

# 阿里云

## 专有云企业版

云原生数据仓库 AnalyticDB  
PostgreSQL 版  
用户指南

产品版本：v3.16.2

文档版本：20220915

## 法律声明

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

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

# 通用约定

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

# 目录

1.什么是云原生数据仓库AnalyticDB PostgreSQL版	06
2.快速入门	07
2.1. 概述	07
2.2. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台	08
2.3. 创建实例	08
2.4. 设置白名单	10
2.5. 创建数据库账号	11
2.6. 获取客户端工具	13
2.7. 连接数据库	14
3.实例	20
3.1. 重置账号密码	20
3.2. 管理网络连接地址	20
3.3. 监控与报警	21
3.4. 变更配置	21
3.5. 重启实例	22
3.6. 释放实例	22
3.7. SQL审计	22
3.8. SSL	23
3.9. 版本升级	24
3.10. 导入数据	25
3.10.1. OSS高速并行导入	25
3.10.2. 从MySQL导入	32
3.10.3. 从PostgreSQL导入	34
3.10.4. 通过COPY数据导入	35
4.数据库	37
4.1. 概述	37

4.2. 创建数据库	37
4.3. 创建分布键	37
4.4. 构造数据	38
4.5. 查询	38
4.6. 管理插件	39
4.7. 管理用户和权限	40
4.8. 实时物化视图	41
4.9. JSON数据类型操作	43
4.10. HyperLogLog的使用	53
4.11. Create Library命令的使用	55
4.12. PL/Java UDF的使用	56
5.表	58
5.1. 创建表	58
5.2. 行存、列存，堆表、AO表的原理和使用场景	63
5.3. 设置列存和压缩	64
5.4. 给列存加字段默认值	65
5.5. 设置表分区	67
5.6. 设置排序键	68
6.最佳实践	70
6.1. 配置内存和负载参数	70

# 1.什么是云原生数据仓库AnalyticDB PostgreSQL版

云原生数据仓库AnalyticDB PostgreSQL版是一种在线分布式云数据库，由多个计算组组成，可提供大规模并行处理（MPP）数据仓库服务。

AnalyticDB PostgreSQL版基于Greenplum Database开源数据库项目开发，由阿里云深度扩展后，具备如下特性：

- 兼容Greenplum，用户可以直接使用所有支持Greenplum的工具。
- 支持OSS存储、JSON数据类型和HyperLogLog预估分析等功能特性。
- 支持符合SQL 2003标准的查询语法和OLAP分析聚合函数，提供灵活的混合分析能力。
- 支持行存储和列存储混合模式，分析性能优越。
- 支持数据压缩技术，存储成本低廉。
- 提供在线扩容和性能监测等服务，用户无需再进行复杂的大规模MPP集群的运维管理工作，使DBA、开发人员及数据分析师能够专注于如何通过SQL提高企业的生产力，实现核心价值。

## 2. 快速入门

### 2.1. 概述

本文介绍如何从创建实例到登录数据库的一系列操作，使用户快速地了解AnalyticDB PostgreSQL版实例的操作流程。

具体操作步骤请参见[实例流程图](#)。

AnalyticDB PostgreSQL版实例流程图



- **登录AnalyticDB for PostgreSQL控制台**

介绍如何登录到AnalyticDB PostgreSQL版控制台。

- **创建实例**

创建AnalyticDB PostgreSQL版实例是一切操作的开始，用户可以通过控制台快速地创建实例。

- **设置白名单**

为了数据库的安全稳定，在开始使用AnalyticDB PostgreSQL版实例前，您需要将访问数据库的IP地址或者IP段加到目标实例的白名单中。

- **创建初始账号**

创建实例后，您需要创建初始账号，用于登录数据库。

- **连接数据库**

用户可以使用支持PostgreSQL或者支持Greenplum的客户端连接数据库。

## 2.2. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台


本文介绍如何登录到AnalyticDB PostgreSQL版控制台。

### 前提条件

- 登录Apsara Uni-manager运营控制台前，确认您已从部署人员处获取Apsara Uni-manager运营控制台的服务域名地址。
- 推荐使用Chrome浏览器。

1. 在浏览器地址栏中，输入Apsara Uni-manager运营控制台的服务域名地址，按回车键。
2. 输入正确的用户名及密码。


请向运营管理员获取登录控制台的用户名和密码。

 **说明** 首次登录Apsara Uni-manager运营控制台时，需要修改登录用户名的密码，请按照提示完成密码修改。为提高安全性，密码长度必须为10~32位，且至少包含以下两种类型：

- 英文大写或小写字母（A~Z、a~z）
- 阿拉伯数字（0~9）
- 特殊符号（感叹号（!）、at（@）、井号（#）、美元符号（\$）、百分号（%）等）

3. 单击**账号登录**。
4. 如果账号已激活MFA多因素认证，请根据以下两种情况进行操作：
  - 管理员强制开启MFA后的首次登录：
    - a. 在绑定虚拟MFA设备页面中，按页面提示步骤绑定MFA设备。
    - b. 按照步骤2重新输入账号和密码，单击**账号登录**。
    - c. 输入6位MFA码后单击**认证**。
  - 您已开启并绑定MFA：

输入6位MFA码后单击**认证**。

 **说明** 绑定并开启MFA的操作请参见Apsara Uni-manager运营控制台用户指南中的**绑定并开启虚拟MFA设备**章节。

5. 在页面顶部的导航栏中，单击**产品**，选择**数据库 > 云原生数据仓库AnalyticDB PostgreSQL版**。

## 2.3. 创建实例

创建AnalyticDB PostgreSQL版实例是一切操作的开始，您可以通过控制台快速地创建实例。

1. [登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
2. 单击页面右上角**新建实例**。
3. 在**创建分析型数据库 PostgreSQL 版**页面，配置相关参数，具体参数说明如下所示。



类别	配置	说明
区域	组织	实例所属组织。
	资源集	实例所属资源集。
	地域	选择AnalyticDB PostgreSQL版所在的地域。   说明 如果要从ECS上通过虚拟私有网络访问使用，AnalyticDB PostgreSQL版实例必须与ECS实例处于相同的地域和可用区。
	可用区	AnalyticDB PostgreSQL版所在的可用区。
基本配置	芯片架构	选择芯片架构。   说明 仅实例资源类型为计算存储一体模式时显示该配置项。
	实例资源类型	选择计算存储一体模式或存储预留模式。
	引擎版本	选择引擎版本。   说明 仅实例资源类型为计算存储一体模式时显示该配置项。
	计算机规格	计算资源单位，不同的计算组规格有不同的存储空间和计算能力。
	计算机节点	选择计算组数量，计算组个数的增加可以线性提升实例性能。   说明 仅实例资源类型为计算存储一体模式时显示该配置项。
	master节点数量	固定为1，实例创建完成后可以自行添加，操作方式，请参见 <a href="#">变更配置</a> 。   说明 仅实例资源类型为存储预留模式时显示该配置项。
	计算节点数量	选择计算节点数量，节点个数的增加可以线性地提升性能。取值范围：2~640。   说明 仅实例资源类型为存储预留模式时显示该配置项。

类别	配置	说明
	计算节点容量	<p>计算节点容量不支持修改，具体容量与节点规格相关。</p> <p>CPU：内存：存储容量比例为1：8：80，例如计算节点规格为2C16G，则存储容量为160 GB。</p> <div> <b>说明</b> 仅实例资源类型为存储预留模式时显示该配置项。</div>
	存储介质	<p>固定为<b>本地SSD盘</b>。</p> <div> <b>说明</b> 仅实例资源类型为存储预留模式时显示该配置项。</div>
网络	网络类型	<p>AnalyticDB PostgreSQL版实例支持的网络类型：</p> <ul style="list-style-type: none"><li>◦ <b>经典网络</b>：经典网络中的云服务在网络上不进行隔离，只能依靠云服务自身的安全组或白名单策略来阻挡非法访问。</li><li>◦ <b>专有网络</b>：（Virtual Private Cloud，简称VPC）专有网络帮助用户在阿里云上构建出一个隔离的网络环境。用户可以自定义专有网络里面的路由表、IP地址范围和网关。建议您选择专有网络，更加安全。</li></ul> <p>您可以事先创建专有网络，也可以在创建实例后切换网络类型时修改专有网络。</p>
	专有网络vpc	<p>AnalyticDB PostgreSQL版实例所在的VPC。</p> <div> <b>说明</b> 仅网络类型为<b>专有网络</b>时显示该配置项。</div>
	交换机vswitch	<p>AnalyticDB PostgreSQL版实例所在的交换机。</p> <div> <b>说明</b> 仅网络类型为<b>专有网络</b>时显示该配置项。</div>
	IP白名单	<p>添加允许访问AnalyticDB PostgreSQL版实例的IP地址。</p>

4. 完成上述参数配置后，单击**提交**。

## 2.4. 设置白名单

为保障数据库的安全稳定，请将需要访问数据库的IP地址或者IP地址段加入白名单。

1. [登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击**数据安全性**。
4. 在**数据安全性**页面，您可以进行以下操作：


- 添加白名单分组：
  - 单击**添加白名单分组**。
  - 在**添加白名单分组**面板中配置以下信息。

配置	说明
分组名称	新建白名单分组的名称，限制如下： <ul style="list-style-type: none"><li>由小写字母、数字或下划线（_）组成。</li><li>以小写字母开头，以小写字母或数字结尾。</li><li>长度为2~32个字符。</li></ul>
组内白名单	设置需要添加的白名单IP地址，说明如下： <ul style="list-style-type: none"><li>IP地址以英文逗号（,）分隔，不可重复，最多999个。</li><li>支持格式为10.23.12.24（具体IP地址）、10.23.12.24/24（CIDR模式，即无类域间路由，/24表示地址中前缀的长度，范围为1~32）。</li><li>地址中的前缀长度设置为0（例如0.0.0.0/0、127.0.0.1/0）表示允许所有IP地址访问该实例，存在高安全风险，请谨慎设置。</li><li>127.0.0.1表示禁止任何外部IP访问本实例。</li></ul>

- 单击**确定**。
- 修改分组内白名单IP：
  - 单击目标白名单右侧的**修改**。
  - 在**组内白名单**下方添加或删除IP地址或IP地址段。

 **说明** 分组名称不支持修改。

- 单击**确定**。
- 删除白名单分组：

 **说明** 默认白名单分组无法删除。

- 单击目标白名单分组右侧的**删除**。
    - 在**删除分组**对话框中单击**确定**即可。
  - 清空默认白名单分组：
    - 单击default白名单分组右侧的**清空**。
    - 在**确认清空默认分组**对话框中单击**确定**即可清空默认白名单分组。
- 清空后默认白名单分组内仅保留127.0.0.1。

后续步骤

- 建议您定期维护白名单，以保证AnalyticDB PostgreSQL版得到有效的访问安全保护。
- 后续操作中，您可以单击分组名称后的**修改**修改已有分组，或者单击**删除**删除已有的自定义分组。


2.5. 创建数据库账号

本文介绍如何在AnalyticDB PostgreSQL版中创建数据库账号。

背景信息

AnalyticDB PostgreSQL版提供了以下两种数据库账号：

- 高权限账号：拥有所有数据库的所有操作权限。控制台上的初始账号为高权限账号。
- 普通账号：拥有已授权数据库（Owner）的所有操作权限。


 **说明** 操作权限包括SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER。

注意事项

- 使用AnalyticDB PostgreSQL版数据库前，需要创建初始账号。
  - 控制台仅支持创建初始账号，如果需要创建其他账号，请参见[使用SQL语句创建账号](#)。
  - 初始账号创建后无法删除。
1. [登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
  2. 找到目标实例，单击实例ID。
  3. 在左侧导航栏中，选择[账号管理](#)。
  4. 单击[创建初始账号](#)。
  5. 在创建账号面板中，填写如下信息。

配置	说明
数据库账号	初始账号的名称，限制如下： <ul style="list-style-type: none"><li>◦ 由小写字母，数字，下划线组成。</li><li>◦ 以小写字母开头，小写字母或数字结尾。</li><li>◦ 不能以gp开头。</li><li>◦ 长度为2~16个字符。</li></ul>
新密码	初始账号的密码，限制如下： <ul style="list-style-type: none"><li>◦ 由大写字母、小写字母、数字、特殊字符其中三种及以上组成。</li><li>◦ 支持的特殊字符如下： !@#%\$^&amp;*()_+~=</li><li>◦ 长度为8~32个字符。</li></ul>
确认密码	再次输入密码。

6. 单击[确定](#)。

 **注意** 完成创建后您可以单击操作列的[重置密码](#)修改初始账号的密码。为保障数据安全，建议您定期更换密码，不要使用曾经用过的密码。

使用SQL语句创建账号

使用SQL语句创建账号需要连接数据库，连接数据库的操作，请参见[连接数据库](#)。

- 创建高权限账号，语句如下：

```
CREATE ROLE <账号名> WITH LOGIN ENCRYPTED PASSWORD <'密码'> RDS_SUPERUSER;
```

示例如下：

```
CREATE role admin0 WITH LOGIN ENCRYPTED PASSWORD '111111' rds_superuser;
```

- 创建普通账号，语句如下：

```
CREATE ROLE <账号名> WITH LOGIN ENCRYPTED PASSWORD <'密码'>;
```

示例如下：

```
CREATE role test1 WITH LOGIN ENCRYPTED PASSWORD '111111';
```

## 2.6. 获取客户端工具

云原生数据仓库AnalyticDB PostgreSQL版的接口协议兼容社区版的Greenplum Database和8.2版本的PostgreSQL。因此，您可以使用Greenplum或PostgreSQL的客户端连接AnalyticDB PostgreSQL版。

### 说明

由于专有云为隔离环境，用户从外部网络获取软件后，需要将其部署到内部环境中。

### 图形客户端工具

AnalyticDB PostgreSQL版用户可以直接使用支持Greenplum的客户端工具，例如[SQL Workbench](#)、[Navicat Premium](#)、[Navicat For PostgreSQL](#)等。

### 命令行客户端psql（RHEL或CentOS版本6和7平台）

对于RHEL（Red Hat Enterprise Linux）和CentOS版本6和7平台，可以通过以下地址进行下载，解压后即可使用：

- RHEL 6或CentOS 6平台，请单击[hybriddb\\_client\\_package\\_el6](#)进行下载。
- RHEL 7或CentOS 7平台，请单击[hybriddb\\_client\\_package\\_el7](#)进行下载。

### 命令行客户端psql（其它Linux平台）

适用于其它Linux平台的客户端工具的编译方法如下所示：

1. 获取源代码。有如下两种方法：

- 直接获取git目录（需要先安装git工具）。

```
git clone https://github.com/greenplum-db/gpdb.git
cd gpdb
git checkout 5d870156
```

- 直接下载代码。

```
wget https://github.com/greenplum-db/gpdb/archive/5d87015609abd330c68a5402c1267fc86bc9e1f.zip
unzip 5d87015609abd330c68a5402c1267fc86bc9e1f.zip
cd gpdb-5d87015609abd330c68a5402c1267fc86bc9e1f
```

2. 使用gcc等编译工具进行编译。

```
./configure
make -j32
make install
```

3. 使用psql和pg\_dump。这两个工具的路径如下：

psql: `/usr/local/pgsql/bin/psql`

pg\_dump: `/usr/local/pgsql/bin/pg_dump`

## 命令行客户端psql（Windows及其它平台）

Windows及其它平台的客户端工具，请到Pivotal网站下载[HybridDB Client](#)。

## 2.7. 连接数据库

Greenplum Database开源数据库基于PostgreSQL 8.2分支开发，完整兼容其消息协议，AnalyticDB PostgreSQL版同样基于此版本。因此，理论上讲，AnalyticDB PostgreSQL版用户可以直接使用支持PostgreSQL 8.2消息协议的工具，例如libpq、JDBC、ODBC、psycpg2等。

