云数据库 GaussDB

用户指南

天翼云科技有限公司

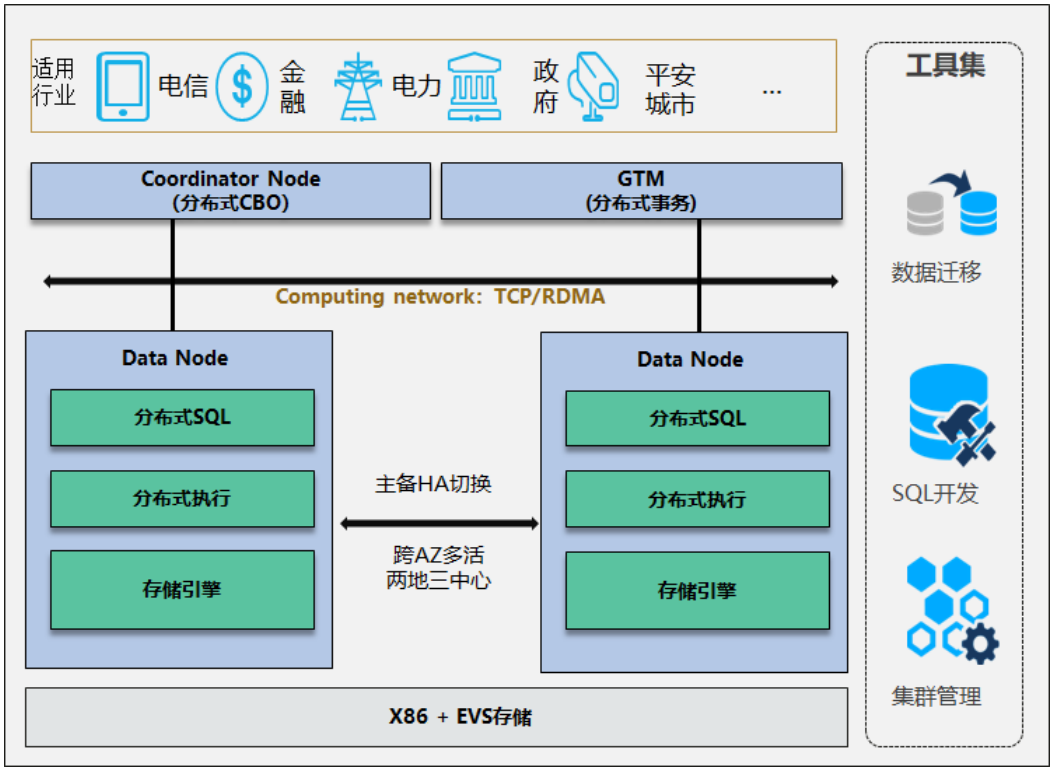
目 录

1 产品介绍

1.1 什么是云数据库 GaussDB

云数据库 GaussDB 是分布式关系型数据库。该产品具备企业级复杂事务混合负载能力，同时支持分布式事务，同城跨 AZ 部署，支持 1000+的扩展能力，PB 级海量存储。同时拥有云上高可用，高可靠，高安全，弹性伸缩，一键部署，快速备份恢复，监控告警等关键能力，能为企业提供功能全面，稳定可靠，扩展性强，性能优越的企业级数据库服务。

云数据库 GaussDB 分布式形态整体架构如下：



1.2 应用场景

- 交易型应用

  大并发、大数据量、以联机事务处理为主的交易型应用，如政务、金融、电商、O2O、电信 CRM/计费等，服务能力支持高扩展、弹性扩缩，应用可按需选择不同的部署规模。

- 详单查询

  具备 PB 级数据负载能力，通过内存分析技术满足海量数据边入库边查询要求，适用于安全、电信、金融、物联网等行业的详单查询业务。

1.3 常用概念

实例

云数据库 GaussDB 的最小管理单元是实例，一个实例代表了一个独立运行的数据库。用户可以在控制台创建和管理云数据库 GaussDB 实例。实例的状态、规格、存储类型、版本，请参考 1.5 实例说明。

实例版本

云数据库 GaussDB 目前支持 2.7 版本。

实例类型

云数据库 GaussDB 支持分布式版和主备版实例。分布式形态能够支撑较大的数据量，且提供了横向扩展的能力，可以通过扩容的方式提高实例的数据容量和并发能力。主备版适用于数据量较小，且长期来看数据不会大幅度增长，但是对数据的可靠性，以及业务的可用性有一定诉求的场景。

实例规格

数据库实例各种规格（vCPU 个数、内存（GB））请参考 1.5.2 数据库实例规格。

CN

Coordinator Node，负责数据库系统元数据存储、查询任务的分解和部分执行，以及将 DN 中查询结果汇聚在一起。

DN

Data Node，和 CN 对应的概念。负责实际执行表数据的存储、查询操作。

自动备份

创建实例时，云数据库 GaussDB 服务默认开启自动备份策略，实例创建成功后，您可对其进行修改，云数据库 GaussDB 服务会根据您的配置，自动创建数据库实例的备份。

手动备份

手动备份是由用户启动的数据库实例的全量备份，它会一直保存，直到用户手动删除。

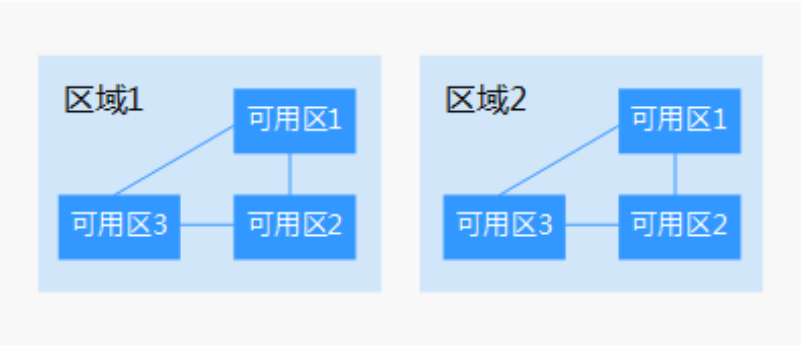
区域和可用区

我们用区域和可用区来描述数据中心的位置，您可以在特定的区域、可用区创建资源。

- 区域（Region）指物理的数据中心。每个区域完全独立，这样可以实现最大程度的容错能力和稳定性。资源创建成功后不能更换区域。
- 可用区（AZ，Availability Zone）是同一区域内，电力和网络互相隔离的物理区域，一个可用区不受其他可用区故障的影响。一个区域内可以有多个可用区，不同可用区之间物理隔离，但内网互通，既保障了可用区的独立性，又提供了低价、低时延的网络连接。

图 1-1 阐明了区域和可用区之间的关系。

图1-1 区域和可用区



1.4 产品优势

- **高安全**

  云数据库 GaussDB 拥有 TOP 级的商业数据库安全特性：数据动态脱敏，TDE 透明加密，行级访问控制，密态计算。能够满足政企&金融级客户的核心安全诉求。

- **健全的工具与服务化能力**

  云数据库 GaussDB 已经拥有云服务商用服务化部署能力，同时支持 DAS、DRS 等生态工具。有效保障用户开发、运维、优化、监控、迁移等日常工作需要。

- **全栈自研**

  云数据库 GaussDB 是当前国内唯一能够做到全栈自主可控的国产品牌。同时云数据库 GaussDB 能够基于硬件优势在底层不断进行优化，提升产品综合性能。

- **开源生态**

  云数据库 GaussDB 已经支持开源社区，并提供主备版版本下载。

1.5 实例说明

1.5.1 数据库实例状态

数据库实例状态

数据库实例状态是数据库实例的运行情况。用户可以使用管理控制台操作查看数据库实例状态。

表1-1 状态及说明

状态	说明
正常	数据库实例正常和可用。
异常	数据库实例不可用。
创建中	正在创建数据库实例。
创建失败	数据库实例创建失败。
重启中	按照用户请求，或修改需要重启才能生效的参数后重启实例。
扩容中	数据库实例磁盘扩容中。
添加节点中	数据库实例节点扩容中。
备份中	正在备份数据库实例。
恢复中	正在恢复备份到实例中。
恢复失败	实例恢复失败。
存储空间满	实例的磁盘空间已满，此时不可进行数据库写入操作。
已删除	数据库实例已被删除，对于已经删除的实例，将不会在实例列表中显示。
版本升级中	实例版本正在升级中。
参数变	数据库参数修改后，有些参数修改，需等待用户重启实例才能生效。

状态	说明
更，等待重启	
变更副本中	数据库实例正在执行降副本操作。

备份状态

表1-2 备份状态及说明

状态	说明
备份完成	表明备份任务执行成功
备份失败	表明备份任务执行失败。
备份中	表明正在进行备份中。

1.5.2 数据库实例规格

表1-3 性能规格

规格	vCPU(个)	内存(GB)	默认最大连接数（单个 CN）
通用增强 II 型 说明 通用增强 II 型基于 X86 架构。	8	64 说明 该规格不能用于生产环境。	2000
	16	128	4000
	32	256	9000
	64	512	18000

1.5.3 数据库实例存储类型

数据库系统通常是 IT 系统最为重要的系统，对存储 IO 性能要求高，云数据库 GaussDB 支持“超高 IO”存储类型，最大吞吐量为 350MB/S。

1.5.4 数据库实例版本

云数据库 GaussDB 目前支持 2.7 版本。

1.6 权限管理

如果您需要对购买的云数据库 GaussDB 资源，为企业中的员工设置不同的访问权限，为达到不同员工之间的权限隔离，您可以使用统一身份认证服务（Identity and Access Management，简称 IAM）进行精细的权限管理。该服务提供用户身份认证、权限分配、访问控制等功能，可以帮助您安全的控制资源的访问。

如果账号已经能满足您的要求，不需要创建独立的 IAM 用户进行权限管理，您可以跳过本章节，不影响您使用云数据库 GaussDB 服务的其它功能。

通过 IAM，您可以在账号中给员工创建 IAM 用户，并授权控制他们对资源的访问范围。例如您的员工中有负责软件开发的人员，您希望开发人员拥有云数据库 GaussDB 的使用权限，但是不希望他们拥有删除云数据库 GaussDB 等高危操作的权限，那么您可以使用 IAM 为开发人员创建用户，通过授予仅能使用云数据库 GaussDB，但是不允许删除云数据库 GaussDB 的权限，控制他们对云数据库 GaussDB 资源的使用范围。

云数据库 GaussDB 权限

默认情况下，管理员创建的 IAM 用户没有任何权限，需要将其加入用户组，并给用户组授予策略或角色，才能使得用户组中的用户获得对应的权限，这一过程称为授权。授权后，用户就可以基于被授予的权限对云服务进行操作。

云数据库 GaussDB 部署时通过物理区域划分，为项目级服务。授权时，“作用范围”需要选择“区域级项目”，然后在指定区域对应的项目中设置相关权限，并且该权限仅对此项目生效；如果在“所有项目”中设置权限，则该权限在所有区域项目中都生效。访问云数据库 GaussDB 时，需要先切换至授权区域。

根据授权精细程度分为角色和策略。

- 角色：IAM 最初提供的一种根据用户的工作职能定义权限的粗粒度授权机制。该机制以服务为粒度，提供有限的服务相关角色用于授权。由于各服务之间存在业务依赖关系，因此给用户授予角色时，可能需要一并授予依赖的其他角色，才能正确完成业务。角色并不能满足用户对精细化授权的要求，无法完全达到企业对权限最小化的安全管控要求。
- 策略：IAM 最新提供的一种细粒度授权的能力，可以精确到具体服务的操作、资源以及请求条件等。基于策略的授权是一种更加灵活的授权方式，能够满足企业对权限最小化的安全管控要求。例如：针对云数据库 GaussDB 服务，管理员能够控制 IAM 用户仅能对某一类数据库资源进行指定的管理操作。多数细粒度策略以 API 接口为粒度进行权限拆分。

如表 1-4 所示，包括了云数据库 GaussDB 的所有系统权限。

表1-4 云数据库 GaussDB 系统权限

策略名称	描述	类别
云数据库 GaussDB FullAccess	云数据库 云数据库 GaussDB 服务的所有执行权限。	系统策略
云数据库 GaussDB	云数据库 云数据库 GaussDB 服务的只读访问	系统策略

策略名称	描述	类别
ReadOnlyAccess	权限。	

表 1-5 列出了云数据库 GaussDB 常用操作与系统权限的授权关系，您可以参照该表选择合适的系统权限。

表1-5 常用操作与系统权限的关系

操作	云数据库 GaussDB FullAccess	云数据库 GaussDB ReadOnlyAccess
创建云数据库 GaussDB 实例	√	x
删除云数据库 GaussDB 实例	√	x
查询云数据库 GaussDB 实例列表	√	√

表1-6 常用操作与对应授权项

操作名称	授权项	备注
创建数据库实例	gaussdb:instance:create gaussdb:param:list	界面选择 VPC、子网、安全组需要配置： vpc:vpcs:list vpc:vpcs:get vpc:subnets:get vpc:securityGroups:get 创建加密实例需要在项目上配置: kms:cmk:get kms:cmk:list 操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
规格变更	gaussdb:instance:modifySpec	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
扩容节点	gaussdb:instance:modifySpec	操作失败上报事件监控

操作名称	授权项	备注
		需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
磁盘扩容	gaussdb:instance:modifySpec	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
重启数据库实例	gaussdb:instance:restart	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
删除数据库实例	gaussdb:instance:delete	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
查询数据库实例列表	gaussdb:instance:list	无
实例详情	gaussdb:instance:list	实例详情界面展示 VPC、子网、安全组，需要对应配置 vpc:*:get 和 vpc:*:list。显示磁盘已使用大小，需要配置 ces:*:list。
修改数据库实例密码	gaussdb:instance:modify	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
修改实例名称	gaussdb:instance:modify	无
绑定/解绑公网 IP	gaussdb:instance:modify	界面列出公网 IP 需要配置： vpc:publicIps:get vpc:publicIps:list 操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
创建参数模板	gaussdb:param:create gaussdb:param:list	无

操作名称	授权项	备注
修改参数模板	gaussdb:param:modify	无
获取参数模板列表	gaussdb:param:list	无
应用参数模板	gaussdb:param:apply	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create
删除参数模板	gaussdb:param:delete	无
创建手动备份	gaussdb:backup:create	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
获取备份列表	gaussdb:backup:list	无
修改备份策略	gaussdb:instance:modifyBackupPolicy	无
删除手动备份	gaussdb:backup:delete	操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create"
恢复到新实例	gaussdb:instance:create	界面选择 VPC、子网、安全组需要配置： vpc:vpcs:list vpc:vpcs:get vpc:subnets:get vpc:securityGroups:get 操作失败上报事件监控需要配置： "ces:alarmsOnOff:put" "ces:alarms:create
查询项目标签	gaussdb:tag:list	无
批量添加删除项目标签	gaussdb:instance:dealTag	无
修改配额	gaussdb:quota:modify	无

1.7 云数据库 GaussDB 的约束与限制

云数据库 云数据库 GaussDB 在使用上有一些固定限制，用来提高实例的稳定性和安全性，具体详见表 1-7。

表1-7 功能约束与限制

功能	使用限制
数据库访问	• 弹性云主机必须处于目标云数据库 GaussDB 实例所属安全组允许访问的范围内。 如果云数据库 GaussDB 实例与弹性云主机处于不同的安全组，系统默认不能访问。需要在云数据库 GaussDB 的安全组添加一条“入”的访问规则。 • 云数据库 GaussDB 实例的默认端口为 8000。
部署	实例所部署的服务器，对用户都不可见，即只允许应用程序通过 IP 地址和端口访问数据库。
数据库的 root 权限	创建实例页面只提供管理员 root 用户权限。
重启云数据库 GaussDB 实例	无法通过命令行重启，必须通过云数据库 GaussDB 的管理控制台操作重启实例。

1.8 云数据库 GaussDB 与其他服务的关系

云数据库 GaussDB 与其他服务的关系如表 1-8。

表1-8 与其他服务的关系

相关服务	交互功能
弹性云主机（ECS）	云数据库 GaussDB 服务通过弹性云主机（Elastic Cloud Server，简称 ECS）远程连接云数据库 GaussDB 可以有效的降低应用响应时间。
虚拟私有云（VPC）	对您的云数据库 GaussDB 实例进行网络隔离和访问控制。
对象存储服务（OBS）	存储云数据库 GaussDB 实例的自动和手动备份数据。

2 快速入门

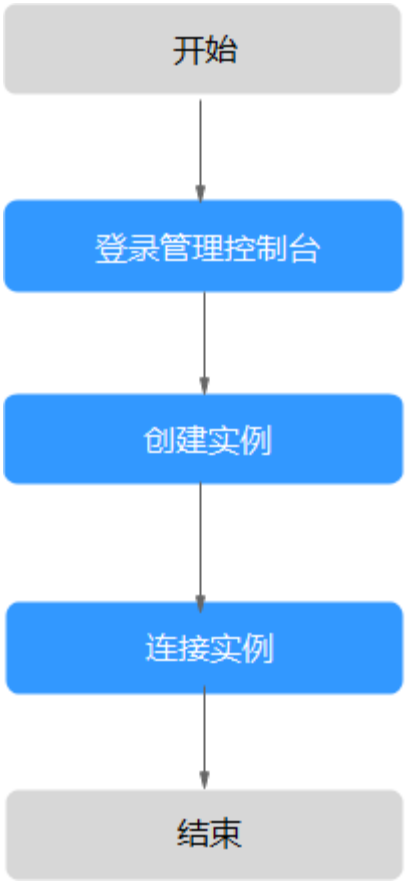
本章指导您快速的创建、连接并使用云数据库 GaussDB。

首次使用云数据库 GaussDB，请先了解 1.7 云数据库 GaussDB 的约束与限制。

2.1 简介

流程图

图2-1 快速入门流程图



操作步骤

表2-1 相关操作及参考手册

相关操作	参考章节
登录管理控制台	2.2 登录管理控制台
创建实例	2.3 创建实例
连接实例	根据业务场景选择连接方式： • 2.4 使用客户端连接实例

2.2 登录管理控制台

操作步骤

步骤 1 登录管理控制台。

步骤 2 单击管理控制台左上角的，选择区域和项目。

步骤 3 在页面左上角单击，选择“数据库 > 云数据库 云数据库 GaussDB”。进入云数据库 云数据库 GaussDB 信息页面。

步骤 4 在左侧导航栏选择云数据库 GaussDB > 实例管理。

进入云数据库 GaussDB 信息页面。

----结束

2.3 创建实例

操作场景

本章将介绍在云数据库 GaussDB 的管理控制台创建实例。

操作步骤

步骤 1 2.2 登录管理控制台。

步骤 2 在“实例管理”页面，单击“创建数据库实例”。

步骤 3 在创建实例页面，选择计费模式，填写并选择实例相关信息后，单击“立即购买”。

表2-2 基本信息

参数	描述
区域	租户当前所在区域，也可在页面左上角切换。 **说明** 不同区域内的产品内网不互通，且创建后不能更换，请谨慎选择。
实例名称	实例名称长度在 4 个到 64 个字符之间，必须以字母开头，可以包含字母、数字、中划线或下划线，不能包含其他特殊字符。
数据库版本	云数据库 GaussDB 目前支持 2.7 版本。
实例类型	“分布式版”：分布式形态能够支撑较大的数据量，且提供了横向扩展的能力，可以通过扩容的方式提高实例的数据容量和并发能力。 “主备版”：适用于数据量较小，且长期来看数据不会大幅度增长，但是对数据的可靠性，以及业务的可用性有一定诉求的场

参数	描述
	景。
部署形态	独立部署：将数据库组件部署在不同节点上。适用于可靠性、稳定性要求较高，实例规模较大的场景。 高可用（1 主 2 备）：采用一主两备三节点的部署模式，包含一个分片。
事务一致性	仅分布式版形态有该参数。 • 强一致性：应用更新数据时，用户都能查询到全部已经成功提交的数据，对性能有影响。 • 最终一致性：应用更新数据时，用户查询到的数据可能不相同，有可能是更新后的值，也有可能是更新前的值，但经过一段时间后，查询到的数据是更新后的值，该种类型通常具有较高的性能。
切换策略	• 数据高可靠：对数据一致性要求高的系统推荐选择数据高可靠，在故障切换的时候优先保障数据一致性。 • 业务高可用：对业务在线时间要求高的系统推荐使用业务高可用，在故障切换的时候优先保证数据库可用性。 **说明** 在业务高可用场景下需要谨慎修改如下数据库参数： • recovery_time_target：不当修改该参数会导致实例频繁进行强制切换，请在技术人员指导下进行修改。 • audit_system_object：不当修改该参数会导致丢失 DDL 审计日志，请在技术人员指导下进行修改。
副本集数量	仅分布式部署形态可选。每个分片下 1 主多副本的部署方案，3 副本就是 1 主 2 备的部署方式。
分片数量	仅分布式部署形态可选。一个分片指的是一组 DN 副本集，分片内的 DN 数量与“副本集数量”参数有关，例如副本集数量为 3，则一个分片就包含一主两备三个 DN 节点。
协调节点数量	仅分布式部署形态可选。数据库中包含的协调节点（CN，Coordinator Node）数量，协调节点负责接收来自应用的访问请求，并向客户端返回执行结果；负责分解任务，并调度在各分片上并行执行。 **须知** CN 数量必须小于或等于两倍的分片数。
可用区	可用区只支持部署在一个或者三个可用区。 可用区指在同一区域下，电力、网络隔离的物理区域，可用区之间内网互通，不同可用区之间物理隔离。
时区	由于世界各国家与地区经度不同，地方时也有所不同，因此会划分为不同的时区。时区可在创建实例时选择。

表2-3 规格与存储

参数	描述
性能规格	实例的 CPU 和内存。不同性能规格对应不同连接数。
存储类型	实例的存储类型决定实例的读写速度。最大吞吐量越高，读写速度越快。
存储空间	您申请的存储空间会有必要的文件系统开销，这些开销包括索引节点和保留块，以及数据库运行必需的空间。

表2-4 网络

参数	描述
虚拟私有云	云数据库 GaussDB 实例所在的虚拟专用网络，可以对不同业务进行网络隔离。您需要创建或选择所需的虚拟私有云。
子网	通过子网提供与其他网络隔离的、可以独享的网络资源，以提高网络安全性。子网在可用区内才会有效，创建云数据库 GaussDB 实例的子网默认开启 DHCP 功能，不可关闭。创建实例时云数据库 GaussDB 会自动为您配置内网地址。
内网安全组	控制网络出/入及端口的访问，默认添加了云数据库 GaussDB 实例所属的内网安全组访问。 创建分布式版实例时，如果需要修改内网安全组，请确保入方向规则 TCP 协议端口包含：40000-60480,20050,5000-5001,2379-2380,6000,6500，<database port> - (<database port> + 100)。（例如设置的数据库端口为 8000，则安全组中需要包含 8000-8100）。 创建主备版实例时，如果需要修改内网安全组，请确保入方向规则 TCP 协议端口包含：20050，5000-5001，2379-2380，6000，6500，<database port> - (<database port> + 100)。（例如设置的数据库端口为 8000，则安全组中需要包含 8000-8100）。 内网安全组限制实例的安全访问规则，加强云数据库 GaussDB 与其他服务间的安全访问。请确保所选取的内网安全组允许客户端访问数据库实例。

表2-5 数据库配置

参数	描述
管理员帐户名	数据库的登录名称默认为 root。
管理员密码	请您输入高强度密码并定期修改，以提高安全性，防止出现密码被暴力破解等安全风险。 须知

参数	描述
	设置的密码需满足以下几个条件： • 8 到 32 个字符。 • 至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（限定为~!@#%^*-_=+?,）四类字符中的三类字符。 请妥善保管您的密码，因为系统将无法获取您的密码信息。
确认密码	必须和管理员密码相同。

表2-6 参数模板

参数	描述
参数模板	数据库参数模板就像是数据库引擎配置值的容器，参数模板中的参数可应用于一个或多个相同类型的数据库实例。实例创建成功后，参数模板可进行修改。

说明

云数据库 GaussDB 的性能，取决于用户申请云数据库 GaussDB 时所选择的配置。可供用户选择的硬件配置项为性能规格、存储类型以及存储空间。

步骤 4 确认详细信息。

进行规格确认。

- 如果需要重新选择实例规格，单击“上一步”，回到上个页面修改实例信息。
- 如果规格确认无误，单击“提交”，完成实例的申请。

步骤 5 云数据库 GaussDB 实例创建成功后，用户可以在“实例管理”页面对其进行查看和管理。

- 创建云数据库 GaussDB 实例过程中，状态显示为“创建中”。
- 在实例列表的右上角，单击  刷新列表，可查看到创建完成的实例状态显示为“正常”。
- 创建完成后会自动进行一次全量备份，用于记录实例的初始状态。

----结束

2.4 使用客户端连接实例

2.4.1 实例连接方式介绍

云数据库 GaussDB 提供使用内网、公网的连接方式。

表2-7 云数据库 GaussDB 连接方式

连接方式	IP 地址	使用场景	说明
2.4.2 通过内网连接实例	内网 IP 地址	系统默认提供内网 IP 地址。 当应用部署在弹性云主机上，且该弹性云主机与云数据库 GaussDB 实例处于同一区域，同一 VPC 时，建议单独使用内网 IP 连接弹性云主机与云数据库 GaussDB 实例。	• 安全性高，可实现云数据库 GaussDB 的较好性能。 • 推荐使用内网连接。
2.4.3 通过公网连接实例	弹性公网 IP	不能通过内网 IP 地址访问云数据库 GaussDB 实例时，使用公网访问，建议单独绑定弹性公网 IP 连接弹性云主机（或公网主机）与云数据库 GaussDB 实例。	• 降低安全性。 • 为了获得更快的传输速率和更高的安全性，建议您将应用迁移到与您的云数据库 GaussDB 实例在同一子网，使用内网连接。

2.4.2 通过内网连接实例

本章介绍如何在管理控制台创建云数据库 GaussDB 实例，并通过内网使用弹性云主机连接云数据库 GaussDB 实例。

准备工作

云数据库 GaussDB 提供 gsql 工具帮助您在命令行下连接数据库，您需要提前创建一台弹性云主机用于安装 gsql 工具。具体请参见《弹性云主机用户指南》。

须知

操作系统需要选择 Euler 操作系统。gsql 支持的操作系统版本如下：

X86：EulerOS V2.0SP5。

设置安全组规则

在访问数据库前，您需要将访问数据库的 IP 地址，或者 IP 段加入安全组入方向的访问规则，操作请参见 3.4.12 设置安全组规则。

远程连接数据库

步骤 1 登录申请的弹性云主机。

步骤 2 在申请的弹性云主机上，上传客户端工具包并配置 gsql 的执行环境变量。

1. 以 root 用户登录客户端机器。
2. 创建“/tmp/tools”目录。

```
mkdir /tmp/tools
```

3. 获取云数据库 GaussDB 软件包并解压。

   unzip GaussDB_opengauss_client_tools.zip

4. 根据申请的弹性云主机的操作系统架构进入不同目录，获取“GaussDB-Kernel-xxx-EULER-64bit-gsql.tar.gz”，并上传到申请的弹性云主机“/tmp/tools”路径下。

