如何批量插入、更新和删除数据。

批量插入数据

您可以通过如下四种方法进行批量插入数据。

使用 insert into ... select 的方法。

```
  postgres=# insert into tbl1 (id, info ,crt_time) select generate_series(1,10000),'test',now();
INSERT 0 10000
postgres=# select count(*) from tbl1;
count
-------
10001
(1 row)
```

使用 values(),(),...(); 的方法。

```
postgres=# insert into tbl1 (id,info,crt_time) values (1,'test',now()), (2,'test2',now()), (3,'test3',now());
INSERT 0 3
```

使用 BEGIN; ...多条insert...; END; 的方法。严格来说，这不属于批量，但可以减少事务提交时的同步等待，同样可以提升性能。

```
postgres=# begin;
BEGIN
postgres=# insert into tbl1 (id,info,crt_time) values (1,'test',now());
INSERT 0 1
postgres=# insert into tbl1 (id,info,crt_time) values (2,'test2',now());
INSERT 0 1
```

```
postgres=# insert into tbl1 (id,info,crt_time) values (3,'test3',now());
INSERT 0 1
postgres=# end;
COMMIT
```

使用 copy 协议。copy 协议与 insert 协议不一样，更加精简，插入效率高。

```
  test03=# \d test
Table "public.test"
Column | Type | Modifiers
----------+-----------------------------+-----------
id | integer | not null
info | text |
crt_time | timestamp without time zone |
Indexes:
"test_pkey" PRIMARY KEY, btree (id)

test03=# copy test from stdin;
Enter data to be copied followed by a newline.
End with a backslash and a period on a line by itself.
>> 8 'test' '2017-01-01'
>> 9 'test9' '2017-02-02'
>> \.
COPY 2
```

说明：不同的语言驱动，对应的 COPY 接口不同，请参见如下文档。

PostgreSQL JDBC Driver - JDBC 4.2 9.4.1209 API

PostgreSQL 9.6.2 Documentation — Functions Associated with the COPY Command

批量更新数据

```
test03=# update test set info=tmp.info from (values (1,'new1'),(2,'new2'),(6,'new6')) as tmp (id,info) where
test.id=tmp.id;
UPDATE 3
test03=# select * from test;
id | info | crt_time
----+--------------+----------------------------
3 | hello | 2017-04-24 15:31:49.14291
4 | digoal0123 | 2017-04-24 15:42:50.912887
5 | hello digoal | 2017-04-24 15:57:29.622045
1 | new1 | 2017-04-24 15:58:55.610072
2 | new2 | 2017-04-24 15:28:20.37392
6 | new6 | 2017-04-24 15:59:12.265915
(6 rows)
```

批量删除数据

```
test03=# delete from test using (values (3),(4),(5)) as tmp(id) where test.id=tmp.id;
DELETE 3
test03=# select * from test;
id | info | crt_time
----+---------+----------------------------
1 | new1 | 2017-04-24 15:58:55.610072
2 | new2 | 2017-04-24 15:28:20.37392
6 | new6 | 2017-04-24 15:59:12.265915
```

如果要清除全表，建议您使用 truncate。

```
test03=# set lock_timeout = '1s';
SET
test03=# truncate test;
TRUNCATE TABLE
test03=# select * from test;
id | info | crt_time
----+------+----------
(0 rows)
```

查找最耗费资源的 SQL（Top SQL）

数据库是较大型的应用，对于繁忙的数据库，需要消耗大量的内存、CPU、IO、网络资源。SQL 优化是数据库优化的手段之一，而为了达到 SQL 优化的最佳效果，您首先需要了解最消耗资源的 SQL（Top SQL），例如 IO 消耗最高的 SQL。

数据库资源分为多个维度、CPU、内存、IO 等，为能够从各个维度层面查找最消耗数据库资源的 SQL，您可以使用 pg_stat_statements 插件统计数据库的资源开销和分析 Top SQL。

本文将通过示例介绍如何创建 pg_stat_statements 插件、如何分析 Top SQL 以及如何重置统计信息。

创建 pg_stat_statements 插件

执行如下命令，在需要查询 TOP SQL 的数据库中，创建 pg_stat_statements 插件。

```
create extension pg_stat_statements;
```

分析 TOP SQL

pg_stat_statements 输出内容介绍

通过查询 pg_stat_statements 视图，您可以得到数据库资源开销的统计信息。SQL 语句中的一些过滤条件在 pg_stat_statements 中会被替换成变量，可以减少重复显示的问题。

pg_stat_statements 视图包含了一些重要信息，例如：

SQL 的调用次数，总耗时，最快执行时间，最慢执行时间，平均执行时间，执行时间的方差（看出抖动），总共扫描、返回或处理了多少行。

shared buffer 的使用情况：命中、未命中、产生脏块、驱逐脏块。

local buffer 的使用情况：命中、未命中、产生脏块、驱逐脏块。

temp buffer 的使用情况：读了多少脏块、驱逐脏块。

数据块的读写时间。

下表列出了 pg_stat_statements 输出内容中各参数的含义。

Name	Type	References	Description
userid	oid	pg_authid.oid	OID of user who executed the statement.
dbid	oid	pg_database.oid	OID of database in which the statement was executed.
queryid	bigint	-	Internal hash code, computed from the statement’ s parse tree.
query	text	-	Text of a representative statement.
calls	bigint	-	Number of times executed.
total_time	double precision	-	Total time spent in the statement, in milliseconds.
min_time	double precision	-	Minimum time spent in the statement, in milliseconds.
max_time	double precision	-	Maximum time spent in the statement, in