### 背景信息

AnalyticDB PostgreSQL版提供了Redhat平台的二进制psql程序，下载链接参见[获取客户端工具](#)。Greenplum官网也提供了一个安装包，包含JDBC、ODBC和libpq，方便安装和使用，详情参见[Greenplum官方文档](#)。

#### 说明

- 由于专有云为隔离环境，用户需要在环境中预先准备好必要的软件安装包。
- 默认情况下，AnalyticDB PostgreSQL版只能通过同一区域和可用区的ECS上的客户端进行访问。

### DMS

数据管理（Data Management Service，简称DMS）支持MySQL、SQL Server、PostgreSQL、PPAS、Petadata等关系型数据库，PolarDB-X等OLTP数据库，AnalyticDB、DLA等OLAP数据库和MongoDB、Redis等NoSQL的数据库管理。它是一种集数据管理、结构管理、用户授权、安全审计、数据趋势、数据追踪、BI图表、性能与优化和服务器管理于一体的数据管理服务。

以下内容将为您介绍如何使用DMS登录AnalyticDB PostgreSQL。

1. 登录[登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
2. （可选）创建AnalyticDB PostgreSQL实例。具体操作，请参见[创建实例](#)。  
若已创建AnalyticDB PostgreSQL实例，请跳过该步骤。
3. 找到目标实例，单击实例ID。
4. （可选）创建数据库账号。具体操作，请参见[创建数据库账号](#)。

如果已经创建初始账号，则可以使用该账号直接登录数据库。

5. 单击实例详情页右上角登录数据库。
6. 在登录实例页面输入数据库账号和数据库密码，单击登录。

## psql

psql是Greenplum中比较常用的工具，提供了丰富的命令，其二进制文件在Greenplum安装后的`bin`目录下。使用步骤如下：

1. 通过如下任意一种方式进行连接：

- 连接串的方式

```
psql "host=yourgpdbaddress.gpdb.rds.aliyuncs.com port=3432 dbname=postgres user=gpdba  
ccount password=gdpdbpassword"
```

- 指定参数的方式

```
psql -h yourgpdbaddress.gp.aliyun-inc.com -p 3432 -d postgres -U gpdbaccount
```

参数说明：

- -h：指定主机地址。
- -p：指定端口号。
- -d：指定数据库（默认的数据库是postgres）。
- -U：指定连接的用户。

可以通过 `psql --help` 查看更多选项。在psql中，可以执行 `\?` 查看更多psql中支持的命令。

2. 输入密码，进入psql的Shell界面。

```
postgres=>
```

参考文档如下：

- 关于Greenplum的psql的更多使用方法，请参见文档[psql](#)。
- AnalyticDB PostgreSQL版也支持PostgreSQL的psql命令，使用时请注意细节上的差异。详情参见[PostgreSQL 8.3.23 Documentation — psql](#)。

## JDBC

下载[PostgreSQL的官方JDBC](#)，下载之后加入到环境变量中。

代码示例如下：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class gp_conn {
    public static void main(String[] args) {
        try {
            Class.forName("org.postgresql.Driver");
            Connection db = DriverManager.getConnection("jdbc:postgresql://mygpdbpub.gpdb.rds.aliyuncs.com:3432/postgres",
                "mygpdb", "mygpdb");
            Statement st = db.createStatement();
            ResultSet rs = st.executeQuery(
                "select * from gp_segment_configuration;");
            while (rs.next()) {
                System.out.print(rs.getString(1));
                System.out.print(" | ");
                System.out.print(rs.getString(2));
                System.out.print(" | ");
                System.out.print(rs.getString(3));
                System.out.print(" | ");
                System.out.print(rs.getString(4));
                System.out.print(" | ");
                System.out.print(rs.getString(5));
                System.out.print(" | ");
                System.out.print(rs.getString(6));
                System.out.print(" | ");
                System.out.print(rs.getString(7));
                System.out.print(" | ");
                System.out.print(rs.getString(8));
                System.out.print(" | ");
                System.out.print(rs.getString(9));
                System.out.print(" | ");
                System.out.print(rs.getString(10));
                System.out.print(" | ");
                System.out.println(rs.getString(11));
            }
            rs.close();
            st.close();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

## Python

Python使用psycopg2连接Greenplum和PostgreSQL。操作步骤如下：

1. 安装psycopg2。在CentOS下，有如下三种安装方法：



- 方法一，执行如下命令：

```
yum -y install python-psycopg2
```

- 方法二，执行如下命令：

```
pip install psycopg2
```

- 方法三，从源码安装：

```
yum install -y postgresql-devel*
wget http://initd.org/psycpg/tarballs/PSYCOPG-2-6/psycpg2-2.6.tar.gz
tar xf psycpg2-2.6.tar.gz
cd psycpg2-2.6
python setup.py build
sudo python setup.py install
```

## 2. 设置 PYTHONPATH，之后就可以引用，如：

```
import psycopg2
sql = 'select * from gp_segment_configuration;'
conn = psycopg2.connect(database='gpdb', user='mygpdb', password='mygpdb', host='mygpdbpub.gpdb.rds.aliyuncs.com', port=3432)
conn.autocommit = True
cursor = conn.cursor()
cursor.execute(sql)
rows = cursor.fetchall()
for row in rows:
    print row
conn.commit()
conn.close()
```

会得到类似以下结果：

```
(1, -1, 'p', 'p', 's', 'u', 3022, '192.**.**.158', '192.**.**.158', None, None) (6, -1, 'm', 'm', 's', 'u', 3019, '192.**.**.47', '192.**.**.47', None, None) (2, 0, 'p', 'p', 's', 'u', 3025, '192.**.**.148', '192.**.**.148', 3525, None) (4, 0, 'm', 'm', 's', 'u', 3024, '192.**.**.158', '192.**.**.158', 3524, None) (3, 1, 'p', 'p', 's', 'u', 3023, '192.**.**.158', '192.**.**.158', 3523, None) (5, 1, 'm', 'm', 's', 'u', 3026, '192.**.**.148', '192.**.**.148', 3526, None)
```

## libpq

libpq是PostgreSQL数据库的C语言接口，在C程序中通过libpq库访问PostgreSQL数据库并进行数据库操作。在安装了Greenplum或者PostgreSQL之后，在其lib目录下可以找到其静态库和动态库。

相关案例请参见[PostgreSQL官方文档](#)，此处不再列举。

关于libpq详情，请参见“[PostgreSQL 9.4.17 Documentation — Chapter 31. libpq - C Library](#)”。

## ODBC

PostgreSQL的ODBC是基于LGPL（GNU Lesser General Public License）协议的开源版本，可以在[PostgreSQL官网](#)下载。

### 1. 安装驱动。

```
yum install -y unixODBC.x86_64
yum install -y postgresql-odbc.x86_64
```

## 2. 查看驱动配置。

```
cat /etc/odbcinst.ini
```

配置示例如下：

```
# Example driver definitions
# Driver from the postgresql-odbc package
# Setup from the unixODBC package
[PostgreSQL]
Description = ODBC for PostgreSQL
Driver = /usr/lib/psqlodbcw.so
Setup = /usr/lib/libodbcpsqlS.so
Driver64 = /usr/lib64/psqlodbcw.so
Setup64 = /usr/lib64/libodbcpsqlS.so
FileUsage = 1
# Driver from the mysql-connector-odbc package
# Setup from the unixODBC package
[MySQL]
Description = ODBC for MySQL
Driver = /usr/lib/libmyodbc5.so
Setup = /usr/lib/libodbcmyS.so
Driver64 = /usr/lib64/libmyodbc5.so
Setup64 = /usr/lib64/libodbcmyS.so
FileUsage = 1
```

## 3. 配置DSN，将如下代码中的 \*\*\*\* 改成对应的连接信息。

```
[mygpdb]
Description = Test to gp
Driver = PostgreSQL
Database = ****
Servername = ****.gpdb.rds.aliyuncs.com
Username = ****
Password = ****
Port = ****
ReadOnly = 0
```

## 4. 测试连通性。

```
echo "select count(*) from pg_class" | isql mygpdb
```

正常连接后返回信息如下：

```
+-----+
| Connected!                                |
|                                           |
| sql-statement                            |
| help [tablename]                         |
| quit                                     |
|                                           |
+-----+
```

测试是否可以正常使用：

```
SELECT count(*) FROM pg_class;
```

正常返回示例如下：

```
+-----+
| count |
+-----+
| 388   |
+-----+
SQLRowCount returns 1
1 rows fetched
```

5. ODBC已连接上实例，将应用连接ODBC即可，请参见[psqlODBC](#)和[C#连接到PostgreSQL](#)。


## 参考文档

- [Pivotal Greenplum官方文档](#)
- [PostgreSQL psqlODBC](#)
- [PostgreSQL ODBC编译](#)
- [Greenplum ODBC下载](#)
- [Greenplum JDBC下载](#)
- [The PostgreSQL JDBC Interface](#)

## 3. 实例

### 3.1. 重置账号密码

在使用AnalyticDB PostgreSQL版过程中，如果忘记数据库账号密码，可以通过AnalyticDB PostgreSQL版数据库管理控制台重新设置密码。

 **说明** 为保障数据安全，建议您定期更换密码。

1. [登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，选择**账号管理**。
4. 单击目标账号右侧**操作列**的**重置密码**。
5. 在**修改账号**面板输入新密码并确认密码。

账号密码限制如下：

- 由大写字母、小写字母、数字、特殊字符其中三种及以上组成。
- 支持的特殊字符如下：  
!@#\$%^&\*()\_+-=
- 长度为8~32个字符。

6. 单击**确定**。

### 3.2. 管理网络连接地址

本文介绍如何查看AnalyticDB PostgreSQL版实例的连接地址以及如何申请和释放外网地址。

#### 查看连接地址

1. [登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击**数据库连接**。
4. 在**数据库连接**区域，即可查看当前实例的连接地址。

#### 申请外网地址

1. [登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击**数据库连接**。
4. 单击**外网地址**右侧的**申请外网地址**即可。

#### 释放外网地址

1. [登录云原生数据仓库AnalyticDB PostgreSQL版控制台](#)。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击**数据库连接**。

4. 单击外网地址右侧的释放外网地址。
5. 在释放外网地址对话框中，单击确定。

## 3.3. 监控与报警

为方便用户掌握实例的运行状态，AnalyticDB PostgreSQL版控制台提供了监控与报警功能。

### 操作步骤

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击监控与报警。

监控与报警页面展示了以下指标信息。

- 监控汇总页签

在该页面您可以查看CPU和内存利用率（%）、当前Master总连接数、IO吞吐量和磁盘空间（MB）。

- 计算组监控页签或计算节点监控页签

计算存储一体模式实例为计算组监控；存储预留模式为计算节点监控。

在该页面您可以查看CPU和内存利用率（%）、写IOPS吞吐量、读IOPS吞吐量和磁盘空间（MB）。

4. （可选）选择不同的查询时间来看距今某一时间段内各项指标的情况，支持的支持的查询时间如下：
  - 1小时：查看最近一个小时的监控信息。
  - 1天：查看最近一天的监控信息。
  - 自定义：查看指定时间范围的监控信息，最长时间跨度为七天。

## 3.4. 变更配置

本文将分别为您介绍计算存储一体模式实例和存储预留模式实例如何变更配置。

### 计算存储一体模式

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击右侧操作列下的变配。
3. 选择需要变更的计算机规格和计算机节点。
4. 单击提交。



**警告** 实例在变配期间将不可读写，请合理安排您的变配任务。

### 存储预留模式

Master节点变配：


1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击右侧操作列下的Master节点变配。
3. 选择需要变更的master节点数量。

4. 单击提交。

 **警告** Master节点在变配期间会短暂阻塞DDL操作，请合理安排您的变配任务。

计算节点变配：

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击右侧操作列下的计算节点变配。
3. 选择需要变更的计算机规格和计算节点数量。
4. 单击提交。

 **警告** 计算节点在变配期间实例将不可读写，请合理安排您的变配任务。


## 3.5. 重启实例

当实例出现连接数满或性能问题时，您可以手动重启实例。

### 注意事项

重启过程大约耗费3到30分钟，在此过程中该实例不对外提供服务，请您提前进行调整。当实例重启结束，对应实例恢复运行中状态，您可以正常访问数据库。

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 单击页面右上角重启实例。
4. 在重启实例对话框中，单击确定。


 **警告** 重启实例将中断数据库服务，请谨慎操作。

## 3.6. 释放实例

您可以根据业务需求手动释放AnalyticDB PostgreSQL版实例。

### 操作步骤

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在基本信息页面的运行状态区域。单击释放。
4. 在释放实例对话框中，单击确定。

 **警告** 实例释放后，所有数据将不再保留，请您在释放前妥善备份您的数据。

## 3.7. SQL审计

AnalyticDB PostgreSQL版控制台提供SQL审计功能，您可通过该功能查看SQL明细、定期审计SQL。开通SQL审计功能后，实例性能不会受到影响。



## 计算存储一体模式

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在实例详情页，单击左侧导航栏的数据安全性。
4. 单击SQL审计页签。
5. （可选）单击开启SQL审计，开启SQL审计。  
如果您已开启SQL审计功能，可跳过本步骤。
6. 在SQL审计页签，您可以通过选择时间范围、DB、User、关键字条件查询SQL信息。

 说明 您可以单击关闭SQL审计关闭SQL审计功能。

## 存储预留模式

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在实例详情页，单击左侧导航栏的SQL审计。
4. 在SQL审计页面，您可以通过数据库名、执行时长、选择时间范围、数据库用户、源端IP、执行状态、操作类别、操作类型、语句文本条件查询SQL信息。

 说明 您可以单击图标导出当前页面的SQL信息。

# 3.8. SSL


为提高链路的安全性，您可以启用SSL（Secure Sockets Layer）加密，SSL加密功能在传输层对网络连接进行加密，在提升通信数据安全性的同时，保证数据的完整性。

## 注意事项

目前仅存储预留模式实例支持SSL加密。


## 开启SSL证书

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击数据安全性。
4. 单击SSL页签。
5. 在SSL页签，单击SSL证书信息右侧开关开启SSL功能。
6. 在开启SSL证书对话框中，单击确定。

 说明 修改SSL状态会重启您的实例，为了不影响业务的运行，建议您在业务低谷期进行操作。


## 更新SSL证书有效期

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击数据安全性。
4. 单击SSL页签。
5. 在SSL页签，单击SSL证书信息右侧的更新有效期。
6. 在更新SSL证书有效期对话框中，单击确定。

 **说明** 修改SSL状态会重启您的实例，为了不影响业务的运行，建议您在业务低谷期进行操作。

## 关闭SSL证书

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在左侧导航栏中，单击数据安全性。
4. 单击SSL页签。
5. 在SSL页签，单击SSL证书信息右侧滑块关闭SSL功能。
6. 在关闭SSL证书对话框中，单击确定。

 **说明** 修改SSL状态会重启您的实例，为了不影响业务的运行，建议您在业务低谷期进行操作。

## 3.9. 版本升级


为了更好的满足您的需求，AnalyticDB PostgreSQL版会不断更新数据库内核版本。在实例创建时，默认使用的是最新版本的数据库内核。当新的内核版本发布后，您可以通过小版本升级实例来更新数据库内核版本，从而使用新版本中扩展的功能。本文将介绍如何升级小版本。