说明

软件包相对位置为安装时所放位置，根据实际情况填写。

5. 解压文件。

```
cd /tmp/tools
tar -zxvf GaussDB-Kernel-xxx-EULER-64bit-gsql.tar.gz
```

   xxx 为版本号，请根据实际情况替换。

6. 设置环境变量。

   打开“~/.bashrc”文件。

```
vi ~/.bashrc
```

   按下 i 键进入 INSERT 模式，在其中输入如下内容后，单击“ESC”退出编辑模式，使用“:wq!”命令保存并退出。

```
export PATH=/tmp/tools/bin:$PATH
export LD_LIBRARY_PATH=/tmp/tools/lib:$LD_LIBRARY_PATH
```

   使环境变量配置生效。

```
source ~/.bashrc
```

步骤 3 执行如下指令，根据提示输入密码，连接数据库。

数据库创建成功后，会默认生成名称为 postgres 的数据库，此处以 postgres 库为例。

```
gsql -d postgres -h 10.0.0.0 -U root -p 8000
Password for user root:
```

postgres 为需要连接的数据库名称，10.0.0.0 分布式为 CN 的 IP 地址，主备版为主 DN 的 IP 地址，root 为登录数据库的用户名，8000 为 CN 的端口号。

----结束

SSL 连接

步骤 1 2.2 登录管理控制台。

步骤 2 在“实例管理”页面，单击实例名称进入“基本信息”页面，单击“数据库信息”模块“SSL”处的，下载根证书或捆绑包。

步骤 3 将根证书上传至需连接云数据库 GaussDB 实例的弹性云主机，或保存到可访问数据库实例的设备。

将根证书导入弹性云主机 Linux 操作系统，您可以使用任何终端连接工具（如 WinSCP、PuTTY 等工具）将证书上传至 Linux 系统任一目录下以实现该目标。

步骤 4 连接云数据库 GaussDB 实例。

以 Linux 系统为例，在弹性云主机设置环境变量，执行如下命令。

```
export PGSSLMODE=<sslmode>
export PGSSLROOTCERT=<ca-file-directory>
```

gsql -h <*host*> **-p** <*port*> **-d** <*database*> **-*U*** <*user*>

表2-8 参数说明

参数	说明
<*host*>	主机 IP，在“实例管理”页面单击实例名称，进入“基本信息”页面。“连接信息”模块的“内网地址”（通过弹性云主机访问）。
<*port*>	端口，默认 8000，当前端口，即在“实例管理”页面单击实例名称，进入“基本信息”页面，“连接信息”模块的“数据库端口”。
<*database*>	需要连接的数据库名，默认管理数据库是 postgres。
<*user*>	用户名，即云数据库 GaussDB 数据库帐号，默认管理员帐号为 root。
<*ca-file-directory*>	ssl 连接 CA 证书路径。
<*sslmode*>	ssl 连接模式，设置为“verify-ca”，通过检查证书链（Certificate Chain，以下简称 CA）来验证服务是否可信任。

在弹性云主机设置环境变量，使用 root 用户 SSL 连接 postgres 数据库实例，具体示例如下：

```
export PGSSLMODE="verify-ca"
export PGSSLROOTCERT="/home/Ruby/ca.pem"
```

gsql -d *postgres* **-h** *10.0.0.0* **-U** *root* **-p** *8000*

```
Password for user root:
```

步骤 5 登录数据库后，出现如下信息，表示通过 SSL 连接成功。

```
SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256)
```

----结束

2.4.3 通过公网连接实例

拥有云数据库 GaussDB 实例后，默认未开启公网访问功能（即未绑定弹性公网 IP）。云数据库 GaussDB 支持用户绑定弹性公网 IP，在公共网络来访问数据库实例，绑定后也可根据需要解绑。

准备工作

云数据库 GaussDB 提供 gsql 工具帮助您在命令行下连接数据库，您需要提前创建一台弹性云主机用于安装 gsql 工具。具体请参见《弹性云主机用户指南》。

须知

操作系统需要选择 Euler 操作系统。gsql 支持的操作系统版本如下：

X86：EulerOS V2.0SP5。

绑定弹性公网 IP

如果您想要通过公网访问数据库实例，那么您需要为数据库实例绑定弹性公网 IP，具体操作请参考 3.4.5 绑定和解绑弹性公网 IP。

设置安全组规则

在访问数据库前，您需要将访问数据库的 IP 地址，或者 IP 段加入安全组入方向的访问规则，操作请参见 3.4.12 设置安全组规则。

远程连接数据库

步骤 1 登录申请的弹性云主机。

步骤 2 在申请的弹性云主机上，上传客户端工具包并配置 gsql 的执行环境变量。

1. 以 root 用户登录客户端机器。
2. 创建“/tmp/tools”目录。

```
mkdir /tmp/tools
```

3. 获取 GaussDB 软件包并解压。

   unzip GaussDB_opengauss_client_tools.zip

4. 根据申请的弹性云主机的操作系统架构进入不同目录，获取“GaussDB-Kernel-xxx-EULER-64bit-gsql.tar.gz”，并上传到申请的弹性云主机“/tmp/tools”路径下。

说明

软件包相对位置为安装时所放位置，根据实际情况填写。

5. 解压文件。

```
cd /tmp/tools
tar -zxvf GaussDB-Kernel-V500R001C00-EULER-64bit-gsql.tar.gz
```

6. 设置环境变量。

   打开“~/.bashrc”文件。

   ```
   vi ~/.bashrc
   ```

   在其中输入如下内容后，单击“ESC”退出编辑模式，使用“:wq!”命令保存并退出。

   ```
   export PATH=/tmp/tools/bin:$PATH
   export LD_LIBRARY_PATH=/tmp/tools/lib:$LD_LIBRARY_PATH
   ```

   使环境变量配置生效。

   ```
   source ~/.bashrc
   ```

步骤 3 执行如下指令，根据提示输入密码，连接数据库。

数据库创建成功后，会默认生成名称为 postgres 的数据库，此处以 postgres 库为例。

```
gsql -d postgres -h 10.0.0.0 -U root -p 8000
Password for user root:
```

postgres 为需要连接的数据库名称，10.0.0.0 为实例绑定的公网 IP 地址，root 为登录数据库的用户名，8000 为实例的端口号。

----结束

SSL 连接

步骤 1 2.2 登录管理控制台。

步骤 2 在“实例管理”页面，单击实例名称进入“基本信息”页面，单击“数据库信息”模块“SSL”处的，下载根证书或捆绑包。

步骤 3 将根证书上传至需连接云数据库 GaussDB 实例的弹性云主机，或保存到可访问数据库实例的设备。

将根证书导入弹性云主机 Linux 操作系统，您可以使用任何终端连接工具（如 WinSCP、PuTTY 等工具）将证书上传至 Linux 系统任一目录下以实现该目标。

步骤 4 连接云数据库 GaussDB 实例。

以 Linux 系统为例，在弹性云主机设置环境变量，执行如下命令。

```
export PGSSLMODE=<sslmode>
export PGSSLROOTCERT=<ca-file-directory>
```

gsql -h <*host*> **-p** <*port*> **-d** <*database*> ***-U*** <*user*>

表2-9 参数说明

参数	说明
<*host*>	主机 IP，在“实例管理”页面单击实例名称，进入“基本信息”页面。“连接信息”模块的“内网地址”（通过弹性云主机访问）。
<*port*>	端口，默认 8000，当前端口，即在“实例管理”页面单击实例名称，进入“基本信息”页面，“连接信息”模块的“数据库端

参数	说明
	口”。
<database>	需要连接的数据库名，默认管理数据库是 postgres。
<user>	用户名，即云数据库 GaussDB 数据库帐号，默认管理员帐号为 root。
<ca-file-directory>	ssl 连接 CA 证书路径。
<sslmode>	ssl 连接模式，设置为“verify-ca”，通过检查证书链（Certificate Chain，以下简称 CA）来验证服务是否可信任。

在弹性云主机设置环境变量，使用 root 用户 SSL 连接 postgres 数据库实例，具体示例如下：

```
export PGSSLMODE="verify-ca"
export PGSSLROOTCERT="/home/Ruby/ca.pem"
```

gsql -d *postgres* **-h** *10.0.0.0* **-U** *root* **-p** *8000*

```
Password for user root:
```

步骤 5 登录数据库后，出现如下信息，表示通过 SSL 连接成功。

```
SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256)
```

----结束

2.5 使用驱动连接实例

本章主要介绍如何使用 JDBC、ODBC 等驱动连接实例。

2.5.1 开发规范

如果用户在 APP 的开发中，使用了连接池机制，那么需要遵循如下规范：

- 如果在连接中设置了 GUC 参数，那么在将连接归还连接池之前，必须使用“SET SESSION AUTHORIZATION DEFAULT;RESET ALL;”将连接的状态清空。
- 如果使用了临时表，那么在将连接归还连接池之前，必须将临时表删除。

否则，连接池里面的连接就是有状态的，会对用户后续使用连接池进行操作的正确性带来影响。

2.5.2 使用 JDBC 连接数据库

前提条件

在创建数据库连接之前，需要先下载数据库驱动程序，

加载驱动有两种方法：

- 在代码中创建连接之前任意位置隐含装载：Class.forName("org.postgresql.Driver");
- 在 JVM 启动时参数传递：java -Djdbc.drivers=org.postgresql.Driver jdbctest

说明

上述 jdbctest 为测试用例程序的名称。

调用函数创建数据库连接

JDBC 提供了三个方法，用于创建数据库连接。

- DriverManager.getConnection(String url);
- DriverManager.getConnection(String url, Properties info);
- DriverManager.getConnection(String url, String user, String password);

分布式版连接参数参见表 2-10，主备版连接参数参见表 2-11。

表2-10 分布式版数据库连接参数

参数	描述
url	gsjdbc4.jar 数据库连接描述符。格式如下： • jdbc:postgresql:(数据库名称缺省则与用户名一致) • jdbc:postgresql:database • jdbc:postgresql://host/database(端口值缺省会使用默认端口） • jdbc:postgresql://host:port/database • jdbc:postgresql://host:port/database?param1=value1&param2=value2 • jdbc:postgresql://host1:port1,host2:port2/database?param1=value1&param2=value2 说明 使用 gsjdbc200.jar 时，将“jdbc:postgresql”修改为“jdbc:gaussdb” • database 为要连接的数据库名称。 • host 为数据库服务器名称或 IP 地址。 建议业务系统单独部署在实例外部，否则可能会影响数据库运行性能。 缺省情况下，连接服务器为 localhost。 • port 为数据库服务器端口。 缺省情况下，会尝试连接到 5432 端口的 database。 • param 为参数名称，即数据库连接属性。 参数可以配置在 URL 中，以"?"开始配置，以"="给参数赋值，以"&"作为不同参数的间隔。也可以采用 info 对象的属性方式进行配置，详细示例会在本节给出。 • value 为参数值，即数据库连接属性值。 • 连接时需配置 connectTimeout 和 socketTimeout，推荐配置为 2（如果未配置，默认为 0，即不会超时）。在 DN 与客户端出现网络故障时，客户端一直未收到 DN 侧 ACK 确认报文，会启动超时重传机制，不断的进行重传。当超时时间达到系统默认的 600s 后才会报超时错误，这也就会导致 RTO 时间很高。

参数	描述
info	数据库连接属性（所有属性大小写敏感）。常用的属性如下： PGDBNAME：String 类型。表示数据库名称（URL 中无需配置该参数，自动从 URL 中解析）。 PGHOST：String 类型。主机 IP 地址。若配置多个 IP，它们的 IP 和端口用“:”分隔，并作为整体以逗号分隔（URL 中无需配置该参数，自动从 URL 中解析）。 PGPORT：Integer 类型。主机端口号。若配置多个，它们的端口号和 IP 用":"分割，并作为整体以逗号分隔其他（URL 中无需配置该参数，自动从 URL 中解析）。 user：String 类型。表示创建连接的数据库用户。 password：String 类型。表示数据库用户的密码。 loggerLevel：String 类型。目前支持 4 种级别：OFF、INFO、DEBUG、TRACE。设置为 OFF 关闭日志。设置为 INFO、DEBUG 和 TRACE 记录的日志信息详细程度不同。 loggerFile：String 类型。用于指定日志输出路径（目录和文件名）。若只指定文件名，未指定目录则日志生成在客户端运行程序目录；若不配置或配置的路径不存在，则日志会默认通过流输出。 logger：String 类型。表示 JDBC Driver 要使用的日志输出框架。JDBC Driver 支持对接用户应用程序使用的日志输出框架。目前仅支持第三方的基于 Slf4j-API 的日志框架。 • 如果不设置或设置为 JDK LOGGER，则 JDBC Driver 使用 JDK LOGGER。 • 否则必须设置采用基于 Slf4j-API 第三方日志框架。 allowEncodingChanges：Boolean 类型。设置该参数值为“true”进行字符集类型更改，配合 characterEncoding=CHARSET 设置字符集，二者使用"&"分隔。 currentSchema：String 类型。在 search-path 中指定要设置的 schema。 loadBalanceHosts：Boolean 类型。在默认模式下（禁用），顺序连接 URL 中指定的多个主机。如果启用，则使用洗牌算法从候选主机中随机选择一个主机建立连接。 autoBalance：String 类型。 • 设置为 true 或 balance 或 roundrobin 表示开启 JDBC 负载均衡功能，将应用程序的多个连接均衡到中的各个节点。 例如：jdbc:postgresql://host1:port1,host2:port2/database?autoBalance=true JDBC 将定期获取（周期刷新可使用参数 refreshCNIpListTime 配置，默认为 10s）整个实例可用 CN 列表，比如获取到的列表为：host1:port1,host2:port2,host3:port3,host4:port4。 host1 和 host2 在 autoBalance 启用时，仅在首次连接做高可用用途，后续 Driver 将从 host1，host2，host3，host4 中依次选择可用的 CN 刷新可用 CN 列表，后续用户新建的 connection 将使用 RoundRobin 算法从 host1，host2，host3，host4 选取 CN 主机进行连接。

参数	描述
	• 设置为 priorityn 表示开启 JDBC 优先级负载均衡功能，将应用程序的多个连接首先均衡到 url 上配置的前 n 个中可用的 CN 数据库主节点，当 url 上配置前 n 个主节点全部不可用时，连接会随机分配到数据库实例中其他可用 CN 数据库主节点。n 为数字，不小于 0，且小于 url 上配置的 CN 数量。 例如： jdbc:postgresql://host1:port1,host2:port2,host3:port3,host4:port4/database?autoBalance=priority2 JDBC 将定期获取（周期按 refreshCNIpListTime 定义）整个实例可用 CN 列表，比如获取到的列表为： host1:port1,host2:port2,host3:port3,host4:port4,host5:port5,host6:port6，其中 host1 和 host2 处于 AZ1，host3 和 host4 处于 AZ2。 Driver 将从优先从 host1,host2 中做负载均衡，host1 和 host2 全部不可用才从 host3, host4, host5, host6 中随机选择 CN 主机连接。 • 设置为 shuffle 表示开启 JDBC 随机负载均衡功能，将应用程序的多个连接随机均衡到数据库实例中的各个可用 CN。 例如： jdbc:postgresql://host1:port1,host2:port2,host3:port3/database?autoBalance=shuffle JDBC 将定期获取(周期刷新可使用参数 refreshCNIpListTime 配置，默认为 10S)整个实例的可用 CN 列表，比如获取到的列表为： host1:port1,host2:port2,host3:port3,host4:port4。 host1:port1,host2:port2,host3:port3,仅在首次连接做高可用，后续连接将在刷新后的 CN 列表中，使用 shuffle 算法随机选用一个 CN 节点进行连接。 • 设置为 false，不开启负载均衡功能。默认为 false。 **注意** 负载均衡是基于连接级别，不是基于事务级别。如果连接是长连接，并且连接上的负载不均衡，无法保证 CN 主机上的负载是均衡的。 负载均衡仅能在分布式场景下使用，主备版环境中不可使用。 使用 priorityn 做负载均衡时，连接串中 IP 需要和 pgxc_node 表中 CN 的 node_host 保持一致，否则无法实现优先级负载均衡功能。 查询实例中可用 CN 的 IP 和 PORT，可使用此语句:“select node_host,node_port from pgxc_node where node_type='C' and nodeis_active = true;” refreshCNIpListTime：Integer 类型。JDBC 定期检测中状态，获取可用的 IP 列表的时间间隔，默认为 10 秒。 hostRecheckSeconds：Integer 类型。JDBC 尝试连接主机后会保存主机状态：连接成功或连接失败。在 hostRecheckSeconds 时间内保持可信，超过则状态失效。缺省值是 10 秒。 ssl：Boolean 类型。以 SSL 方式连接。 ssl=true 可支持 NonValidatingFactory 通道和使用证书的方式： • NonValidatingFactory 通道需要配置用户名和密码，同时将 SSL 设置为 true。

参数	描述
	• 配置客户端证书、密钥、根证书，将 SSL 设置为 true。 sslmode：String 类型。SSL 认证方式。取值范围为：require、verify-ca、verify-full。 • require 只尝试 SSL 连接，不会检查服务器证书是否由受信任的 CA 签发，且不会检查服务器主机名与证书中的主机名是否一致。 • verify-ca 只尝试 SSL 连接，并且验证服务器是否具有由可信任的证书机构签发的证书。 • verify-full 只尝试 SSL 连接，并且验证服务器是否具有由可信任的证书机构签发的证书，以及验证服务器主机名是否与证书中的一致。 sslcert：String 类型。提供证书文件的完整路径。客户端和服务端证书的类型为 End Entity。 sslkey：String 类型。提供密钥文件的完整路径。 sslrootcert：String 类型。SSL 根证书的文件名。根证书的类型为 CA。 sslpassword：String 类型。提供给 ConsoleCallbackHandler 使用。 sslpasswordcallback：String 类型。SSL 密码提供者的类名。缺省值：org.postgresql.ssl.jdbc4.LibPQFactory.ConsoleCallbackHandler。 sslfactory：String 类型。提供的值是 SSLSocketFactory 在建立 SSL 连接时用的类名。 sslprivatekeyfactory: String 类型。提供的值是实现私钥解密方法的接口 org.postgresql.ssl.PrivateKeyFactory 的实现类的完整限定类名。如果不提供，首先尝试默认的 jdk 私钥解密算法，如果无法解密，则使用 org.postgresql.ssl.BouncyCastlePrivateKeyFactory，用户需要自己提供 bcpkix-jdk15on.jar 包，版本建议：1.65 以上。 sslfactoryarg：String 类型。此值是上面提供的 sslfactory 类的构造函数的可选参数（不推荐使用本参数）。 sslhostnameverifier：String 类型。主机名验证程序的类名。接口实现 javax.net.ssl.HostnameVerifier，默认使用 org.postgresql.ssl.PGjdbcHostnameVerifier。 loginTimeout：Integer 类型。指建立数据库连接的等待时间。超时时间单位为秒。 connectTimeout：Integer 类型。用于连接服务器操作的超时值。如果连接到服务器花费的时间超过此值，则连接断开。超时时间单位为秒，值为 0 时表示已禁用，timeout 不发生。 socketTimeout：Integer 类型。用于 socket 读取操作的超时值。如果从服务器读取所花费的时间超过此值，则连接关闭。超时时间单位为秒，值为 0 时表示已禁用，timeout 不发生。 cancelSignalTimeout：Integer 类型。发送取消消息本身可能会阻塞，此属性控制用于取消命令的“connect 超时”和“socket 超时”。超时时间单位为秒，默认值为 10 秒。 tcpKeepAlive：Boolean 类型。启用或禁用 TCP 保活探测功能。默认为 false。 logUnclosedConnections：Boolean 类型。客户端可能由于未调用

参数	描述
	Connection 对象的 close()方法而泄漏 Connection 对象。最终这些对象将被垃圾回收，并且调用 finalize()方法。如果调用者自己忽略了此操作，该方法将关闭 Connection。 assumeMinServerVersion（废弃）：String 类型。该参数设置要连接的服务器版本。 ApplicationName：String 类型。设置正在使用连接的应用程序名称。通过在上查询 pgxc_stat_activity 表可以看到正在连接的客户端信息，显示在 application_name 列。缺省值为 PostgreSQL JDBC Driver。 connectionExtraInfo：Boolean 类型。表示驱动是否上报当前驱动的部署路径、进程属主用户到数据库。 取值范围：true 或 false，默认值为 false。设置 connectionExtraInfo 为 true，JDBC 驱动会将当前驱动的部署路径、进程属主用户上报到数据库中，记录在 connection_info 参数里；同时可以在 PG_STAT_ACTIVITY 和 PGXC_STAT_ACTIVITY 中查询到。 autosave：String 类型。共有 3 种："always", "never", "conservative"。如果查询失败，指定驱动程序应该执行的操作。在 autosave=always 模式下，JDBC 驱动程序在每次查询之前设置一个保存点，并在失败时回滚到该保存点。在 autosave=never 模式（默认）下，无保存点。在 autosave=conservative 模式下，每次查询都会设置保存点，但是只会在“statement XXX 无效”等情况下回滚并重试。 protocolVersion：Integer 类型。连接协议版本号，目前仅支持 3。注意：设置该参数时将采用 md5 加密方式，需要同步修改数据库的加密方式：gs_guc set -N all -I all -Z coordinator -c "password_encryption_type=1" ，重启实例生效后需要创建用 md5 方式加密口令的用户。同时修改 pg_hba.conf，将客户端连接方式修改为 md5。用新建用户进行登录（因为设置这个值后，只能使用低等级的加密方式（md5）,降低安全性，所以此值不推荐设置）。 **说明** MD5 加密算法安全性低，存在安全风险，建议使用更安全的加密算法。 prepareThreshold：Integer 类型。该值决定着 PreparedStatement 对象在执行多少次以后使用服务端已经准备好的 statement。默认值是 5，意味着在执行同一个 PreparedStatement 对象时，在第五次以及以上执行时不再向服务端发送 parse 消息对 statement 进行解析，而使用之前在服务端已经解析好的 statement。 preparedStatementCacheQueries：Integer 类型。该参数确定了每个连接的 cache 缓存 Statement 对象生成 query 的最大个数。默认值为 256，若 Statement 对象生成 query 个大于 256 则会将最近最少使用的 query 从缓存中丢弃。0 表示禁用缓存。 preparedStatementCacheSizeMiB：Integer 类型，该参数确定了每个连接的 cache 缓存 Statement 对象所生成 query 的最大值（以兆字节为单位），默认情况下是 5。若缓存了超过 5MB 的 query，则最近最少使用的查询缓存将被丢弃。0 表示禁用缓存。 databaseMetadataCacheFields：Integer 类型。默认值是 65536。指定每个连接可缓存的最大字段的个数。“0”表示禁用缓存。

参数	描述
	databaseMetadataCacheFieldsMiB：Integer 类型。默认值是 5。指定每个连接可缓存的字段的最大值，单位是 MB。“0”表示禁用缓存。 stringtype：String 类型，可选字段为："unspecified", "varchar"。设置通过 setString()方法使用的 PreparedStatement 参数的类型，如果 stringtype 设置为 VARCHAR（默认值），则这些参数将作为 varchar 参数发送给服务器。若 stringtype 设置为 unspecified，则参数将作为 untyped 值发送到服务器，服务器将尝试推断适当的类型。 batchMode：Boolean 类型。用于确定是否使用 batch 模式连接。默认值为 on，表示开启 batch 模式。 fetchsize：Integer 类型。用于设置数据库连接所创建 statement 的默认 fetchsize。默认值为 0，表示一次获取所有结果。 reWriteBatchedInserts：Boolean 类型。批量导入时，该参数设置为 on，可将 N 条插入语句合并为一条：insert into TABLE_NAME values(values1, ..., valuesN), ..., (values1, ..., valuesN);使用该参数时，需设置 batchMode=off。 unknownLength：Integer 类型，默认为 Integer.MAX_VALUE。某些 postgresql 类型（例如 TEXT）没有明确定义的长度，当通过 ResultSetMetaData.getColumnDisplaySize 和 ResultSetMetaData.getPrecision 等函数返回关于这些类型的数据时，此参数指定未知长度类型的长度。 defaultRowFetchSize：Integer 类型。确定一次 fetch 在 ResultSet 中读取的行数。限制每次访问数据库时读取的行数可以避免不必要的内存消耗，从而避免 OutOfMemoryException。缺省值是 0，这意味着 ResultSet 中将一次获取所有行。本参数不允许设置为负值。 binaryTransfer：Boolean 类型。使用二进制格式发送和接收数据，默认值为“false”。 binaryTransferEnable：String 类型。启用二进制传输的类型列表，以逗号分隔。OID 编号和名称二选一，例如 binaryTransferEnable=INT4_ARRAY,INT8_ARRAY。 比如：OID 名称为 BLOB，编号为 88，可以如下配置： binaryTransferEnable=BLOB 或 binaryTransferEnable=88 binaryTransferDisEnable：String 类型。禁用二进制传输的类型列表，以逗号分隔。OID 编号和名称二选一。覆盖 binaryTransferEnable 的设置。 blobMode：String 类型。用于设置 setBinaryStream(int, InputStream, int)方法为不同类型的数据赋值，设置为 on 时表示为 blob 类型数据赋值，设置为 off 时表示为 bytea 类型数据赋值，默认为 on;setBinaryStream(int, InputStream, long)与 setBinaryStream(int, InputStream)用于为 bytea 数据类型赋值。 socketFactory：String 类型。用于创建与服务器 socket 连接的类的名称。该类必须实现了接口“javax.net.SocketFactory”，并定义无参或单 String 参数的构造函数。 socketFactoryArg：String 类型。此值是上面提供的 socketFactory 类的构造函数的可选参数，不推荐使用。 receiveBufferSize：Integer 类型。该值用于设置连接流上的

参数	描述
	SO_RCVBUF。 sendBufferSize：Integer 类型。该值用于设置连接流上的 SO_SNDBUF。 preferQueryMode：String 类型。共有 4 种："extended", "extendedForPrepared", "extendedCacheEverything", "simple"。用于指定执行查询的模式，默认值为 extended。simple 模式只发送 Q 消息，仅支持文本模式，不支持 parse 与 bind；extended 模式会使用 parse、bind 和 execute 消息；extendedForPrepared 模式下只有 Prepared Statement 对象使用扩展查询，Statement 对象只使用简单查询；extendedCacheEverything 模式会缓存每个 Statement 对象所生成的 query。 ApplicationType：String 类型。共有 2 种："not_perfect_sharding_type"，"perfect_sharding_type"。用于设置是否开启分布式写入和查询，默认值为 "not_perfect_sharding_type"。not_perfect_sharding_type 模式下开启分布式写入和查询；perfect_sharding_type 模式下默认禁止分布式写入和查询，只有在 sql 文中加入/* multinode */ 才能执行分布式写入和查询。该项设置只有数据库处于 gtm free 场景的情况下才会有效。
user	数据库用户。
password	数据库用户的密码。