			milliseconds.
mean_time	double precision	-	Mean time spent in the statement, in milliseconds.
stddev_time	double precision	-	Population standard deviation of time spent in the statement, in milliseconds.
rows	bigint	-	Total number of rows retrieved or affected by the statement.
shared_blks_hit	bigint	-	Total number of shared block cache hits by the statement.
shared_blks_read	bigint	-	Total number of shared blocks read by the statement.
shared_blks_dirtied	bigint	-	Total number of shared blocks dirtied by the statement.
shared_blks_written	bigint	-	Total number of shared blocks written by the statement.
local_blks_hit	bigint	-	Total number of local block cache hits by the statement.
local_blks_read	bigint	-	Total number of local blocks read by the statement.
local_blks_dirtied	bigint	-	Total number of local blocks dirtied by the statement.
local_blks_written	bigint	-	Total number of local blocks written by the statement.
temp_blks_read	bigint	-	Total number of temp blocks read by the statement.
temp_blks_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 track_io_timing is enabled, otherwise zero).
blk_write_time	double precision	-	Total time the statement spent writing blocks, in milliseconds (if track_io_timing is enabled, otherwise zero).

最耗 IO SQL

执行如下命令，查询单次调用最耗 IO SQL TOP 5。

```
select userid::regrole, dbid, query from pg_stat_statements order by (blk_read_time+blk_write_time)/calls desc limit 5;
```

执行如下命令，查询总最耗 IO SQL TOP 5。

```
select userid::regrole, dbid, query from pg_stat_statements order by (blk_read_time+blk_write_time) desc limit 5;
```

最耗时 SQL

执行如下命令，查询单次调用最耗时 SQL TOP 5。

```
select userid::regrole, dbid, query from pg_stat_statements order by mean_time desc limit 5;
```

执行如下命令，查询总最耗时 SQL TOP 5。

```
select userid::regrole, dbid, query from pg_stat_statements order by total_time desc limit 5;
```

响应时间抖动最严重 SQL

执行如下命令，查询响应时间抖动最严重 SQL。

```
select userid::regrole, dbid, query from pg_stat_statements order by stddev_time desc limit 5;
```

最耗共享内存 SQL

执行如下命令，查询最耗共享内存 SQL。

```
select userid::regrole, dbid, query from pg_stat_statements order by (shared_blks_hit+shared_blks_dirtied) desc limit
5;
```

最耗临时空间 SQL

执行如下命令，查询最耗临时空间 SQL。

```
select userid::regrole, dbid, query from pg_stat_statements order by temp_blks_written desc limit 5;
```

重置统计信息

pg_stat_statements 是累积的统计，如果要查看某个时间段的统计，需要打快照，建议您参见文档《PostgreSQL AWR报告(for 阿里云ApsaraDB PgSQL)》。

您也可以通过执行如下命令，来定期清理历史统计信息。

```
select pg_stat_statements_reset();
```

参考文档

PostgreSQL 9.6.2 Documentation — F.29. pg_stat_statements

模糊查询、正则查询和相似查询优化

PostgreSQL 拥有很强的文本搜索能力，除了支持全文检索，还支持模糊查询和正则查询。PostgreSQL 不仅内置了一般数据库都没有的 pg_trgm 插件，还内置了表达式索引和 GIN 索引的功能，为加速各类需求的查询提供了有力条件。

对于正则查询，PostgreSQL 可以通过 pg_trgm 插件来加速查询。模糊查询包括前模糊（有前缀的模糊）、后模糊（有后缀的模糊）和前后模糊（无前后缀的模糊），针对不同的模糊查询需求，PostgreSQL 有不同的优化方法：

对于前模糊和后模糊，PostgreSQL 与其他数据库一样，可以使用 B-tree 来加速查询。对于后模糊，也可以使用反转函数（reverse）的函数索引来加速查询。

对于前后模糊，一种方法是使用 pg_trgm 插件，利用 GIN 索引加速查询（特别是对于输入 3 个或 3 个以上字符的模糊查询有很好的效果）。另一种方法是自定义 GIN 表达式索引的方法，适合于定制的

模糊查询。

模糊查询和正则查询都是找出完全符合条件的记录，还有一种需求是相似查询。本文将通过示例介绍如何进行模糊查询、正则查询和相似查询的优化。

背景信息

pg_trgm 模糊查询的原理

pg_trgm 先将字符串的前端添加 2 个空格，末尾添加 1 个空格。每连续的 3 个字符为一个 TOKEN，然后将各 TOKEN 拆开。最后，对 TOKEN 建立 GIN 倒排索引。