### 注意事项

实例升级小版本时，会执行实例重启操作，在重启期间实例将不可用，请在业务低谷时执行该操作。

### 操作步骤

1. 登录云原生数据仓库AnalyticDB PostgreSQL版控制台。
2. 找到目标实例，单击实例ID。
3. 在基本信息页面，单击右上方的小版本升级。
4. 在小版本升级对话框中，单击确定。

 **说明** 小版本升级过程一般耗时3到15分钟，在此过程中该实例不能对外提供服务，请您提前做好调整。当升级结束，对应实例恢复运行中状态，您可以正常访问数据库。

5. 完成了上述操作后，您可以回到控制台查看目标实例的运行状态。如果实例小版本升级操作完成，则实例运行状态为运行中，否则为版本升级中。

小版本升级时会检查您的实例当前的版本号，如果您的实例已经是当前最新的小版本，实例的升级和重启过程将会被跳过。



## 3.10. 导入数据

### 3.10.1. OSS高速并行导入

云原生数据仓库AnalyticDB PostgreSQL版支持通过OSS外部表（即gpossex功能），将数据并行从OSS导入或导出到OSS，并支持通过gzip进行OSS外部表文件压缩，大量节省存储空间及成本。目前的gpossex支持读写text/csv格式的文件或者gzip压缩格式的text/csv文件。

- 创建OSS外部表插件（oss\_ext）

使用OSS外部表时，需要在AnalyticDB PostgreSQL版中先创建OSS外部表插件（每个数据库需要单独创建）。

- 创建命令为：

```
CREATE EXTENSION IF NOT EXISTS oss_ext;
```
- 删除命令为：

```
DROP EXTENSION IF EXISTS oss_ext;
```

- 并行导入数据

- i. 将数据均匀分散存储在多个OSS文件中，文件的数目最好为AnalyticDB PostgreSQL版数据节点数（Segment个数）的整数倍。
- ii. 在AnalyticDB PostgreSQL版中，创建READABLE外部表。
- iii. 执行如下操作，并行导入数据。

```
INSERT INTO <目标表> SELECT * FROM <外部表>
```

#### ② 说明

- 数据导入的性能和AnalyticDB PostgreSQL版集群的资源（CPU、IO、内存、网络等）相关，也和OSS相关。为了获取最大的导入性能，建议在创建表时，使用列式存储+压缩功能。例如，指定子句 `WITH (APPENDONLY=true, ORIENTATION=column, COMPRESSTYPE=zlib, COMPRESSLEVEL=5, BLOCKSIZE=1048576)`，具体信息，请参见[Greenplum Database表创建语法官方文档](#)。
- 为保证数据导入的性能，OSS访问域名的Region需要匹配AnalyticDB PostgreSQL版的Region，建议OSS和AnalyticDB PostgreSQL版在同一个Region内以获得最好的性能。

- 并行导出数据

- i. 在AnalyticDB PostgreSQL版中，创建WRITABLE外部表。
- ii. 执行如下操作，并行把数据导出到OSS中。

```
INSERT INTO <外部表> SELECT * FROM <源表>
```

- 创建OSS外部表语法

② 说明 创建和使用外部表的语法，除了location相关的参数，其余部分和Greenplum相同。

```
CREATE [READABLE] EXTERNAL TABLE tablename
( columnname datatype [, ...] | LIKE othertable )
LOCATION ('ossprotocol')
FORMAT 'TEXT'
    [( [HEADER]
    [DELIMITER [AS] 'delimiter' | 'OFF']
    [NULL [AS] 'null string']
    [ESCAPE [AS] 'escape' | 'OFF']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [FILL MISSING FIELDS] )]
| 'CSV'
    [( [HEADER]
    [QUOTE [AS] 'quote']
    [DELIMITER [AS] 'delimiter']
    [NULL [AS] 'null string']
    [FORCE NOT NULL column [, ...]]
    [ESCAPE [AS] 'escape']
    [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
    [FILL MISSING FIELDS] )]
[ ENCODING 'encoding' ]
[ [LOG ERRORS [INTO error_table]] SEGMENT REJECT LIMIT count
[ROWS | PERCENT] ]
CREATE WRITABLE EXTERNAL TABLE table_name
( column_name data_type [, ...] | LIKE other_table )
LOCATION ('ossprotocol')
FORMAT 'TEXT'
    [( [DELIMITER [AS] 'delimiter']
    [NULL [AS] 'null string']
    [ESCAPE [AS] 'escape' | 'OFF'] )]
| 'CSV'
    [( [QUOTE [AS] 'quote']
    [DELIMITER [AS] 'delimiter']
    [NULL [AS] 'null string']
    [FORCE QUOTE column [, ...]] ]
    [ESCAPE [AS] 'escape'] )]
[ ENCODING 'encoding' ]
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
ossprotocol:
    oss://oss_endpoint prefix=prefix_name
    id=userossid key=userosskey bucket=ossbucket compressiontype=[none|gzip] async=[true|false]
ossprotocol:
    oss://oss_endpoint dir=[folder/[folder/]...]/file_name
    id=userossid key=userosskey bucket=ossbucket compressiontype=[none|gzip] async=[true|false]
ossprotocol:
    oss://oss_endpoint filepath=[folder/[folder/]...]/file_name
    id=userossid key=userosskey bucket=ossbucket compressiontype=[none|gzip] async=[true|false]
```

## 参数解释

### 常用参数解释

参数	说明
协议和endpoint	<p>格式为 “协议名://oss_endpoint” ，协议名为oss。oss_endpoint为OSS 对应区域的域名。</p> <p> <b>说明</b> 如果是用虚拟专有网络的主机进行访问，应该使用内网域名（即带有“internal”的域名），避免产生公网流量。</p>
id	OSS账号的ID。
key	OSS账号的key。
bucket	指定数据文件所在的bucket，需要通过OSS预先创建。
prefix	<p>指定数据文件对应路径名的前缀，不支持正则表达式，仅是匹配前缀，且与filepath、dir互斥，三者只能设置其中一个。</p> <ul style="list-style-type: none"> <li>如果创建的是用于数据导入的READABLE外部表，则在导入时含有这一前缀的所有OSS 文件都会被导入。 <ul style="list-style-type: none"> <li>如果指定prefix=test/filename，以下文件都会被导入： <ul style="list-style-type: none"> <li>test/filename</li> <li>test/filenameexxx</li> <li>test/filename/aa</li> <li>test/filenameyyy/aa</li> <li>test/filenameyyy/bb/aa</li> </ul> </li> <li>如果指定 prefix=test/filename/，只有以下文件会被导入（上面列的其他文件不会被导入）： <p>test/filename/aa</p> </li> </ul> </li> <li>如果创建的是用于数据导出的WRITABLE 外部表，在导出数据时，将根据该前缀自动生成一个唯一的文件名来给导出文件命名。</li> </ul> <p> <b>说明</b> 导出文件将不止有一个，每个数据节点都会导出一个或多个文件。导出文件名格式为 <code>prefix_tablename_uuid.x</code>，其中uuid是生成的int64整型值（微秒时间戳），x为节点ID。支持使用同一外部表多次导出，每次导出的文件将通过uuid区分，而同一次导出的文件uuid相同。</p>

参数	说明
dir	<p>OSS中的虚拟文件夹路径，与prefix、filepath互斥，三者只能设置其中一个。</p> <ul style="list-style-type: none"><li>文件夹路径需要以“/”结尾，如 <code>test/mydir/</code>。</li><li>在导入数据时，使用此参数创建外部表，会导入指定虚拟目录下的所有文件，但不包括它的子目录和子目录下的文件。与filepath不同，dir下的文件没有命名要求。</li><li>在导出数据时，使用此参数创建外部表，所有数据会导出到此目录下的多个文件中，输出文件名的形式为 <code>filename.x</code>，x为数字，但可能不是连续的。</li></ul>
filepath	<p>OSS中带路径的文件名，与prefix、dir互斥，三者只能设置其中一个，且filepath只能在创建READABLE外部表时指定（即只支持在导入数据时使用）。</p> <ul style="list-style-type: none"><li>文件名包含文件路径，但不包含bucket名。</li><li>在导入数据时，文件命名方式必须为 <code>filename</code> 或 <code>filename.x</code>，x要求从1开始，且是连续的。</li></ul> <p>例如，如果指定 <code>filepath = filename</code>，而OSS中含有如下文件，则将被导入的文件有 <code>filename</code>、<code>filename.1</code>和<code>filename.2</code>，而因为<code>filename.3</code>不存在，<code>filename.4</code> 不会被导入。</p> <pre>filename filename.1 filename.2 filename.4</pre>

导入模式参数解释

参数	说明
async	<p>是否启用异步模式装载数据。</p> <ul style="list-style-type: none"><li>默认情况下异步模式是打开的，如果需要关掉，可以使用参数<code>async = false</code>或<code>async = f</code>。</li><li>开启辅助线程从OSS装载数据，加速导入性能，默认导入模式是异步模式。</li><li>异步模式和普通模式比，会消耗更多的硬件资源。</li></ul>
compressiontype	<p>导入文件的压缩格式。</p> <ul style="list-style-type: none"><li>none（缺省值）：说明导入的文件没经过压缩。</li><li>gzip：则导入的格式为gzip。目前仅支持gzip压缩格式。</li></ul>
compressionlevel	<p>设置写入OSS的文件的压缩等级，取值范围为1~9，默认值为6。</p>

导出模式参数说明

参数	说明
oss_flush_block_size	单次刷出数据到OSS的buffer大小，默认为32 MB，可选范围是1 MB~128 MB。
oss_file_max_size	写到OSS的最大文件大小，超出之后会切换到另一个文件继续写。默认为1024 MB，可选范围是8 MB~4000 MB。
num_parallel_worker	设置写入OSS的压缩数据的并行压缩线程个数，取值范围为1~8，默认值为3。

另外，针对导出模式，需要注意如下事项：

- WRITABLE是导出模式外部表的关键字，创建外部表时需要明确指明。
- 导出模式目前只支持prefix和dir参数模式，不支持filepath。
- 导出模式的DISTRIBUTED BY子句可以使数据节点（Segment）按指定的分布键将数据写入OSS。

#### 其他通用参数

针对导入模式和导出模式，还有以下容错相关参数。

#### 容错相关参数

参数	说明
oss_connect_timeout	设置连接超时，单位秒，默认是10秒。
oss_dns_cache_timeout	设置DNS超时，单位秒，默认是60秒。
oss_speed_limit	控制能容忍的最小速率，默认是1024，即1 K。
oss_speed_time	控制能容忍的最长时间，默认是15秒。

上述参数如果使用默认值，则如果连续15秒的传输速率小于1 K，就会触发超时。详细描述请参见对象存储OSS开发指南C-SDK中 [错误处理](#) 章节。

其他参数兼容Greenplum EXTERNAL TABLE的原有语法，具体语法解释请参见[Greenplum外部表语法](#)。这部分参数主要有：

- FORMAT：支持文件格式，支持text、csv等。
- ENCODING：文件中数据的编码格式，如utf8。
- LOG ERRORS：指定该子句可以忽略掉导入中出错的数据，将这些数据写入error\_table，并可以使用count参数指定报错的阈值。

#### 使用示例

1. 创建OSS导入外部表。

```
CREATE READABLE EXTERNAL TABLE ossexample
(date text, time text, open float, high float,
 low float, volume int)
location('oss://oss-cn-hangzhou.aliyuncs.com
prefix=osstest/example id=XXX
key=XXX bucket=testbucket compressiontype=gzip')
FORMAT 'csv' (QUOTE '' DELIMITER E'\t')
ENCODING 'utf8'
LOG ERRORS INTO my_error_rows SEGMENT REJECT LIMIT 5;
CREATE READABLE EXTERNAL TABLE ossexample
(date text, time text, open float, high float,
 low float, volume int)
location('oss://oss-cn-hangzhou.aliyuncs.com
dir=osstest/ id=XXX
key=XXX bucket=testbucket')
FORMAT 'csv'
LOG ERRORS SEGMENT REJECT LIMIT 5;
CREATE READABLE EXTERNAL TABLE ossexample
(date text, time text, open float, high float,
 low float, volume int)
location('oss://oss-cn-hangzhou.aliyuncs.com
filepath=osstest/example.csv id=XXX
key=XXX bucket=testbucket')
FORMAT 'csv'
LOG ERRORS SEGMENT REJECT LIMIT 5;
```

## 2. 创建OSS导出外部表。

```
CREATE WRITABLE EXTERNAL TABLE ossexample_exp
(date text, time text, open float, high float,
 low float, volume int)
location('oss://oss-cn-hangzhou.aliyuncs.com
prefix=osstest/exp/outfromhdb id=XXX
key=XXX bucket=testbucket') FORMAT 'csv'
DISTRIBUTED BY (date);
CREATE WRITABLE EXTERNAL TABLE ossexample_exp
(date text, time text, open float, high float,
 low float, volume int)
location('oss://oss-cn-hangzhou.aliyuncs.com
dir=osstest/exp/ id=XXX
key=XXX bucket=testbucket') FORMAT 'csv'
DISTRIBUTED BY (date);
```

## 3. 创建堆表，数据就装载到这张表中。

```
CREATE TABLE example
(date text, time text, open float,
 high float, low float, volume int)
DISTRIBUTED BY (date);
```

## 4. 将数据并行地从ossexample表装载到example表中。

```
INSERT INTO example SELECT * FROM ossexample;
```

## 5. 将数据并行地从example表导出到OSS。

```
INSERT INTO ossexample_exp SELECT * FROM example;
```

6. 执行查询计划，从以下查询计划可以看出每个Segment都会参与工作。

```
EXPLAIN INSERT INTO example SELECT * FROM ossexample;
```

每个Segment从OSS并行拉取数据，然后通过Redistribution Motion这个执行节点将拿到的数据HASH计算后分发给对应的Segment，接受数据的Segment通过Insert执行节点进行入库。返回信息如下：

```
QUERY PLAN
-----
Insert (slice0; segments: 4) (rows=250000 width=92)
  -> Redistribute Motion 4:4 (slice1; segments: 4) (cost=0.00..11000.00 rows=250000 width=92)
      Hash Key: ossexample.date
      -> External Scan on ossexample (cost=0.00..11000.00 rows=250000 width=92)
(4 rows)
```

```
EXPLAIN INSERT INTO ossexample_exp SELECT * FROM example;
```

从下面的查询计划可以看出，Segment把本地数据直接导出到OSS，没有进行数据重分布。返回信息如下：

```
QUERY PLAN
-----
Insert (slice0; segments: 3) (rows=1 width=92)
  -> Seq Scan on example (cost=0.00..0.00 rows=1 width=92)
(2 rows)
```

## TEXT/CSV格式说明


下列几个参数可以在外表DDL参数中指定，用于规定读写OSS上的文件格式：

- TEXT/CSV的行分隔符是'\n'，也就是换行符。
- DELIMITER用于定义列的分隔符。
  - 当用户数据中包括DELIMITER时，则需要配合QUOTE参数。
  - 推荐的列分隔符有','、'\t'、'|'或一些不常出现的字符。
- QUOTE以列为单位包裹有特殊字符的用户数据。
  - 用户包含有特殊字符的字符串必须用QUOTE包裹，用于区分用户数据和控制字符。
  - 如果不必要，例如整数，数据不必用QUOTE包裹（用于优化效率）。
  - QUOTE不能和DELIMITER相同，默认QUOTE是双引号。
  - 当用户数据中包含了QUOTE字符，则需要使用转义字符ESCAPE加以区分。
- ESCAPE特殊字符转义。
  - 转义字符出现在需要转义的特殊字符前，表示它不是一个特殊字符。
  - ESCAPE默认和QUOTE相同，也就是双引号。
  - 也支持设置成'\（MySQL默认的转义字符）或别的字符。