表2-11 主备版数据库连接参数

参数	描述
url	postgresql.jar 数据库连接描述符。格式如下： • jdbc:postgresql:database • jdbc:postgresql://host/database • jdbc:postgresql://host:port/database • jdbc:postgresql://host:port/database?param1=value1&param2=value2 • jdbc:postgresql://host1:port1,host2:port2/database?param1=value1&param2=value2 说明 • database 为要连接的数据库名称。 • host 为数据库服务器名称或 IP 地址。 由于安全原因，数据库主节点禁止数据库内部其他节点无认证接入。如果要在数据库内部访问数据库主节点，请将 JDBC 程序部署在数据库主节点所在机器，host 使用"127.0.0.1"。否则可能会出现“FATAL: Forbid remote connection with trust method!”错误。 建议业务系统单独部署在数据库外部，否则可能会影响数据库运行性能。 缺省情况下，连接服务器为 localhost。 • port 为数据库服务器端口。 缺省情况下，会尝试连接到 5432 端口的 database。 • param 为参数名称，即数据库连接属性。

参数	描述
	参数可以配置在 URL 中，以"?"开始配置，以"="给参数赋值，以"&"作为不同参数的间隔。也可以采用 info 对象的属性方式进行配置，详细示例会在本节给出。 • value 为参数值，即数据库连接属性值。 • 连接时需配置 connectTimeout 和 socketTimeout，推荐配置为 2（如果未配置，默认为 0，即不会超时）。在 DN 与客户端出现网络故障时，客户端一直未收到 DN 侧 ACK 确认报文，会启动超时重传机制，不断的进行重传。当超时时间达到系统默认的 600s 后才会报超时错误，这也就会导致 RTO 时间很高。
info	数据库连接属性（所有属性大小写敏感）。常用的属性如下： • PGDBNAME：String 类型。表示数据库名称。（URL 中无需配置该参数，自动从 URL 中解析） • PGHOST：String 类型。主机 IP 地址。详细示例见下。 • PGPORT：Integer 类型。主机端口号。详细示例见下。 • user：String 类型。表示创建连接的数据库用户。 • password：String 类型。表示数据库用户的密码。 • loggerLevel：String 类型。目前支持 3 种级别：OFF、DEBUG、TRACE。设置为 OFF 关闭日志，设置为 DEBUG 和 TRACE 记录的日志信息详细程度不同。 • loggerFile：String 类型。Logger 输出的文件名。需要显示指定日志文件名，若未指定目录则生成在客户端运行程序目录。 • allowEncodingChanges：Boolean 类型。设置该参数值为“true”进行字符集类型更改，配合 characterEncoding=CHARSET 设置字符集，二者使用"&"分隔。 • currentSchema：String 类型。在 search-path 中指定要设置的 schema。 • hostRecheckSeconds：Integer 类型。JDBC 尝试连接主机后会保存主机状态：连接成功或连接失败。在 hostRecheckSeconds 时间内保持可信，超过则状态失效。缺省值是 10 秒。 • ssl：Boolean 类型。以 SSL 方式连接。 ssl=true 可支持 NonValidatingFactory 通道和使用证书的方式： 1、NonValidatingFactory 通道需要配置用户名和密码。 2、配置客户端证书、密钥、根证书，将 SSL 设置为 true。 • sslmode：String 类型。SSL 认证方式。取值范围为：require、verify-ca、verify-full。 – require 只尝试 SSL 连接，如果存在 CA 文件，则应设置成 verify-ca 的方式验证。 – verify-ca 只尝试 SSL 连接，并且验证服务器是否具有由可信任的证书机构签发的证书。 – verify-full 只尝试 SSL 连接，并且验证服务器是否具有由可信任的证书机构签发的证书，以及验证服务器主机名是否与证书中的一致。 • sslcert：String 类型。提供证书文件的完整路径。客户端和服务端证书

参数	描述
	的类型为 End Entity。 • sslkey：String 类型。提供密钥文件的完整路径。使用时将客户端证书转换为 DER 格式： `openssl pkcs8 -topk8 -outform DER -in client.key -out client.key.pk8 -nocrypt` • sslrootcert：String 类型。SSL 根证书的文件名。根证书的类型为 CA。 • sslpassword：String 类型。提供给 ConsoleCallbackHandler 使用。 • sslpasswordcallback：String 类型。SSL 密码提供者的类名。缺省值：org.postgresql.ssl.jdbc4.LibPQFactory.ConsoleCallbackHandler。 • sslfactory：String 类型。提供的值是 SSLSocketFactory 在建立 SSL 连接时用的类名。 • sslfactoryarg：String 类型。此值是上面提供的 sslfactory 类的构造函数的可选参数（不推荐使用）。 • sslhostnameverifier：String 类型。主机名验证程序的类名。接口实现 javax.net.ssl.HostnameVerifier，默认使用 org.postgresql.ssl.PGjdbcHostnameVerifier。 • loginTimeout：Integer 类型。指建立数据库连接的等待时间。超时时间单位为秒。 • connectTimeout：Integer 类型。用于连接服务器操作的超时值。如果连接到服务器花费的时间超过此值，则连接断开。超时时间单位为秒，值为 0 时表示已禁用，timeout 不发生。 • socketTimeout：Integer 类型。用于 socket 读取操作的超时值。如果从服务器读取所花费的时间超过此值，则连接关闭。超时时间单位为秒，值为 0 时表示已禁用，timeout 不发生。 • cancelSignalTimeout：Integer 类型。发送取消消息本身可能会阻塞，此属性控制用于取消命令的“connect 超时”和“socket 超时”。超时时间单位为秒，默认值为 10 秒。 • tcpKeepAlive：Boolean 类型。启用或禁用 TCP 保活探测功能。默认为 false。 • logUnclosedConnections：Boolean 类型。客户端可能由于未调用 Connection 对象的 close()方法而泄漏 Connection 对象。最终这些对象将被垃圾回收，并且调用 finalize()方法。如果调用者自己忽略了此操作，该方法将关闭 Connection。 • assumeMinServerVersion：String 类型。客户端会发送请求进行 float 精度设置。该参数设置要连接的服务器版本，如 assumeMinServerVersion=9.0，可以在建立时减少相关包的发送。 • ApplicationName：String 类型。设置正在使用连接的 JDBC 驱动的名称。通过在数据库主节点上查询 pg_stat_activity 表可以看到正在连接的客户端信息，JDBC 驱动名称显示在 application_name 列。缺省值为 PostgreSQL JDBC Driver。 • connectionExtraInfo：Boolean 类型。表示驱动是否上报当前驱动的部署路径、进程属主用户到数据库。

参数	描述
	取值范围：true 或 false，默认值为 false。设置 connectionExtraInfo 为 true，JDBC 驱动会将当前驱动的部署路径、进程属主用户上报到数据库中，记录在 connection_info 参数里；同时可以在 PG_STAT_ACTIVITY 中查询到。 • autosave：String 类型。共有 3 种："always", "never", "conservative"。如果查询失败，指定驱动程序应该执行的操作。在 autosave=always 模式下，JDBC 驱动程序在每次查询之前设置一个保存点，并在失败时回滚到该保存点。在 autosave=never 模式（默认）下，无保存点。在 autosave=conservative 模式下，每次查询都会设置保存点，但是只会在“statement XXX 无效”等情况下回滚并重试。 • protocolVersion：Integer 类型。连接协议版本号，目前仅支持 3。注意：设置该参数时将采用 md5 加密方式，需要同步修改数据库的加密方式：gs_guc set -N all -I all -c "password_encryption_type=1" ，重启数据库生效后需要创建用 md5 方式加密口令的用户。同时修改 pg_hba.conf，将客户端连接方式修改为 md5。用新建用户进行登录（不推荐）。 说明 MD5 加密算法安全性低，存在安全风险，建议使用更安全的加密算法。 • prepareThreshold：Integer 类型。控制 parse 语句何时发送。默认值是 5。第一次 parse 一个 SQL 比较慢，后面再 parse 就会比较快，因为有缓存了。如果一个会话连续多次执行同一个 SQL，在达到 prepareThreshold 次数以上时，JDBC 将不再对这个 SQL 发送 parse 命令。 • preparedStatementCacheQueries：Integer 类型。确定每个连接中缓存的查询数，默认情况下是 256。若在 prepareStatement()调用中使用超过 256 个不同的查询，则最近最少使用的查询缓存将被丢弃。0 表示禁用缓存。 • preparedStatementCacheSizeMiB：Integer 类型。确定每个连接可缓存的最大值（以兆字节为单位），默认情况下是 5。若缓存了超过 5MB 的查询，则最近最少使用的查询缓存将被丢弃。0 表示禁用缓存。 • databaseMetadataCacheFields：Integer 类型。默认值是 65536。指定每个连接可缓存的最大值。“0”表示禁用缓存。 • databaseMetadataCacheFieldsMiB：Integer 类型。默认值是 5。每个连接可缓存的最大值，单位是 MB。“0”表示禁用缓存。 • stringtype：String 类型，可选字段为：false, "unspecified", "varchar"。设置通过 setString()方法使用的 PreparedStatement 参数的类型，如果 stringtype 设置为 VARCHAR（默认值），则这些参数将作为 varchar 参数发送给服务器。若 stringtype 设置为 unspecified，则参数将作为 untyped 值发送到服务器，服务器将尝试推断适当的类型。 • batchMode：Boolean 类型。用于确定是否使用 batch 模式连接。默认值为 on，表示开启 batch 模式。 • fetchsize：Integer 类型。用于设置数据库连接所创建 statement 的默认 fetchsize。默认值为 0，表示一次获取所有结果。 • reWriteBatchedInserts：Boolean 类型。批量导入时，该参数设置为

参数	描述
	on，可将 N 条插入语句合并为一条：insert into TABLE_NAME values(values1, ..., valuesN), ..., (values1, ..., valuesN);使用该参数时，需设置 batchMode=off。 • unknownLength：Integer 类型，默认为 Integer.MAX_VALUE。某些 postgresql 类型（例如 TEXT）没有明确定义的长度，当通过 ResultSetMetaData.getColumnDisplaySize 和 ResultSetMetaData.getPrecision 等函数返回关于这些类型的数据时，此参数指定未知长度类型的长度。 • defaultRowFetchSize：Integer 类型。确定一次 fetch 在 ResultSet 中读取的行数。限制每次访问数据库时读取的行数可以避免不必要的内存消耗，从而避免 OutOfMemoryException。缺省值是 0，这意味着 ResultSet 中将一次获取所有行。没有负数。 • binaryTransfer：Boolean 类型。使用二进制格式发送和接收数据，默认值为“false”。 • binaryTransferEnable：String 类型。启用二进制传输的类型列表，以逗号分隔。OID 编号和名称二选一，例如 binaryTransferEnable=Integer4_ARRAY,Integer8_ARRAY。 比如：OID 名称为 BLOB，编号为 88，可以如下配置： binaryTransferEnable=BLOB 或 binaryTransferEnable=88 • binaryTransferDisEnable：String 类型。禁用二进制传输的类型列表，以逗号分隔。OID 编号和名称二选一。覆盖 binaryTransferEnable 的设置。 • blobMode：String 类型。用于设置 setBinaryStream 方法为不同类型的数据赋值，设置为 on 时表示为 blob 类型数据赋值，设置为 off 时表示为 bytea 类型数据赋值，默认为 on。 • socketFactory：String 类型。用于创建与服务器 socket 连接的类的名称。该类必须实现了接口“javax.net.SocketFactory”，并定义无参或单 String 参数的构造函数。 • socketFactoryArg：String 类型。此值是上面提供的 socketFactory 类的构造函数的可选参数，不推荐使用。 • receiveBufferSize：Integer 类型。该值用于设置连接流上的 SO_RCVBUF。 • sendBufferSize：Integer 类型。该值用于设置连接流上的 SO_SNDBUF。 • preferQueryMode：String 类型。共有 4 种："extended", "extendedForPrepared", "extendedCacheEverything", "simple"。用于指定执行查询的模式，simple 模式会 excute，不 parse 和 bind；extended 模式会 bind 和 excute；extendedForPrepared 模式为 prepared statement 扩展使用；extendedCacheEverything 模式会缓存每个 statement。 • targetServerType： String 类型。该参数识别主备数据节点是通过查询 URL 连接串中，数据节点是否允许写操作来实现的，默认为"any"。共有四种："any"，"master"，"slave"，"preferSlave"： – master 则尝试连接到 URL 连接串中的主节点，如果找不到就抛出

参数	描述
	异常。 – slave 则尝试连接到 URL 连接串中的备节点，如果找不到就抛出异常。 – preferSlave 则尝试连接到 URL 连接串中的备数据节点（如果有可用的话），否则连接到主数据节点。 – any 则尝试连接 URL 连接串中的任何一个数据节点。
user	数据库用户。
password	数据库用户的密码。

示例

```
//以下代码将获取数据库连接操作封装为一个接口，可通过给定用户名和密码来连接数据库。
public static Connection getConnect(String username, String passwd)
    {
       //驱动类。
       String driver = "org.postgresql.Driver";
       //数据库连接描述符。
       String sourceURL = "jdbc:postgresql://10.10.0.13:8000/postgres";
       Connection conn = null;

       try
       {
          //加载驱动。
          Class.forName(driver);
       }
       catch( Exception e )
       {
          e.printStackTrace();
          return null;
       }

       try
       {
           //创建连接。
          conn = DriverManager.getConnection(sourceURL, username, passwd);
          System.out.println("Connection succeed!");
       }
       catch(Exception e)
       {
          e.printStackTrace();
          return null;
       }

       return conn;
    };
// 以下代码将使用 Properties 对象作为参数建立连接
public static Connection getConnectUseProp(String username, String passwd)
    {
```

```
    //驱动类。
    String driver = "org.postgresql.Driver";
    //数据库连接描述符。
    String sourceURL = "jdbc:postgresql://10.10.0.13:8000/postgres?";
    Connection conn = null;
    Properties info = new Properties();

    try
    {
        //加载驱动。
        Class.forName(driver);
    }
    catch( Exception e )
    {
        e.printStackTrace();
        return null;
    }

    try
    {
         info.setProperty("user", username);
         info.setProperty("password", passwd);
         //创建连接。
         conn = DriverManager.getConnection(sourceURL, info);
         System.out.println("Connection succeed!");
    }
    catch(Exception e)
    {
        e.printStackTrace();
        return null;
    }

    return conn;
};
```

2.5.3 使用 ODBC 连接数据库

ODBC（Open Database Connectivity，开放数据库互连）是由 Microsoft 公司基于 X/OPEN CLI 提出的用于访问数据库的应用程序编程接口。应用程序通过 ODBC 提供的 API 与数据库进行交互，在避免了应用程序直接操作数据库系统的同时，增强了应用程序的可移植性、扩展性和可维护性。

云数据库 GaussDB 目前在以下环境中提供对 ODBC3.5 的支持。

表2-12 ODBC 支持平台

操作系统	平台
EulerOS 2.5	x86_64 位
EulerOS 2.8	ARM64 位
Windows 7	x86_32 位
Windows 7	x86_64 位

操作系统	平台
Windows Server 2008	x86_32 位
Windows Server 2008	x86_64 位

UNIX/Linux 系统下的驱动程序管理器主要有 unixODBC 和 iODBC，在这选择驱动管理器 unixODBC-2.3.0 作为连接数据库的组件。

Windows 系统自带 ODBC 驱动程序管理器，在控制面板->管理工具中可以找到数据源（ODBC）选项。

说明

当前数据库 ODBC 驱动基于开源版本，对于自研的数据类型，tinyint、smalldatetime、nvarchar2 在获取数据类型的时候，可能会出现不兼容。

前提条件

- 已下载 Linux 版本的 ODBC 驱动包和 Windows 版本的 ODBC 驱动包，

  Linux 环境下，开发应用程序要用到 unixODBC 提供的头文件（sql.h、sqlext.h 等）和库 libodbc.so。这些头文件和库可从 unixODBC-2.3.0 的安装包中获得。
- 已下载开源 unixODBC 代码文件，支持版本为 2.3.0，下载地址：https://sourceforge.net/projects/unixodbc/files/unixODBC/2.3.0/unixODBC-2.3.0.tar.gz/download
- 将提供的 ODBC DRIVER（psqlodbcw.so）配置到数据源中便可使用。配置数据源需要配置“odbc.ini”和“odbcinst.ini”两个文件（在编译安装 unixODBC 过程中生成且默认放在“/usr/local/etc”目录下），并在服务器端进行配置。

在 Linux 下使用 ODBC 连接数据库

步骤 1 安装 unixODBC。如果机器上已经安装了其他版本的 unixODBC，可以直接覆盖安装。

目前不支持 unixODBC-2.2.1 版本。以 unixODBC-2.3.0 版本为例，在客户端执行如下命令安装 unixODBC。默认安装到“/usr/local”目录下，生成数据源文件到“/usr/local/etc”目录下，库文件生成在“/usr/local/lib”目录。

```
tar zxvf unixODBC-2.3.0.tar.gz
cd unixODBC-2.3.0
#修改 configure 文件，找到 LIB_VERSION
#将它的值修改为"1:0:0"，这样将编译出*.so.1 的动态库，与 psqlodbcw.so 的依赖关系相同。
vim configure

./configure --enable-gui=no
make
#安装可能需要 root 权限
make install
```

步骤 2 替换客户端云数据库 GaussDB 驱动程序。

将 GaussDB-Kernel-VxxxRxxxCxx-EULER-64bit-Odbc.tar.gz 解压到“/usr/local/lib”目录下。解压会得到“psqlodbcw.la”和“psqlodbcw.so”两个文件。

步骤 3 配置数据源。

1. 配置 ODBC 驱动文件。

   在“/usr/local/etc/odbcinst.ini”文件中追加以下内容。

```
[GaussMPP]
Driver64=/usr/local/lib/psqlodbcw.so
setup=/usr/local/lib/psqlodbcw.so
```

   odbcinst.ini 文件中的配置参数说明如表 2-13 所示。

表2-13 odbcinst.ini 文件配置参数

参数	描述	示例
[DriverName]	驱动器名称，对应数据源 DSN 中的驱动名。	[DRIVER_N]
Driver64	驱动动态库的路径。	Driver64=/xxx/odbc/lib/psqlodbcw.so
setup	驱动安装路径，与 Driver64 中动态库的路径一致。	setup=/xxx/odbc/lib/psqlodbcw.so

2. 配置数据源文件。

   在“/usr/local/etc/odbc.ini ”文件中追加以下内容。

```
[gaussdb]
Driver=GaussMPP
Servername=10.10.0.13（数据库 Server IP）
Database=postgres （数据库名）
Username=omm （数据库用户名）
Password= （数据库用户密码）
Port=8000 （数据库侦听端口）
Sslmode=allow
```

   odbc.ini 文件配置参数说明如表 2-14 所示。

表2-14 odbc.ini 文件配置参数

参数	描述	示例
[DSN]	数据源的名称。	[gaussdb]
Driver	驱动名，对应 odbcinst.ini 中的 DriverName。	Driver=DRIVER_N
Servername	服务器的 IP 地址。	Servername=10.145.130.26
Database	要连接的数据库的名称。	Database=postgres
Username	数据库用户名称。	Username=omm
Password	数据库用户密码。	Password= 说明

参数	描述	示例
		ODBC 驱动本身已经对内存密码进行过清理，以保证用户密码在连接后不会再在内存中保留。 但是如果配置了此参数，由于 UnixODBC 对数据源文件等进行缓存，可能导致密码长期保留在内存中。 推荐在应用程序连接时，将密码传递给相应 API，而非写在数据源配置文件中。同时连接成功后，应当及时清理保存密码的内存段。
Port	服务器的端口号。	Port=8000
Sslmode	开启 SSL 模式	Sslmode=allow
UseServerSidePrepare	是否开启数据库端扩展查询协议。 可选值 0 或 1，默认为 1，表示打开扩展查询协议。	UseServerSidePrepare=1
UseBatchProtocol	是否开启批量查询协议（打开可提高 DML 性能）；可选值 0 或者 1，默认为 1。 当此值为 0 时，不使用批量查询协议（主要用于与早期数据库版本通信兼容）。 当此值为 1，并且数据库 support_batch_bind 参数存在且为 on 时，将打开批量查询协议。	UseBatchProtocol=1
ConnectionExtraInfo	GUC 参数 connection_info 中显示驱动部署路径和进程属主用户的开关。	ConnectionExtraInfo=1 **说明** 默认值为 0。当设置为 1 时，ODBC 驱动会将当前驱动的部署路径、进程属主用户上报到数据库中，记录在 connection_info 参数中。

其中关于 Sslmode 的选项的允许值，具体信息见下表：

表2-15 sslmode 的可选项及其描述

sslmode	是否会启用 SSL 加密	描述

sslmode	是否会启用SSL 加密	描述
disable	否	不使用 SSL 安全连接。
allow	可能	如果数据库服务器要求使用，则可以使用 SSL 安全加密连接，但不验证数据库服务器的真实性。
prefer	可能	如果数据库支持，那么首选使用 SSL 安全加密连接，但不验证数据库服务器的真实性。
require	是	必须使用 SSL 安全连接，但是只做了数据加密，而并不验证数据库服务器的真实性。
verify-ca	是	必须使用 SSL 安全连接，并且验证数据库是否具有可信证书机构签发的证书。
verify-full	是	必须使用 SSL 安全连接，在 verify-ca 的验证范围之外，同时验证数据库所在主机的主机名是否与证书内容一致。如果不一致，需要使用 root 用户修改/etc/hosts 文件，将连接的数据库节点的 IP 地址和主机名加入。

步骤 4 SSL 模式。

声明如下环境变量，同时保证 client.key*系列文件为 600 权限：

```
export PGSSLCERT="/YOUR/PATH/OF/client.crt" #请修改该路径到 client.crt 的绝对路径
export PGSSLKEY="/YOUR/PATH/OF/client.key"  #请修改该路径到 client.key 的绝对路径
将根证书 cacert.pem 文件放至客户端用户 home 目录下的.postgresql 目录下（如果没有请创建该目录），
并将 cacert.pem 重命名为 root.crt，文件权限设置为 600。
```

同时将数据源中的 Sslmode 选项调整至“require”。

步骤 5 配置环境变量。

```
vim ~/.bashrc
```

在配置文件中追加以下内容。

```
export LD LIBRARY PATH=/usr/local/lib/:$LD LIBRARY PATH
export ODBCSYSINI=/usr/local/etc
export ODBCINI=/usr/local/etc/odbc.ini
```

步骤 6 执行如下命令使设置生效。

```
source ~/.bashrc
```

步骤 7 执行以下命令，开始连接数据库。

isql -v *GaussODBC*

GaussODBC 为数据源名称

- 如果显示如下信息，表明配置正确，连接成功。

```
+---------------------------------------+
| Connected!                            |
```

```
|                                    |
| sql-statement                        |
| help [tablename]                      |
| quit                                |
|                                    |
+------------------------------------+
SQL>
```

- 若显示 ERROR 信息，则表明配置错误。请检查上述配置是否正确。

----结束

在 Windows 下使用 ODBC 连接数据库

Windows 操作系统自带 ODBC 数据源管理器，无需用户手动安装管理器便可直接进行配置。

步骤 1 替换客户端云数据库 GaussDB 驱动程序

将 GaussDB-Kernel-V500R001C20-Windows-Odbc-X86.tar.gz 解压后，根据需要，点击 psqlodbc.msi（32 位）或者 psqlodbc_x64.msi（64 位）进行驱动安装。

步骤 2 打开驱动管理器。

在配置数据源时，请使用对应的驱动管理器（假设操作系统安装盘符为 C 盘，如果是其他盘符，请对路径做相应修改）：

- **64** 位操作系统上进行 **32** 位程序开发，安装 **32** 位驱动程序后，使用 **32** 位的驱动管理器：C:\Windows**SysWOW64**\odbcad32.exe

  请勿直接使用“控制面板> 管理工具 > 数据源(ODBC)”。

  说明

  WoW64 的全称是"Windows 32-bit on Windows 64-bit"，C:\Windows\SysWOW64\存放的是 64 位系统上的 32 位运行环境。而 C:\Windows\System32\存放的是与操作系统一致的运行环境，具体的技术信息请查阅 Windows 的相关技术文档。

- **64** 位操作系统上进行 **64** 位程序开发，安装 **64** 位驱动程序后，使用 **64** 位的驱动管理器：C:\Windows**System32**\odbcad32.exe

  请勿直接使用“控制面板 > 管理工具 > 数据源(ODBC)”。

- 32 位操作系统请使用：C:\Windows\System32\odbcad32.exe

  或者点击“计算机 > 控制面板 > 管理工具 > 数据源(ODBC)”打开驱动管理器。

步骤 3 配置数据源。

在打开的驱动管理器上，选择“用户 DSN > 添加 > PostgreSQL Unicode”（如果是 64 位驱动，将会有 64 位标识），然后进行配置：

须知

此界面上配置的用户名及密码信息，将会被记录在 Windows 注册表中，再次连接数据库时就不再需要输入认证信息。但是出于安全考虑，建议在单击"Save"按钮保存配置信息前，清空相关敏感信息；在使用 ODBC 的连接 API 时，再传入所需的用户名、密码信息。

步骤 4 SSL 模式。

将 client.crt、client.key、client.key.cipher、client.key.rand 文件放至%APPDATA%\postgresql(该目录需手动建立)目录下，并且将文件名中的 client 改为 postgres，例如 client.key 修改为 postgres.key；将 cacert.pem 文件放至%APPDATA%\postgresql 目录，并更名为 root.crt。

说明

%APPDATA% 该值在安装时由客户指定，默认位置在 C:\Users\[username]\AppData。

同时将步骤 2 中的设置窗口的“SSL Mode”选项调整至“require”。

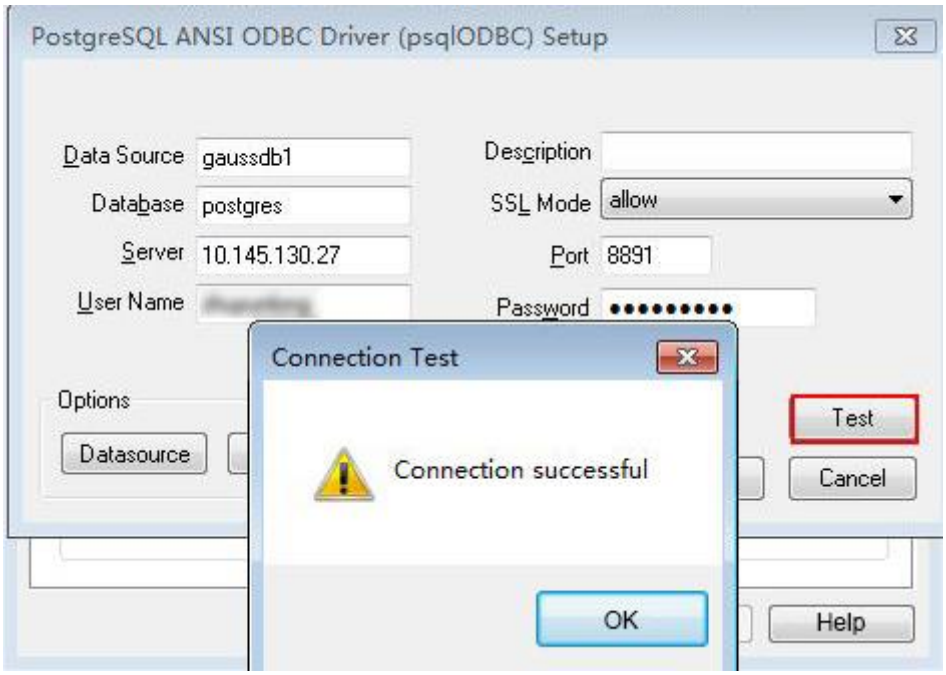
表2-16 sslmode 的可选项及其描述

sslmode	是否会启用 SSL 加密	描述
disable	否	不使用 SSL 安全连接。
allow	可能	如果数据库服务器要求使用，则可以使用 SSL 安全加密连接，但不验证数据库服务器的真实性。
prefer	可能	如果数据库支持，那么首选使用 SSL 安全加密连接，但不验证数据库服务器的真实性。
require	是	必须使用 SSL 安全连接，但是只做了数据加密，而并不验证数据库服务器的真实性。

sslmode	是否会启用SSL 加密	描述
verify-ca	是	必须使用 SSL 安全连接，并且验证数据库是否具有可信证书机构签发的证书。当前 windows ODBC 不支持 cert 方式认证。
verify-full	是	必须使用 SSL 安全连接，在 verify-ca 的验证范围之外，同时验证数据库所在主机的主机名是否与证书内容一致。当前 windows odbc 不支持 cert 方式认证。