您可以通过如下方法查看字符串的 TOKEN。

```
test=# select show_trgm('123');
show_trgm
-------------------------
{" 1"," 12",123,"23 "}
(1 row)
```

pg_trgm 优化查询时要求字符个数的原因

使用 pg_trgm 优化前后模糊查询时，若要获得最好的效果，需满足如下条件：

对于前模糊查询，例如 a%，至少需要提供 1 个字符。（搜索的是token=' a'）

对于后模糊查询，例如 %ab，至少需要提供 2 个字符。（搜索的是token='ab '）

对于前后模糊查询，例如 %abcd%，至少需要提供 3 个字符。（这个使用数组搜索，搜索的是包含{" a"," ab",abc,bcd,"cd "}的 TOKEN。）

由于 pg_trgm 生成的 TOKEN 是三个字符，只有在以上三个条件下，才能匹配到对应的 TOKEN。

使用示例

```
test=# select show_trgm('123');
show_trgm
-------------------------
{" 1"," 12",123,"23 "}
(1 row)
```

前模糊和后模糊查询的优化

前模糊查询的优化方法

PostgreSQL 支持使用 B-tree 来加速前模糊的查询。

当使用类型默认的 index ops class 时，仅适合于collate="C"的查询（当数据库默认的 lc_collate <> C 时，索引和查询都需要明确指定 collate “C” ）。

注意：索引和查询条件的 collate 必须一致才能使用索引。

使用示例

```
 test=# create table test(id int, info text);
CREATE TABLE
test=# insert into test select generate_series(1,1000000),md5(random()::text);
INSERT 0 1000000
test=# create index idx on test(info collate "C");
CREATE INDEX

test=# explain (analyze,verbose,timing,costs,buffers) select * from test where info like 'abcd%' collate "C";
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Index Scan using idx on public.test (cost=0.42..16.76 rows=100 width=37) (actual time=0.057..0.093
rows=18 loops=1)
Output: id, info
Index Cond: ((test.info >= 'abcd'::text) AND (test.info < 'abce'::text))
Filter: (test.info ~~ 'abcd%'::text COLLATE "C")
Buffers: shared hit=18 read=3
Planning time: 0.424 ms
Execution time: 0.124 ms
(7 rows)
```

当数据库默认的 lc_collate <> C 时，还可以使用对应类型的 pattern ops 让 B-tree 索引支持模糊查询。使用 pattern ops 将使用字符查询而非 binary 查询的搜索方式。详情请参见文档 PostgreSQL 9.6.2 Documentation — 11.9. Operator Classes and Operator Families。

使用示例

```
 test=# drop table test;
DROP TABLE
test=# create table test(id int, info text);
CREATE TABLE
test=# insert into test select generate_series(1,1000000),md5(random()::text);
INSERT 0 1000000
test=# create index idx on test(info text_pattern_ops);
CREATE INDEX
test=# explain (analyze,verbose,timing,costs,buffers) select * from test where info like 'abcd%' collate
"zh_CN";
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
```

```
-
Index Scan using idx on public.test (cost=0.42..16.76 rows=100 width=37) (actual time=0.038..0.059
rows=12 loops=1)
Output: id, info
Index Cond: ((test.info ~>=~ 'abcd'::text) AND (test.info ~<~ 'abce'::text))
Filter: (test.info ~~ 'abcd%'::text COLLATE "zh_CN")
Buffers: shared hit=12 read=3
Planning time: 0.253 ms
Execution time: 0.081 ms
(7 rows)

test=# explain (analyze,verbose,timing,costs,buffers) select * from test where info like 'abcd%' collate "C";
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
-
Index Scan using idx on public.test (cost=0.42..16.76 rows=100 width=37) (actual time=0.027..0.050
rows=12 loops=1)
Output: id, info
Index Cond: ((test.info ~>=~ 'abcd'::text) AND (test.info ~<~ 'abce'::text))
Filter: (test.info ~~ 'abcd%'::text COLLATE "C")
Buffers: shared hit=15
Planning time: 0.141 ms
Execution time: 0.072 ms
(7 rows)
```

另外，使用类型对应的 pattern ops 时，索引搜索不仅支持 LIKE 的写法，还支持规则表达式的写法，如下所示：

```
 test=# explain (analyze,verbose,timing,costs,buffers) select * from test where info ~ '^abcd';
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
-
Index Scan using idx on public.test (cost=0.42..16.76 rows=100 width=37) (actual time=0.031..0.061
rows=12 loops=1)
Output: id, info
Index Cond: ((test.info ~>=~ 'abcd'::text) AND (test.info ~<~ 'abce'::text))
Filter: (test.info ~ '^abcd'::text)
Buffers: shared hit=15
Planning time: 0.213 ms
Execution time: 0.083 ms
(7 rows)
```

后模糊查询的优化方法

PostgreSQL 支持使用反转函数索引来加速后模糊的查询。

当使用类型默认的 index ops class 时，仅适合于collate="C"的查询（当数据库默认的 lc_collate <> C 时，索引和查询都需要明确指定 collate “C” ）。