## 典型的TEXT/CSV默认控制字符

典型的TEXT/CSV默认控制字符

控制字符\格式	TEXT	CSV
DELIMITER（列分割符）	\t（tab）	,（comma）
QUOTE（摘引）	“（double-quote）	“（double-quote）
ESCAPE（转义）	（不适用）	和QUOTE相同
NULL（空值）	\N（backslash-N）	（无引号的空字符串）

 **说明** 所有的控制字符都必须是单字节字符。

SDK错误处理

当导入或导出操作出错时，错误日志可能会出现如所示信息。

错误日志信息

关键词	说明
code	出错请求的HTTP状态码。
error_code	OSS的错误码。
error_msg	OSS的错误信息。
req_id	标识该次请求的UUID。当您无法解决问题时，可以凭req_id来请求OSS开发工程师的帮助。

超时相关的错误可以使用oss\_ext相关参数处理。

参考文档

- [Greenplum Database外部表语法官方文档](#)
- [Greenplum Database表创建语法官方文档](#)

3.10.2. 从MySQL导入

工具mysql2pgsql支持不落地的把MySQL中的表迁移到AnalyticDB PostgreSQL版/Greenplum Database/PostgreSQL/PPAS。

背景信息

mysql2pgsql工具的原理是，同时连接源端MySQL数据库和目的端AnalyticDB PostgreSQL版数据库，从MySQL库中通过查询得到要导出的数据，然后通过COPY 命令导入到目的端。此工具支持多线程导入（每个工作线程负责导入一部分数据库表）。

下载[mysql2pgsql](#)二进制安装包。

查看[mysql2pgsql](#)源码编译说明。




## 操作步骤

- 1. 修改配置文件my.cfg，配置源库和目标库连接信息。
  - i. 修改源库mysql的连接信息。

 说明 MySQL源库的连接信息中，用户需要对所有用户表的读权限。

```
[src.mysql]
host = "192.168.1.1"
port = "3306"
user = "test"
password = "test"
db = "test"
encodingdir = "share"
encoding = "utf8"
```

- ii. 修改目标库pgsql（包括Postgresql、PPAS和AnalyticDB PostgreSQL版）的连接信息。

 说明 目标库pgsql的连接信息，用户需要对目标表有写权限。

```
[desc.pgsql]
connect_string = "host=192.168.1.2 dbname=test port=3432 user=test password=pgsql"
```

- 2. 通过mysql2pgsql导入数据。

```
./mysql2pgsql -l <tables_list_file> -d -n -j <number of threads> -s <schema of target a
ble>
```

## 参数说明

参数	说明
-l	可选参数，指定一个文本文件，文件中含有需要同步的表；如果不指定此参数，则同步配置文件中指定数据库下的所有表。<tables_list_file> 为一个文件名，里面含有需要同步的表集合以及表上查询的条件，其内容格式示例如下： <div><pre>table1 : select * from table_big where column1 &lt; '2016-08-05' table2 : table3 table4: select column1, column2 from tableX where column1 != 10 table5: select * from table_big where column1 &gt;= '2016-08-05'</pre></div>
-d	可选参数，表示只生成目标表的建表DDL语句，不实际进行数据同步。
-n	可选参数，需要与-d一起使用，指定在DDL语句中不包含表分区定义。
-j	可选参数，指定使用多少线程进行数据同步；如果不指定此参数，会使用5个线程并发。
-s	可选参数，指定目标表的schema，一次命令只能指定一个schema。如果不指定此参数，则数据会导入到public下的表。

## 典型用法

### 全库迁移

1. 通过如下命令，获取目的端对应表的DDL。

```
./mysql2pgsql -d
```

2. 根据这些DDL，再加入distribution key等信息，在目的端创建表。

3. 执行如下命令，同步所有表：

```
./mysql2pgsql
```

此命令会把配置文件中指定数据库中的所有MySQL表数据迁移到目的端。过程中使用5个线程（即默认线程数为5），读取和导入所有涉及的表数据。

### 部分表迁移

1. 编辑一个新文件tab\_list.txt，放入如下内容：

```
t1
t2 : select * from t2 where c1 > 138888
```

2. 执行如下命令，同步指定的t1和t2表（注意t2表只迁移符合c1 > 138888 条件的数据）：

```
./mysql2pgsql -l tab_list.txt
```

## 3.10.3. 从PostgreSQL导入

工具pgsql2pgsql支持将AnalyticDB PostgreSQL版/Greenplum Database/PostgreSQL/PPAS中的表迁移到AnalyticDB PostgreSQL版/Greenplum Database/PostgreSQL/PPAS。

### 背景信息

pgsql2pgsql支持如下功能：


- PostgreSQL/PPAS/Greenplum Database/AnalyticDB PostgreSQL版全量数据迁移到PostgreSQL/PPAS/Greenplum Database/AnalyticDB PostgreSQL版。
- PostgreSQL/PPAS（版本大于9.4）全量+增量迁移到PostgreSQL/PPAS。

请在[dbsync项目](#)库中下载软件包：

- 下载[pgsql2pgsql](#)二进制安装包。
- 查看[pgsql2pgsql](#)源码编译说明。

### 操作步骤

1. 修改配置文件my.cfg，配置源和目的库连接信息。
  - i. 修改源库pgsql连接信息。

 说明 源库pgsql的连接信息中，用户最好是对应DB的owner。

```
[src.pgsql]
connect_string = "host=192.168.0.1 dbname=test port=3432 user=test password=pgsql"
```

## ii. 修改本地临时Database postgresql连接信息。

```
[local.postgresql]
connect_string = "host=192.168.0.2 dbname=test port=3432 user=test2 password=postgresql"
```

## iii. 修改目的库postgresql连接信息。

② 说明 目的库postgresql的连接信息，用户需要对目标表有写权限。

```
[desc.postgresql]
connect_string = "host=192.168.0.2 dbname=test port=3432 user=test3 password=postgresql"
```

② 说明

- 如果要做增量数据同步，连接源库需要有创建replication slot的权限。
- 由于PostgreSQL 9.4及以上版本支持逻辑流复制，所以支持作为数据源的增量迁移。打开下列内核参数才能让内核支持逻辑流复制功能。

```
wal_level = logical
max_wal_senders = 6
max_replication_slots = 6
```

## 2. 使用pgsql2pgsql进行全库迁移。

```
./pgsql2pgsql
```

迁移程序会默认把对应pgsql库中所有用户的表数据将迁移到pgsql。

## 3. 查询状态信息。

连接本地临时Database，可以查看到单次迁移过程中的状态信息。这些信息被放在表db\_sync\_status中，包括全量迁移的开始和结束时间、增量迁移的开始时间和增量同步的数据情况。

## 3.10.4. 通过COPY数据导入

用户可以直接使用 `\COPY` 命令，将本地的文本文件数据导入云原生数据仓库AnalyticDB PostgreSQL版。但要求用户本地的文本文件是格式化的，如通过英文半角逗号(,)、英文半角分号(;)或特有符号作为分割符号的文件。

### 背景信息

- 由于 `\COPY` 命令需要通过 Master 节点进行串行数据写入处理，因此无法实现并行写入大批量数据。如果要进行大量数据的并行写入，请使用基于OSS的数据导入方式。
- `\COPY` 命令是psql的操作指令，如果您使用的不是 `\COPY`，而是数据库指令 `COPY`，则需要注意只支持 STDIN，不支持file，因为“根用户”并没有superuser权限，不可以进行file文件操作。
- 云原生数据仓库AnalyticDB PostgreSQL版还支持用户使用JDBC执行COPY语句，JDBC中封装了CopyIn方法，详细用法请参见文档“[Interface CopyIn](#)”。
- COPY 命令使用方法请参见文档[COPY](#)。

## 操作步骤

1. 使用类似如下COPY命令将数据导入。

```
\COPY table [(column [, ...])] FROM {'file' | STDIN}
[ [WITH]
  [OIDS]
  [HEADER]
  [DELIMITER [ AS ] 'delimiter']
  [NULL [ AS ] 'null string']
  [ESCAPE [ AS ] 'escape' | 'OFF']
  [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
  [CSV [QUOTE [ AS ] 'quote']
    [FORCE NOT NULL column [, ...]]
  [FILL MISSING FIELDS]
  [[LOG ERRORS [INTO error_table] [KEEP]
  SEGMENT REJECT LIMIT count [ROWS | PERCENT] ]
\COPY {table [(column [, ...])] | (query)} TO {'file' | STDOUT}
[ [WITH]
  [OIDS]
  [HEADER]
  [DELIMITER [ AS ] 'delimiter']
  [NULL [ AS ] 'null string']
  [ESCAPE [ AS ] 'escape' | 'OFF']
  [CSV [QUOTE [ AS ] 'quote']
    [FORCE QUOTE column [, ...]] ]
[IGNORE EXTERNAL PARTITIONS ]
```

## 4. 数据库

### 4.1. 概述

对基于Greenplum Database的操作，AnalyticDB PostgreSQL版与Greenplum Database开源数据库基本一致，包括schema、类型支持、用户权限等。除了Greenplum Database开源数据库一些特有的操作，如分布键、AO表等，其它操作与原生PostgreSQL一致。

#### 参考

- [Pivotal Greenplum官方文档](#)
- [GP 4.3 Best Practice](#)
- [Greenplum数据分布黄金法则](#)

### 4.2. 创建数据库

登录数据库后，用户可以使用SQL语句创建数据库。

AnalyticDB PostgreSQL版中创建数据库的操作与PostgreSQL相同，可以通过SQL来执行，例如在psql连接到Greenplum后执行如下命令：

```
CREATE DATABASE mygpdb;
```

查看数据库信息：

```
\c mygpdb
```

返回信息如下：

```
You are now connected to database "mygpdb" as user "mygpdb".
```

### 4.3. 创建分布键

由于AnalyticDB PostgreSQL版是一个分布式的数据库，数据是分散存储在各个数据节点的，所以需要告诉AnalyticDB PostgreSQL版数据应该如何分布。分布键对于查询性能至关重要。均匀为分布键选择的第一大原则，选取更有业务意义的字段，将有助于显著提高性能。

#### 指定分布键

在AnalyticDB PostgreSQL版中，表分布在所有的Segment上，其分布规则是HASH或者随机。在建表时，指定分布键；当导入数据时，会根据分布键计算得到的HASH值分配到特定的Segment上。

```
CREATE TABLE vtbl(id serial, key integer, value text, shape cuboid, location geometry, comment text) DISTRIBUTED BY (key);
```

当不指定分布键时（即不带后面的DISTRIBUTED BY时），AnalyticDB PostgreSQL版会默认对id字段以Round-Robin的方式进行随机分配。

#### 选择分布键的法则

- 尽量选择分布均匀的列、或者多列，以防止数据倾斜。
- 尽量选择常用于连接运算的字段，对于并发较高的语句，该选择更为重要。
- 尽量选择高并发查询、过滤性高的条件列。
- 不要轻易使用随机分布。

## 4.4. 构造数据

在某些测试场景下，我们需要构造数据来填充数据库。

1. 创建函数，用于生成随机字符串。

```
CREATE OR REPLACE FUNCTION random_string(integer) RETURNS text AS $body$
SELECT array_to_string(array
                        (SELECT substring('0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
                                          FROM (ceil(random()*62))::int
                                          FOR 1)
                        FROM generate_series(1, $1)), '');
$body$
LANGUAGE SQL VOLATILE;
```

2. 创建分布键。

```
CREATE TABLE tbl(id serial, KEY integer, locate geometry, COMMENT text) distributed by
(key);
```

3. 构造数据。

```
INSERT INTO tbl(KEY, COMMENT, locate)
SELECT
    KEY,
    COMMENT,
    ST_GeomFromText(locate) AS locate
FROM
    (SELECT
        (a + 1) AS KEY,
        random_string(ceil(random() * 24)::integer) AS COMMENT,
        'POINT(' || ceil(random() * 36 + 99) || ' ' || ceil(random() * 24 + 50) || ') '
    AS locate
    FROM
        generate_series(0, 99999) AS a)
AS t;
```

## 4.5. 查询

介绍查询语句，以及如何查看查询计划。

### 查询语句示例

```
SELECT * FROM tbl WHERE key = 751;
```

返回示例如下：

```

| id | key | value | shape | locate | comment |
|-----+-----+-----+-----+-----+-----+
| 751 | 751 | red   | 01010000000000000000C05B4000000000004A40 | B9hPhjeNWPqV |
(1 row)
Time: 513.101 ms

```

## 查看查询计划

```
EXPLAIN SELECT * FROM tbl WHERE key = 751;
```

返回示例如下：

```

Gather Motion 1:1 (slice1; segments: 1) (cost=0.00..1519.28 rows=1 width=53)
-> Seq Scan on tbl (cost=0.00..1519.28 rows=1 width=53)
    Filter: key = 751
Settings:  effective_cache_size=8GB; gp_statistics_use_fkeys=on
Optimizer status: legacy query optimizer

```

## 4.6. 管理插件

用户可以通过插件扩展数据库的功能，AnalyticDB PostgreSQL版提供了便捷的插件管理功能。

### 插件类型

云原生数据仓库AnalyticDB PostgreSQL版支持如下插件：

- PostGIS：支持地理信息数据。
- MADlib：机器学习方面的函数库。
- fuzzystrmatch：字符串模糊匹配。
- orafunc：兼容Oracle的部分函数。
- oss\_ext：支持从OSS读取数据。
- hll：支持用HyperLogLog算法进行统计。
- pljava：支持使用PL/Java语言编写用户自定义函数（UDF）。
- pgcrypto：支持加密函数。
- intarray：整数数组相关的函数、操作符和索引支持。

### 创建插件

创建插件的方法如下所示：

```

CREATE EXTENSION <extension name>;
CREATE SCHEMA <schema name>;
CREATE EXTENSION IF NOT EXISTS <extension name> WITH SCHEMA <schema name>;

```

### ② 说明

创建MADlib插件时，需要先创建plpythonu插件，如下所示：

```
CREATE EXTENSION plpythonu;
CREATE EXTENSION madlib;
```

## 删除插件

删除插件的方法如下所示：

```
DROP EXTENSION <extension name>;
DROP EXTENSION IF EXISTS <extension name> CASCADE;
```

② 说明 如果插件被其它对象依赖，需要加入CASCADE（级联）关键字，删除所有依赖对象。

## 4.7. 管理用户和权限

介绍云原生数据仓库AnalyticDB PostgreSQL版如何管理用户和权限。

### 管理用户

实例创建过程中，会提示您指定初始用户名和密码，这个初始用户为“根用户”。实例创建好后，您可以使用该根用户连接数据库。同时，系统也会创建aurora、replicator等超级用户，用于内部管理。

使用PostgreSQL或Greenplum的客户端工具连接数据库后，通过 `\du+` 命令可以查看所有用户的信息，返回示例如下：

List of roles			
Role name	Attributes	Member of	Description
root_user		rds_superuser	
...			