步骤 5 点击 Test 进行测试连接。

- 如果显示如下，则表明配置正确，连接成功。



- 若显示 ERROR 信息，则表明配置错误。请检查上述配置是否正确。

----结束

2.5.4 使用 libpq 连接数据库

云数据库 GaussDB 主备版部署形态未对此接口在应用程序开发场景下的使用做验证。因此对使用此接口做应用程序开发存在的风险未知，故不推荐用户使用此套接口做应用程序开发。推荐用户使用 ODBC 或 JDBC 接口来替代。

前提条件

编译并且链接一个 libpq 的源程序，需要做下面的一些事情：

- 获取 libpq 驱动，

- 解压 GaussDB-Kernel-VxxxRxxxCxx-EULER-64bit-Libpq.tar.gz 文件，其中 include 文件夹下的头文件为所需的头文件，lib 文件夹中为所需的 libpq 库文件。

说明

除 libpq-fe.h 外，include 文件夹下默认还存在头文件 postgres_ext.h，gs_thread.h，gs_threadlocal.h，这三个头文件是 libpq-fe.h 的依赖文件。

- 包含 libpq-fe.h 头文件：

```
#include <libpq-fe.h>
```

- 通过-I *directory* 选项，提供头文件的安装位置（有些时候编译器会查找缺省的目录，因此可以忽略这些选项）。如：

```
gcc -I （头文件所在目录） -L （libpq 库所在目录） testprog.c -lpq
```

- 如果要使用制作文件(makefile)，向 CPPFLAGS、LDFLAGS、LIBS 变量中增加如下选项：

```
CPPFLAGS += -I （头文件所在目录）
LDFLAGS += -L （libpq 库所在目录）
LIBS += -lpq
```

常用功能示例代码

此处以示例来说明连接方式：

示例 1：

```
/*
 * testlibpq.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

static void
exit nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,j;

    /*
     * 用户在命令行上提供了 conninfo 字符串的值时使用该值；
     * 否则环境变量或者所有其它连接参数
     * 都使用缺省值。
     */
    if (argc > 1)
        conninfo = argv[1];
```

```
    else
        conninfo = "dbname=postgres port=42121 host='10.44.133.171'
application_name=test connect_timeout=5 sslmode=allow user='test' password='xxxx'";

    /* 连接数据库 */
    conn = PQconnectdb(conninfo);

    /* 检查后端连接成功建立 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * 测试实例涉及游标的使用时候必须使用事务块。
     *把全部放在一个  "select * from pg_database"
     * PQexec() 里，过于简单，不推荐使用。
     */

    /* 开始一个事务块 */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * 在结果不需要的时候 PQclear PGresult，以避免内存泄漏
     */
    PQclear(res);

    /*
     * 从系统表 pg_database（数据库的系统目录）里抓取数据
     */
    res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in myportal");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
```

```
    /* 打印属性名称 */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* 打印行 */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* 关闭入口 ... 不用检查错误 ... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* 结束事务 */
    res = PQexec(conn, "END");
    PQclear(res);

    /* 关闭数据库连接并清理 */
    PQfinish(conn);

    return 0;
}
```

示例 2：

```
/*
 * testlibpq2.c
 *     测试外联参数和二进制 I/O。
 *
 * 在运行这个例子之前，用下面的命令填充一个数据库
 *
 *
 * CREATE TABLE test1 (i int4, t text);
 *
  * INSERT INTO test1 values (2, 'ho there');
 *
 * 期望的输出是：
 *
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <libpq-fe.h>
```

```
/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * 这个函数打印查询结果，这些结果是二进制格式，从上面的
 * 注释里面创建的表中抓取出来的。
 */
static void
show_binary_results(PGresult *res)
{
    int         i;
    int         i_fnum,
                t_fnum;

    /* 使用 PQfnumber 来避免对结果中的字段顺序进行假设 */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");

    for (i = 0; i < PQntuples(res); i++)
    {
        char      *iptr;
        char      *tptr;
        int        ival;

        /* 获取字段值（忽略可能为空的可能） */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);

        /*
         * INT4 的二进制表现形式是网络字节序，
         * 建议转换成本地字节序。
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * TEXT 的二进制表现形式是文本，因此 libpq 能够给它附加一个字节零，
         * 把它看做 C 字符串。
         *
         */

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf("\n\n");
```

```
    }
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * 如果用户在命令行上提供了参数，
     * 那么使用该值为 conninfo 字符串；否则
     * 使用环境变量或者缺省值。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname=postgres port=42121 host='10.44.133.171'
application name=test connect timeout=5 sslmode=allow user='test' password='xxxx'";

    /* 和数据库建立连接 */
    conn = PQconnectdb(conninfo);

    /* 检查与服务器的连接是否成功建立 */
    if (PQstatus(conn) != CONNECTION OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit nicely(conn);
    }

    /* 把整数值 "2" 转换成网络字节序 */
    binaryIntVal = htonl((uint32 t) 2);

    /* 为 PQexecParams 设置参数数组 */
    paramValues[0] = (char *) &binaryIntVal;
    paramLengths[0] = sizeof(binaryIntVal);
    paramFormats[0] = 1;        /* 二进制 */

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE i = $1::int4",
                       1,       /* 一个参数 */
                       NULL,    /* 让后端推导参数类型 */
                       paramValues,
                       paramLengths,
                       paramFormats,
                       1);      /* 要求二进制结果 */

    if (PQresultStatus(res) != PGRES TUPLES OK)
    {
```

```
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /* 关闭与数据库的连接并清理 */
    PQfinish(conn);

    return 0;
}
```

2.5.5 使用 PyGreSQL 连接数据库

PyGreSQL 是一款开源的 PostgreSQL 的 Python 驱动接口。它支持许多数据库原生功能，是主流的 PostgreSQL 的 Python 客户端驱动之一。目前仅分布式支持该驱动接口。

软件环境要求

- 云数据库 GaussDB 已经安装完成。
- 安装了 PyGreSQL 的 Python 环境。

说明

GaussDB 并不提供，也不维护 PyGreSQL 的源码及发布包，如有使用问题，请参考 PyGreSQL 的官方网站。

云数据库 GaussDB 推荐使用 Python3.5.3 及 PyGreSQL5.0.3。

- 数据库的连接及 PyGreSQL 的具体使用方式，请参见 PyGreSQL 的官方网站。

2.5.6 使用 Psycopg 连接数据库

Psycopg 是一种用于执行 SQL 语句的 PythonAPI，可以为 PostgreSQL、云数据库 GaussDB 数据库提供统一访问接口，应用程序可基于它进行数据操作。Psycopg2 是对 libpq 的封装，主要使用 C 语言实现，既高效又安全。它具有客户端游标和服务器端游标、异步通信和通知、支持“COPY TO/COPY FROM”功能。支持多种类型 Python 开箱即用，适配 PostgreSQL 数据类型；通过灵活的对象适配系统，可以扩展和定制适配。Psycopg2 兼容 Unicode 和 Python 3。

云数据库 GaussDB 数据库提供了对 Psycopg2 特性的支持，并且支持 psycopg2 通过 SSL 模式链接。

表2-17 Psycopg 支持平台

操作系统	平台
EulerOS 2.5	x86_64 位
EulerOS 2.8	ARM64 位

前提条件

- 获取 Python 驱动包，

  解压后有两个文件夹：

  - psycopg2：psycopg2 库文件。
  - lib：lib 库文件。
- 在使用驱动之前，需要做如下操作：

  a. 先解压版本对应驱动包，使用 root 用户将 psycopg2 拷贝到 python 安装目录下的 site-packages 文件夹下。

  b. 修改 psycopg2 目录权限为 755。

  c. 将 psycopg2 目录添加到环境变量$PYTHONPATH，并使之生效。

  d. 对于非数据库用户，需要将解压后的 lib 目录，配置在 LD_LIBRARY_PATH 中。
- 在创建数据库连接之前，需要先加载如下数据库驱动程序：

```
import  psycopg2
```

连接数据库

步骤 1 使用*.ini 文件（python 的 configparser 包可以解析这种类型的配置文件）保存数据库连接的配置信息。

步骤 2 使用 psycopg2.connect 函数获得 connection 对象。

步骤 3 使用 connection 对象创建 cursor 对象。

----结束

3 用户指南

3.1 登录管理控制台

操作步骤

步骤 1 登录管理控制台。

步骤 2 单击管理控制台左上角的，选择区域和项目。

步骤 3 在页面左上角单击，选择“数据库 > 云数据库 云数据库 GaussDB ”。进入云数据库 云数据库 GaussDB 信息页面。

步骤 4 在左侧导航栏选择云数据库 GaussDB > 实例管理。

进入云数据库 GaussDB 信息页面。

----结束

3.2 性能调优

3.2.1 总体调优思路

云数据库 GaussDB 的总体性能调优思路为性能瓶颈点分析、关键参数调整以及 SQL 调优。在调优过程中，通过系统资源、吞吐量、负载等因素来帮助定位和分析性能问题，使系统性能达到可接受的范围。

云数据库 GaussDB 性能调优过程需要综合考虑多方面因素，因此，调优人员应对系统软件架构、软硬件配置、数据库配置参数、并发控制、查询处理和数据库应用有广泛而深刻的理解。

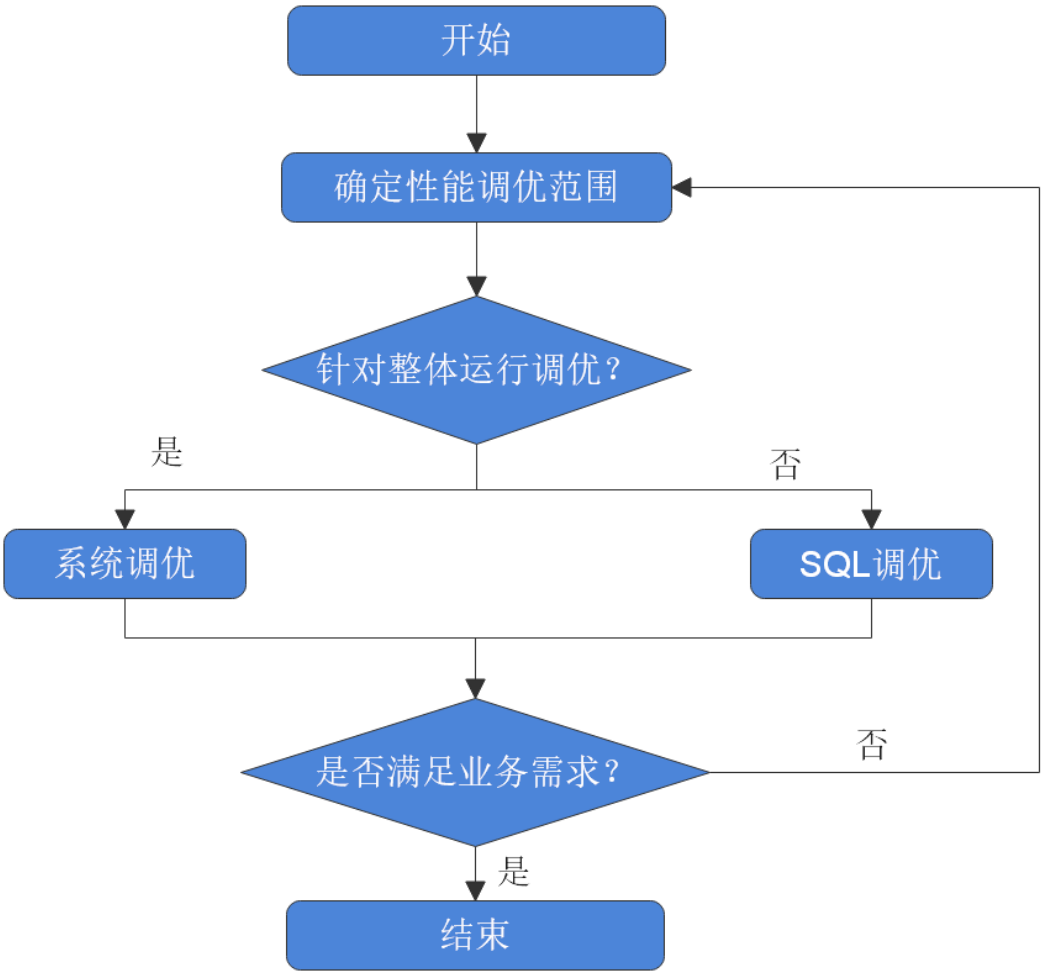
须知

性能调优过程有时候需要重启集群，可能会中断当前业务。因此，业务上线后，当性能调优操作需要重启集群时，操作窗口时间需向管理部门提出申请，经批准后方可执行。

调优流程

调优流程如图 3-1 所示。

图3-1 云数据库 GaussDB 性能调优流程



3.2.2 确定性能调优范围

数据库性能调优通常发生在用户对业务的执行效率不满意，期望通过调优加快业务执行的情况下。正如“3.2.2 确定性能调优范围”小节所述，数据库性能受影响因素多，从而性能调优是一项复杂的工程，有些时候无法系统性地说明和解释，而是依赖于 DBA 的经验判断。尽管如此，此处还是期望能尽量系统性的对性能调优方法加以说明，方便应用开发人员和刚接触云数据库 GaussDB 的 DBA 参考。

性能因素

多个性能因素会影响数据库性能，了解这些因素可以帮助定位和分析性能问题。

- 系统资源

  数据库性能在很大程度上依赖于磁盘的 I/O 和内存使用情况。为了准确设置性能指标，用户需要了解集群部署硬件的基本性能。CPU，硬盘，磁盘控制器，内存和网络接口等这些硬件性能将显著影响数据库的运行速度。

- 负载

  负载等于数据库系统的需求总量，它会随着时间变化。总体负载包含用户查询，应用程序，并行作业，事务以及数据库随时传递的系统命令。比如：多用户在执行多个查询时会提高负载。负载会显著地影响数据库的性能。了解工作负载高峰期可以帮助用户更合理地利用系统资源，更有效地完成系统任务。

- 吞吐量

  使用系统的吞吐量来定义处理数据的整体能力。数据库的吞吐量以每秒的查询次数、每秒的处理事务数量或平均响应时间来测量。数据库的处理能力与底层系统（磁盘 I/O，CPU 速度，存储器带宽等）有密切的关系，所以当设置数据库吞吐量目标时，需要提前了解硬件的性能。

- 竞争

  竞争是指两组或多组负载组件尝试使用冲突的方式使用系统的情况。比如，多条查询视图同一时间更新相同的数据，或者多个大量的负载争夺系统资源。随着竞争的增加，吞吐量下降。

- 优化

  数据库优化可以影响到整个系统的性能。在执行 SQL 制定、数据库配置参数、表设计、数据分布等操作时，启用数据库查询优化器打造最有效的执行计划。

调优范围确定

性能调优主要通过查看集群各节点的 CPU、内存、I/O 和网络这些硬件资源的使用情况，确认这些资源是否已被充分利用，是否存在瓶颈点，然后针对性调优。

- 如果某个资源已达瓶颈，则：
  a. 通过查询最耗时的 SQL 语句、跑不出来的 SQL 语句，找出耗资源的 SQL，进行 3.2.3 SQL 调优指南。
- 如果所有资源均未达瓶颈，则表明性能仍有提升潜力。可以查询最耗时的 SQL 语句，或者跑不出来的 SQL 语句，进行针对性的 3.2.3 SQL 调优指南。

3.2.2.1 查询最耗性能的 SQL

系统中有些 SQL 语句运行了很长时间还没有结束，这些语句会消耗很多的系统性能，请根据本章内容查询长时间运行的 SQL 语句。

操作步骤

步骤 1 使用 DAS 或者 gsql 连接实例。

步骤 2 查询系统中长时间运行的查询语句。

```
SELECT current_timestamp - query_start AS runtime, datname, usename, query FROM
pg_stat_activity where state != 'idle' ORDER BY 1 desc;
```

查询后会按执行时间从长到短顺序返回查询语句列表，第一条结果就是当前系统中执行时间最长的查询语句。返回结果中包含了系统调用的 SQL 语句和用户执行 SQL 语句，请根据实际找到用户执行时间长的语句。

若当前系统较为繁忙，可以通过限制 current_timestamp - query_start 大于某一阈值来查看执行时间超过此阈值的查询语句。

```
SELECT query FROM pg stat activity WHERE current timestamp - query start > interval
'1 days';
```

步骤 3 设置参数 track_activities 为 on。

```
SET track_activities = on;
```

当此参数为 on 时，数据库系统才会收集当前活动查询的运行信息。

步骤 4 查看正在运行的查询语句。

以查看视图 pg_stat_activity 为例：

```
SELECT datname, usename, state FROM pg stat activity;
 datname  | usename | state  |
----------+---------+--------+
 postgres |   omm   | idle   |
 postgres |   omm   | active |
(2 rows)
```

如果 state 字段显示为 idle，则表明此连接处于空闲，等待用户输入命令。

如果仅需要查看非空闲的查询语句，则使用如下命令查看：

```
SELECT datname, usename, state FROM pg_stat_activity WHERE state != 'idle';
```

步骤 5 分析长时间运行的查询语句状态。

- 若查询语句处于正常状态，则等待其执行完毕。
- 若查询语句阻塞，则通过如下命令查看当前处于阻塞状态的查询语句：

  ```
  SELECT datname, usename, state, query FROM pg stat activity WHERE waiting =
  true;
  ```

  查询结果中包含了当前被阻塞的查询语句，该查询语句所请求的锁资源可能被其他会话持有，正在等待持有会话释放锁资源。

只有当查询阻塞在系统内部锁资源时，waiting 字段才显示为 true。尽管等待锁资源是数据库系统最常见的阻塞行为，但是在某些场景下查询也会阻塞在等待其他系统资源上，例如写文件、定时器等。但是这种情况的查询阻塞，不会在视图 pg_stat_activity 中体现。

----结束

3.2.2.2 分析作业是否被阻塞

数据库系统运行时，在某些业务场景下查询语句会被阻塞，导致语句运行时间过长，可以强制结束有问题的会话。

操作步骤

步骤 1 使用 DAS 或者 gsql 连接实例。

步骤 2 查看阻塞的查询语句及阻塞查询的表、模式信息。

```
SELECT w.query as waiting query,
w.pid as w pid,
w.usename as w user,
l.query as locking query,
l.pid as l pid,
l.usename as l user,
t.schemaname || '.' || t.relname as tablename
from pg stat activity w join pg locks l1 on w.pid = l1.pid
and not l1.granted join pg locks l2 on l1.relation = l2.relation
and l2.granted join pg stat activity l on l2.pid = l.pid join pg stat user tables t
on l1.relation = t.relid
where w.waiting;
```

该查询返回线程 ID、用户信息、查询状态，以及导致阻塞的表、模式信息。

步骤 3 使用如下命令结束相应的会话。

```
SELECT PG_TERMINATE_BACKEND(139834762094352);
```

其中，139834762094352 为线程 ID。

显示类似如下信息，表示结束会话成功。

```
 PG_TERMINATE_BACKEND
----------------------
 t
(1 row)
```

显示类似如下信息，表示用户正在尝试结束当前会话。

```
FATAL:  terminating connection due to administrator command
FATAL:  terminating connection due to administrator command
```

说明

- gsql 客户端使用 PG_TERMINATE_BACKEND 函数结束当前正在执行会话的后台线程时，如果当前的用户是初始用户，客户端不会退出而是自动重连，即还会返回“The connection to the server was lost. Attempting reset: Succeeded.”；否则客户端会重连失败，即返回“The connection to the server was lost. Attempting reset: Failed.”。这是因为只有初始用户可以免密登录，普遍用户不能免密登录，从而重连失败。
- 对于使用 PG_TERMINATE_BACKEND 函数结束非活跃的后台线程时。如果打开了线程池，此时空闲的会话没有线程 ID，无法结束会话。非线程池模式下，结束的会话不会自动重连。

----结束

3.2.2.3 参数调优建议

数据库参数是数据库系统运行的关键配置信息，设置不合适的参数值可能会影响业务。本文列举了一些重要参数说明，更多参数详细说明，请参考 3.5.3 导出参数，将参数导出后查看。

通过控制台界面修改参数值，请参见 3.4.9 查看和修改实例参数。

查询

- **track_stmt_session_slot**

  作用：设置一个 session 缓存的最大的全量/慢 SQL 的数量。

  影响：缓存的 SQL 定期会被写入到系统表，如果业务量很大，超过这个数量语句执行将不会被跟踪，直到落盘线程将缓存语句落盘，留出空闲的空间，但不影响 SQL 的执行。

- **effective_cache_size**

  作用：设置节点优化器在一次单一的查询中可用的磁盘缓冲区的有效大小。设置这个参数，还要考虑的共享缓冲区以及内核的磁盘缓冲区。另外，还要考虑预计的在不同表之间的并发查询数目，因为它们将共享可用的空间。这个参数对分配的共享内存大小没有影响，它也不会使用内核磁盘缓冲，它只用于估算。数值是用磁盘页来计算的，通常每个页面是 8192 字节。

  取值范围：整型，1～INT_MAX，单位为 8KB。

  影响：比默认值高的数值可能会导致使用索引扫描，更低的数值可能会导致选择顺序扫描。

- **enable_stream_operator**

  控制优化器对 stream 的使用。当该参数关闭时，可能会有大量关于计划不能下推的日志记录到日志文件中。

- **log_min_duration_statement**

  作用：当某条语句的持续时间大于或者等于特定的毫秒数时，记录每条完成语句的持续时间。设置 log_min_duration_statement 可以很方便地跟踪需要优化的查询语句。对于使用扩展查询协议的客户端，语法分析、绑定、执行每一步所花时间被独立记录。

  影响：设置过低的阈值可能影响负载吞吐，-1 表示关闭此功能。

审计参数

- **audit_system_object**

  作用：该参数决定是否对数据库对象的 CREATE、DROP、ALTER 操作进行审计。数据库对象包括 DATABASE、USER、schema、TABLE 等。通过修改该配置参数的值，可以只审计需要的数据库对象的操作,在主备强制选主场景建议。

  影响：不当修改该参数会导致丢失 DDL 审计日志，请在客服人员指导下进行修改。

锁管理

- **update_lockwait_timeout**

  设置并发更新同一行数据时单个锁的最长等待时间，当申请的锁等待时间超过设定值时系统会报错。0 表示不会超时，默认值为 2min。

连接与认证

- **session_timeout**

  表明与服务器建立连接后，不进行任何操作一定时间后超时的限制，0 表示关闭超时设置。

- **failed_login_attempts**

设置密码错误次数上限，输入密码错误的次数达到该参数所设置的值时，账户将会被自动锁定，配置为 0 时表示不限制密码输入错误的次数。

- **password_effect_time**

  设置帐户密码的有效时间，0 表示不开启有效期限制功能。

- **password_lock_time**

  设置账户被锁定后的自动解锁时间，单位为天。

3.2.3 SQL 调优指南

SQL 调优的唯一目的是“资源利用最大化”，即 CPU、内存、磁盘 IO、网络 IO 四种资源利用最大化。所有调优手段都是围绕资源使用开展的。所谓资源利用最大化是指 SQL 语句尽量高效，节省资源开销，以最小的代价实现最大的效益。比如做典型点查询的时候，可以用 seqscan+filter(即读取每一条元组和点查询条件进行匹配)实现，也可以通过 indexscan 实现，显然 indexscan 可以以更小的代价实现相同的效果。根据硬件资源和客户的业务特征确定合理的集群部署方案和表定义是数据库在多数情况下满足性能要求的基础。

3.2.3.1 Query 执行流程

SQL 引擎从接受 SQL 语句到执行 SQL 语句需要经历的步骤如图 3-2 和表 3-1 所示。其中，红色字体部分为 DBA 可以介入实施调优的环节。

图3-2 SQL 引擎执行查询类 SQL 语句的流程

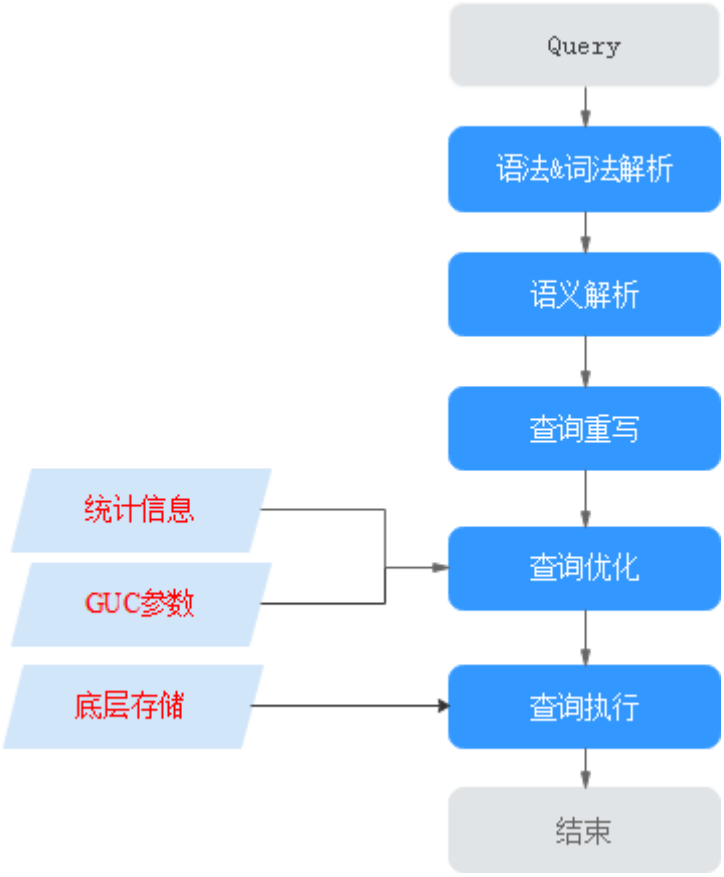


表3-1 SQL 引擎执行查询类 SQL 语句的步骤说明

步骤	说明
语法&词法解析	按照约定的 SQL 语句规则，把输入的 SQL 语句从字符串转化为格式化结构(Stmt)。
语义解析	将“语法&词法解析”输出的格式化结构转化为数据库可以识别的对象。
查询重写	根据规则把“语义解析”的输出等价转化为执行上更为优化的结构。
查询优化	根据“查询重写”的输出和数据库内部的统计信息规划 SQL 语句具体的执行方式，也就是执行计划。统计信息和 GUC 参数对查询优化（执行计划）的影响，请参见调优手段之统计信息和调优手段之 GUC 参数。
查询执行	根据“查询优化”规划的执行路径执行 SQL 查询语句。底层存储方式的选择合理性，将影响查询执行效率。详见调优手段之底层存储。