注意：索引和查询条件的 collate 必须一致才能使用索引。

使用示例

```
 test=# create index idx1 on test(reverse(info) collate "C");
CREATE INDEX
test=# select * from test limit 1;
id | info
----+----------------------------------
1 | b3275976cdd437a033d4329775a52514
(1 row)

test=# explain (analyze,verbose,timing,costs,buffers) select * from test where reverse(info) like '4152%'
collate "C";
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
----
Index Scan using idx1 on public.test (cost=0.42..4009.43 rows=5000 width=37) (actual time=0.061..0.097
rows=18 loops=1)
Output: id, info
Index Cond: ((reverse(test.info) >= '4152'::text) AND (reverse(test.info) < '4153'::text))
Filter: (reverse(test.info) ~~ '4152%'::text COLLATE "C")
Buffers: shared hit=18 read=3
Planning time: 0.128 ms
Execution time: 0.122 ms
(7 rows)

test=# select * from test where reverse(info) like '4152%' collate "C";
id | info
--------+----------------------------------
847904 | abe2ecd90393b5275df8e34a39702514
414702 | 97f66d26545329321164042657d02514
191232 | 7820972c6220c2b01d46c11ebb532514
752742 | 93232ac39c6632e2540df44627c42514
217302 | 39e518893a1a7b1e691619bd1fc42514
1 | b3275976cdd437a033d4329775a52514
615718 | 4948f94c484c13dc6c4fae8a3db52514
308815 | fc2918ceff7c7a4dafd2e04031062514
149521 | 546d963842ea5ca593e622c810262514
811093 | 4b6eca2eb6b665af67b2813e91a62514
209000 | 1dfd0d4e326715c1739f031cca992514
937616 | 8827fd81f5b673fb5afecbe3e11b2514
419553 | bd6e01ce360af16137e8b6abc8ab2514
998324 | 7dff51c19dc5e5d9979163e7d14c2514
771518 | 8a54e30003a48539fff0aedc73ac2514
691566 | f90368348e3b6bf983fcbe10db2d2514
652274 | 8bf4a97b5f122a5540a21fa85ead2514
233437 | 739ed715fc203d47e37e79b5bcbe2514
(18 rows)
```

当数据库默认的 lc_collate <> C 时，还可以使用对应类型的 pattern ops 让 B-tree 索引支持模糊查询。使用 pattern ops 将使用字符查询而非 binary 查询的搜索方式。使用类型对应的 pattern ops 时，索引搜索不仅支持 LIKE 的写法，还支持规则表达式的写法。

使用示例

```
 test=# create index idx1 on test(reverse(info) text_pattern_ops);
CREATE INDEX
test=# explain (analyze,verbose,timing,costs,buffers) select * from test where reverse(info) like '4152%';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Index Scan using idx1 on public.test (cost=0.42..4009.43 rows=5000 width=37) (actual time=0.026..0.049 rows=12
loops=1)
Output: id, info
Index Cond: ((reverse(test.info) ~>=~ '4152'::text) AND (reverse(test.info) ~<~ '4153'::text))
Filter: (reverse(test.info) ~~ '4152%'::text)
Buffers: shared hit=15
Planning time: 0.102 ms
Execution time: 0.072 ms
(7 rows)
test=# explain (analyze,verbose,timing,costs,buffers) select * from test where reverse(info) ~ '^4152';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Index Scan using idx1 on public.test (cost=0.42..4009.43 rows=5000 width=37) (actual time=0.031..0.063 rows=12
loops=1)
Output: id, info
Index Cond: ((reverse(test.info) ~>=~ '4152'::text) AND (reverse(test.info) ~<~ '4153'::text))
Filter: (reverse(test.info) ~ '^4152'::text)
Buffers: shared hit=15
Planning time: 0.148 ms
Execution time: 0.087 ms
(7 rows)
```

前模糊和后模糊查询的单一索引优化方法

PostgreSQL 支持使用 pg_trgm 索引来加速同时包含前模糊和后模糊的查询。

为使索引过滤有较好的效果，前模糊查询至少需输入 1 个字符，后模糊查询至少需输入 2 个字符。若要高效支持多字节字符（如中文），数据库的 lc_ctype 不能为 “C” ，只有 TOKEN 分割正确才能有较好的效果。

注意：索引和查询条件的 collate 必须一致才能使用索引。

使用示例

执行如下代码，创建一个表。

```
 test=# \l+ test
List of databases
Name | Owner | Encoding | Collate | Ctype | Access privileges | Size | Tablespace | Description
------+----------+----------+------------+------------+-------------------+--------+------------+-------------
test | postgres | UTF8 | zh_CN.utf8 | zh_CN.utf8 | | 245 MB | pg_default |
(1 row)

test=# create extension pg_trgm;

test=# create table test001(c1 text);
```

```
CREATE TABLE
```

执行如下代码，生成随机中文字符串的函数。

```
 test=# create or replace function gen_hanzi(int) returns text as $$
declare
res text;
begin
if $1 >=1 then
select string_agg(chr(19968+(random()*20901)::int), '') into res from generate_series(1,$1);
return res;
end if;
return null;
end;
$$ language plpgsql strict;
CREATE FUNCTION
```