目前，AnalyticDB PostgreSQL版没有开放SUPERUSER权限，对应的是RDS\_SUPERUSER，这一点与云数据库RDS（PostgreSQL）中的权限体系一致。所以，根用户（如上面示例中的 root\_user）具有RDS\_SUPERUSER权限，这个权限属性只能通过查看用户的描述（Description）来识别。

根用户具有如下权限：

- 具有CREATEROLE、CREATEDB和LOGIN权限，即可以用来创建数据库和用户，但没有SUPERUSER权限。
- 查看和修改其它非超级用户的数据表，执行SELECT、UPDATE、DELETE或更改所有者（Owner）等操作。
- 查看其它非超级用户的连接信息、撤销（Cancel）其SQL或终止（Kill）其连接。
- 执行CREATE EXTENSION和DROP EXTENSION命令，创建和删除插件。
- 创建其他具有RDS\_SUPERUSER权限的用户，示例如下：

```
CRATE ROLE root_user2 RDS_SUPERUSER LOGIN PASSWORD 'xyz' ;
```



## 管理权限


您可以在数据库（Database）、模式（Schema）、表等多个层次管理权限。例如，赋予一个用户表上的读取权限，但收回修改权限，示例如下。

```
GRANT SELECT ON TABLE t1 TO normal_user1;
REVOKE UPDATE ON TABLE t1 FROM normal_user1;
REVOKE DELETE ON TABLE t1 FROM normal_user1;
```

## 4.8. 实时物化视图

AnalyticDB PostgreSQL版提供了实时物化视图功能，相较于普通（非实时）物化视图，实时物化视图无需手动调用刷新命令，即可实现数据更新时自动同步刷新物化视图。

当基表发生变化时，构建在基表上的实时物化视图将会自动更新，目前AnalyticDB PostgreSQL版的实时物化视图仅支持语句（STATEMENT）级别的自动更新，即当基表的INSERT、COPY、UPDATE、DELETE语句执行成功时，构建于基表之上的物化视图都会实时更新，保证数据强一致。

 **说明** 该模式下会对基表的写入性能会有一定影响，建议您不要在同一张基表上创建过多的实时物化视图。

更多关于物化视图的介绍，请参见[CREATE MATERIALIZED VIEW](#)。

### STATEMENT级别的刷新

STATEMENT（语句）级别的一致性表示当基表的某一条语句执行成功时，实时物化视图中的数据将会同步变更。语句级的实时物化视图可以实现当对基表的更新语句（INSERT、COPY、UPDATE、DELETE）返回成功时，对应的物化视图的数据已经完成变化。更新逻辑如下：

- 在数据库内核中，会先对基表执行更新，然后再执行对物化视图的更新。当基表更新失败时，物化视图中的数据不会发生变化。
- 如果对物化视图更新失败，则基表的更新也将失败，基表将不会有任何变化，同时对基表执行的语句将返回失败。

如果使用显式事务（例如BEGIN+COMMIT），当基表的更新语句成功更新后，物化视图中的数据变更也同样在这个事务中：

- 如果AnalyticDB PostgreSQL版为READ COMMITTED隔离级别（默认），当事务未提交时，物化视图中的更新对其他事务也不可见。
- 如果事务回滚，基表和物化视图都会进行相应的回滚。

### 使用限制

AnalyticDB PostgreSQL版对实时物化视图的查询语句有所限制，您只能创建如下类型查询语句的实时物化视图：

- 如果查询语句包含JOIN，仅支持使用INNER JOIN语句，不支持OUTER JOIN和SELF JOIN。
- 查询语句可以包含大部分的过滤和投影操作。
- 当查询语句包含聚合操作时，只支持COUNT、SUM、AVG、MAX、MIN，且不支持HAVING子句。
- 查询语句只能为简单的语句，不支持子查询、CTE等复杂的语句。

当您在基表上创建了实时物化视图，对基表执行的DDL将受到限制，限制如下：

- 对基表执行TRUNCATE命令时，实时物化视图不会同步变化，需要手动刷新物化视图或重建物化视图。

- 只有指定了CASCADE选项，才能成功对基表执行DROP TABLE命令。
- 基表上执行ALTER TABLE命令无法删除或修改物化视图引用的字段。

目前实时物化视图还存在部分限制，限制如下：

- 暂时仅支持HEAP普通表和HEAP分区表，不支持AO表。

## 使用场景

建议您在具有如下特征的场景使用实时物化视图：

- 查询结果相对于对基表仅包含少量的行或列。例如具有很高过滤性的过滤条件，或者高度集中的聚合函数等场景。
- 获取查询结果需要经过大量的计算处理，包括：
  - 半结构化数据分析。
  - 需要很长时间才能计算完成的聚合操作。
- 视图的基表不会经常更改。

实时物化视图适用于所有适合使用物化视图的场景。与普通物化视图相比，实时物化视图具有高度的一致性，当基表发生改变时，实时物化视图将以较低的性能消耗同步发生改变，而普通物化视图如果每次基表发生改变，都执行刷新操作，大部分时候将消耗较大。所以当基表具有一定程度的更改，甚至需要流式导入更新时，实时物化视图相比于普通物化视图将具有很大优势。

## 实时物化视图代价

实时物化视图类似实时维护的索引，在对查询性能进行大幅度优化的同时，对写入性能有一定影响。

当您创建了一个只包含单表的实时物化视图时，由于需要同步更新物化视图中的数据，将会使数据库的写入性能有所下降，写入的延迟会比直接写基表的延迟多1~3倍左右，建议您在同一张基表上不要创建超过5个以上的实时物化视图。

批量写入数据有利于降低实时物化视图带来的维护开销，在使用COPY或INSERT时，适当增加在单条语句内BATCH的数据行数，可以有效降低实时物化视图的开销维护。

当创建实时物化视图的查询语句包含两表联结（JOIN）时，实时物化视图的写入性能需要进行特别调优，如果您没有相关的经验，或者测试过程中发现性能较差，建议只使用包含单表的实时物化视图。对于两表JOIN场景有如下使用建议：

- 两张基表的JOIN KEY作为各自的分布键。
- 两张基表的JOIN KEY必须都创建索引。

## 创建或删除实时物化视图

- 使用 `CREATE INCREMENTAL MATERIALIZED VIEW` 命令创建一个名为 `mv` 实时物化视图，示例如下：

```
CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT * FROM base WHERE id > 40;
```

- 使用 `DROP MATERIALIZED VIEW` 命令删除物化视图 `mv`，示例如下：

```
DROP MATERIALIZED VIEW mv;
```

## 使用示例

1. 创建基表。示例如下：

```
CREATE TABLE test (a int, b int) DISTRIBUTED BY (a);
```

## 2. 创建实时物化视图。示例如下：

```
CREATE INCREMENTAL MATERIALIZED VIEW mv AS SELECT * FROM TEST WHERE b > 40 DISTRIBUTED BY (a);
```

## 3. 向基表插入数据。示例如下：

```
INSERT INTO test VALUES (1, 30), (2, 40), (3, 50), (4, 60);
```

## 4. 查看基表，示例如下：

```
SELECT * FROM test;
```

查询结果如下：

```
a | b
---+---
1 | 30
2 | 40
3 | 50
4 | 60
(4 rows)
```

## 5. 查看物化视图，示例如下：

```
SELECT * FROM mv;
```

物化视图已经修改成功，查询结果如下：

```
a | b
---+---
3 | 50
4 | 60
(2 rows)
```

# 4.9. JSON数据类型操作

JSON (JavaScript Object Notation) 类型几乎已成为互联网及物联网 (IoT) 的基础数据类型，其重要性不言而喻，具体协议请参见[JSON官网](#)。PostgreSQL对JSON的支持已经比较完善，阿里云深度优化云数据库AnalyticDB PostgreSQL版，基于PostgreSQL语法实现了对JSON 数据类型的支持。

## 检查现有版本是否支持JSON

执行如下命令，检查是否已经支持JSON。

```
SELECT ''::json;
```

若系统出现如下信息，则说明已经支持JSON类型，可以使用实例了。若执行不成功，请重新启动实例。

```
json
-----
""
(1 row)
```

若系统出现如下信息，则说明尚未支持JSON类型。

```
ERROR:  type "json" does not exist
LINE 1: SELECT ' '::json;
           ^
```

上述命令是将一个字符串强制转换成JSON格式，这基本上就是JSON操作的本质。

## JSON在数据库中的转换

数据库的操作主要分为读和写，JSON的数据写入一般是字符串到JSON。字符串中的内容必须符合JSON标准，包括字符串、数字、数组、对象等内容。如：

### 字符串

```
SELECT '"hijson"'::json;
```

返回信息如下：

```
json
-----
"hijson"
(1 row)
```

`::` 在PostgreSQL/Greenplum/AnalyticDB PostgreSQL版中代表强制类型转换。在进行该转换的时候，数据库会调用JSON类型的输入函数，因此，进行类型转换的同时会做JSON 格式的检查，如下所示：

```
SELECT '{hijson:1024}'::json;
```

错误信息如下：

```
ERROR:  invalid input syntax for type json
LINE 1: SELECT '{hijson:1024}'::json;
           ^

DETAIL:  Token "hijson" is invalid.
CONTEXT:  JSON data, line 1: {hijson...
```

上述 `"hijson"` 两边的 `"` 是必不可少的，因为在标准中，KEY 值对应的是一个字符串，所以这里的 `{hijson:1024}` 在语法上会报错。

除了类型上的强制转换，还有从数据库记录到JSON串的转换。

我们一般使用JSON，不会只用一个String或一个Number，而是一个包含一个或多个键值对的对象。所以，对AnalyticDB PostgreSQL版而言，支持到对象的转换，即支持了JSON的绝大多数场景，如：

```
SELECT row_to_json(row('{"a":"a"}', 'b'));
```

返回信息如下：

```

row_to_json
-----
{"f1":"\\a\\":"a\\","f2":"b"}
(1 row)

```

```
SELECT row_to_json(row('{"a":"a"}'::json, 'b'));
```

返回信息如下：

```

row_to_json
-----
{"f1":{"a":"a"},"f2":"b"}
(1 row)

```

由此也可以看出字符串和JSON的区别，这样就可以很方便地将一整条记录转换成JSON。

## JSON内部数据类型的定义

- 对象

JSON中最常用的是对象，如：

```
SELECT '{"key":"value"}'::json;
```

```

json
-----
{"key":"value"}
(1 row)

```

- 整数&浮点数

JSON的协议只有三种数字：整数、浮点数和常数表达式，当前AnalyticDB PostgreSQL版对这三种都有很好的支持。

```
SELECT '1024'::json;
```

返回信息如下：

```

json
-----
1024
(1 row)

```

```
SELECT '0.1'::json;
```

返回信息如下：

```

json
-----
0.1
(1 row)

```

特殊情况下，需要如下信息：

```
SELECT '1e100'::json;
```

返回信息如下：

```
      json
-----
      1e100
(1 row)
```

```
SELECT '{"f":1e100}'::json;
```

返回信息如下：

```
      json
-----
{"f":1e100}
(1 row)
```

并且，包括下面这个长度超长的数字：

```
SELECT '9223372036854775808'::json;
```

返回信息如下：

```
      json
-----
9223372036854775808
(1 row)
```

- 数组

```
SELECT '[[1,2], [3,4,5]]'::json;
```

返回信息如下：

```
      json
-----
[[1,2], [3,4,5]]
(1 row)
```

## 操作符

### JSON支持的操作符类型

```
SELECT oprname,oprcode FROM pg_operator WHERE oprleft = 3114;
```

返回信息如下：

```

oprname |          opcode
-----+-----
->      | json_object_field
->>     | json_object_field_text
->      | json_array_element
->>     | json_array_element_text
#>      | json_extract_path_op
#>>     | json_extract_path_text_op
(6 rows)

```

## 基本使用方法

```
SELECT '{"f":"1e100"}'::json -> 'f';
```

返回信息如下：

```

?column?
-----
"1e100"
(1 row)

```

```
SELECT '{"f":"1e100"}'::json ->> 'f';
```

返回信息如下：

```

?column?
-----
1e100
(1 row)

```

```
SELECT '{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}'::json#>array['f4','f6'];
```

返回信息如下：

```

?column?
-----
"stringy"
(1 row)

```

```
SELECT '{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}'::json#>'f4,f6';
```

返回信息如下：

```

?column?
-----
"stringy"
(1 row)

```

```
SELECT '{"f2":["f3",1],"f4":{"f5":99,"f6":"stringy"}}'::json#>>'f2,0';
```

返回信息如下：

```
?column?
-----
f3
(1 row)
```

JSON函数

支持的函数

```
\df *json*
```

返回信息如下：

List of functions			
Schema	Name	Result data type	Argument data types
Type	Type		
pg_catalog	array_to_json	json	anyarray
normal			
pg_catalog	array_to_json	json	anyarray, boolean
normal			
pg_catalog	json_array_element	json	from_json json, element_index integer
normal			
pg_catalog	json_array_element_text	text	from_json json, element_index integer
normal			
pg_catalog	json_array_elements	SETOF json	from_json json, OUT value json
normal			
pg_catalog	json_array_length	integer	json
normal			
pg_catalog	json_each	SETOF record	from_json json, OUT key text, OUT value json
normal			
pg_catalog	json_each_text	SETOF record	from_json json, OUT key text, OUT value text
normal			
pg_catalog	json_extract_path	json	from_json json, VARIADIC path_elems text[]
normal			
pg_catalog	json_extract_path_op	json	from_json json, path_elems text[]
normal			
pg_catalog	json_extract_path_text	text	from_json json, VARIADIC path_elems text[]
normal			
pg_catalog	json_extract_path_text_op	text	from_json json, path_elems text[]
normal			
pg_catalog	json_in	json	cstring
normal			
pg_catalog	json_object_field	json	from_json json, field_name text
normal			
pg_catalog	json_object_field_text	text	from_json json, field_name text
normal			
pg_catalog	json_object_keys	SETOF text	json
normal			
pg_catalog	json_out	cstring	json
normal			
pg_catalog	json_populate_record	anyelement	base anyelement, from_json json, use_json_as_text boolean
normal			



```

n, use_json_as_text boolean | normal
pg_catalog | json_populate_recordset | SETOF anyelement | base anyelement, from_json json
n, use_json_as_text boolean | normal
pg_catalog | json_recv | json | internal
| normal
pg_catalog | json_send | bytea | json
| normal
pg_catalog | row_to_json | json | record
| normal
pg_catalog | row_to_json | json | record, boolean
| normal
pg_catalog | to_json | json | anyelement
| normal
(24 rows)

```

## 基本使用方法

```
SELECT array_to_json('{{1,5},{99,100}}'::int[]);
```

返回信息如下：

```

array_to_json
-----
[[1,5],[99,100]]
(1 row)

```

```
SELECT row_to_json(row(1, 'foo'));
```

返回信息如下：

```

row_to_json
-----
{"f1":1,"f2":"foo"}
(1 row)

```

```
SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]');
```

返回信息如下：

```

json_array_length
-----
5
(1 row)

```

```
SELECT * FROM json_each('{ "f1": [1,2,3], "f2": { "f3": 1}, "f4": null, "f5": 99, "f6": "stringy" }')
q;
```

返回信息如下：

```
key | value
-----+-----
f1  | [1,2,3]
f2  | {"f3":1}
f4  | null
f5  | 99
f6  | "stringy"
(5 rows)
```

```
SELECT json_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null,"f5":"null"}');
```

返回信息如下：

```
json_each_text
-----
(f1,"[1,2,3]")
(f2,"{"f3":1}")
(f4,)
(f5,null)
(4 rows)
```

```
SELECT json_array_elements('[1,true,[1,[2,3]],null,{"f1":1,"f2":[7,8,9]},false]');
```

返回信息如下：

```
json_array_elements
-----
1
true
[1,[2,3]]
null
{"f1":1,"f2":[7,8,9]}
false
(6 rows)
```

```
CREATE TYPE jpop AS (a text, b int, c timestamp);
```

```
SELECT * FROM json_populate_record(null::jpop,'{"a":"blurfl","x":43.2}', false) q;
```

返回信息如下：

```
 a    | b | c
-----+---+---
blurfl |   | 
(1 row)
```

```
SELECT * FROM json_populate_recordset(null::jpop,'[{"a":"blurfl","x":43.2},{ "b":3,"c":"2012-01-20 10:42:53"}]',false) q;
```