调优手段之统计信息

云数据库 GaussDB 优化器是典型的基于代价的优化 (Cost-Based Optimization，简称 CBO)。在这种优化器模型下，数据库根据表的元组数、字段宽度、NULL 记录比率、distinct 值、MCV 值、HB 值等表的特征值，以及一定的代价计算模型，计算出每一个执行步骤的不同执行方式的输出元组数和执行代价(cost)，进而选出整体执行代价最小/首元组返回代价最小的执行方式进行执行。这些特征值就是统计信息。从上面描述可以看出统计信息是查询优化的核心输入，准确的统计信息将帮助规划器选择最合适的查询规划，一般来说我们通过 analyze 语法收集整个表或者表的若干个字段的统计信息，周期性地运行 ANALYZE，或者在对表的大部分内容做了更改之后马上运行它是个好习惯。

调优手段之 GUC 参数

查询优化的主要目的是为查询语句选择高效的执行方式。

如下 SQL 语句:

```
select count(1)
from customer inner join store_sales on (ss_customer_sk = c_customer_sk);
```

在执行 customer inner join store_sales 的时候，云数据库 GaussDB 支持 Nested Loop、Merge Join 和 Hash Join 三种不同的 Join 方式。优化器会根据表 customer 和表 store_sales 的统计信息估算结果集的大小以及每种 join 方式的执行代价，然后对比选出执行代价最小的执行计划。

正如前面所说，执行代价计算都是基于一定的模型和统计信息进行估算，当因为某些原因代价估算不能反映真实的 cost 的时候，我们就需要通过 guc 参数设置的方式让执行计划倾向更优规划。

调优手段之底层存储

云数据库 GaussDB 的表支持行存表、列存表，底层存储方式的选择严格依赖于客户的具体业务场景。一般来说计算型业务查询场景(以关联、聚合操作为主)建议使用列存表；点查询、大批量 UPDATE/DELETE 业务场景适合行存表。

对于每种存储方式还有对应的存储层优化手段，这部分会在后续的调优章节深入介绍。

调优手段之 SQL 重写

除了上述干预 SQL 引擎所生成执行计划的执行性能外，根据数据库的 SQL 执行机制以及大量的实践发现，有些场景下，在保证客户业务 SQL 逻辑的前提下，通过一定规则由 DBA 重写 SQL 语句，可以大幅度的提升 SQL 语句的性能。

这种调优场景对 DBA 的要求比较高，需要对客户业务有足够的了解，同时也需要扎实的 SQL 语句基本功，后续会介绍几个常见的 SQL 改写场景。

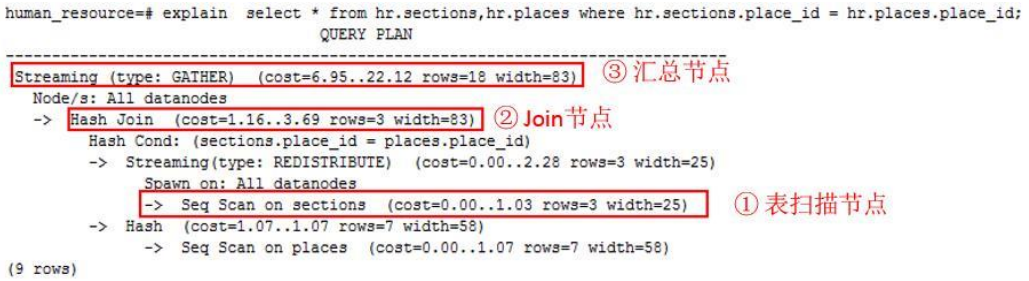
3.2.3.2 SQL 执行计划介绍

3.2.3.2.1 SQL 执行计划概述

SQL 执行计划是一个节点树，显示云数据库 GaussDB 执行一条 SQL 语句时执行的详细步骤。每一个步骤为一个数据库运算符。

使用 EXPLAIN 命令可以查看优化器为每个查询生成的具体执行计划。EXPLAIN 给每个执行节点都输出一行，显示基本的节点类型和优化器为执行这个节点预计的开销值。如图 3-3 所示。

图3-3 SQL 执行计划示例



- 最底层节点是表扫描节点，它扫描表并返回原始数据行。不同的表访问模式有不同的扫描节点类型：顺序扫描、索引扫描等。最底层节点的扫描对象也可能是非表行数据（不是直接从表中读取的数据），如 VALUES 子句和返回行集的函数，它们有自己的扫描节点类型。
- 如果查询需要连接、聚集、排序、或者对原始行做其它操作，那么就会在扫描节点之上添加其它节点。 并且这些操作通常都有多种方法，因此在这些位置也有可能出现不同的执行节点类型。
- 第一行(最上层节点)是执行计划总执行开销的预计。这个数值就是优化器试图最小化的数值。

执行计划显示格式

云数据库 GaussDB 对执行计划提供了 normal、pretty、summary、run 四种显示格式：

- normal：代表使用默认的打印格式。图 3-3 中即为此显示格式。
- pretty：代表使用云数据库 GaussDB 改进后的新显示格式。新的格式层次清晰，计划包含了 plan node id，性能分析简单直接。如图 3-4。
- summary：是在 pretty 的基础上增加了对打印信息的分析。
- run：在 summary 的基础上，将统计的信息输出到 csv 格式的文件中，以便于进一步分析。

图3-4 pretty 格式执行计划示例

```
postgres=# explain select cjxh, count(1) from dwcjk group by cjxh;
 id |               operation                | E-rows | E-memory | E-width | E-costs
----+----------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                        |      1 |          |      52 | 58.42
  2 |    ->  Vector Streaming (type: GATHER) |      1 |          |      52 | 58.42
  3 |       ->  Vector Hash Aggregate        |      1 | 16MB     |      52 | 58.02
  4 |          ->  CStore Scan on dwcjk      |      1 | 1MB      |      44 | 58.00
(4 rows)
```

通过设置 GUC 参数 explain_perf_mode，可以显示不同格式的执行计划。下文的用例默认显示 pretty 格式。

执行计划显示信息

除了设置不同的执行计划显示格式外，还可以通过不同的 EXPLAIN 用法，显示不同详细程度的执行计划信息。常见有如下几种，

- EXPLAIN *statement*： 只生成执行计划，不实际执行。其中 statement 代表 SQL 语句。
- EXPLAIN ANALYZE *statement*：生成执行计划，进行执行，并显示执行的概要信息。显示中加入了实际的运行时间统计，包括在每个规划节点内部花掉的总时间(以毫秒计)和它实际返回的行数。
- EXPLAIN PERFORMANCE *statement*：生成执行计划，进行执行，并显示执行期间的全部信息。

为了测量运行时在执行计划中每个节点的开销，EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 会在当前查询执行上增加性能分析的开销。在一个查询上运行 EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 有时会比普通查询明显的花费更多的时间。超支的数量依赖于查询的本质和使用的平台。

因此，当定位 SQL 运行慢问题时，如果 SQL 长时间运行未结束，建议通过 EXPLAIN 命令查看执行计划，进行初步定位。如果 SQL 可以运行出来，则推荐使用 EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 查看执行计划及其实际的运行信息，以便更精准地定位问题原因。

EXPLAIN PERFORMANCE 轻量化执行方式与 EXPLAIN PERFORMANCE 保持一致，在原来的基础上减少了性能分析的时间，执行时间与 SQL 执行时间的差异显著减少。

3.2.3.2.2 详解

如 3.2.3.2.1 SQL 执行计划概述节中所说，EXPLAIN 会显示执行计划，但并不会实际执行 SQL 语句。EXPLAIN ANALYZE 和 EXPLAIN PERFORMANCE 两者都会实际执行 SQL 语句并返回执行信息。在这一节将详细解释执行计划及执行信息。

执行计划

以如下 SQL 语句为例：

```
select
    cjxh,
```

```
    count(1)
from dwcjk
group by cjxh;
```

执行 EXPLAIN 的输出为：

```
postgres=# explain select cjxh, count(1) from dwcjk group by cjxh;
 id |               operation                | E-rows | E-memory | E-width | E-costs
----+----------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                        |      1 |          |      52 | 58.42
  2 |    ->  Vector Streaming (type: GATHER) |      1 |          |      52 | 58.42
  3 |       ->  Vector Hash Aggregate        |      1 | 16MB     |      52 | 58.02
  4 |          ->  CStore Scan on dwcjk      |      1 | 1MB      |      44 | 58.00
(4 rows)
```

执行计划字段解读（横向）：

- id：执行算子节点编号。
- operation：具体的执行节点算子名称。

  Vector 前缀的算子是指向量化执行引擎算子，一般出现含有列存表的 Query 中。

  Streaming 是一个特殊的算子，它实现了分布式架构的核心数据 shuffle 功能，Streaming 共有三种形态，分别对应了分布式结构下不同的数据 shuffle 功能：

  - Streaming (type: GATHER)：作用是 coordinator 从 DN 收集数据。
  - Streaming(type: REDISTRIBUTE)：作用是 DN 根据选定的列把数据重分布到所有的 DN。
  - Streaming(type: BROADCAST)：作用是把当前 DN 的数据广播给其他所有的 DN
- E-rows：每个算子估算的输出行数。
- E-memory：DN 上每个算子估算的内存使用量，只有 DN 上执行的算子会显示。某些场景会在估算的内存使用量后使用括号显示该算子在内存资源充足下可以自动扩展的内存上限。
- E-width：每个算子输出元组的估算宽度。
- E-costs：每个算子估算的执行代价。
  - E-costs 是优化器根据成本参数定义的单位来衡量的，习惯上以磁盘页面抓取为 1 个单位， 其它开销参数将参照它来设置。
  - 每个节点的开销（E-costs 值）包括它的所有子节点的开销。
  - 开销只反映了优化器关心的东西，并没有把结果行传递给客户端的时间考虑进去。虽然这个时间可能在实际的总时间里占据相当重要的分量，但是被优化器忽略了，因为它无法通过修改规划来改变。

执行计划层级解读（纵向）：

1. 第一层：CStore Scan on dwcjk

   表扫描算子，用 CStore Scan 的方式扫描表 dwcjk。这一层的作用是把表 dwcjk 的数据从 buffer 或者磁盘上读上来输送给上层节点参与计算。
2. 第二层：Vector Hash Aggregate

   聚合算子，作用是把下层计算输送上来的算子做聚合操作(group by)。
3. 第三层：Vector Streaming (type: GATHER)

Shuffle 算子，此处 GATHER 类型的 Shuffle 算子作用是把数据从 DN 汇聚到 CN。

4. 第四层：Row Adapter

   存储格式转化算子，主要作用是把内存中列式格式数据转为行式数据，以便客户端展示。

需要注意的是最顶层算子为 Data Node Scan 时，需要设置 enable_fast_query_shipping 为 off 才能看到具体的执行计划，如下面这个计划：

```
postgres=#  explain select cjxh, count(1) from dwcjk group by cjxh;
                QUERY PLAN
--------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes
(2 rows)
```

设置 enable_fast_query_shipping 参数之后，执行计划显示如下：

```
postgres=# set enable_fast_query_shipping=off;
SET
postgres=# explain select cjxh, count(1) from dwcjk group by cjxh;
 id |               operation               | E-rows | E-memory | E-width | E-costs
----+---------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                       |      1 |          |      52 | 58.42
  2 |    ->  Vector Streaming (type: GATHER) |      1 |          |      52 | 58.42
  3 |       ->  Vector Hash Aggregate       |      1 | 16MB     |      52 | 58.02
  4 |          ->  CStore Scan on dwcjk     |      1 | 1MB      |      44 | 58.00
(4 rows)
```

执行计划中的关键字说明：

1. 表访问方式

   - Seq Scan

     全表顺序扫描。

   - Index Scan

     优化器决定使用两步的规划：最底层的规划节点访问一个索引，找出匹配索引条件的行的位置，然后上层规划节点真实地从表中抓取出那些行。独立地抓取数据行比顺序地读取它们的开销高很多，但是因为并非所有表的页面都被访问了，这么做实际上仍然比一次顺序扫描开销要少。使用两层规划的原因是，上层规划节点在读取索引标识出来的行位置之前，会先将它们按照物理位置排序，这样可以最小化独立抓取的开销。

     如果在 WHERE 里面使用的好几个字段上都有索引，那么优化器可能会使用索引的 AND 或 OR 的组合。但是这么做要求访问两个索引，因此与只使用一个索引，而把另外一个条件只当作过滤器相比，这个方法未必是更优。

     索引扫描可以分为以下几类，他们之间的差异在于索引的排序机制。

     - Bitmap Index Scan

       使用位图索引抓取数据页。

     - Index Scan using index_name

       使用简单索引搜索，该方式表的数据行是以索引顺序抓取的，这样就令读取它们的开销更大，但是这里的行少得可怜，因此对行位置的额外排序并不值得。最常见的就是看到这种规划类型只抓取一行，以及那些要

求 ORDER BY 条件匹配索引顺序的查询。因为那时候没有多余的排序步骤是必要的以满足 ORDER BY。

2. 表连接方式
   - Nested Loop

     嵌套循环，适用于被连接的数据子集较小的查询。在嵌套循环中，外表驱动内表，外表返回的每一行都要在内表中检索找到它匹配的行，因此整个查询返回的结果集不能太大（不能大于 10000），要把返回子集较小的表作为外表，而且在内表的连接字段上建议要有索引。
   - (Sonic) Hash Join

     哈希连接，适用于数据量大的表的连接方式。优化器使用两个表中较小的表，利用连接键在内存中建立 hash 表，然后扫描较大的表并探测散列，找到与散列匹配的行。Sonic 和非 Sonic 的 Hash Join 的区别在于所使用 hash 表结构不同，不影响执行的结果集。
   - Merge Join

     归并连接，通常情况下执行性能差于哈希连接。如果源数据已经被排序过，在执行融合连接时，并不需要再排序，此时融合连接的性能优于哈希连接。
3. 运算符
   - sort

     对结果集进行排序。
   - filter

     EXPLAIN 输出显示 WHERE 子句当作一个"filter"条件附属于顺序扫描计划节点。这意味着规划节点为它扫描的每一行检查该条件，并且只输出符合条件的行。预计的输出行数降低了，因为有 WHERE 子句。不过，扫描仍将必须访问所有 10000 行，因此开销没有降低；实际上它还增加了一些（确切的说，通过 10000 * cpu_operator_cost）以反映检查 WHERE 条件的额外 CPU 时间。
   - LIMIT

     LIMIT 限定了执行结果的输出记录数。如果增加了 LIMIT，那么不是所有的行都会被检索到。

执行信息

在 SQL 调优过程中经常需要执行 EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 查看 SQL 语句实际执行信息，通过对比实际执行与优化器的估算之间的差别来为优化提供依据。EXPLAIN PERFORMANCE 相对于 EXPLAIN ANALYZE 增加了每个 DN 上的执行信息。

以如下 SQL 语句为例：

```
select count(1) from tb1;
```

执行 EXPLAIN PERFORMANCE 输出为：

```
postgres=# explain performance select count(1) from tbl;
 id |              operation               |    A-time     | A-rows | E-rows | E-distinct | Peak Memory  | E-memory | A-width | E-width | E-costs
----+--------------------------------------+---------------+--------+--------+------------+--------------+----------+---------+---------+---------
  1 | ->  Aggregate                        | 19.269        |      1 |      1 |            | 10KB         |          |         |       8 | 2.19
  2 |    ->  Streaming (type: GATHER)      | 19.191        |      4 |      4 |            | 144KB        |          |         |       8 | 2.19
  3 |       ->  Aggregate                  | [0.004,0.069] |      4 |      4 |            | [10KB, 10KB] | 1MB      |         |       8 | 2.03
  4 |          ->  Seq Scan on public.tbl | [0.001,0.060] |      5 |      5 |            | [12KB, 12KB] | 1MB      |         |       0 | 2.02
(4 rows)


                Memory Information (identified by plan id)
-----------------------------------------------------------------------
 Coordinator Query Peak Memory:
        Query Peak Memory: 0MB
 DataNode Query Peak Memory
        datanode1 Query Peak Memory: 0MB
        datanode2 Query Peak Memory: 0MB
        datanode3 Query Peak Memory: 0MB
        datanode4 Query Peak Memory: 0MB
   1 --Aggregate
        Peak Memory: 10KB, Estimate Memory: 64MB
   2 --Streaming (type: GATHER)
        Peak Memory: 144KB, Estimate Memory: 64MB
   3 --Aggregate
        datanode1 Peak Memory: 10KB, Estimate Memory: 1024KB
        datanode2 Peak Memory: 10KB, Estimate Memory: 1024KB
        datanode3 Peak Memory: 10KB, Estimate Memory: 1024KB
        datanode4 Peak Memory: 10KB, Estimate Memory: 1024KB
        datanode1 Stream Send time: 0.000; Data Serialize time: 0.006
        datanode2 Stream Send time: 0.000; Data Serialize time: 0.004
        datanode3 Stream Send time: 0.000; Data Serialize time: 0.005
        datanode4 Stream Send time: 0.000; Data Serialize time: 0.003
   4 --Seq Scan on public.tbl
        datanode1 Peak Memory: 12KB, Estimate Memory: 1024KB
        datanode2 Peak Memory: 12KB, Estimate Memory: 1024KB
        datanode3 Peak Memory: 12KB, Estimate Memory: 1024KB
        datanode4 Peak Memory: 12KB, Estimate Memory: 1024KB
(25 rows)


 Targetlist Information (identified by plan id)
------------------------------------------------
   1 --Aggregate
        Output: count((count(1)))
   2 --Streaming (type: GATHER)
        Output: (count(1))
        Node/s: All datanodes
   3 --Aggregate
        Output: count(1)
   4 --Seq Scan on public.tbl
        Output: a, b
        Distribute Key: a
(10 rows)


                    Datanode Information (identified by plan id)
-------------------------------------------------------------------------------
   1 --Aggregate
        (actual time=19.269..19.269 rows=1 loops=1)
        (Buffers: 0)
        (CPU: ex c/r=50593, ex row=4, ex cyc=202372, inc cyc=50094480)
   2 --Streaming (type: GATHER)
        (actual time=12.371..19.191 rows=4 loops=1)
        (Buffers: 0)
        (CPU: ex c/r=12473027, ex row=4, ex cyc=49892108, inc cyc=49892108)
   3 --Aggregate
        datanode1 (actual time=0.041..0.041 rows=1 loops=1)
        datanode2 (actual time=0.063..0.063 rows=1 loops=1)
        datanode3 (actual time=0.069..0.069 rows=1 loops=1)
        datanode4 (actual time=0.004..0.004 rows=1 loops=1)
        datanode1 (Buffers: shared hit=1)
        datanode2 (Buffers: shared hit=1)
        datanode3 (Buffers: shared hit=1)
        datanode4 (Buffers: 0)
        datanode1 (CPU: ex c/r=32888, ex row=1, ex cyc=32888, inc cyc=107740)
        datanode2 (CPU: ex c/r=9992, ex row=2, ex cyc=19984, inc cyc=163264)
        datanode3 (CPU: ex c/r=12272, ex row=2, ex cyc=24544, inc cyc=180008)
        datanode4 (CPU: ex c/r=0, ex row=0, ex cyc=8100, inc cyc=10768)
   4 --Seq Scan on public.tbl
        datanode1 (actual time=0.027..0.029 rows=1 loops=1)
        datanode2 (actual time=0.054..0.055 rows=2 loops=1)
        datanode3 (actual time=0.059..0.060 rows=2 loops=1)
        datanode4 (actual time=0.001..0.001 rows=0 loops=1)
        datanode1 (Buffers: shared hit=1)
        datanode2 (Buffers: shared hit=1)
        datanode3 (Buffers: shared hit=1)
        datanode4 (Buffers: 0)
        datanode1 (CPU: ex c/r=74852, ex row=1, ex cyc=74852, inc cyc=74852)
        datanode2 (CPU: ex c/r=71640, ex row=2, ex cyc=143280, inc cyc=143280)
        datanode3 (CPU: ex c/r=77732, ex row=2, ex cyc=155464, inc cyc=155464)
        datanode4 (CPU: ex c/r=0, ex row=0, ex cyc=2668, inc cyc=2668)
(34 rows)
```

```
                    User Define Profiling
--------------------------------------------------------------------
 Plan Node id: 2  Track name: coordinator get datanode connection
       coordinator1: (time=0.054 total_calls=1 loops=1)
(2 rows)

                        ====== Query Summary =====
--------------------------------------------------------------------------
 Datanode executor start time [datanode1, datanode2]: [1.352 ms,2.276 ms]
 Datanode executor end time [datanode4, datanode3]: [0.121 ms,0.187 ms]
 Remote query poll time: 10.992 ms, Deserialze time: 0.000 ms
 System available mem: 9188966KB
 Query Max mem: 9473228KB
 Query estimated mem: 2048KB
 Coordinator executor start time: 0.505 ms
 Coordinator executor run time: 19.277 ms
 Coordinator executor end time: 0.380 ms
 Planner runtime: 2.260 ms
 Query Id: 79375943434081342
 Total runtime: 20.803 ms
(12 rows)
```

图中显示执行信息分为以下 7 个部分

1. 以表格的形式将计划显示出来，包含有 11 个字段，分别是：id、operation、A-time、A-rows、E-rows、E-distinct、Peak Memory、E-memory、A-width、E-width 和 E-costs。其中计划类字段（id、operation 以及 E 开头字段）的含义与执行 EXPLAIN 时的含义一致，详见执行计划小节中的说明。A-time、A-rows、E-distinct、Peak Memory、A-width 的含义说明如下：
   - A-time：当前算子执行完成时间，一般 DN 上执行的算子的 A-time 是由[]括起来的两个值，分别表示此算子在所有 DN 上完成的最短时间和最长时间。
   - A-rows：表示当前算子的实际输出元组数。
   - E-distinct：表示 hashjoin 算子的 distinct 估计值。
   - Peak Memory：此算子在每个 DN 上执行时使用的内存峰值。
   - A-width：表示当前算子每行元组的实际宽度，仅对于重内存使用算子会显示，包括：(Vec)HashJoin、(Vec)HashAgg、(Vec) HashSetOp、(Vec)Sort、(Vec)Materialize 算子等，其中(Vec)HashJoin 计算的宽度是其右子树算子的宽度，会显示在其右子树上。
2. Predicate Information (identified by plan id)：

   这一部分主要显示的是静态信息，即在整个计划执行过程中不会变的信息，主要是一些 join 条件和一些 filter 信息。
3. Memory Information (identified by plan id)：

   这一部分显示的是整个计划中会将内存的使用情况打印出来的算子的内存使用信息，主要是 Hash、Sort 算子，包括算子峰值内存（peak memory），控制内存（control memory），估算内存使用（operator memory），执行时实际宽度（width），内存使用自动扩展次数（auto spread num），是否提前下盘（early spilled），以及下盘信息，包括重复下盘次数（spill Time(s)），内外表下盘分区数（inner/outer partition spill num），下盘文件数（temp file num），下盘数据量及最小和最大分区的下盘数据量（written disk IO [min, max] ）。
4. Targetlist Information (identified by plan id)

   这一部分显示的是每一个算子输出的目标列。
5. DataNode Information (identified by plan id)：

这一部分会将各个算子的执行时间、CPU、buffer 的使用情况全部打印出来。

6. User Define Profiling

   这一部分显示的是 CN 和 DN、DN 和 DN 建连的时间，以及存储层的一些执行信息。

7. ====== Query Summary =====：

   这一部分主要打印总的执行时间和网络流量，包括了各个 DN 上初始化和结束阶段的最大最小执行时间、CN 上的初始化、执行、结束阶段的时间，以及当前语句执行时系统可用内存、语句估算内存等信息。

须知

- A-rows 和 E-rows 的差异体现了优化器估算和实际执行的偏差度。一般来说，他们偏差越大，我们越可以认为优化器生成的计划的越不可信，人工干预调优的必要性越大。
- A-time 中的两个值偏差越大，表明此算子的计算偏斜(在不同 DN 上执行时间差异)越大，人工干预调优的必要性越大。
- Max Query Peak Memory 经常用来估算 SQL 语句耗费内存，也被用来作为 SQL 语句调优时运行态内存参数设置的重要依据。一般会以 EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 的输出作为进一步调优的输入。

3.2.3.3 调优流程

对慢 SQL 语句进行分析，通常包括以下步骤：

操作步骤

步骤 1 收集 SQL 中涉及到的所有表的统计信息。在数据库中，统计信息是规划器生成计划的源数据。没有收集统计信息或者统计信息陈旧往往会造成执行计划严重劣化，从而导致性能问题。从经验数据来看，10%左右性能问题是因为没有收集统计信息。具体请参见 3.2.3.4 更新统计信息。

步骤 2 通过查看执行计划来查找原因。如果 SQL 长时间运行未结束，通过 EXPLAIN 命令查看执行计划，进行初步定位。如果 SQL 可以运行出来，则推荐使用 EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 查看执行计划及实际运行情况，以便更精准地定位问题原因。有关执行计划的详细介绍请参见 3.2.3.2 SQL 执行计划介绍。

步骤 3 3.2.3.5 审视和修改表定义。

步骤 4 针对 EXPLAIN 或 EXPLAIN PERFORMANCE 信息，定位 SQL 慢的具体原因以及改进措施，具体参见 3.2.3.6 典型 SQL 调优点。

步骤 5 通常情况下，有些 SQL 语句可以通过查询重写转换成等价的，或特定场景下等价的语句。重写后的语句比原语句更简单，且可以简化某些执行步骤达到提升性能的目的。查询重写方法在各个数据库中基本是通用的。3.2.3.7 经验总结：SQL 语句改写规则介绍了几种常用的通过改写 SQL 进行调优的方法。

----结束

3.2.3.4 更新统计信息

在数据库中，统计信息是规划器生成计划的源数据。没有收集统计信息或者统计信息陈旧往往会造成执行计划严重劣化，从而导致性能问题。

背景信息

ANALYZE 语句可收集与数据库中表内容相关的统计信息，统计结果存储在系统表 PG_STATISTIC 中。查询优化器会使用这些统计数据，以生成最有效的执行计划。