执行如下代码，生成随机数据。

```
 test=# insert into test001 select gen_hanzi(20) from generate_series(1,100000);
INSERT 0 100000

test=# create index idx_test001_1 on test001 using gin (c1 gin_trgm_ops);
CREATE INDEX

test=# select * from test001 limit 5;
c1
-------------------------------------------
埳噪办甾讷昃碇珅陧箖燋邢賀浮媊跕蔺喃谇橆
烸橑籛鷗拟俱敤龐竈醢轇坙騉鏦纗藒婃坑嫻儅
葰纏鎧爪鵬二悲膕膜麻瀷鍪楨竆違舔樱嘓索籓
馳泅薬鐧慬撞窍浉渗蛁灩勑撍摐瞪拡擨詨隝緼
襳铺熣匶漉懲荼粛樆惺篋搔狃憯墆鋃碵蔓恧頬
(5 rows)
```

执行如下代码，进行模糊查询。

```
 test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 like '你%';
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
----
Bitmap Heap Scan on public.test001 (cost=5.08..15.20 rows=10 width=61) (actual time=0.030..0.034
rows=3 loops=1)
Output: c1
Recheck Cond: (test001.c1 ~~ '你%'::text)
Heap Blocks: exact=3
Buffers: shared hit=7
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..5.08 rows=10 width=0) (actual time=0.020..0.020
rows=3 loops=1)
Index Cond: (test001.c1 ~~ '你%'::text)
```

```
Buffers: shared hit=4
Planning time: 0.119 ms
Execution time: 0.063 ms
(10 rows)

test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 like '%恖顙';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
----
Bitmap Heap Scan on public.test001 (cost=5.08..15.20 rows=10 width=61) (actual time=0.031..0.034
rows=1 loops=1)
Output: c1
Recheck Cond: (test001.c1 ~~ '%恖顙'::text)
Rows Removed by Index Recheck: 1
Heap Blocks: exact=2
Buffers: shared hit=6
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..5.08 rows=10 width=0) (actual time=0.020..0.020
rows=2 loops=1)
Index Cond: (test001.c1 ~~ '%恖顙'::text)
Buffers: shared hit=4
Planning time: 0.136 ms
Execution time: 0.062 ms
(11 rows)
```

前后模糊查询的优化

大于或等于 3 个输入字符的前后模糊查询优化

PostgreSQL 支持使用 pg_trgm 插件来加速前后模糊的查询。为达到更好的加速效果，建议您输入 3 个或 3 个以上字符，详细原因请参见本文背景信息中关于 pg_trgm 优化查询时要求字符个数的原因。

若要让 pg_trgm 高效支持多字节字符（如中文），数据库的 lc_ctype 不能为 “C” ，只有 TOKEN 分割正确才能达到加速效果。

使用示例

```
test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 like '%燋邢賀%';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on public.test001 (cost=5.08..15.20 rows=10 width=61) (actual time=0.038..0.038 rows=1
loops=1)
Output: c1
Recheck Cond: (test001.c1 ~~ '%燋邢賀%'::text)
Heap Blocks: exact=1
Buffers: shared hit=5
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..5.08 rows=10 width=0) (actual time=0.025..0.025 rows=1
loops=1)
Index Cond: (test001.c1 ~~ '%燋邢賀%'::text)
Buffers: shared hit=4
Planning time: 0.170 ms
```

```
Execution time: 0.076 ms
(10 rows)

test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 like '%燋邢%';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
------
Bitmap Heap Scan on public.test001 (cost=7615669.08..7615679.20 rows=10 width=61) (actual
time=147.524..178.232 rows=1 loops=1)
Output: c1
Recheck Cond: (test001.c1 ~~ '%燋邢%'::text)
Rows Removed by Index Recheck: 99999
Heap Blocks: exact=1137
Buffers: shared hit=14429
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..7615669.08 rows=10 width=0) (actual time=147.377..147.377
rows=100000 loops=1)
Index Cond: (test001.c1 ~~ '%燋邢%'::text)
Buffers: shared hit=13292
Planning time: 0.133 ms
Execution time: 178.265 ms
(11 rows)
```