返回信息如下：

```

      a      | b |      c
-----+---+-----
blurfl      |   |
            | 3 | Fri Jan 20 10:42:53 2012
(2 rows)

```

## 完整操作示例

### 创建表

创建并插入数据：

```

CREATE TABLE tj(id serial, ary int[], obj json, num integer);
INSERT INTO tj(ary, obj, num) VALUES('{1,5}'::int[], '{"obj":1}', 5);

```

查询表：

```

SELECT row_to_json(q) FROM (select id, ary, obj, num from tj) AS q;

```

返回信息如下：

```

              row_to_json
-----
{"f1":1,"f2":[1,5],"f3":{"obj":1},"f4":5}
(1 row)

```

插入数据：

```

INSERT INTO tj(ary, obj, num) VALUES('{2,5}'::int[], '{"obj":2}', 5);

```

查询表：

```

SELECT row_to_json(q) FROM (select id, ary, obj, num from tj) AS q;

```

返回信息如下：

```

              row_to_json
-----
{"f1":1,"f2":[1,5],"f3":{"obj":1},"f4":5}
{"f1":2,"f2":[2,5],"f3":{"obj":2},"f4":5}
(2 rows)

```

### 多表 JOIN

创建表并插入数据：

```

CREATE TABLE tj2(id serial, ary int[], obj json, num integer);
INSERT INTO tj2(ary, obj, num) VALUES('{2,5}'::int[], '{"obj":2}', 5);

```

查询表：

```
SELECT * FROM tj, tj2 where tj.obj->>'obj' = tj2.obj->>'obj';
```

返回信息如下：

```
id | ary | obj | num | id | ary | obj | num
----+-----+-----+-----+----+-----+-----+-----
  2 | {2,5} | {"obj":2} | 5 | 1 | {2,5} | {"obj":2} | 5
(1 row)
```

查询表：

```
SELECT * FROM tj, tj2 WHERE json_object_field_text(tj.obj, 'obj') = json_object_field_text(tj2.obj, 'obj');
```

返回信息如下：

```
id | ary | obj | num | id | ary | obj | num
----+-----+-----+-----+----+-----+-----+-----
  2 | {2,5} | {"obj":2} | 5 | 1 | {2,5} | {"obj":2} | 5
(1 row)
```

## JSON函数索引

创建表并插入数据：

```
CREATE TEMP TABLE test_json (
    json_type text,
    obj json
);
INSERT INTO test_json VALUES('aa', '{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}');
INSERT INTO test_json VALUES('cc', '{"f7":{"f3":1},"f8":{"f5":99,"f6":"foo"}}');
```

查询表：

```
SELECT obj->'f2' FROM test_json WHERE json_type = 'aa';
```

返回信息如下：

```
?column?
-----
{"f3":1}
(1 row)
```

创建索引：


```
CREATE INDEX i ON test_json (json_extract_path_text(obj, '{f4}'));
```

查询表：

```
SELECT * FROM test_json where json_extract_path_text(obj, '{f4}') = '{"f5":99,"f6":"foo"}';
```

返回信息如下：

```
json_type |          obj
-----+-----
aa        | {"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}
(1 row)
```

 **说明** JSON类型暂时不能支持作为分布键来使用；也不支持JSON聚合函数。

下面是Python访问的一个例子：

```
#!/bin/env python
import time
import json
import psycopg2
def gpquery(sql):
    conn = None
    try:
        conn = psycopg2.connect("dbname=sanity1x2")
        conn.autocommit = True
        cur = conn.cursor()
        cur.execute(sql)
        return cur.fetchall()
    except Exception as e:
        if conn:
            try:
                conn.close()
            except:
                pass
            time.sleep(10)
        print e
    return None
def main():
    sql = "select obj from tj;"
    #rows = Connection(host, port, user, pwd, dbname).query(sql)
    rows = gpquery(sql)
    for row in rows:
        print json.loads(row[0])
if __name__ == "__main__":
    main()
```

## 4.10. HyperLogLog的使用

阿里云深度优化云原生数据仓库AnalyticDB PostgreSQL版，除了原生Greenplum Database功能外，还支持HyperLogLog，为互联网广告分析及有类似预估分析计算需求的行业提供解决方案，以便于快速预估PV、UV等业务指标。

### 创建HyperLogLog插件

执行如下命令，创建HyperLogLog插件：

```
CREATE EXTENSION hll;
```

## 基本类型

- 执行如下命令，创建一个含有hll字段的表：

```
CREATE TABLE agg (id int primary key, usersids hll);
```

- 执行如下命令，进行int转hll\_hashval：

```
SELECT 1::hll_hashval;
```

## 基本操作符

- hll类型支持=、!=、<>、||和#。

```
SELECT hll_add_agg(1::hll_hashval) = hll_add_agg(2::hll_hashval);  
SELECT hll_add_agg(1::hll_hashval) || hll_add_agg(2::hll_hashval);  
SELECT #hll_add_agg(1::hll_hashval);
```

- hll\_hashval类型支持=、!=和<>。

```
SELECT 1::hll_hashval = 2::hll_hashval;  
SELECT 1::hll_hashval <> 2::hll_hashval;
```

## 基本函数

- hll\_hash\_boolean、hll\_hash\_smallint和hll\_hash\_bigint等HASH函数。

```
SELECT hll_hash_boolean(true);  
SELECT hll_hash_integer(1);
```

- hll\_add\_agg：可以将int转hll格式。

```
SELECT hll_add_agg(1::hll_hashval);
```

- hll\_union：hll并集。

```
SELECT hll_union(hll_add_agg(1::hll_hashval), hll_add_agg(2::hll_hashval));
```

- hll\_set\_defaults：设置精度。

```
SELECT hll_set_defaults(15, 5, -1, 1);
```

- hll\_print：用于debug信息。

```
SELECT hll_print(hll_add_agg(1::hll_hashval));
```

## 示例

- 创建测试表。

```
CREATE TABLE access_date (acc_date date unique, usersids hll);
```

- 插入测试数据。

```
INSERT INTO access_date SELECT current_date, hll_add_agg(hll_hash_integer(user_id)) FROM generate_series(1,10000) t(user_id);
INSERT INTO access_date SELECT current_date-1, hll_add_agg(hll_hash_integer(user_id)) FROM generate_series(5000,20000) t(user_id);
INSERT INTO access_date SELECT current_date-2, hll_add_agg(hll_hash_integer(user_id)) FROM generate_series(9000,40000) t(user_id);
```

### 3. 查看使用HyperLogLog插件测试结果。

```
select #userids from access_date where acc_date=current_date;
```

返回信息如下：

```
?column?
-----
 9725.85273370708
(1 row)
```

```
select #userids from access_date where acc_date=current_date-1;
```

返回信息如下：

```
?column?
-----
14968.6596883279
(1 row)
```

```
select #userids from access_date where acc_date=current_date-2;
```

返回信息如下：

```
?column?
-----
29361.5209149911
(1 row)
```

## 4.11. Create Library命令的使用

为支持用户导入自定义软件包，AnalyticDB PostgreSQL版引入了Create/Drop Library命令。

### 语法

```
CREATE LIBRARY library_name LANGUAGE [JAVA] FROM oss_location OWNER ownername
CREATE LIBRARY library_name LANGUAGE [JAVA] VALUES file_content_hex OWNER ownername
DROP LIBRARY library_name
```

### 参数说明

参数	说明
library_name	要安装的库的名称。若已安装的库与要安装的库的名称相同，则必须先删除现有的库，然后再安装新库。

参数	说明
LANGUAGE [JAVA]	要使用的语言。目前仅支持PL/Java。
oss_location	包文件的位置。您可以指定OSS 存储桶和对象名称，仅可以指定一个文件，且不能为压缩文件。其格式为： <pre>oss://oss_endpoint filepath=[folder/[folder/]...]/file_name id=userossid key=userosskey bucket=ossbucket</pre>
file_content_hex	文件内容，字节流为16进制，例如“73656c6563742031”（“select 1”的16进制字节流）。借助这个语法，可以直接导入包文件，不必通过OSS。
ownername	指定用户。
DROP LIBRARY	删除一个库。

## 示例

- 示例1：安装名为analytics.jar的jar包。

```
create library example language java from 'oss://oss-cn-hangzhou.aliyuncs.com filepath=analytics.jar id=xxx key=yyy bucket=zzz';
```

- 示例2：直接导入文件内容，字节流为16进制。

```
create library pglib LANGUAGE java VALUES '73656c6563742031' OWNER "myuser";
```

- 示例3：删除一个库。

```
drop library example;
```

- 示例4：查看已经安装的库。

```
select name, lanname from pg_library;
```

## 4.12. PL/Java UDF的使用

AnalyticDB PostgreSQL版支持用户使用PL/Java 语言，编写并上传jar软件包，并利用这些jar包创建用户自定义函数（UDF）。AnalyticDB PostgreSQL版支持的PL/Java语言版本对应社区的PL/Java 1.5.0，使用的JVM版本为1.8。下面是一个创建PL/Java UDF的示例步骤。更多的PL/Java例子，请参见[PL/Java代码](#)（编译方法见其[文档](#)）。

### 操作步骤

1. 在AnalyticDB PostgreSQL版中，执行如下命令，创建PL/Java插件（每个数据库只需执行一次）。

```
create extension pljava;
```

2. 根据业务需要，编写自定义函数。例如，使用如下代码编写Test.java文件。



```
public class Test
{
    public static String substring(String text, int beginIndex,
        int endIndex)
    {
        try {
            Process process = null;
            process = Runtime.getRuntime().exec("echo Test running");
        } catch (Exception e) {
            return "" + e;
        }
        return text.substring(beginIndex, endIndex);
    }
}
```

3. 编写manifest.txt文件。

```
Manifest-Version: 1.0
Main-Class: Test
Specification-Title: "Test"
Specification-Version: "1.0"
Created-By: 1.7.0_99
Build-Date: 01/20/2016 21:00 AM
```

4. 执行如下命令，将程序编译打包。

```
javac Test.java
jar cfm analytics.jar manifest.txt Test.class
```

5. 将上一步生成的analytics.jar文件，通过OSS控制台命令上传到OSS。

```
osscmd put analytics.jar oss://zzz
```

6. 在AnalyticDB PostgreSQL版中，执行Create Library命令，将文件导入到AnalyticDB PostgreSQL版中。

```
create library example language java from 'oss://oss-cn-hangzhou.aliyuncs.com filepath=
analytics.jar id=xxx key=yyy bucket=zzz';
```

 **说明** Create Library只支持filepath，一次导入一个文件。另外，Create Library还支持字节流形式，可以不通过OSS 直接导入，详情请参见[Create Library命令的使用](#)。

7. 在AnalyticDB PostgreSQL版中，执行如下命令，创建和使用相应UDF。

```
create table temp (a varchar) distributed randomly;
insert into temp values ('my string');
create or replace function java_substring(varchar, int, int) returns varchar as 'Test.
substring' language java;
select java_substring(a, 1, 5) from temp;
```

# 5.表

## 5.1. 创建表

您可以在数据库中创建表。

### 语法

创建表的完整语法如下。不是所有子句都必选，实际使用时请根据需要选择相应的子句。

```
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
  [ { column_namedata_type [ DEFAULT default_expr ]
    [column_constraint [ ... ]
  [ ENCODING ( storage_directive [,...] ) ]
]
  | table_constraint
  | LIKE other_table [{INCLUDING | EXCLUDING}
    {DEFAULTS | CONSTRAINTS}] ...}
[, ... ] ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter=value [, ... ] )
[ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
[ TABLESPACE tablespace ]
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
[ PARTITION BY partition_type (column)
  [ SUBPARTITION BY partition_type (column) ]
  [ SUBPARTITION TEMPLATE ( template_spec ) ]
  [...]
  ( partition_spec )
  | [ SUBPARTITION BY partition_type (column) ]
  [...]
  ( partition_spec
    [ ( subpartition_spec
      [ (...) ]
    ) ]
  ) ]
)
```

column\_constraint子句可定义为：

```
[CONSTRAINT constraint_name]
  NOT NULL | NULL
  | UNIQUE [USING INDEX TABLESPACE tablespace]
    [WITH ( FILLFACTOR = value )]
  | PRIMARY KEY [USING INDEX TABLESPACE tablespace]
    [WITH ( FILLFACTOR = value )]
  | CHECK ( expression )
  | REFERENCES table_name [ ( column_name [, ... ] ) ]
    [ key_match_type ]
    [ key_action ]
```

列的storage\_directive子句可定义为：

```
COMPRESSTYPE={ZLIB | QUICKLZ | RLE_TYPE | NONE}  
[COMPRESSLEVEL={0-9} ]  
[BLOCKSIZE={8192-2097152} ]
```

表的storage\_parameter子句可定义为：

```
APPENDONLY={TRUE|FALSE}  
BLOCKSIZE={8192-2097152}  
ORIENTATION={COLUMN|ROW}  
CHECKSUM={TRUE|FALSE}  
COMPRESSTYPE={ZLIB|QUICKLZ|RLE_TYPE|NONE}  
COMPRESSLEVEL={0-9}  
FILLFACTOR={10-100}  
OIDS [=TRUE|FALSE]
```

table\_constraint子句可定义为：

```
[CONSTRAINT constraint_name]  
UNIQUE ( column_name [, ... ] )  
    [USING INDEX TABLESPACE tablespace]  
    [WITH ( FILLFACTOR=value )]  
| PRIMARY KEY ( column_name [, ... ] )  
    [USING INDEX TABLESPACE tablespace]  
    [WITH ( FILLFACTOR=value )]  
| CHECK ( expression )  
| FOREIGN KEY ( column_name [, ... ] )  
    REFERENCES table_name [ ( column_name [, ... ] ) ]  
    [ key_match_type ]  
    [ key_action ]  
    [ key_checking_mode ]
```

其中key\_match\_type的值包括：

```
MATCH FULL  
| SIMPLE
```

其中key\_action的值包括：

```
ON DELETE  
| ON UPDATE  
| NO ACTION  
| RESTRICT  
| CASCADE  
| SET NULL  
| SET DEFAULT
```

其中key\_checking\_mode的值包括：

```
DEFERRABLE
| NOT DEFERRABLE
| INITIALLY DEFERRED
| INITIALLY IMMEDIATE
```

其中partition\_type的值包括：

```
LIST
| RANGE
```

partition\_specification子句可定义为：

```
partition_element [, ...]
```

partition\_element子句可定义为：

```
DEFAULT PARTITION name
| [PARTITION name] VALUES (list_value [,...])
| [PARTITION name]
  START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
  [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
| [PARTITION name]
  END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

其中subpartition\_spec或template\_spec子句可定义为：

```
subpartition_element [, ...]
```

subpartition\_element子句可定义为：

```
DEFAULT SUBPARTITION name
| [SUBPARTITION name] VALUES (list_value [,...])
| [SUBPARTITION name]
  START ([datatype] 'start_value') [INCLUSIVE | EXCLUSIVE]
  [ END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE] ]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
| [SUBPARTITION name]
  END ([datatype] 'end_value') [INCLUSIVE | EXCLUSIVE]
  [ EVERY ([datatype] [number | INTERVAL] 'interval_value') ]
[ WITH ( partition_storage_parameter=value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

其中分区的storage\_parameter子句可定义为：

```
APPENDONLY={ TRUE | FALSE }
BLOCKSIZE={ 8192-2097152 }
ORIENTATION={ COLUMN | ROW }
CHECKSUM={ TRUE | FALSE }
COMPRESSTYPE={ ZLIB | QUICKLZ | RLE_TYPE | NONE }
COMPRESSLEVEL={ 1-9 }
FILLFACTOR={ 10-100 }
OIDS [=TRUE | FALSE]
```