建议在执行了大批量插入/删除操作后，例行对表或全库执行 ANALYZE 语句更新统计信息。目前默认收集统计信息的采样比例是 30000 行（即：guc 参数 default_statistics_target 默认设置为 100），如果表的总行数超过一定行数（大于 1600000），建议设置 guc 参数 default_statistics_target 为-2，即按 2%收集样本估算统计信息。

对于在批处理脚本或者存储过程中生成的中间表，也需要在完成数据生成之后显式的调用 ANALYZE。

对于表中多个列有相关性且查询中有同时基于这些列的条件或分组操作的情况，可尝试收集多列统计信息，以便查询优化器可以更准确地估算行数，并生成更有效的执行计划。

操作步骤

使用以下命令更新某个表或者整个 database 的统计信息。

```
ANALYZE tablename;                    --更新单个表的统计信息
ANALYZE;                              --更新全库的统计信息
```

使用以下命令进行多列统计信息相关操作。

```
ANALYZE tablename ((column 1, column 2));                    --收集 tablename 表的
column 1、column 2 列的多列统计信息

ALTER TABLE tablename ADD STATISTICS ((column 1, column 2));    --添加 tablename 表的
column 1、column 2 列的多列统计信息声明
ANALYZE tablename;                                           --收集单列统计信息，并收集已声明
的多列统计信息

ALTER TABLE tablename DELETE STATISTICS ((column 1, column 2)); --删除 tablename 表的
column_1、column_2 列的多列统计信息或其声明
```

须知

在使用 ALTER TABLE tablename ADD STATISTICS 语句添加了多列统计信息声明后，系统并不会立刻收集多列统计信息，而是在下次对该表或全库进行 ANALYZE 时，进行多列统计信息的收集。

说明

如果想直接收集多列统计信息，请使用 ANALYZE 命令进行收集。

使用 EXPLAIN 查看各 SQL 的执行计划时，如果发现某个表 SEQ SCAN 的输出中 rows=10，rows=10 是系统给的默认值，有可能该表没有进行 ANALYZE，需要对该表执行 ANALYZE。

3.2.3.5 审视和修改表定义

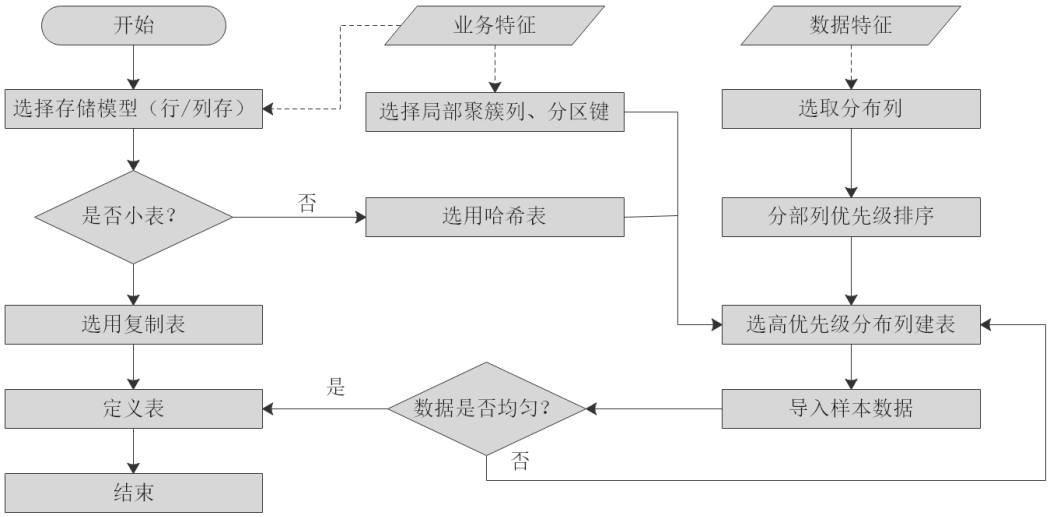
3.2.3.5.1 审视和修改表定义概述

在分布式框架下，数据分布在各个 DN 上。一个或者几个 DN 的数据存在一块物理存储设备上，好的表定义至少需要达到以下几个目标：

1. **表数据均匀分布在各个 DN 上**，以防止单个 DN 对应的存储设备空间不足造成集群有效容量下降。选择合适分布列，避免数据分布倾斜可以实现该点。
2. **表 Scan 压力均匀分散在各个 DN 上**，以避免单 DN 的 Scan 压力过大，形成 Scan 的单节点瓶颈。分布列不选择基表上等值 filter 中的列可以实现该点。
3. **减少扫描数据数据量**。通过分区的剪枝机制可以实现该点。
4. **尽量极少随机 IO**。通过聚簇/局部聚簇可以实现该点。
5. **尽量避免数据 shuffle**，减小网络压力。通过选择 join-condition 或者 group by 列为分布列可以最大程度的实现这点。

从上述描述来看表定义中最重要的一点是分布列的选择。创建表定义一般遵循图 3-5 所示流程。表定义在数据库设计阶段创建，在 SQL 调优过程中进行审视和修改。

图3-5 表定义流程



3.2.3.5.2 选择存储模型

进行数据库设计时，表设计上的一些关键项将严重影响后续整库的查询性能。表设计对数据存储也有影响：好的表设计能够减少 I/O 操作及最小化内存使用，进而提升查询性能。

表的存储模型选择是表定义的第一步。客户业务属性是表的存储模型的决定性因素，依据下面表格选择适合当前业务的存储模型。

存储模型	适用场景
行存	点查询(返回记录少，基于索引的简单查询)。 增删改比较多的场景。
列存	统计分析类查询 (group , join 多的场景)。

3.2.3.5.3 选择分布方式

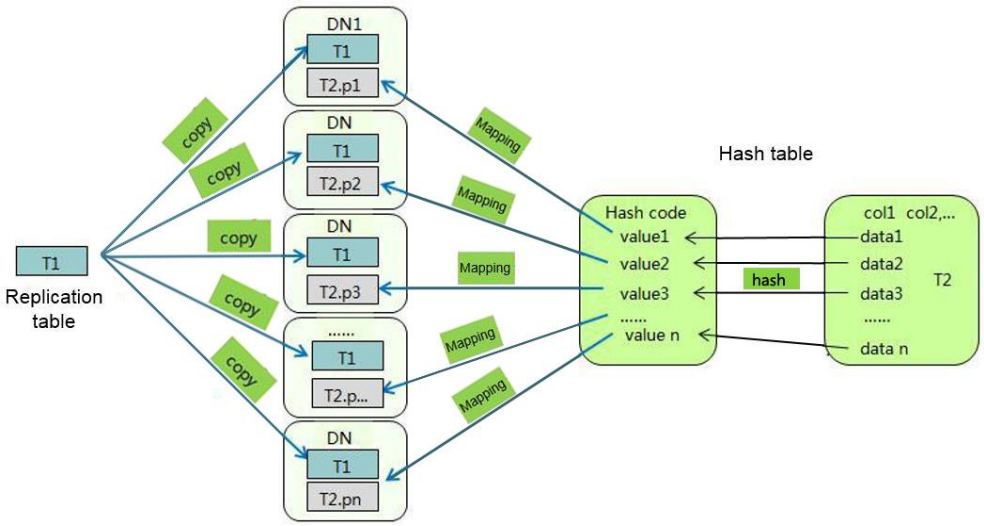
复制表（Replication）方式将表中的全量数据在集群的每一个 DN 实例上保留一份。主要适用于记录集较小的表。这种存储方式的优点是每个 DN 上都有该表的全量数据，在 join 操作中可以避免数据重分布操作，从而减小网络开销，同时减少了 plan segment(每个 plan segment 都会起对应的线程)；缺点是每个 DN 都保留了表的完整数据，造成数据的冗余。一般情况下只有较小的维度表才会定义为 Replication 表。

哈希（Hash）表将表中某一个或几个字段进行 hash 运算后，生成对应的 hash 值，根据 DN 实例与哈希值的映射关系获得该元组的目标存储位置。对于 Hash 分布表，在读/写数据时可以利用各个节点的 IO 资源，大大提升表的读/写速度。一般情况下大表定义为 Hash 表。

策略	描述	适用场景
Hash	表数据通过 hash 方式散列到集群中的所有 DN 实例上。	数据量较大的事实表。
Replication	集群中每一个 DN 实例上都有一份全量表数据。	小表、维度表。

如图 3-6 所示，复制表如图中的表 T1，哈希表如图中的表 T2。

图3-6 复制表和哈希表



3.2.3.5.4 选择分布列

Hash 分布表的分布列选取至关重要，需要满足以下原则：

1. **列值应比较离散，以便数据能够均匀分布到各个 DN**。例如，考虑选择表的主键为分布列，如在人员信息表中选择身份证号码为分布列。
2. **在满足第一条原则的情况下尽量不要选取存在常量 filter 的列**。例如，表 dwcjk 相关的部分查询中出现 dwcjk 的列 zqdh 存在常量的约束(例如 zqdh=’000001’)，那么就应当尽量不用 zqdh 做分布列。
3. **在满足前两条原则的情况，考虑选择查询中的连接条件为分布列**，以便 Join 任务能够下推到 DN 中执行，且减少 DN 之间的通信数据量。

对于 Hash 分表策略，如果分布列选择不当，可能导致数据倾斜，查询时出现部分 DN 的 I/O 短板，从而影响整体查询性能。因此在采用 Hash 分表策略之后需对表的数据进行数据倾斜性检查，以确保数据在各个 DN 上是均匀分布的。可以使用以下 SQL 检查数据倾斜性

```
select
xc node id, count(1)
from tablename
group by xc node id
order by xc_node_id desc;
```

其中 xc_node_id 对应 DN，一般来说，**不同 DN 的数据量相差 5%以上即可视为倾斜，如果相差 10%以上就必须要调整分布列**。

云数据库 GaussDB 支持多分布列特性，可以更好地满足数据分布的均匀性要求。

3.2.3.5.5 使用局部聚簇

局部聚簇（Partial Cluster Key）是列存下的一种技术。这种技术可以通过 min/max 稀疏索引较快的实现基表扫描的 filter 过滤。Partial Cluster Key 可以指定多列，但是一般不建议超过 2 列。Partial Cluster Key 的选取原则：

1. 受基表中的简单表达式约束。这种约束一般形如 col op const，其中 col 为列名，op 为操作符 =、>、>=、<=、<，const 为常量值。
2. 尽量采用选择度比较高(过滤掉更多数据)的简单表达式中的列。
3. 尽量把选择度比较低的约束 col 放在 Partial Cluster Key 中的前面。
4. 尽量把枚举类型的列放在 Partial Cluster Key 中的前面。

3.2.3.5.6 使用分区表

分区表是把逻辑上的一张表根据某种方案分成几张物理块进行存储。这张逻辑上的表称之为分区表，物理块称之为分区。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的。分区表和普通表相比具有以下优点：

1. 改善查询性能：对分区对象的查询可以仅搜索自己关心的分区，提高检索效率。
2. 增强可用性：如果分区表的某个分区出现故障，表在其他分区的数据仍然可用。
3. 方便维护：如果分区表的某个分区出现故障，需要修复数据，只修复该分区即可。

云数据库 GaussDB 支持的分区表为范围分区表。

范围分区表：将数据基于范围映射到每一个分区。这个范围是由创建分区表时指定的分区键决定的。分区键经常采用日期，例如将销售数据按照月份进行分区。

3.2.3.5.7 选择数据类型

高效数据类型，主要包括以下三方面：

1. **尽量使用执行效率比较高的数据类型**

   一般来说整型数据运算(包括=、＞、＜、≧、≦、≠等常规的比较运算，以及group by)的效率比字符串、浮点数要高。比如某客户场景中对列存表进行点查询，filter 条件在一个 numeric 列上，执行时间为 10+s；修改 numeric 为 int 类型之后，执行时间缩短为 1.8s 左右。

2. **尽量使用短字段的数据类型**

   长度较短的数据类型不仅可以减小数据文件的大小，提升 IO 性能；同时也可以减小相关计算时的内存消耗，提升计算性能。比如对于整型数据，如果可以用 smallint 就尽量不用 int，如果可以用 int 就尽量不用 bigint。

3. **使用一致的数据类型**

   表关联列尽量使用相同的数据类型。如果表关联列数据类型不同，数据库必须动态地转化为相同的数据类型进行比较，这种转换会带来一定的性能开销。

3.2.3.6 典型 SQL 调优点

SQL 调优是一个不断分析与尝试的过程：试跑 Query，判断性能是否满足要求；如果不满足要求，则通过 3.2.3.2 SQL 执行计划介绍分析原因并进行针对性优化；然后重新试跑和优化，直到满足性能目标。

3.2.3.6.1 SQL 自诊断

用户在执行查询或者执行 INSERT/DELETE/UPDATE/CREATE TABLE AS 语句时，可能会遇到性能问题。这种情况下，通过查询 GS_WLM_SESSION_STATISTICS，GS_WLM_SESSION_HISTORY，GS_WLM_SESSION_QUERY_INFO_ALL 视图的 warning 字段可以获得对应查询可能导致性能问题的告警信息，为性能调优提供参考。

告警场景

目前支持对以下 7 种导致性能问题的场景上报告警。

- 多列/单列统计信息未收集

如果存在单列或者多列统计信息未收集，则上报相关告警。调优方法可以参考 3.2.3.4 更新统计信息和 3.2.3.6.4 统计信息调优。

需要特别注意的是，对于基于 OBS 外表的查询，如果未收集统计信息也会上报统计信息未收集的告警，但是由于 OBS 外表的 analyze 的性能比较差，因此，需要用户对这种场景下告警是否通过 analyze 收集统计信息，以获取更优的性能，和查询本身的复杂度做权衡。

告警信息示例：

整表的统计信息未收集：

```
Statistic Not Collect:
   schema_test.t1
```

单列统计信息未收集：

```
Statistic Not Collect:
   schema_test.t2(c1,c2)
```

多列统计信息未收集:

```
Statistic Not Collect:
   schema_test.t3((c1,c2))
```

单列和多列统计信息未收集：

```
Statistic Not Collect:
   schema_test.t4(c1,c2)    schema_test.t4((c1,c2))
```

- SQL 不下推

  对于不下推的 SQL，尽可能详细上报导致不下推的原因。调优方法可以参考案例 3.2.3.6.2 语句下推调优。

  - 对于函数导致的不下推，告警导致不下推的函数名信息；
  - 对于不支持下推的语法，会告警对应语法不支持下推，例如：含有 With Recursive，Distinct On，row 表达式，返回值为 record 类型的，会告警相应语法不支持下推等等。

告警信息示例：

```
SQL is not plan-shipping, reason : "With Recursive" can not be shipped"
SQL is not plan-shipping, reason : "Function now() can not be shipped"
SQL is not plan-shipping, reason : "Function string_agg() can not be shipped"
```

- HashJoin 中大表做内表

如果在表连接过程中使用了 Hashjoin(可以在 GS_WLM_SESSION_HISTORY 视图的 query_plan 字段中查看到)，且连接的内表行数是外表行数的 10 倍或以上；同时内表在每个 DN 上的平均行数大于 10 万行，且发生了下盘，则上报相关告警。调优方法可以参考 3.2.3.9 使用 Plan Hint 进行调优。

告警信息示例：

```
PlanNode[7] Large Table is INNER in HashJoin “Vector Hash Aggregate”
```

- 大表等值连接使用 Nestloop

如果在表连接过程中使用了 nestloop(可以在 GS_WLM_SESSION_HISTORY 视图的 query_plan 字段中查看到)，并且两个表中较大表的行数平均每个 DN 上的行数大于 10 万行、表的连接中存在等值连接，则上报相关告警。调优方法可以参考 3.2.3.9 使用 Plan Hint 进行调优。

告警信息示例：

```
PlanNode[5] Large Table with Equal-Condition use Nestloop"Nested Loop"
```

- 大表 Broadcast

如果在 Broadcast 算子中，平均每 DN 的行数大于 10 万行，则告警大表 broadcast。调优方法可以参考 3.2.3.9 使用 Plan Hint 进行调优。

告警信息示例：

```
PlanNode[5] Large Table in Broadcast "Streaming(type: BROADCAST dop: 1/2)"
```

- 数据倾斜

某表在各 DN 上的分布，存在某 DN 上的行数是另一 DN 上行数的 10 倍或以上，且有 DN 中的行数大于 10 万行，则上报相关告警。调优方法可以参考案例 3.2.3.6.6 数据倾斜调优。

告警信息示例：

```
PlanNode[6] DataSkew:"Seq Scan", min_dn_tuples:0, max_dn_tuples:524288
```

- 估算不准

如果优化器的估算行数和实际行数中的较大值平均每 DN 行数大于 10 万行，并且估算行数和实际行数中较大值是较小值的 10 倍或以上，则上报相关告警。调优方法可以参考 3.2.3.9 使用 Plan Hint 进行调优。

告警信息示例：

```
PlanNode[5] Inaccurate Estimation-Rows: "Hash Join" A-Rows:0, E-Rows:52488
```

规格约束

1. 告警字符串长度上限为 2048。如果告警信息超过这个长度（例如存在大量未收集统计信息的超长表名，列名等信息）则不告警，只上报 warning：

   WARNING, "Planner issue report is truncated, the rest of planner issues will be skipped"
2. 如果 query 存在 limit 节点（即查询语句中包含 limit），则不会上报 limit 节点以下的 Operator 级别的告警。
3. 对于“数据倾斜”和“估算不准”两种类型告警，在某一个 plan 树结构下，只上报下层节点的告警，上层节点不再重复告警。这主要是因为这两种类型的告警可能是因为底层触发上层的。例如，如果在 scan 节点已经存在数据倾斜，那么在上层的 hashagg 等其他算子很可能也出现数据倾斜。

3.2.3.6.2 语句下推调优

语句下推介绍

目前，云数据库 GaussDB 优化器在分布式框架下制定语句的执行策略时，有三种执行计划方式：生成下推语句计划、生成分布式执行计划、生成发送语句的分布式执行计划。

- 下推语句计划：指直接将查询语句从 CN 发送到 DN 进行执行，然后将执行结果返回给 CN。
- 分布式执行计划：指 CN 对查询语句进行编译和优化，生成计划树，再将计划树发送给 DN 进行执行，并在执行完毕后返回结果到 CN。
- 发送语句的分布式执行计划：上述两种方式都不可行时，将可下推的查询部分组成查询语句（多为基表扫描语句）下推到 DN 进行执行，获取中间结果到 CN，然后在 CN 执行剩下的部分。

在第 3 种策略中，要将大量中间结果从 DN 发送到 CN，并且要在 CN 运行不能下推的部分语句，会导致 CN 成为性能瓶颈（带宽、存储、计算等）。在进行性能调优的时候，应尽量避免只能选择第 3 种策略的查询语句。

执行语句不能下推是因为语句中含有不支持下推的函数或者不支持下推的语法。一般都可以通过等价改写规避执行计划不能下推的问题。

语句下推典型场景

通常而言 explain 语句后没有显示具体的执行计划算子，仅存在类似关键字“Data Node Scan on”则说明语句已下推给 DN 去执行。下面我们从三个维度场景介绍下语句下推以及其支持的范围。

1 单表查询语句下推

在分布式数据库中对于单表查询而言，当前语句是否可以下推需要判断 CN 是否要进一步参与计算而不是简单收集数据。如果 CN 要进一步对 DN 结果进行计算则语句不可下推。通常带有 agg, windows function, limit/offset, sort, distinct 等关键字都不可下推。

- 可下推：简单查询，无需在 CN 进一步计算则可以下推。

```
postgres=# explain select * from t where c1 > 1;
                            QUERY PLAN
------------------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes
(2 rows)
```

- 不可下推：带有 limit 子句，对于 CN 而言不能简单发语句给 DN 并收集数据，明显与 limit 语义不符。

```
postgres=# explain select * from t limit 1;
                                QUERY PLAN
---------------------------------------------------------------------------------
------
 Limit  (cost=0.00..0.00 rows=1 width=12)
   ->  Data Node Scan on "  REMOTE LIMIT QUERY  "  (cost=0.00..0.00 rows=1
width=12)
         Node/s: All datanodes
(3 rows)
```

- 不可下推：带有聚集函数 CN 不能简单下推语句，而应该对从 DN 收集结果进一步聚集运算处理。

```
postgres=# explain select sum(c1), count(*) from t;
                                QUERY PLAN
---------------------------------------------------------------------------------
------
 Aggregate  (cost=0.10..0.11 rows=1 width=20)
   ->  Data Node Scan on "__REMOTE_GROUP_QUERY__"  (cost=0.00..0.00 rows=20
width=4)
         Node/s: All datanodes
(3 rows)
```

2 多表查询语句下推

多表查询场景下语句能否下推通常与 join 条件以及分布列有关，即如果 join 条件与表分布列匹配得上则可下推，否则无法下推。对于复制表来说通常可以下推。

- 创建两个 hash 分布表。

```
postgres=# create table t(c1 int, c2 int, c3 int)distribute by hash(c1);
CREATE TABLE
postgres=# create table t1(c1 int, c2 int, c3 int)distribute by hash(c1);
CREATE TABLE
```

- 可下推：join 条件满足两个表 hash 分布列属性。

```
postgres=# explain select * from t1 join t on t.c1 = t1.c1;
                            QUERY PLAN
------------------------------------------------------------------------------
 Data Node Scan on "  REMOTE FQS QUERY  "  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes
(2 rows)
```

- 不可下推：join 条件不满足 hash 分布列属性，即 t1.c2 不是 t1 表的分布列。

```
postgres=# explain select * from t1 join t on t.c1 = t1.c2;
                                  QUERY PLAN
------------------------------------------------------------------------------
-------------
 Hash Join  (cost=0.25..0.53 rows=20 width=24)
   Hash Cond: (t1.c2 = t.c1)
   ->  Data Node Scan on t1 " REMOTE TABLE QUERY "  (cost=0.00..0.00 rows=20
width=12)
         Node/s: All datanodes
   ->  Hash  (cost=0.00..0.00 rows=20 width=12)
         ->  Data Node Scan on t " REMOTE TABLE QUERY "  (cost=0.00..0.00 rows=20
width=12)
               Node/s: All datanodes
(7 rows)
```

3 特殊场景

对于有一些特殊场景通常无法下推，例如语句中带有 with recursive 子句，列存表等不支持下推。

实例分析：自定义函数

对于自定义函数，如果对于确定的输入，有确定的输出，则应将函数定义为 immutable 类型。

利用 TPCDS 的销售信息举个例子，比如我们要写一个函数，获取商品的打折情况，需要一个计算折扣的函数，我们可以将这个函数定义为：

```
CREATE FUNCTION func percent 2 (NUMERIC, NUMERIC) RETURNS NUMERIC
AS 'SELECT $1 / $2 WHERE $2 > 0.01'
LANGUAGE SQL
VOLATILE;
```

执行下列语句：

```
SELECT func percent 2(ss sales price, ss list price)
FROM store_sales;
```

其执行计划为：

```
 Data Node Scan on store_sales "_REMOTE_TABLE_QUERY_"
   Output: func_percent_2(store_sales.ss_sales_price, store_sales.ss_list_price)
   Remote query: SELECT ss_sales_price, ss_list_price FROM ONLY store_sales WHERE true
(3 rows)
```

可见，func_percent_2 并没有被下推，而是将 ss_sales_price 和 ss_list_price 收到 CN 上，再进行计算，消耗大量 CN 的资源，而且计算缓慢。

由于该自定义函数对确定的输入有确定的输出，如果将该自定义函数改为：

```
CREATE FUNCTION func percent 1 (NUMERIC, NUMERIC) RETURNS NUMERIC
AS 'SELECT $1 / $2 WHERE $2 > 0.01'
LANGUAGE SQL
IMMUTABLE;
```

执行语句：

```
SELECT func percent 1(ss sales price, ss list price)
FROM store_sales;
```

其执行计划为：

```
 Data Node Scan
   Output: (func_percent_1(store_sales.ss_sales_price, store_sales.ss_list_price))
   Remote query: SELECT func_percent_1(ss_sales_price, ss_list_price) AS func_percent_1 FROM store_sales
(3 rows)
```

可见函数 func_percent_1 被下推到 DN 执行，提升了执行效率（TPCDS 1000X，3CN18DN，查询效率提升 100 倍以上）。

不支持下推的函数

首先介绍函数的易变性。在云数据库 GaussDB 中共分三种形态：

- **IMMUTABLE**

  表示该函数在给出同样的参数值时总是返回同样的结果。
- **STABLE**

  表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同 SQL 语句之间变化。
- **VOLATILE**

  表示该函数值可以在一次表扫描内改变，因此不会做任何优化。

函数易变性可以查询 pg_proc 的 provolatile 字段获得，i 代表 IMMUTABLE，s 代表 STABLE，v 代表 VOLATILE。另外，在 pg_proc 中的 proshippable 字段，取值范围为 t/f/NULL，这个字段与 provolatile 字段一起用于描述函数是否下推。

- 如果函数的 provolatile 属性为 i，则无论 proshippable 的值是否为 t，则函数始终可以下推。
- 如果函数的 provolatile 属性为 s 或 v，则仅当 proshippable 的值为 t 时，函数可以下推。
- random，exec_hadoop_sql，exec_on_extension 如果出现 CTE 中，也不下推。因为这种场景下下推可能出现结果错误。

对于用户自定义函数，可以在创建函数的时候指定 provolatile 和 proshippable 属性的值，详细请参考 CREATE FUNCTION 语法。

对于函数不能下推的场景：

- 如果是系统函数，建议根据业务等价替换这个函数。
- 如果是自定义函数，建议分析客户业务场景，看函数的 provolatile 和 proshippable 属性定义是否正确。

不支持下推的语法

以如下三个表定义说明不支持下推的 SQL 语法。

```
postgres=# CREATE TABLE CUSTOMER1
(
   C CUSTKEY     BIGINT NOT NULL
  , C NAME       VARCHAR(25) NOT NULL
  , C ADDRESS    VARCHAR(40) NOT NULL
  , C NATIONKEY   INT NOT NULL
  , C PHONE      CHAR(15) NOT NULL
  , C ACCTBAL    DECIMAL(15,2)   NOT NULL
  , C MKTSEGMENT  CHAR(10) NOT NULL
  , C COMMENT    VARCHAR(117) NOT NULL
)
DISTRIBUTE BY hash(C CUSTKEY);
postgres=# CREATE TABLE test_stream(a int,b float); --float 不支持重分布
postgres=# CREATE TABLE sal_emp ( c1 integer[] ) DISTRIBUTE BY replication;
```

- 不支持 returning 语句下推

```
postgres=# explain update customer1 set C_NAME = 'a' returning c_name;
                          QUERY PLAN
------------------------------------------------------------------
 Update on customer1  (cost=0.00..0.00 rows=30 width=187)
   Node/s: All datanodes
   Node expr: c_custkey
   ->  Data Node Scan on customer1 "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00
rows=30 width=187)
         Node/s: All datanodes
(5 rows)
```