小于 3 个输入字符的前后模糊查询优化

当需要前后模糊搜索 1 个或者 2 个字符时，pg_trgm 无法满足需求，但您可以使用表达式 GIN 索引。

使用表达式，将字符串拆成 1 个单字以及两个连续字符的数组，然后对数组建立 GIN 索引即可。

使用示例

```
test=# create or replace function split001(text) returns text[] as $$
declare
res text[];
begin
select regexp_split_to_array($1,'') into res;
for i in 1..length($1)-1 loop
res := array_append(res, substring($1,i,2));
end loop;
return res;
end;
$$ language plpgsql strict immutable;
CREATE FUNCTION

test=# create index idx_test001_2 on test001 using gin (split001(c1));

test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where split001(c1) @> array['你好'];
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on public.test001 (cost=8.87..550.12 rows=500 width=61) (actual time=0.041..0.041 rows=0
loops=1)
Output: c1
Recheck Cond: (split001(test001.c1) @> '{你好}'::text[])
Buffers: shared hit=4
```

```
-> Bitmap Index Scan on idx_test001_2 (cost=0.00..8.75 rows=500 width=0) (actual time=0.039..0.039 rows=0
loops=1)
Index Cond: (split001(test001.c1) @> '{你好}'::text[])
Buffers: shared hit=4
Planning time: 0.104 ms
Execution time: 0.068 ms
(9 rows)

test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where split001(c1) @> array['你'];
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on public.test001 (cost=8.87..550.12 rows=500 width=61) (actual time=0.063..0.183 rows=86
loops=1)
Output: c1
Recheck Cond: (split001(test001.c1) @> '{你}'::text[])
Heap Blocks: exact=80
Buffers: shared hit=84
-> Bitmap Index Scan on idx_test001_2 (cost=0.00..8.75 rows=500 width=0) (actual time=0.048..0.048 rows=86
loops=1)
Index Cond: (split001(test001.c1) @> '{你}'::text[])
Buffers: shared hit=4
Planning time: 0.101 ms
Execution time: 0.217 ms
(10 rows)

test=# select * from test001 where split001(c1) @> array['你'];
c1
------------------------------------------
殊蹤洪冨垓刂贤閌偉垢胸鍘崩你萬隡劭荂雫隻
黷慨热脸鐀涝寘鰻总襠戌譈枨陪幷倫柆套你仮
.....
```

正则查询的优化

PostgreSQL 正则查询的语法为字符串 ~ 'pattern' 或字符串 ~* 'pattern'。详情请参见文档 PostgreSQL 9.6.2 Documentation — 9.7. Pattern Matching。

另外，关于正则查询索引原理，请参见 PostgreSQL 源码。

使用示例

```
test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 ~ '12[0-9]{3,9}';
QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on public.test001 (cost=65.08..75.20 rows=10 width=61) (actual time=0.196..0.196 rows=0
loops=1)
Output: c1
Recheck Cond: (test001.c1 ~ '12[0-9]{3,9}'::text)
Rows Removed by Index Recheck: 1
Heap Blocks: exact=1
Buffers: shared hit=50
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..65.08 rows=10 width=0) (actual time=0.183..0.183 rows=1
```

```
loops=1)
Index Cond: (test001.c1 ~ '12[0-9]{3,9}'::text)
Buffers: shared hit=49
Planning time: 0.452 ms
Execution time: 0.221 ms
(11 rows)

test01=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 ~ '宸杠啣' collate "zh_CN";
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on public.test001 (cost=6.58..19.42 rows=10 width=61) (actual time=0.061..0.061 rows=1
loops=1)
Output: c1
Recheck Cond: (test001.c1 ~ '宸杠啣'::text COLLATE "zh_CN")
Heap Blocks: exact=1
Buffers: shared hit=5
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..6.58 rows=10 width=0) (actual time=0.049..0.049 rows=1
loops=1)
Index Cond: (test001.c1 ~ '宸杠啣'::text COLLATE "zh_CN")
Buffers: shared hit=4
Planning time: 0.238 ms
Execution time: 0.082 ms
(10 rows)
```

相似查询优化

相似查询是指通过相似度匹配输入的字符串和需要查询的字符串，例如查询 postgresql 字符串时，输入 p0stgresgl 也能根据相似度匹配到 postgresql。

PostgreSQL 支持使用 pg_trgm 插件加速相似查询。为达到更好的加速效果，建议您输入 3 个或 3 个以上字符，详细原因请参见本文背景信息中关于 pg_trgm 优化查询时要求字符个数的原因。

若要高效支持中文相似查询，数据库的 lc_ctype 不能为 “C” ，只有 TOKEN 分割正确才能有较好的效果。

使用示例

```
test=# create index idx_test001_3 on test001 using gist (c1 gist_trgm_ops);
CREATE INDEX

test=# explain (analyze,verbose,timing,costs,buffers) SELECT t, c1 <-> '犖磛鹐蠌鰓躅123鶹垂雉鳊茉奶垮惇溴蔑筴爌憡'
AS dist
FROM test001 t
ORDER BY dist LIMIT 5;
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
-----------------
Limit (cost=0.28..0.52 rows=5 width=89) (actual time=37.462..37.639 rows=5 loops=1)
Output: t.*, ((c1 <-> '犖磛鹐蠌鰓躅123鶹垂雉鳊茉奶垮惇溴蔑筴爌憡'::text))
Buffers: shared hit=1631
-> Index Scan using idx_test001_3 on public.test001 t (cost=0.28..4763.28 rows=100000 width=89) (actual
time=37.461..37.636 rows=5 loops=1)
Output: t.*, (c1 <-> '犖磛鹐蠌鰓躅123鶹垂雉鳊茉奶垮惇溴蔑筴爌憡'::text)
```

```
Order By: (t.c1 <-> '丵蹔鹚蠌鳃斸123鶡垂婎鳊茉奶圬憌溴蔻筴糖愫'::text)
Buffers: shared hit=1631
Planning time: 0.089 ms
Execution time: 37.668 ms
(9 rows)

test=# SELECT t, c1 <-> '丵蹔鹚蠌鳃斸123鶡垂婎鳊茉奶圬憌溴蔻筴糖愫' AS dist
FROM test001 t
ORDER BY dist LIMIT 5;
t | dist
-------------------------------------------+----------
(丵蹔鹚蠌鳃斸你鶡垂婎鳊茉奶圬憌溴蔻筴糖愫) | 0.307692
(坆栙悁旆眩瑚豌腏炁悿隝轲盉挜棆睟礓蜮铅涪) | 0.976744
(丵鉗餯祂鼃恫蝁爬蘱癊梎蛸歡僷贊敔欉侑韧鐹) | 0.976744
(丵嚁皍戚蹪熼胘橥仳欔韜挹蔵瘭惶踓项鑟鞖媗) | 0.976744
(丵饎瞲餿焜歔峡盾狣擔禞嵪豦咢脉馄踔济隘缄) | 0.976744
(5 rows)
```