参数说明

创建表的关键参数说明如所示。

创建表的参数说明

参数	说明
TABLE_NAME	待创建的表的名称。
column_name	新表中列的名称。
data_type	设置列的数据类型。 对于包含文本数据的表列，请指定数据类型为VARCHAR或TEXT。不推荐使用CHAR数据类型。
DEFAULT default_expr	为列定义一个默认数值。系统将为所有不指定列值的列插入默认值。该值可以是任何无变量表达式（不允许对当前表中的其他列进行子查询和交叉引用）。默认表达式的数据类型必须与列的数据类型相匹配。如果列没有默认值，那么默认值为空。
ENCODING storage_directive	指定列数据的压缩类型和块大小。 该子句仅适用于附加优化列表。 列压缩设置从表级继承到分区级到子分区级。最低级别的设置具有优先级。
INHERITS	指定新表中的所有列自动继承某个父表。用INHERITS在新的子表和它的父表之间创建一个永久的关系。对父表的模式修改通常也同步至子表，并且默认情况下子项的数据包含在父项的扫描中。
LIKE other_table	指定一个表，新表自动复制其所有列名、数据类型、非空约束和分配策略。但存储属性例如append-optimized或分区结构不会被复制。 与INHERITS不同，创建完成后新表与原表完全解耦。
CONSTRAINT constraint_name	设置列或表的约束。如果违反约束条件，那么约束名称会出现在错误消息中，因此约束名称可用于将有用的约束信息传递给客户端应用程序。（需要使用双引号来指定包含空格的约束名称。）
WITH ( storage_option=value )	设置表或其索引的存储选项。

参数	说明
ON COMMIT	当事务处理结束后对临时表采取的行为。包括三个选项： <ul style="list-style-type: none"><li>• <b>PRESERVE ROWS</b>：默认为不采取特殊行动。即完成事务后数据保留，只有连接会话断开后数据才消失。</li><li>• <b>DELETE ROWS</b>：删除临时表中的所有行。</li><li>• <b>DROP</b>：删除临时表。</li></ul>
TABLESPACE tablespace	设置表空间的名称。如果未指定，则使用数据库的默认表空间。
DISTRIBUTED BY	设置数据库分布策略。 <ul style="list-style-type: none"><li>• <b>DISTRIBUTED BY (column, [ ... ])</b>：指定分布键，系统会根据分布键对数据进行哈希分布。 为了使数据均匀分布，分布键应该是表的主键或均匀分布的一列或多列。</li><li>• <b>DISTRIBUTED RANDOMLY</b>：数据随机分布。</li></ul> <div> <b>说明</b> 建议不要轻易使用随机分布。</div>
PARTITION BY	设置表分区的分区键。对特别大的表进行分区，可以提高数据访问的效率。 对一张表做分区，实际上是创建了一张顶层（父级）表和多个低层（子级）表。一旦创建了分区表，顶层表总是空的。数据值储存在最低层的表中。在多级分区表中，仅仅在层级最低的子分区中有数据。 支持RANGE分区、LIST分区或两者混合。
SUBPARTITION BY	设置多级分区表。
SUBPARTITION TEMPLATE	您可以设置一个子分区模板来创建子分区（低层子表），来确保每个分区具有相同的子分区结构。

示例

创建表格和表格分布键。默认情况下，主键将用作AnalyticDB PostgreSQL版分布键。

```
CREATE TABLE films (  
code          char(5) CONSTRAINT firstkey PRIMARY KEY,  
title         varchar(40) NOT NULL,  
did           integer NOT NULL,  
date_prod     date,  
kind          varchar(10),  
len           interval hour to minute  
);  
CREATE TABLE distributors (  
did           integer PRIMARY KEY DEFAULT nextval('serial'),  
name          varchar(40) NOT NULL CHECK (name <> '')  
);
```

创建一个压缩表，设置分布键。

```
CREATE TABLE sales (txn_id int, qty int, date date)
WITH (appendonly=true, compresslevel=5)
DISTRIBUTED BY (txn_id);
```

使用每个级别的子分区模板和默认分区创建三级分区表。

```
CREATE TABLE sales (id int, year int, month int, day int,
region text)
DISTRIBUTED BY (id)
PARTITION BY RANGE (year)
SUBPARTITION BY RANGE (month)
SUBPARTITION TEMPLATE (
START (1) END (13) EVERY (1),
DEFAULT SUBPARTITION other_months )
SUBPARTITION BY LIST (region)
SUBPARTITION TEMPLATE (
SUBPARTITION usa VALUES ('usa'),
SUBPARTITION europe VALUES ('europe'),
SUBPARTITION asia VALUES ('asia'),
DEFAULT SUBPARTITION other_regions)
( START (2008) END (2016) EVERY (1),
DEFAULT PARTITION outlying_years);
```

## 5.2. 行存、列存，堆表、AO表的原理和使用场景

AnalyticDB PostgreSQL版支持行存和列存，支持堆表和AO表。本文简要介绍他们的原理和使用场景。

### 行存和列存

#### 行存和列存的比较

维度	行存	列存
概念	以行的形式组织存储，一行是一个tuple，存在一起。当需要读取某列时，需要将这列前面的所有列都进行deform，所以访问第一列和访问最后一列的成本实际上是不一样的。	以列为形式组织存储，每列对应一个或一批文件。读取任一列的成本是一样的，但是如果要读取多列，需要访问多个文件，访问的列越多，开销越大。
压缩比	较低。	较高。
读取任意列的成本	不一样，越靠后的列，成本越高。	一样。
是否适合向量计算、JIT架构	不适合向量计算、JIT架构。简单来说，就是不适合批处理形式的计算。	非常适合向量计算、JIT架构。对大批量数据的访问和统计，效率更高。

维度	行存	列存
使用场景	<p>如果OLTP的需求偏多，例如经常需要查询表的明细（输出很多列），需要更多的更新和删除操作时。可以考虑行存。</p> <p>如果用户有混合需求，可以采用分区表，例如按时间维度的需求分区，近期的数据明细查询多，那就使用行存，对历史的数据统计需求多那就使用列存。</p>	<p>如果OLAP的需求偏多，经常需要对数据进行统计时，选择列存。</p> <p>需要比较高的压缩比时，选择列存。</p>

## 堆表

堆表，实际上就是的堆存储，堆表的所有变更都会产生REDO，可以实现时间点恢复。但是堆表不能实现逻辑增量备份，因为表的任意一个数据块都有可能变更，不方便通过堆存储来记录位点。

一个事务结束时，通过clog以及REDO来实现它的可靠性。同时支持通过REDO来构建MIRROR节点实现数据冗余。

## AO表

AO表，看名字就知道，只追加的存储，删除更新数据时，通过另一个BITMAP文件来标记被删除的行，通过bit以及偏移对齐来判定AO表上的某一行是否被删除。

事务结束时，需要调用FSYNC，记录最后一次写入对应的数据块的偏移。并且这个数据块即使只有一条记录，下次再发起事务又会重新追加一个数据块。同时发送对应的数据块给MIRROR实现数据冗余。

因此AO表不适合小事务，因为每次事务结束都会FSYNC，同时事务结束后这个数据块即使有空余也不会被复用。

虽然如此，AO表非常适合OLAP场景，批量的数据写入，高压缩比，逻辑备份支持增量备份。备份时只需记录备份到的偏移量，加上每次备份全量的BITMAP删除标记（占用空间很小）即可。

## 什么时候选择堆表

- 当数据写入时，小事务偏多时选择堆表。
- 当需要时间点恢复时，选择堆表。

## 什么时候选择AO表

- 当需要列存时，选择AO表。
- 当数据批量写入时，选择AO表。

# 5.3. 设置列存和压缩

对于更新不频繁、字段较多的表，如果想提高性能、提高导入速度或者降低成本，建议使用列存加压缩。这样在保证性能的基础上，压缩比率一般可以达到三倍以上，而且通常导入速度更快。

如果要使用列存和压缩功能，您需要在建表时设置指定列存和压缩选项。例如，可以在建表语句中，加入如下的子句，来启用列存和压缩功能。创建表的具体语法请参见[创建表](#)。

```
with (APPENDONLY=true, ORIENTATION=column, COMPRESSTYPE=zlib, COMPRESSLEVEL=5, BLOCKSIZE=1048576, OIDS=false)
```



❓ 说明 目前AnalyticDB for PostgreSQL仅支持zlib和RLE\_TYPE压缩算法。如果指定了quicklz算法，会自动转换为zlib。

## 5.4. 给列存加字段默认值

介绍如何给列存的表格增加字段默认值，并且通过analyze语句查看更新数据对列存表格大小的影响。

### 背景信息

列存储将每列存储为一个文件，同一行的列通过偏移即对应起来。例如INT8的两个字段，通过偏移很快能找到某一行的A列对应的B列。

在加字段时，AO不会REWRITE TABLE。如果AO表有一些垃圾（被删除的数据）记录，添加的字段需要填充已删除的记录后才会使用相对偏移。

### 操作步骤

1. 创建3张AO列存表。

```
CREATE TABLE tbl1 (id int, info text) WITH (appendonly=true, blocksize=8192, compressype=none, orientation=column);
CREATE TABLE tbl2 (id int, info text) WITH (appendonly=true, blocksize=8192, compressype=none, orientation=column);
CREATE TABLE tbl3 (id int, info text) WITH (appendonly=true, blocksize=8192, compressype=none, orientation=column);
```

2. 前两张分别插入1000万记录，最后一张插入2000万记录。

```
INSERT INTO tbl1 SELECT generate_series(1,10000000),'test';
INSERT INTO tbl2 SELECT generate_series(1,10000000),'test';
INSERT INTO tbl3 SELECT generate_series(1,20000000),'test';
```

3. 分析表，并记录它们的大小。

分析表语句如下：

```
ANALYZE tbl1;
ANALYZE tbl2;
ANALYZE tbl3;
```

查看表的大小：

```
SELECT pg_size_pretty(pg_relation_size('tbl1'));
```

返回信息如下：

```
pg_size_pretty
-----
88 MB
(1 row)
```

```
SELECT pg_size_pretty(pg_relation_size('tbl2'));
```

返回信息如下：

```
pg_size_pretty
-----
88 MB
(1 row)
```

```
SELECT pg_size_pretty(pg_relation_size('tbl3'));
```

返回信息如下：

```
pg_size_pretty
-----
173 MB
(1 row)
```

4. 更新第一张表，全表更新。并记录更新后的大小，翻了一倍。

全表更新，语句如下：

```
UPDATE tbl1 SET INFO='test';
```

重新分析该表，语句如下：

```
ANALYZE tbl1;
```

查看第一张表的大小，查询语句如下：

```
SELECT pg_size_pretty(pg_relation_size('tbl1'));
```

返回信息如下：

```
pg_size_pretty
-----
173 MB
(1 row)
```

5. 对三个表添加字段，设置默认值。

```
ALTER TABLE tbl1 ADD column c1 int8 default 1;
ALTER TABLE tbl2 ADD column c1 int8 default 1;
ALTER TABLE tbl3 add column c1 int8 default 1;
```

6. 分析表，查看表的大小。

分析表语句如下：

```
ANALYZE tbl1;
ANALYZE tbl2;
ANALYZE tbl3;
```

查看表的大小：

```
SELECT pg_size_pretty(pg_relation_size('tbl1'));
```

返回信息如下：

```
pg_size_pretty
-----
325 MB
(1 row)
```

```
SELECT pg_size_pretty(pg_relation_size('tbl2'));
```

返回信息如下：

```
pg_size_pretty
-----
163 MB
(1 row)
```

```
SELECT pg_size_pretty(pg_relation_size('tbl3'));
```

返回信息如下：

```
pg_size_pretty
-----
325 MB
(1 row)
```

经测试，AO表在添加字段时，以已有文件的记录数为准，即使全部都删除了，也需要在新增字段上初始化这个值。

## 5.5. 设置表分区

对于数据库中的事实表，以及一些比较大的表，通常建议使用表分区。

### 设置表分区

使用表分区功能，方便定期地进行数据的删除（通过 `alter table drop partition` 命令即可删除整个分区的数据）和导入（使用交换分区的方式即 `alter table exchange partition` 命令可以加入新数据分区）。

AnalyticDB PostgreSQL版支持 Range Partition（范围分区）、List Partition（列表分区）和Composite Partition（多级分区）。注意Range Partition只支持利用数值或时间类型的字段来分区。

下面是一个使用Range Partition的表的例子。

```
CREATE TABLE LINEITEM (  
  L_ORDERKEY          BIGINT NOT NULL,  
  L_PARTKEY           BIGINT NOT NULL,  
  L_SUPPKEY           BIGINT NOT NULL,  
  L_LINENUMBER        INTEGER,  
  L_QUANTITY          FLOAT8,  
  L_EXTENDEDPRICE     FLOAT8,  
  L_DISCOUNT         FLOAT8,  
  L_TAX               FLOAT8,  
  L_RETURNFLAG        CHAR(1),  
  L_LINESTATUS        CHAR(1),  
  L_SHIPDATE          DATE,  
  L_COMMITDATE        DATE,  
  L_RECEIPTDATE       DATE,  
  L_SHIPINSTRUCT      CHAR(25),  
  L_SHIPMODE          CHAR(10),  
  L_COMMENT            VARCHAR(44)  
) WITH (APPENDONLY=true, ORIENTATION=column, COMPRESSTYPE=zlib, COMPRESSLEVEL=5, BLOCKSIZE=1048576, OIDS=false) DISTRIBUTED BY (l_orderkey)  
PARTITION BY RANGE (L_SHIPDATE) (START (date '1992-01-01') INCLUSIVE END (date '2000-01-01'  
) EXCLUSIVE EVERY (INTERVAL '1 month' ));
```

## 表分区的设计原则

分区的目的是尽可能地减少QUERY需要扫描的数据量，因此必须和查询条件相关联。

- 法则1，尽量选择 and 查询条件相关的字段，减少QUERY需要扫描的数据。
- 法则2，当有多个查询条件时，可以使用子分区，进一步减少需要扫描的数据。

## 5.6. 设置排序键

排序键是表的一种属性，可以将数据按照排序键顺序存储在磁盘文件中。

### 背景信息

排序键主要有两大优势：

- 加速列存优化，收集的min、max元信息很少重叠，过滤性很好。
- 对于含有order by和group by等需要排序的SQL可以避免再次排序，直接从磁盘中读取出来就是满足条件的有序数据。

### 创建表

```

Command:      CREATE TABLE
Description:  define a new table
Syntax:
CREATE [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]      [column_constraint [ ... ]
[ ENCODING ( storage_directive [,...] ) ]
]
| table_constraint
| LIKE other_table [{INCLUDING | EXCLUDING}
                    {DEFAULTS | CONSTRAINTS}] ...}
[, ... ] ]
[column_reference_storage_directive [, ] ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter=value [, ... ] )
[ ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP} ]
[ TABLESPACE tablespace ]
[ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
[ SORTKEY (column, [ ... ] ) ]
[ PARTITION BY partition_type (column)
  [ SUBPARTITION BY partition_type (column) ]
  [ SUBPARTITION TEMPLATE ( template_spec ) ]
  [...]
  ( partition_spec )
  | [ SUBPARTITION BY partition_type (column) ]
  [...]
  ( partition_spec
    [ ( subpartition_spec
        [ (...)]
      ) ]
  ) ]
]
)

```

**样例：**

```

create table test(date text, time text, open float, high float, low float, volume int) with
(APPENDONLY=true,ORIENTATION=column) sortkey (volume);