- 不支持聚集函数中使用 order by 语句的下推

```
postgres=# explain verbose select count ( c custkey order by c custkey) from
customer1;

                    QUERY PLAN
------------------------------------------------------------------ Aggregate
(cost=2.50..2.51 rows=1 width=8)
   Output: count(customer1.c custkey ORDER BY customer1.c custkey)
   ->  Data Node Scan on customer1 "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00
rows=30 width=8)
         Output: customer1.c_custkey
         Node/s: All datanodes
         Remote query: SELECT c_custkey FROM ONLY public.customer1 WHERE true
(6 rows)
```

- count(distinct expr)中的字段不支持重分布，则不支持下推

```
postgres=# explain verbose select count(distinct b) from test_stream;
                                  QUERY PLAN
------------------------------------------------------------------------ Aggregate
(cost=2.50..2.51 rows=1 width=8)
  Output: count(DISTINCT test_stream.b)
  ->  Data Node Scan on test_stream "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00
rows=30 width=8)
       Output: test_stream.b
       Node/s: All datanodes
       Remote query: SELECT b FROM ONLY public.test_stream WHERE true
(6 rows)
```

- 不支持 distinct on 用法下推

```
postgres=# explain verbose select distinct on (c custkey) c custkey from
customer1 order by c custkey;
                                    QUERY PLAN
------------------------------------------------------------------------ Unique
(cost=49.83..54.83 rows=30 width=8)
  Output: customer1.c custkey
  ->  Sort  (cost=49.83..52.33 rows=30 width=8)
       Output: customer1.c custkey
       Sort Key: customer1.c custkey
       ->  Data Node Scan on customer1 " REMOTE TABLE QUERY "  (cost=0.00..0.00
rows=30 width=8)
            Output: customer1.c custkey
            Node/s: All datanodes
            Remote query: SELECT c custkey FROM ONLY public.customer1 WHERE
true
(9 rows)
```

- 不支持数组表达式下推

```
postgres=# explain verbose select array[c custkey,1] from customer1 order by
c custkey;

                     QUERY PLAN
------------------------------------------------------------------------ Sort
(cost=49.83..52.33 rows=30 width=8)
  Output: (ARRAY[customer1.c custkey, 1::bigint]), customer1.c custkey
  Sort Key: customer1.c custkey
  ->  Data Node Scan on "  REMOTE SORT QUERY  "  (cost=0.00..0.00 rows=30
width=8)
       Output: (ARRAY[customer1.c custkey, 1::bigint]), customer1.c custkey
       Node/s: All datanodes
       Remote query: SELECT ARRAY[c custkey, 1::bigint], c custkey FROM ONLY
public.customer1 WHERE true ORDER BY 2
(7 rows)
```

- With Recursive 当前版本不支持下推的场景和原因如下：

序号	场景	不下推原因
1	包含外表的查询场景	LOG: SQL can't be shipped, reason: RecursiveUnion contains HDFS Table or ForeignScan is not shippable（LOG 为 CN 日志中打印的不下推原因，下同）

序号	场景	不下推原因
		外表，当前版本暂不支持下推。
2	多 nodegroup 场景	LOG: SQL can't be shipped, reason: With-Recursive under multi-nodegroup scenario is not shippable 基表存储 nodegroup 不相同，或者计算 nodegroup 与基表不相同，当前版本暂不支持下推。
3	UNION 不带 ALL，需要去重	LOG: SQL can't be shipped, reason: With-Recursive does not contain "ALL" to bind recursive & none-recursive branches 例如: `WITH recursive t result AS ( SELECT dm,sj dm,name,1 as level FROM test rec part WHERE sj dm > 10 UNION SELECT t2.dm,t2.sj dm,t2.name||' > '||t1.name,t1.level+1 FROM t result t1 JOIN test rec part t2 ON t2.sj dm = t1.dm ) SELECT * FROM t_result t;`
4	基表中有系统表	LOG: SQL can't be shipped, reason: With-Recursive contains system table is not shippable 例如： `WITH RECURSIVE x(id) AS ( select count(1) from pg class where oid=1247 UNION ALL SELECT id+1 FROM x WHERE id < 5 ), y(id) AS ( select count(1) from pg class where oid=1247 UNION ALL SELECT id+1 FROM x WHERE id < 10 ) SELECT y.*, x.* FROM y LEFT JOIN x USING (id) ORDER BY 1;`
5	基表扫描只有 VALUES 子句，仅在 CN 上即可完成执行。	LOG: SQL can't be shipped, reason: With-Recursive contains only values rte is not shippable 例如：

序号	场景	不下推原因
		WITH RECURSIVE t(n) AS (VALUES (1) UNION ALL SELECT n+1 FROM t WHERE n < 100) SELECT sum(n) FROM t;
6	相关子查询的关联条件仅在递归部分，非递归部分无关联条件。	LOG: SQL can't be shipped, reason: With-Recursive recursive term correlated only is not shippable 例如： select a.ID,a.Name, (with recursive cte as (select ID, PID, NAME from b where b.ID = 1 union all select parent.ID,parent.PID,parent.NAME from cte as child join b as parent on child.pid=parent.id where child.ID = a.ID) select NAME from cte limit 1) cName from (select id, name, count(*) as cnt from a group by id,name) a order by 1,2;
7	非递归部分带 limit 为 Replicate 计划，递归部分为 Hash 计划，计划存在冲突。	LOG: SQL can't be shipped, reason: With-Recursive contains conflict distribution in none-recursive(Replicate) recursive(Hash) 例如： WITH recursive t result AS (select * from(SELECT dm,sj dm,name,1 as level FROM test rec part WHERE sj dm < 10 order by dm limit 6 offset 2) UNION all SELECT t2.dm,t2.sj dm,t2.name||' > '||t1.name,t1.level+1 FROM t result t1 JOIN test rec part t2 ON t2.sj dm = t1.dm) SELECT * FROM t_result t;
8	多层 Recursive 嵌套，即 recursive	LOG: SQL can't be shipped, reason: Recursive CTE references recursive

序号	场景	不下推原因
	的递归部分又嵌套另一个 recursive 查询。	CTE "cte" 例如： `with recursive cte as (select * from rec tb4 where id<4 union all select h.id,h.parentID,h.name from (with recursive cte as (select * from rec tb4 where id<4 union all select h.id,h.parentID,h.name from rec tb4 h inner join cte c on h.id=c.parentID) SELECT id ,parentID,name from cte order by parentID) h inner join cte c on h.id=c.parentID) SELECT id ,parentID,name from cte order by parentID,1,2,3;`

3.2.3.6.3 子查询调优

子查询背景介绍

应用程序通过 SQL 语句来操作数据库时会使用大量的子查询，这种写法比直接对两个表做连接操作在结构上和思路上更清晰，尤其是在一些比较复杂的查询语句中，子查询有更完整、更独立的语义，会使 SQL 对业务逻辑的表达更清晰更容易理解，因此得到了广泛的应用。

云数据库 GaussDB 根据子查询在 SQL 语句中的位置把子查询分成了子查询、子链接两种形式。

- 子查询 SubQuery：对应于查询解析树中的范围表 RangeTblEntry，更通俗一些指的是出现在 FROM 语句后面的独立的 SELECT 语句。
- 子链接 SubLink：对应于查询解析树中的表达式，更通俗一些指的是出现在 where/on 子句、targetlist 里面的语句。

  综上，对于查询解析树而言，SubQuery 的本质是范围表、而 SubLink 的本质是表达式。针对 SubLink 场景而言，由于 SubLink 可以出现在约束条件、表达式中，按照云数据库 GaussDB 对 sublink 的实现，sublink 可以分为以下几类：
  - exist_sublink：对应 EXIST、NOT EXIST 语句
  - any_sublink：对应 op ALL(select…)语句，其中 OP 可以是 IN,<,>,=操作符
  - all_sublink：对应 op ALL(select…)语句，其中 OP 可以是 IN,<,>,=操作符

- rowcompare_sublink：对应 record op (select …)语句
- expr_sublink：对应(SELECT with single targetlist item ...)语句
- array_sublink：对应 ARRAY(select…)语句
- cte_sublink：对应 with query(…)语句

其中 OLAP、HTAP 场景中常用的 sublink 为 exist_sublink、any_sublink，在云数据库 GaussDB 的优化引擎中对其应用场景做了优化（子链接提升），由于 SQL 语句中子查询的使用的灵活性，会带来 SQL 子查询过于复杂造成性能问题。子查询从大类上来看，分为非相关子查询和相关子查询：

- **非相关子查询 None-Correlated SubQuery**

  子查询的执行不依赖于外层父查询的任何属性值。这样子查询具有独立性，可独自求解，形成一个子查询计划先于外层的查询求解。

  例如：

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
   select c2
   from t2
   where t2.c2 IN (2,3,4)
);
                     QUERY PLAN
-------------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Hash Right Semi Join
       Hash Cond: (t2.c2 = t1.c1)
       ->  Streaming(type: REDISTRIBUTE)
            Spawn on: All datanodes
            ->  Seq Scan on t2
                 Filter: (c2 = ANY ('{2,3,4}'::integer[]))
       ->  Hash
            ->  Seq Scan on t1
(10 rows)
```

- **相关子查询 Correlated-SubQuery**

  子查询的执行依赖于外层父查询的一些属性值（如下列示例 t2.c1 = t1.c1 条件中的 t1.c1）作为内层查询的一个 AND-ed 条件。这样的子查询不具备独立性，需要和外层查询按分组进行求解。

  例如：

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
   select c2
   from t2
   where t2.c1 = t1.c1 AND t2.c2 in (2,3,4)
);
                           QUERY PLAN
------------------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Seq Scan on t1
```

```
        Filter: (SubPlan 1)
        SubPlan 1
          ->  Result
                Filter: (t2.c1 = t1.c1)
                ->  Materialize
                      ->  Streaming(type: BROADCAST)
                            Spawn on: All datanodes
                            ->  Seq Scan on t2
                                  Filter: (c2 = ANY ('{2,3,4}'::integer[]))
(12 rows)
```

云数据库 GaussDB 对 SubLink 的优化

针对 SubLink 的优化策略主要是让内层的子查询提升(pullup)，能够和外表直接做关联查询，从而避免生成 SubPlan+Broadcast 内表的执行计划。判断子查询是否存在性能风险，可以通过 explain 查询语句查看 Sublink 的部分是否被转换成 SubPlan+Broadcast 的执行计划。

例如：

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
    select c2
    from t2
    where t2.c1 = t1.c1
);
```

```
QUERY PLAN
-----------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Seq Scan on t1
        Filter: (SubPlan 1)
        SubPlan 1
          ->  Result
                Filter: (t2.c1 = t1.c1)
                ->  Materialize
                      ->  Streaming(type: BROADCAST)
                            Spawn on: All datanodes
                            ->  Seq Scan on t2
(11 rows)
```

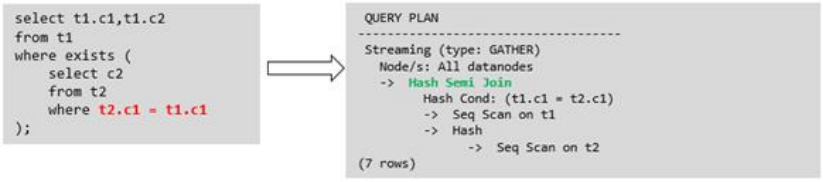
- **目前云数据库 GaussDB 支持的 Sublink-Release 场景**
  - IN-Sublink 无相关条件
    - 不能包含上一层查询的表中的列（可以包含更高层查询表中的列）。
    - 不能包含易变函数。

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
    select c2
    from t2
    where t2.c1 = 1
);
```

```
QUERY PLAN
-----------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Nested Loop Semi Join
        Join Filter: (t1.c1 = t2.c2)
        ->  Seq Scan on t1
        ->  Materialize
              ->  Streaming(type: REDISTRIBUTE)
                    Spawn on: datanode1
                    ->  Seq Scan on t2
                          Filter: (c1 = 1)
(10 rows)
```

  - Exist-Sublink 包含相关条件

    Where 子句中必须包含上一层查询的表中的列，子查询的其它部分不能含有上层查询的表中的列。其它限制如下。
    - 子查询必须有 from 子句。
    - 子查询不能含有 with 子句。
    - 子查询不能含有聚集函数。
    - 子查询里不能包含集合操作、排序、limit、windowagg、having 操作。
    - 不能包含易变函数。



- 包含聚集函数的等值相关子查询的提升

  子查询的 where 条件中必须含有来自上一层的列，而且此列必须和子查询本层涉及表中的列做相等判断，且这些条件必须用 and 连接。其它地方不能包含上层的列。其它限制条件如下。

  - 子查询中 where 条件包含的表达式(列名)必须是表中的列。
  - 子查询的 Select 关键字后，必须有且仅有一个输出列，此输出列必须是聚集函数(如 max)，并且聚集函数的参数(t2.c2)不能是来自外层表(t1)中的列。聚集函数不能是 count。

    例如，下列示例可以提升。

    ```
    select * from t1 where c1 >(
          select max(t2.c1) from t2 where t2.c1=t1.c1
    );
    ```

    下列示例不能提升，因为子查询没有聚集函数。

    ```
    select * from t1 where c1 >(
          select  t2.c1 from t2 where t2.c1=t1.c1
    );
    ```

    下列示例不能提升，因为子查询有两个输出列。

    ```
    select * from t1 where (c1,c2) >(
          select  max(t2.c1),min(t2.c2) from t2 where t2.c1=t1.c1
    );
    ```

  - 子查询必须是 from 子句。
  - 子查询中不能有 groupby、having、集合操作。
  - 子查询只能是 inner join。

    例如：下列示例不能提升。

    ```
    select * from t1 where c1 >(
          select max(t2.c1) from t2 full join t3 on (t2.c2=t3.c2) where
    t2.c1=t1.c1
    );
    ```

  - 子查询的 targetlist 中不能包含返回 set 的函数。
  - 子查询的 where 条件中必须含有来自上一层的列，而且此列必须和子查询层涉及表中的列做相等判断，且这些条件必须用 and 连接。其它地方不能包含上层的上层中的列。例如：下列示例中的最内层子链接可以提升。

    ```
    select * from t3 where t3.c1=(
           select t1.c1
           from t1 where c1 >(
                  select max(t2.c1) from t2 where t2.c1=t1.c1
    ));
    ```

基于上面的示例，再加一个条件，则不能提升，因为最内侧子查询引用了上层中的列。示例如下：

```
select * from t3 where t3.c1=(
       select t1.c1
       from t1 where c1 >(
             select max(t2.c1) from t2 where t2.c1=t1.c1 and
t3.c1>t2.c2

));
```

- 提升 OR 子句中的 SubLink

  当 WHERE 过滤条件中有 OR 连接的 EXIST 相关 SubLink，

  例如：

```
select a, c from t1
where t1.a = (select avg(a) from t3 where t1.b = t3.b) or
exists (select * from t4 where t1.c = t4.c);
```

  将 OR-ed 连接的 EXIST 相关子查询 OR 字句的提升过程：

  i. 提取 where 条件中，or 子句中的 opExpr。为：t1.a = (select avg(a) from t3 where t1.b = t3.b)

  ii. 这个 op 操作中包含 subquery，判断是否可以提升，如果可以提升，重写 subquery 为：select avg(a), t3.b from t3 group by t3.b，生成 not null 条件 t3.b is not null，并将这个 opexpr 用这个 not null 条件替换。此时 SQL 变为：

```
select a, c
from t1 left join (select avg(a) avg, t3.b from t3 group by t3.b)  as
t3 on (t1.a = avg and t1.b = t3.b)
where t3.b is not null or exists (select * from t4 where t1.c = t4.c);
```

  iii. 再次提取 or 子句中的 exists sublink，exists (select * from t4 where t1.c = t4.c)，判断是否可以提升，如果可以提升，转换 subquery 为：select t4.c from t4 group by t4.c 生成 NotNull 条件 t4.c is not null 提升查询，SQL 变为：

```
select t1.a, t1.c from t1 left join (select avg(a) avg, t3.b from t3
group by t3.b) as t3 on (t1.a = avg and t1.b = t3.b) left join (select
t5.c from t5 group by t5.c) as t5 on (t1.c = t5.c) where t3.b is not
null or t5.c is not null;
```

```
select * from t1
where exists (
    select t2.c1 from t2
    where t2.c1 = t1.c1
) OR
exists (
    select t3.c1 from t3
    where t3.c1 = t1.c1
);
```

```
QUERY PLAN
------------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Hash Left Join
        Hash Cond: (t1.c1 = t3.c1)
        Filter: ((t2.c1 IS NOT NULL) OR (t3.c1 IS NOT NULL))
        ->  Hash Left Join
              Hash Cond: (t1.c1 = t2.c1)
              ->  Seq Scan on t1
              ->  Hash
                    ->  HashAggregate
                          Group By Key: t2.c1
                          ->  Seq Scan on t2
        ->  Hash
              ->  HashAggregate
                    Group By Key: t3.c1
                    ->  Seq Scan on t3
(16 rows)
```

- **目前云数据库 GaussDB 不支持的 Sublink-Release 场景**

  除了以上场景之外都不支持 Sublink 提升，因此关联子查询会被计划成 SubPlan+Broadcast 的执行计划，当 inner 表的数据量较大时则会产生性能风险。

如果相关子查询中跟外层的两张表做 join，那么无法提升该子查询，需要通过将父 SQL 创建成 with 子句，然后再跟子查询中的表做相关子查询查询。

例如：

```
select distinct t1.a, t2.a
from t1 left join t2 on t1.a=t2.a and not exists (select a,b from test1 where
test1.a=t1.a and test1.b=t2.a);
```

改写为

```
with temp as
(
       select * from (select t1.a as a, t2.a as b from t1 left join t2 on
t1.a=t2.a)

)
select distinct a,b
from temp
where not exists (select a,b from test1 where temp.a=test1.a and
temp.b=test1.b);
```

- 出现在 targetlist 里的相关子查询无法提升(不含 count)

  例如：

  ```
  explain (costs off)
  select (select c2 from t2 where t1.c1 = t2.c1) ssq, t1.c2
  from t1
  where t1.c2 > 10;
  ```

  执行计划为：

  ```
  explain (costs off)
  select (select c2 from t2 where t1.c1 = t2.c1) ssq, t1.c2
  from t1
  where t1.c2 > 10;
                    QUERY PLAN
  ------------------------------------------------------
   Streaming (type: GATHER)
     Node/s: All datanodes
     ->  Seq Scan on t1
           Filter: (c2 > 10)
           SubPlan 1
             ->  Result
                   Filter: (t1.c1 = t2.c1)
                   ->  Materialize
                         ->  Streaming(type: BROADCAST)
                               Spawn on: All datanodes
                               ->  Seq Scan on t2
  (11 rows)
  ```

  由于相关子查询出现在 targetlist(查询返回列表)里，对于 t1.c1=t2.c1 不匹配的场景仍然需要输出值，因此使用 left-outerjoin 关联 T1&T2 确保 t1.c1=t2.c1 在不匹配时，子 SSQ 能够返回不匹配的补空值。

说明

SSQ 和 CSSQ 的解释如下：

- SSQ：ScalarSubQuery 一般指返回 1 行 1 列 scalar 值的 sublink，简称 SSQ。
- CSSQ：Correlated-ScalarSubQuery 和 SSQ 相同不过是指包含相关条件的 SSQ。

上述 SQL 语句可以改写为：

```
with ssq as
(
   select t2.c2 from t2
)
select ssq.c2, t1.c2
from t1 left join ssq on t1.c1 = ssq.c2
where t1.c2 > 10;
```

改写后的执行计划为：

```
             QUERY PLAN
-----------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Hash Right Join
         Hash Cond: (t2.c2 = t1.c1)
         ->  Streaming(type: REDISTRIBUTE)
               Spawn on: All datanodes
               ->  Seq Scan on t2
         ->  Hash
               ->  Seq Scan on t1
                     Filter: (c2 > 10)
(10 rows)
```

可以看到出现在 SSQ 返回列表里的相关子查询 SSQ，已经被提升成 Right Join，从而避免当内表 T2 较大时出现 SubPlan+Broadcast 计划导致性能变差。

- 出现在 targetlist 里的相关子查询无法提升(带 count)

例如：

```
select (select count(*) from t2 where t2.c1=t1.c1) cnt, t1.c1, t3.c1
from t1,t3
where t1.c1=t3.c1 order by cnt, t1.c1;
```

执行计划为

```
                        QUERY PLAN
-----------------------------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Sort
         Sort Key: ((SubPlan 1)), t1.c1
         ->  Hash Join
               Hash Cond: (t1.c1 = t3.c1)
               ->  Seq Scan on t1
               ->  Hash
                     ->  Seq Scan on t3
               SubPlan 1
                 ->  Aggregate
                       ->  Result
                             Filter: (t2.c1 = t1.c1)
                             ->  Materialize
                                   ->  Streaming(type: BROADCAST)
                                         Spawn on: All datanodes
                                         ->  Seq Scan on t2
(17 rows)
```

由于相关子查询出现在 targetlist(查询返回列表)里，对于 t1.c1=t2.c1 不匹配的场景仍然需要输出值，因此使用 left-outerjoin 关联 T1&T2 确保 t1.c1=t2.c1 在不匹配时子 SSQ 能够返回不匹配的补空值，但是这里带了 count 语句及时在 t1.c1=t2.t1 不匹配时需要输出 0，因此可以使用一个 case-when NULL then 0 else count(*)来代替。

上述 SQL 语句可以改写为：

```
with ssq as
(
   select count(*) cnt, c1 from t2 group by c1
)
select case when
          ssq.cnt is null then 0
          else ssq.cnt
      end cnt, t1.c1, t3.c1
from t1 left join ssq on ssq.c1 = t1.c1,t3
where t1.c1 = t3.c1
order by ssq.cnt, t1.c1;
```

改写后的执行计划为

```
 QUERY PLAN
-----------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Sort
        Sort Key: (count(*)), t1.c1
        ->  Hash Join
             Hash Cond: (t1.c1 = t3.c1)
             ->  Hash Left Join
                  Hash Cond: (t1.c1 = t2.c1)
                  ->  Seq Scan on t1
                  ->  Hash
                       ->  HashAggregate
                            Group By Key: t2.c1
                            ->  Seq Scan on t2
             ->  Hash
                  ->  Seq Scan on t3
(15 rows)
```

- 相关条件为不等值场景

例如：

```
select t1.c1, t1.c2
from t1
where t1.c1 = (select agg() from t2.c2 > t1.c2);
```

对于非等值相关条件的 SubLink 目前无法提升，从语义上可以通过做 2 次 join（一次 CorrelationKey，一次 rownum 自关联）达到提升改写的目的。

改写方案有两种。

- 子查询改写方式

```
select t1.c1, t1.c2
from t1, (
   select t1.rowid, agg() aggref
   from t1,t2
   where t1.c2 > t2.c2 group by t1.rowid
```

```
) dt /* derived table */
where t1.rowid = dt.rowid AND t1.c1 = dt.aggref;
```

- CTE 改写方式

```
WITH dt as
(
    select t1.rowid, agg() aggref
    from t1,t2
    where t1.c2 > t2.c2 group by t1.rowid
)
select t1.c1, t1.c2
from t1, derived table
where t1.rowid = derived table.rowid AND
t1.c1 = derived_table.aggref;
```

须知

- 目前尚无高效的实现表、中间结果集的全局唯一 rowid，因此目前此类场景很难改写，建议通过业务层进行规避，或者可以使用 t1.xc_nodeid + t1.ctid 进行 rowid 关联，但是 xc_nodeid 的重复率较高会导致 join 关联效率变低，而 xc_node_id+ctid 类型无法作为 hashjoin 的关联条件。
- 对于 AGG 类型为 count(*)时需要进行 CASE-WHEN 对没有 match 的场景补 0 处理，非 COUNT(*)场景 NULL 处理。
- CTE 改写方式如果有 sharescan 支持性能上能够更优。

更多优化示例

示例 1：修改基表为 replicate 表，并且在过滤列上创建索引。

```
create table master table (a int);
create table sub table(a int, b int);
select a from master_table group by a having a in (select a from sub_table);
```

上述事例中存在一个相关性子查询，为了提升查询的性能，可以将 sub_table 修改为一个 relication 表，并且在字段 a 上创建一个 index。

示例 2：修改 select 语句，将子查询修改为和主表的 join，或者修改为可以提升的 subquery，但是在修改前后需要保证语义的正确性。

```
explain (costs off)select * from master table as t1 where t1.a in (select t2.a from
sub table as t2 where t1.a = t2.b);
                    QUERY PLAN
-------------------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Seq Scan on master table t1
        Filter: (SubPlan 1)
        SubPlan 1
          ->  Result
               Filter: (t1.a = t2.b)
               ->  Materialize
                    ->  Streaming(type: BROADCAST)
```

```
                        Spawn on: All datanodes
                        ->  Seq Scan on sub_table t2
(11 rows)
```

上面事例计划中存在一个 subPlan，为了消除这个 subPlan 可以修改语句为：

```
explain(costs off) select * from master_table as t1 where exists (select t2.a from
sub_table as t2 where t1.a = t2.b and t1.a = t2.a);
                QUERY PLAN
-----------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Hash Semi Join
        Hash Cond: (t1.a = t2.b)
        ->  Seq Scan on master_table t1
        ->  Hash
             ->  Streaming(type: REDISTRIBUTE)
                  Spawn on: All datanodes
                  ->  Seq Scan on sub_table t2
(9 rows)
```

从计划可以看出，subPlan 消除了，计划变成了两个表的 semi join，这样会大大提高执行效率。

3.2.3.6.4 统计信息调优

统计信息调优介绍

云数据库 GaussDB 是基于代价估算生成的最优执行计划。优化器需要根据 analyze 收集的统计信息进行行数估算和代价估算，因此统计信息对优化器行数估算和代价估算起着至关重要的作用。通过 analyze 收集全局统计信息，主要包括：pg_class 表中的 relpages 和 reltuples；pg_statistic 表中的 stadistinct、stanullfrac、stanumbersN、stavaluesN、histogram_bounds 等。

实例分析 1：未收集统计信息导致查询性能差

在很多场景下，由于查询中涉及到的表或列没有收集统计信息，会对查询性能有很大的影响。

表结构如下所示：

```
CREATE TABLE LINEITEM
(
L_ORDERKEY        BIGINT       NOT NULL
, L_PARTKEY       BIGINT       NOT NULL
, L_SUPPKEY       BIGINT       NOT NULL
, L_LINENUMBER    BIGINT       NOT NULL
, L_QUANTITY      DECIMAL(15,2) NOT NULL
, L_EXTENDEDPRICE  DECIMAL(15,2) NOT NULL
, L_DISCOUNT      DECIMAL(15,2) NOT NULL
, L_TAX           DECIMAL(15,2) NOT NULL
, L_RETURNFLAG    CHAR(1)      NOT NULL
, L_LINESTATUS    CHAR(1)      NOT NULL
, L_SHIPDATE      DATE         NOT NULL
, L_COMMITDATE    DATE         NOT NULL
```

```
, L_RECEIPTDATE    DATE         NOT NULL
, L_SHIPINSTRUCT   CHAR(25)      NOT NULL
, L_SHIPMODE      CHAR(10)      NOT NULL
, L_COMMENT       VARCHAR(44)   NOT NULL
) with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(L_ORDERKEY);

CREATE TABLE ORDERS
(
O_ORDERKEY       BIGINT       NOT NULL
, O_CUSTKEY      BIGINT       NOT NULL
, O_ORDERSTATUS   CHAR(1)      NOT NULL
, O_TOTALPRICE    DECIMAL(15,2) NOT NULL
, O_ORDERDATE     DATE NOT NULL
, O_ORDERPRIORITY CHAR(15)     NOT NULL
, O_CLERK        CHAR(15)     NOT NULL
, O_SHIPPRIORITY  BIGINT       NOT NULL
, O_COMMENT      VARCHAR(79)   NOT NULL
)with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(O_ORDERKEY);
```