总结

若只有前模糊查询需求（字符串 like 'xx%'），使用 collate “C” 的 B-tree 索引。当 collate 不为 “C” 时，可以使用类型对应的 pattern ops（如 text_pattern_ops）建立 B-tree 索引。

若只有后模糊的查询需求（字符串 like '%abc'等价于reverse(字符串) like cba%），使用 collate “C” 的 reverse() 表达式的 B-tree 索引。当 collate 不为 “C” 时，可以使用类型对应的 pattern ops（如text_pattern_ops）建立 B-tree 索引。

若有前后模糊查询需求，并且包含中文，请使用 lc_ctype <> “C” 的数据库，同时使用 pg_trgm 插件的 GIN 表达式索引。

若有前后模糊查询需求，并且不包含中文，请使用 pg_trgm 插件的 GIN 表达式索引。

若有正则查询需求，请使用 pg_trgm 插件的 GIN 表达式索引。

若有输入条件少于 3 个字符的模糊查询需求，可以使用 GIN 表达式索引，通过数组包含的方式进行搜索，性能一样非常好。

模糊查询性能测试

如下示例对优化前后模糊查询后的性能进行了测试。该示例中的测试数据为 1 亿条记录，且每条记录有 15 个随机中文。

操作步骤

执行如下代码，生成测试数据。

```
 vi test.sql
insert into test001 select gen_hanzi(15) from generate_series(1,2500000);
pgbench -n -r -P 1 -f ./test.sql -c 40 -j 40 -t 1 test
test=# select count(*) from test001;
count
-----------
100000000
(1 row)
test=# select * from test001 limit 10;
c1
--------------------------------
釾笏皜鰈确艄騟馺腃彊鈀忰采汦擇
槮摽圮墔婂蹾飄孡鶇镇赀瞶線麯欑
孨郑韞蓸赫炧暤蟠檼駧餪嵧娲譌筋
烸呫醝椲怩鷟梵棐妛譥佰飛饡绘韦
撐豁襇皀炠映罱襄彊鰮莆壏妒辷闤
蝊愊鶱磹貳惋眲嚉榑苍潵檐簷椰魨
瑄翁羸巨躋詎蛸湗鑣頼櫟砣八癱栵
馇巍咎鞒装棊嘢恓煓焜锠鋈踷煿屓
訆韄踔牤嘇糺絢輧鶻燿螛梋鰢謷鄣
撲蒨伤釱糕鮄孌攉鰼繩圆醷燜霎掑
(10 rows)
```

执行如下代码，创建索引。

```
test=# set maintenance_work_mem ='32GB';
test=# create index idx_test001_1 on test001 using gin (c1 gin_trgm_ops);
```

执行如下代码，查询表和索引的大小。

```
 test=# \di+
List of relations
Schema | Name | Type | Owner | Table | Size | Description
--------+---------------+-------+----------+---------+-------+-------------
public | idx_test001_1 | index | postgres | test001 | 12 GB |
(1 row)

test=# \dt+
List of relations
Schema | Name | Type | Owner | Size | Description
--------+---------+-------+----------+---------+-------------
public | test001 | table | postgres | 7303 MB |
(1 row)
```

分别执行如下代码，进行模糊查询性能测试。

执行如下代码，进行前模糊查询性能测试。

响应时间：9 毫秒。

返回 4701 行。

```
  select * from test001 where c1 like '你%';

test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 like '你%';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on public.test001 (cost=89.50..10161.50 rows=10000 width=46) (actual time=1.546..8.868 rows=4701 loops=1)
Output: c1
Recheck Cond: (test001.c1 ~~ '你%'::text)
Rows Removed by Index Recheck: 85
Heap Blocks: exact=4776
Buffers: shared hit=4784
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..87.00 rows=10000 width=0) (actual time=0.879..0.879 rows=4786 loops=1)
Index Cond: (test001.c1 ~~ '你%'::text)
Buffers: shared hit=8
Planning time: 0.099 ms
Execution time: 9.166 ms
(11 rows)
```

执行如下代码，进行后模糊查询性能测试。

响应时间：0.25 毫秒。

返回 2 行。

```
  select * from test001 where c1 like '%霰掑';

test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 like '%霰掑';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on public.test001 (cost=89.50..10161.50 rows=10000 width=46) (actual time=0.049..0.223 rows=2 loops=1)
Output: c1
Recheck Cond: (test001.c1 ~~ '%霰掑'::text)
Rows Removed by Index Recheck: 87
Heap Blocks: exact=89
Buffers: shared hit=94
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..87.00 rows=10000 width=0) (actual time=0.031..0.031 rows=89 loops=1)
Index Cond: (test001.c1 ~~ '%霰掑'::text)
Buffers: shared hit=5
Planning time: 0.113 ms
Execution time: 0.249 ms
```

```
(11 rows)
```