```

**对表进行排序**

```
VACUUM SORT ONLY [tablename]
```

**修改排序键**

这个命令只改catalog不会对数据立即排序，需要通过 `vaccum sort only` 命令排序。

```
ALTER [[GLOBAL | LOCAL] {TEMPORARY | TEMP}] TABLE table_name SET SORTKEY (column, [ ... ] )
```

**样例：**

```
alter table test set sortkey (high,low);
```

## 6. 最佳实践

### 6.1. 配置内存和负载参数

为了提高数据库的稳定性，您需要合理配置内存和负载参数。

#### 背景信息

AnalyticDB PostgreSQL版是一个重计算和重资源的MPP数据库，可谓有多少资源就能消耗多少资源，带来的好处是处理速度变快了，坏处就是容易超出限制。

CPU、网络、硬盘超出限制的话，关系不大，因为大不了就是到硬件瓶颈，但是内存超出限制的话会带来较大的负面影响，例如操作系统提示用户进程内存不足，导致数据库崩溃等。

#### 如何避免OOM

OOM（Out of memory）指遇到了进程申请内存不足的错误，会出现如下类似提示：

```
Out of memory (seg27 host.example.com pid=47093) VM Protect failed to allocate 4096 bytes,
0 MB available
```

#### 问题原因

导致数据库OOM报错的原因可能有：

- 数据库节点的内存不足。
- 操作系统内存相关的内核参数配置不当。
- 数据倾斜，导致某些查询时，某个Segment需要申请的内存超大。
- 查询倾斜，例如某些聚合、窗口函数的分组不是分布键，那么需要对数据进行重分布，重分布后导致某个Segment的数据出现倾斜，导致某些查询时，某个Segment需要申请的内存超大。

#### 处理方法

1. 调整QUERY，使之需要更少的内存。
2. 使用资源队列（AnalyticDB PostgreSQL版控制资源的一种手段），限制并发QUERY数。降低集群内同时运行的QUERY数，从而减少系统整体的内存资源的使用。
3. 减少单个主机部署的Segment数量，例如有128G内存的主机，部署16个和部署8个Segment节点，每个节点能使用的内存相差了一倍。
4. 增加单台主机的内存数。
5. 设置数据库参数gp\_vmem\_protect\_limit，限制单个Segment可以使用的VMEM上限。单个主机的内存以及部署多少个Segment决定了平均单个Segment最多可以使用多少内存。
6. 对于某些对内存使用量不可预知的SQL，通过在会话中设置statement\_mem参数，限制单条SQL对内存的使用，从而避免单条SQL把内存用光的情况。
7. 也可以在库级别设置statement\_mem参数。对这个数据库的所有会话生效。
8. 使用资源队列（AnalyticDB PostgreSQL版控制资源的一种手段），限制这个资源组的内存使用上限，将数据库用户加入资源组，控制这些用户共同使用内存的上限。

#### 配置内存相关参数

正确地配置操作系统、数据库参数、资源队列管理，可以有效地降低OOM发生的概率。

在计算单主机内单个Segment的平均可使用内存时，不能只考虑Primary Segment，还需要考虑Mirror Segment，因为当集群出现主机故障时，会将Segment切换到对应的Mirror Segment，此时，主机上跑的Segment数就比平时更多了。因此我们必须考虑到failover时，Mirror Segment需要占用的资源。

接下来我们分析一下如何进行操作系统内核配置、数据库配置，从而避免OOM。

操作系统内核参数配置说明如所示。

## 操作系统内核参数说明

参数	说明
huge page	不要配置系统的huge page，因为AnalyticDB PostgreSQL版的PG版本较老，还没有支持huge page。而操作系统的huge page会锁定内存的分配，导致这部分内存不能被数据库节点使用。
vm.overcommit_memory	<p>如果使用SWAP建议设置为2，如果不使用SWAP建议设置为0。</p> <p>各参数值的说明如下：</p> <ul style="list-style-type: none"> <li>0：比较友好，允许申请的内存空间通常不能超过"总内存-不可释放内存(RSS部分)"的部分。超过时申请内存才会报错。</li> <li>1：强制，大多数进程先使用malloc申请内存空间，但是不使用它或者是只使用部分。所以设置为1时，不管什么情况都允许malloc申请成功，除非遇到真的内存不足的情况。</li> <li>2：最友好，在计算允许申请的内存空间时，将SWAP也算进去，也就是说申请大量内存时，可能触发SWAP但是允许你申请成功。</li> </ul>
overcommit_ratio	<p>值越大，允许用户进程申请的内存空间越大，但是给操作系统保留的空间就越小。需要用公式来计算，具体参考接下来的例子。</p> <p>当设置为2时，允许申请的内存地址范围不能超过“swap+内存大小*overcommit_ratio”。</p>

数据库参数配置说明如[数据库参数说明](#)所示。

## 数据库参数说明

参数	说明
gp_vmem_protect_limit	控制每个Segment上所有进程可以申请到的最大内存。如果这个值太高，可能触发系统的OOM或者更严重的问题。如果设置太低，则可能导致系统有足够内存的情况下，SQL也无法运行。
runaway_detector_activation_percent	<p>这个参数默认为90，是一个百分比值。当任一Segment使用的内存超过(runaway_detector_activation_percent*gp_vmem_protect_limit/100)时，主动终止查询，防止OOM。</p> <p>终止的顺序从使用最多内存的QUERY依次开始，直到内存降低到(runaway_detector_activation_percent*gp_vmem_protect_limit/100)以下。</p> <p>通过 gp_toolkit.session_level_memory_consumption 视图可以观察每个会话的内存使用情况，以及runaway的信息。</p>

参数	说明
statement_mem	<p>设置单条SQL最多可以申请的内存，当超过这个内存时，写spill file文件。默认为125MB。</p> <p>建议设置为单个Segment的保护内存乘以0.9除以期望的最大SQL并行运行的值。</p> <pre>(gp_vmem_protect_limit * 0.9) / max_expected_concurrent_queries</pre> <div> <p> <b>说明</b></p> <ul style="list-style-type: none"> <li>statement_mem在会话中设置，如果当前并行度很低，而某个会话需要运行一条需要大量内存的查询，就需要在会话中设置该参数。</li> <li>statement_mem比较适合于低并发环境对内存的使用控制。对于高并发环境，如果使用statement_mem来控制内存，您会发现每条查询可以使用的内存极少，会影响高并发情况下少量对内存需求较高的查询的性能。建议高并发的情况下，使用资源队列（resource queue）来控制内存的使用上限。</li> </ul> </div>
gp_workfile_limit_files_per_query	<p>限制每个查询可以使用的最大spill文件数，当查询申请的内存超过statement_mem的限制时，使用spill file(workfiles)，类似操作系统的swap空间。当使用的spill file超过限制时，查询会被终止。</p> <p>默认为0，表示无限制。</p>
gp_workfile_compress_algorithm	<p>设置spill file的压缩算法。取值范围包括NONE或ZLIB。</p> <p>设置压缩，CPU换空间，或CPU换I/O能力。当磁盘紧张或者磁盘spill file有写入瓶颈时可以设置该参数。</p>

## 内存参数计算举例

环境如下：

- 主机配置：

```
Total RAM = 256GB
SWAP = 64GB
```

- 4台主机，每台主机上有8个Primary Segment 和 Mirror Segment。

当一台主机发生故障时，8个Primary Segment 要分摊到剩余的3台主机，最多单台额外承担3个Primary Segment。所以是8+3=11。

1. 计算给AnalyticDB PostgreSQL版SQL的总内存。

给操作系统保留 "7.5G + 5%内存" 的余量，算出整个系统给应用软件的实际可用内存。然后用实际可用内存除以1.7的经验系数。



```
gp_vmem = ((SWAP + RAM) - (7.5GB + 0.05 * RAM)) / 1.7
         = ((64 + 256) - (7.5 + 0.05 * 256)) / 1.7
         = 176
```

2. 计算overcommit\_ratio，需要使用到经验系数0.026。

```
vm.overcommit_ratio = (RAM - (0.026 * gp_vmem)) / RAM
                    = (256 - (0.026 * 176)) / 256
                    = .982
Set vm.overcommit_ratio to 98.
```

3. 计算每个Segment的内存使用上限保护参数：gp\_vmem\_protect\_limit，除以一台节点故障后其余单台节点需要运行的Primary Segment 数。

```
gp_vmem_protect_limit calculation
gp_vmem_protect_limit = gp_vmem / maximum_acting_primary_segments
                     = 176 / 11
                     = 16GB
                     = 16384MB
```

## 配置资源队列

AnalyticDB PostgreSQL版资源队列可以用来限制“并发的QUERY数和总的内存使用”。当QUERY运行时，会添加到对应的队列中，使用的资源将记录到对应的队列中，对应队列的资源控制限制对该队列内的所有会话起作用。

AnalyticDB PostgreSQL版资源队列控制资源的思想Linux 的CGROUP非常类似。

创建资源队列的语法。

```
Command:      CREATE RESOURCE QUEUE
Description:  create a new resource queue for workload management
Syntax:
CREATE RESOURCE QUEUE name WITH (queue_attribute=value [, ... ])
where queue_attribute is:
    ACTIVE_STATEMENTS=integer
    [ MAX_COST=float [ COST_OVERCOMMIT={TRUE|FALSE} ] ]
    [ MIN_COST=float ]
    [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
    [ MEMORY_LIMIT='memory_units' ]
| MAX_COST=float [ COST_OVERCOMMIT={TRUE|FALSE} ]
  [ ACTIVE_STATEMENTS=integer ]
  [ MIN_COST=float ]
  [ PRIORITY={MIN|LOW|MEDIUM|HIGH|MAX} ]
  [ MEMORY_LIMIT='memory_units' ]
```

参数说明如所示。

## 创建资源队列参数说明

参数	说明
----	----

参数	说明
ACTIVE_STATEMENTS	<p>允许同时运行（active状态）的SQL数。</p> <p>-1表示不限。</p>
MEMORY_LIMIT 'memory_units kB, MB or GB'	<p>设置资源队列中所有SQL允许的最大内存使用量。-1表示不限，但是受前面提到的数据库或系统参数限制，容易触发OOM错误。</p> <p>SQL的内存使用限制不仅受资源队列限制，同时还受到参数限制：</p> <ul style="list-style-type: none"> <li>参数gp_resqueue_memory_policy=none时，限制同Greenplum Database releases 4.1之前的版本。</li> <li>参数gp_resqueue_memory_policy=auto时，如果设置了会话的statement_mem参数，或者设置了statement_mem参数时，单条QUERY允许申请的内存将突破资源队列的MEMORY_LIMIT限制。</li> </ul> <p>例子：</p> <pre>SET statement_mem='2GB'; SELECT * FROM my_big_table WHERE column='value' ORDER BY id; RESET statement_mem;</pre> <ul style="list-style-type: none"> <li>系统参数max_statement_mem，可以控制Segment级别的内存使用的上限，单个QUERY申请的memory不能超过max_statement_mem。</li> </ul> <p>您可以随便更改会话级的statement_mem参数，但是不要随便改max_statement_mem参数。建议max_statement_mem设置如下：</p> <pre>(seghost_physical_memory) / (average_number_concurrent_queries)</pre> <ul style="list-style-type: none"> <li>参数gp_resqueue_memory_policy=eager_free时，表示数据库在评估SQL对内存的申请渴望时，分阶段统计，也就是说一个SQL可能总共需要申请1GB内存，但是每个阶段只申请100MB，所以需要的内存实际上是100MB。使用eager_free策略，可以降低QUERY出现内存不足的可能性。</li> </ul>
MAX_COST float	<p>表示资源组允许同时执行的QUERY加起来的COST上限。COST是SQL执行计划中的总成本。</p> <p>设置为浮点（如 100.0）或指数（如 1e+2），-1表示不限制。</p>
COST_OVERCOMMIT boolean	<p>当系统空闲时，是否允许超过max_cost的限制。TRUE表示允许</p>
MIN_COST float	<p>资源超限时，是需要排队的，但是，当QUERY的成本低于min_cost时，不需要排队，直接运行。</p>
PRIORITY= {MIN LOW MEDIUM HIGH MAX}	<p>指当前资源队列的优先级，当资源紧张时，优先将CPU资源分配给高优先级的资源队列。处于高优先级的资源队列中的SQL，可以获得更高优先级的CPU资源。建议将实时性要求高的查询对应的用户分配到高优先级的资源队列中。</p> <p>类似Linux CGROUP中的CPU资源组，实时task和普通task的时间片策略。</p>

修改资源队列限制示例。

```
ALTER RESOURCE QUEUE myqueue WITH (MAX_COST=-1.0, MIN_COST= -1.0);
```

将用户放到资源队列中示例。

```
ALTER ROLE sammy RESOURCE QUEUE poweruser;
```

## 资源队列参数说明

参数	说明
gp_resqueue_memory_policy	资源队列的内存管理策略。
gp_resqueue_priority	是否使用资源队列的优先级。 <ul style="list-style-type: none"><li>ON: 使用。</li><li>OFF: 不使用。不使用资源队列优先级时，所有队列公平对待。</li></ul>
gp_resqueue_priority_cpucore_per_segment	每个Segment可以使用的CPU核数，例如8核的主机上运行了2个Primary Segment，则配置为4。master上面如果没有其他节点，则配置为8。 当发生CPU抢占时，优先级高的资源组中运行的SQL，优先分配CPU资源。
gp_resqueue_priority_sweeper_interval	CPU时间片统计间隔，SQL执行时，计算它的share值。根据优先级以及计算gp_resqueue_priority_cpucore_per_segment出来。 越小越频繁，优先级设置带来的效果越好。但是开销越大。

资源队列的推荐使用方法。


- 建议为每个用户创建一个资源队列。

AnalyticDB PostgreSQL版默认的资源队列为pg\_default，如果不创建队列，那么所有的用户都会被指定给pg\_default。这是非常不建议的。建议的做法是为每个用户创建一个资源队列。因为通常一个数据库用户对应一个业务。不同的数据库用户可能对应不同的业务或者使用者。例如业务用户、分析师用户、开发者、DBA等。

- 不建议业务使用超级用户来执行QUERY。

超级用户发起的SQL请求不受资源队列的限制，仅仅受前面讲到的参数的限制。因此如果要使用resource queue来限制资源的使用，就不建议业务使用超级用户来执行QUERY。

- ACTIVE\_STATEMENTS表示资源队列中，允许同时执行的SQL。注意当QUERY的成本低于min\_cost时，不需要排队，直接运行。
- MEMORY\_LIMIT，设置资源队列中所有SQL允许的最大内存使用量。前面讲了突破方法，statement\_mem设置的优先级更高，可以突破resource queue的限制。

 说明 所有资源队列的内存加起来不要超过gp\_vmem\_protect\_limit的限制。

- 通过配置资源队列的优先级，可以区分不同的业务。

例如出报表的业务优先级最高，其次是普通业务，其次是分析师。这样的情况，我们可以创建3个资源队列，分别使用MEDIUM|HIGH|MAX的优先级。

- 如果每个时间段的资源需求不一样，可以写一个CRONTAB任务，定时地调整资源队列的限制。

例如白天分析师的优先级更高，晚上处理报表的队列优先级更高。目前AnalyticDB PostgreSQL版还不支持按时间段来设置资源限制，所以只能外部部署任务，通过alter resource queue来实现。

- 通过gp\_toolkit提供的视图，可以观察资源队列的资源使用。

```
gp_toolkit.gp_resq_activity
gp_toolkit.gp_resq_activity_by_queue
gp_toolkit.gp_resq_priority_backend
gp_toolkit.gp_resq_priority_statement
gp_toolkit.gp_resq_role
gp_toolkit.gp_resqueue_status
```