执行如下代码，进行前后模糊查询性能测试。

响应时间：0.2 毫秒。

返回 1 行。

```
 select * from test001 where c1 like '%螛梋鱊%';

test=# explain (analyze,verbose,timing,costs,buffers) select * from test001 where c1 like '%螛梋鱊%';
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
---------
Bitmap Heap Scan on public.test001 (cost=89.50..10161.50 rows=10000 width=46) (actual
time=0.044..0.175 rows=1 loops=1)
Output: c1
Recheck Cond: (test001.c1 ~~ '%螛梋鱊%'::text)
Rows Removed by Index Recheck: 81
Heap Blocks: exact=82
Buffers: shared hit=87
-> Bitmap Index Scan on idx_test001_1 (cost=0.00..87.00 rows=10000 width=0) (actual
time=0.027..0.027 rows=82 loops=1)
Index Cond: (test001.c1 ~~ '%螛梋鱊%'::text)
Buffers: shared hit=5
Planning time: 0.112 ms
Execution time: 0.201 ms
(11 rows)
```

通过DMS将逻辑备份导入RDS数据库

适用场景

用于将逻辑备份导入到RDS数据库中。本文介绍了通过DMS将逻辑备份导入RDS数据库的方法。

前提条件

已在RDS实例中创建本地数据要迁移至的目标数据库及可以登录该目标库的账号。关于如何创建数据库和账号，请参见：

创建数据库和账号MySQL 5.5/5.6版

创建数据库和账号MySQL 5.7版

创建数据库和账号SQL Server 2008 R2版

创建数据库和账号SQL Server 2012/2016版

创建数据库和账号PostgreSQL版

创建数据库和账号PPAS版

注意事项

要导入的文件类型是CSV、SQL或ZIP格式。

单次导入的文件大小不能超过100MB。

要导入的文件中不能包含原数据库的信息。

操作步骤

登录RDS管理控制台。

选择目标实例所在地域。

单击目标实例的ID，进入**基本信息**页面。

在页面右上角，单击**登录数据库**，详细步骤请参见通过DMS登录RDS数据库。

成功登录RDS实例后，在页面上方的菜单栏中选择**导入**。

单击**新增任务**。

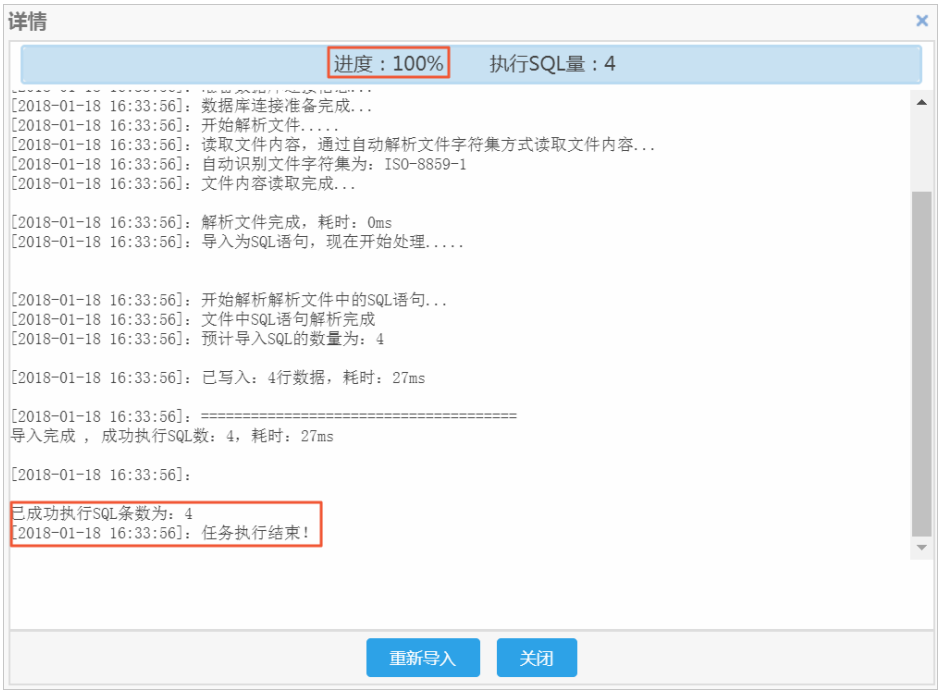
填写导入任务信息。

参数说明：

参数	说明
文件类型	可选SQL或CSV，请选择与导入文件一致的文件类型。
文件字符集	可选自动判定、utf8或gbk，请选择与导入文件一致的字符集。
数据库	选择要导入文件的数据库。
附件	添加要导入的文件。单击**选择文件**即可添加要导入的文件。
描述	为便于管理，可添加导入文件的备注信息。

单击**开始**。

当弹出的对话框中提示进度为100%及任务执行结束时，则说明文件已成功导入。



单击**关闭**。

刷新页面，即可在目标数据中找到成功导入的文件。

相关文档

若您要迁移数据库，请参见**数据迁移方案概览**。