查询语句如下所示：

```
explain verbose select
count(*) as numwait
from
lineitem l1,
orders
where
o orderkey = l1.l orderkey
and o orderstatus = 'F'
and l1.l receiptdate > l1.l commitdate
and not exists (
select
*
from
lineitem l3
where
l3.l orderkey = l1.l orderkey
and l3.l suppkey <> l1.l suppkey
and l3.l receiptdate > l3.l commitdate
)
order by
numwait desc;
```

当出现该问题时，可以通过如下方法确认查询中涉及到的表或列有没有做过 analyze 收集统计信息。

1. 通过 explain verbose 执行 query 分析执行计划时会提示 WARNING 信息，如下所示：

   ```
   WARNING:Statistics in some tables or columns(public.lineitem.l_receiptdate,
   public.lineitem.l_commitdate, public.lineitem.l_orderkey,
   public.lineitem.l_suppkey, public.orders.o_orderstatus,
   public.orders.o_orderkey) are not collected.
   HINT:Do analyze for them in order to generate optimized plan.
   ```

2. 可以通过在 pg_log 目录下的日志文件中查找以下信息来确认当前执行的 query 是否由于没有收集统计信息导致查询性能变差。

```
2017-06-14 17:28:30.336 CST 140644024579856 20971684 [BACKEND] LOG:Statistics
in some tables or columns(public.lineitem.l_receiptdate,
public.lineitem.l_commitdate, public.lineitem.l_orderkey, public.linei
tem.l_suppkey, public.orders.o_orderstatus, public.orders.o_orderkey) are not
collected.
2017-06-14 17:28:30.336 CST 140644024579856 20971684 [BACKEND] HINT:Do analyze
for them in order to generate optimized plan.
```

当通过以上方法查看到哪些表或列没有做 analyze，可以通过对 WARNING 或日志中上报的表或列做 analyze 来解决由于未收集统计信息导致查询变慢的问题。

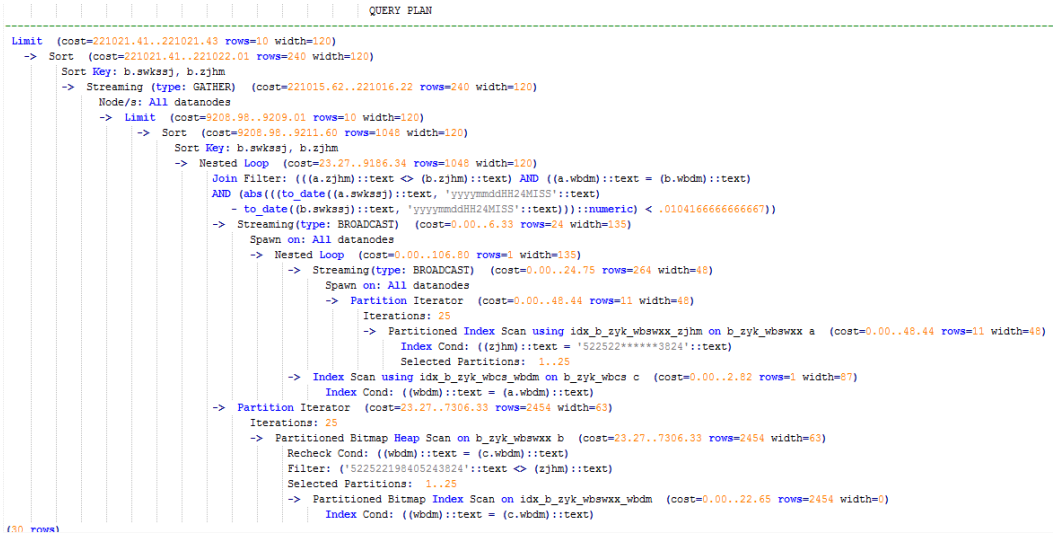
实例分析 3：多表 join 的复杂查询存在中间结果不准调优

现象描述：查询与指定人在前后 15 分钟内、同一网吧登记上网的人员信息：

```
SELECT
C.WBM,
C.DZQH,
C.DZ,
B.ZJHM,
B.SWKSSJ,
B.XWSJ
FROM
b_zyk_wbswxx A,
b_zyk_wbswxx B,
b_zyk_wbcs C
WHERE
A.ZJHM = '522522******3824'
AND A.WBDM = B.WBDM
AND A.WBDM = C.WBDM
AND abs(to date(A.SWKSSJ,'yyyymmddHH24MISS') - to date(B.SWKSSJ,'yyyymmddHH24MISS'))
< INTERVAL '15 MINUTES'
ORDER BY
B.SWKSSJ,
B.ZJHM
limit 10 offset 0
;
```

执行计划如图 3-7 所示。该查询实际耗时约 12 秒。

图3-7 应用 unlogged table 案例（一）



优化分析：分析过程如下：

1. 分析该执行计划发现，扫描节点已使用 Index Scan，耗时主要在最外层 Nest Loop Join 的 Join Filter 计算中，且该计算执行了字符串的加减法和不等值比较。
2. 考虑使用 unlogged table 保存目标人的上网信息，且在插入时处理上网开始时间和终止时间，以避免后续进行时间加减。

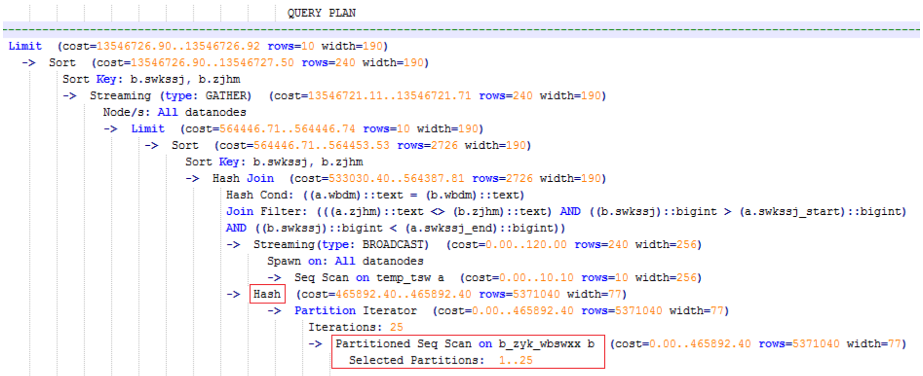
```
//创建临时 unlogged table
CREATE UNLOGGED TABLE temp tsw
(
ZJHM        NVARCHAR2(18),
WBDM        NVARCHAR2(14),
SWKSSJ START NVARCHAR2(14),
SWKSSJ END   NVARCHAR2(14),
WBM         NVARCHAR2(70),
DZQH        NVARCHAR2(6),
DZ          NVARCHAR2(70),
IPDZ        NVARCHAR2(39)
)
;
//插入目标人的上网记录，并处理上网开始和结束时间。
INSERT INTO
temp_tsw
SELECT
A.ZJHM,
A.WBDM,
to_char((to_date(A.SWKSSJ,'yyyymmddHH24MISS') - INTERVAL '15
MINUTES'),'yyyymmddHH24MISS'),
to_char((to_date(A.SWKSSJ,'yyyymmddHH24MISS') + INTERVAL '15
MINUTES'),'yyyymmddHH24MISS'),
B.WBM,B.DZQH,B.DZ,B.IPDZ
FROM
b_zyk_wbswxx A,
b_zyk_wbcs B
WHERE
```

```
A.ZJHM='522522******3824' AND A.WBDM = B.WBDM
;

//查询和目标人在前后十五分钟内在同一网吧上网的人员信息，比较大小时强制转换为 int8。
SELECT
A.WBM,
A.DZQH,
A.DZ,
A.IPDZ,
B.ZJHM,
B.XM,
to_date(B.SWKSSJ,'yyyymmddHH24MISS') as SWKSSJ,
to_date(B.XWSJ,'yyyymmddHH24MISS') as XWSJ,
B.SWZDH
FROM temp_tsw A,
b_zyk_wbswxx B
WHERE
A.ZJHM <> B.ZJHM
AND A.WBDM = B.WBDM
AND (B.SWKSSJ)::int8 > (A.swkssj_start)::int8
AND (B.SWKSSJ)::int8 < (A.swkssj_end)::int8
order by
B.SWKSSJ,
B.ZJHM
limit 10 offset 0
;
```

上述查询耗时约 7 秒，执行计划如图 3-8 所示。

图3-8 应用 unlogged table 案例（二）



```
QUERY PLAN
-----------------------------------------------------------------------------------------------
Limit  (cost=13546726.90..13546726.92 rows=10 width=190)
  ->  Sort  (cost=13546726.90..13546727.50 rows=240 width=190)
        Sort Key: b.swkssj, b.zjhm
        ->  Streaming (type: GATHER)  (cost=13546721.11..13546721.71 rows=240 width=190)
              Node/s: All datanodes
              ->  Limit  (cost=564446.71..564446.74 rows=10 width=190)
                    ->  Sort  (cost=564446.71..564453.53 rows=2726 width=190)
                          Sort Key: b.swkssj, b.zjhm
                          ->  Hash Join  (cost=533030.40..564387.81 rows=2726 width=190)
                                Hash Cond: ((a.wbdm)::text = (b.wbdm)::text)
                                Join Filter: (((a.zjhm)::text <> (b.zjhm)::text) AND ((b.swkssj)::bigint > (a.swkssj_start)::bigint)
                                AND ((b.swkssj)::bigint < (a.swkssj_end)::bigint))
                                ->  Streaming(type: BROADCAST)  (cost=0.00..120.00 rows=240 width=256)
                                      Spawn on: All datanodes
                                      ->  Seq Scan on temp_tsw a  (cost=0.00..10.10 rows=10 width=256)
                                ->  Hash  (cost=465892.40..465892.40 rows=5371040 width=77)
                                      ->  Partition Iterator  (cost=0.00..465892.40 rows=5371040 width=77)
                                            Iterations: 25
                                            ->  Partitioned Seq Scan on b_zyk_wbswxx b  (cost=0.00..465892.40 rows=5371040 width=77)
                                                  Selected Partitions:  1..25
```

3. 分析上述执行计划，发现执行了 Hash Join，对大表 b_zyk_wbswxx 建立了 Hash Table。由于该表数据量大，创建过程耗时较长。

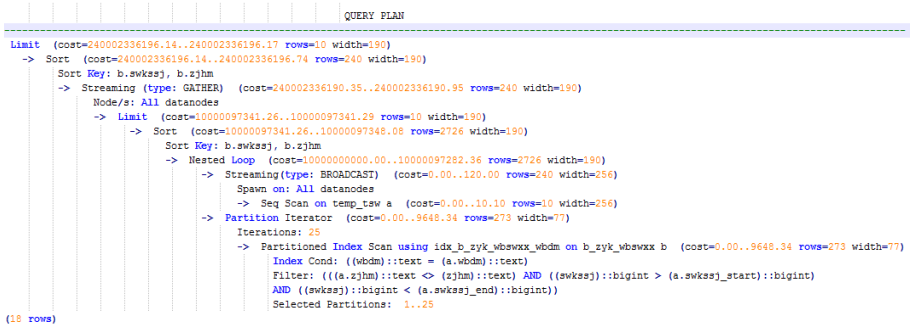
   由于 temp_tsw 中仅包含几百条记录，且 temp_tsw 和 b_zyk_wbswxx 均通过 wbdm（网吧代码）执行等值连接。因此，如果 Join 方式改为 Nest Loop Join，则扫描节点可以实现 Index Scan，性能预计将会提升。

4. 执行如下语句，将 Join 方式改为 Nest Loop Join。

```
SET enable_hashjoin = off;
```

执行计划如图 3-9 所示。查询耗时约 3 秒。

图3-9 应用 unlogged table 案例（三）



5. 使用 unlogged table 保存结果集并用于分页显示。

   如果需要在上层应用页面实现分页显示，需要修改 offset 值确定显示目标页的结果集。按此实现，每次翻页时均执行上面查询语句，耗时较长。

   为解决上述问题，建议使用 unlogged table 保存结果集。

```
//创建保存结果集的 unlogged table
CREATE UNLOGGED TABLE temp result
(
WBM      NVARCHAR2(70),
DZQH     NVARCHAR2(6),
DZ       NVARCHAR2(70),
IPDZ     NVARCHAR2(39),
ZJHM     NVARCHAR2(18),
XM       NVARCHAR2(30),
SWKSSJ   date,
XWSJ     date,
SWZDH    NVARCHAR2(32)
);

//将结果集插入 unlogged table，插入耗时约 3 秒。
INSERT INTO
temp_result
SELECT
A.WBM,
A.DZQH,
A.DZ,
A.IPDZ,
B.ZJHM,
B.XM,
to_date(B.SWKSSJ,'yyyymmddHH24MISS') as SWKSSJ,
to_date(B.XWSJ,'yyyymmddHH24MISS') as XWSJ,
B.SWZDH
FROM temp tsw A,
b zyk wbswxx B
WHERE
A.ZJHM <> B.ZJHM
AND A.WBDM = B.WBDM
AND (B.SWKSSJ)::int8 > (A.swkssj start)::int8
AND (B.SWKSSJ)::int8 < (A.swkssj end)::int8
;
```

```
//查询结果集表进行分页显示，分页查询耗时约 10ms。
SELECT
*
FROM
temp_result
ORDER BY
SWKSSJ,
ZJHM
LIMIT 10 OFFSET 0;
```

⚠ 注意

收集更准确的统计信息，通常会改善查询性能，但是也有可能使性能劣化。如果遇到性能劣化，可以考虑：

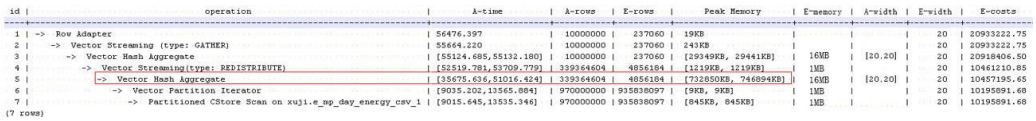
- 恢复默认的统计信息。
- 使用 plan hint 来调整到之前的查询计划。（详细参见 3.2.3.9 使用 Plan Hint 进行调优）

3.2.3.6.5 算子级调优

算子级调优介绍

一个查询语句要经过多个算子步骤才会输出最终的结果。由于各别算子耗时过长导致整体查询性能下降的情况比较常见。这些算子是整个查询的瓶颈算子。通用的优化手段是 EXPLAIN ANALYZE/PERFORMANCE 命令查看执行过程的瓶颈算子，然后进行针对性优化。

如下面的执行过程信息中，Hashagg 算子的执行时间占总时间的：(51016-13535)/ 56476 ≈66%，此处 Hashagg 算子就是这个查询的瓶颈算子，在进行性能优化时应当优先考虑此算子的优化。



id	operation	A-time	A-rows	E-rows	Peak Memory	E-memory	A-width	E-width	E-costs
1	-> Row Adapter	56476.397	10000000	237060	19KB			20	20933222.75
2	-> Vector Streaming (type: GATHER)	55664.220	10000000	237060	243KB			20	20933222.75
3	-> Vector Hash Aggregate	[55124.685,55132.180]	10000000	237060	[29349KB, 29441KB]	16MB	[20,20]	20	20918406.50
4	-> Vector Streaming(type: REDISTRIBUTE)	[52519.781,53709.779]	339364604	4856184	[1219KB, 1219KB]	1MB		20	10461210.85
5	-> Vector Hash Aggregate	[35675.636,51016.424]	339364604	4856184	[732850KB, 746894KB]	16MB	[20,20]	20	10457195.65
6	-> Vector Partition Iterator	[9035.202,13565.864]	970000000	935838097	[9KB, 9KB]	1MB		20	10195891.68
7	-> Partitioned CStore Scan on xuji.e_mp_day_energy_csv_1	[9015.645,13535.346]	970000000	935838097	[845KB, 845KB]	1MB		20	10195891.68

(7 rows)

算子级调优示例

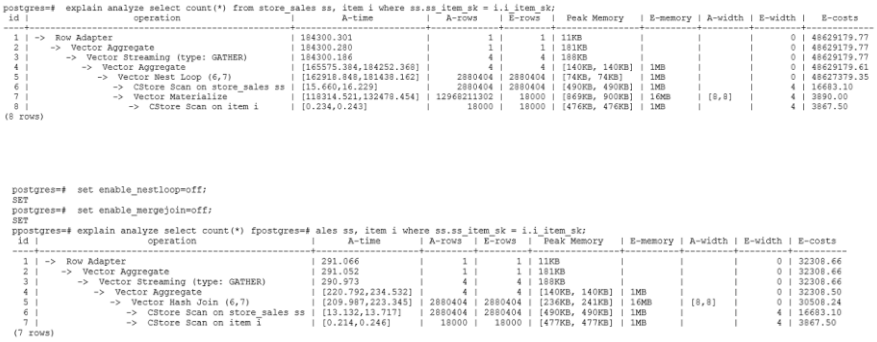
示例 1：基表扫描时，对于点查或者范围扫描等过滤大量数据的查询，如果使用 SeqScan 全表扫描会比较耗时，可以在条件列上建立索引选择 IndexScan 进行索引扫描提升扫描效率。

```
postgres=#  explain (analyze on, costs off) select * from store sales where
ss sold date sk = 2450944;
 id |           operation          |      A-time       | A-rows | Peak Memory  | A-
width
----+------------------------------+-------------------+--------+--------------
+---------
  1 | ->  Streaming (type: GATHER)   | 3666.020          |   3360 | 195KB        |
  2 |    ->  Seq Scan on store sales | [3594.611,3594.611] |   3360 | [34KB, 34KB] |
(2 rows)
```

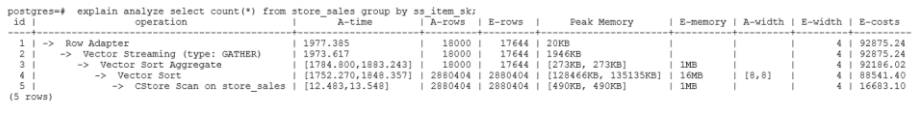
```
 Predicate Information (identified by plan id)
-----------------------------------------------
   2 --Seq Scan on store_sales
         Filter: (ss_sold_date_sk = 2450944)
         Rows Removed by Filter: 4968936
postgres=#  create index idx on store_sales_row(ss_sold_date_sk);
CREATE INDEX
postgres=#  explain (analyze on, costs off) select * from store_sales_row where
ss_sold_date_sk = 2450944;
 id |                  operation                  |     A-time      | A-rows | Peak
Memory  | A-width
----+---------------------------------------------+-----------------+--------+--
------------+----------
  1 | ->  Streaming (type: GATHER)                | 81.524          |   3360 | 195KB
|
  2 |    ->  Index Scan using idx on store_sales_row | [13.352,13.352] |   3360 |
[34KB, 34KB] |
(2 rows)
```

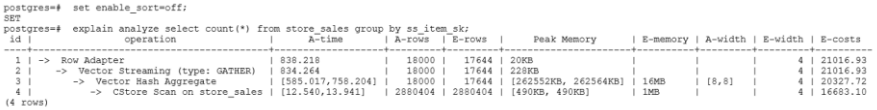
上述例子中，全表扫描返回 3360 条数据，过滤掉大量数据，在 ss_sold_date_sk 列上建立索引后，使用 IndexScan 扫描效率显著提高，从 3.6 秒提升到 13 毫秒。

示例 2：如果从执行计划中看，两表 join 选择了 NestLoop，而实际行数比较大时，NestLoop Join 可能执行比较慢。如下的例子中 NestLoop 耗时 181 秒，如果设置参数 enable_mergejoin=off 关掉 Merge Join，同时设置参数 enable_nestloop=off 关掉 NestLoop，让优化器选择 HashJoin，则 Join 耗时提升至 200 多毫秒。



示例 3：通常情况下 Agg 选择 HashAgg 性能较好，如果大结果集选择了 Sort+GroupAgg，则需要设置 enable_sort=off，HashAgg 耗时明显优于 Sort+GroupAgg。





3.2.3.6.6 数据倾斜调优

数据倾斜问题是分布式架构的重要难题，它破坏了 MPP 架构中各个节点对等的要求，导致单节点（倾斜节点）所存储或者计算的数据量远大于其他节点，所以会造成以下危害：

- 存储上的倾斜会严重限制系统容量，在系统容量不饱和的情况下，由于单节点倾斜的限制，使得整个系统容量无法继续增长。
- 计算上的倾斜会严重影响系统性能，由于倾斜节点所需要运算的数据量远大于其它节点，导致倾斜节点降低系统整体性能。
- 数据倾斜还严重影响了 MPP 架构的扩展性。由于在存储或者计算时，往往会将相同值的数据放到同一节点，因此当倾斜数据（大量数据的值相同）出现之后，即使我们增加节点，系统瓶颈仍然受限于倾斜节点的容量或者性能。

云数据库 GaussDB 数据库针对数据倾斜问题给出了完整的解决方案，包括存储倾斜和计算倾斜两大问题，下面分别进行介绍。

存储层数据倾斜

GaussDB 数据库中，数据分布存储在各个 DN 上，通过分布式执行提高查询的效率。但是，如果数据分布存在倾斜，则会导致分布式执行某些 DN 成为瓶颈，影响查询性能。这种情况通常是由于分布列选择不合理，可以通过调整分布列的方式解决。

例如下例：

```
postgres=# explain performance select count(*) from inventory;
5 --CStore Scan on lmz.inventory
       dn_6001_6002 (actual time=0.444..83.127 rows=42000000 loops=1)
       dn_6003_6004 (actual time=0.512..63.554 rows=27000000 loops=1)
       dn_6005_6006 (actual time=0.722..99.033 rows=45000000 loops=1)
       dn_6007_6008 (actual time=0.529..100.379 rows=51000000 loops=1)
       dn_6009_6010 (actual time=0.382..71.341 rows=36000000 loops=1)
       dn_6011_6012 (actual time=0.547..100.274 rows=51000000 loops=1)
       dn_6013_6014 (actual time=0.596..118.289 rows=60000000 loops=1)
       dn_6015_6016 (actual time=1.057..132.346 rows=63000000 loops=1)
       dn_6017_6018 (actual time=0.940..110.310 rows=54000000 loops=1)
       dn_6019_6020 (actual time=0.231..41.198 rows=21000000 loops=1)
       dn_6021_6022 (actual time=0.927..114.538 rows=54000000 loops=1)
       dn_6023_6024 (actual time=0.637..118.385 rows=60000000 loops=1)
       dn_6025_6026 (actual time=0.288..32.240 rows=15000000 loops=1)
       dn_6027_6028 (actual time=0.566..118.096 rows=60000000 loops=1)
       dn_6029_6030 (actual time=0.423..82.913 rows=42000000 loops=1)
       dn_6031_6032 (actual time=0.395..78.103 rows=39000000 loops=1)
       dn_6033_6034 (actual time=0.376..51.052 rows=24000000 loops=1)
       dn_6035_6036 (actual time=0.569..79.463 rows=39000000 loops=1)
```

在 performance 信息中，可以看到 inventory 表各 DN 的 scan 行数，发现各 DN 的行数差距较大，最大的为 63000000，最小的只有 15000000，差了 4 倍。这个差距对于数据扫描的性能影响还可以接受，但如果上层有 join 算子，则影响较大。

通常，数据表在各 DN 上是 hash 分布的，因此分布列的选择很重要。通过 table_skewness()来查看上述 inventory 表在各 DN 的数据分布倾斜，查询结果如下：

```
postgres=# select table_skewness('inventory');
            table_skewness
-----------------------------------------
 ("dn_6015_6016       ",63000000,8.046%)
 ("dn_6013_6014       ",60000000,7.663%)
 ("dn_6023_6024       ",60000000,7.663%)
 ("dn_6027_6028       ",60000000,7.663%)
 ("dn_6017_6018       ",54000000,6.897%)
 ("dn_6021_6022       ",54000000,6.897%)
 ("dn_6007_6008       ",51000000,6.513%)
 ("dn_6011_6012       ",51000000,6.513%)
 ("dn_6005_6006       ",45000000,5.747%)
 ("dn_6001_6002       ",42000000,5.364%)
 ("dn_6029_6030       ",42000000,5.364%)
 ("dn_6031_6032       ",39000000,4.981%)
 ("dn_6035_6036       ",39000000,4.981%)
 ("dn_6009_6010       ",36000000,4.598%)
 ("dn_6003_6004       ",27000000,3.448%)
 ("dn_6033_6034       ",24000000,3.065%)
 ("dn_6019_6020       ",21000000,2.682%)
 ("dn_6025_6026       ",15000000,1.916%)
(18 rows)
```

通过查询建表定义，可以发现，目前该表是以 inv_date_sk 作为分布列的，导致存在倾斜。通过查看各列的数据分布情况，改为 inv_item_sk 作为分布列，则倾斜情况分布如下：

```
postgres=# select table_skewness('inventory');
            table_skewness
-----------------------------------------
 ("dn_6001_6002       ",43934200,5.611%)
 ("dn_6007_6008       ",43829420,5.598%)
 ("dn_6003_6004       ",43781960,5.592%)
 ("dn_6031_6032       ",43773880,5.591%)
 ("dn_6033_6034       ",43763280,5.589%)
 ("dn_6011_6012       ",43683600,5.579%)
 ("dn_6013_6014       ",43551660,5.562%)
 ("dn_6027_6028       ",43546340,5.561%)
 ("dn_6009_6010       ",43508700,5.557%)
 ("dn_6023_6024       ",43484540,5.554%)
 ("dn_6019_6020       ",43466800,5.551%)
 ("dn_6021_6022       ",43458500,5.550%)
 ("dn_6017_6018       ",43448040,5.549%)
 ("dn_6015_6016       ",43247700,5.523%)
 ("dn_6005_6006       ",43200240,5.517%)
 ("dn_6029_6030       ",43181360,5.515%)
 ("dn_6025_6026       ",43179700,5.515%)
 ("dn_6035_6036       ",42960080,5.487%)
(18 rows)
```

数据分布倾斜的问题得到解决。

计算层数据倾斜

即使通过修改表的分布键，使得数据存储在各个节点上是均衡的，但是在执行查询的过程中，仍然可能出现数据倾斜的问题。在运算过程中某个算子在 DN 上输出的结果集出现倾斜，从而导致此算子上层的运算出现计算倾斜。一般来说，这是由于在执行过程中，数据重分布导致的。

在查询执行的过程中，join key、group by key 等往往不是表的分布列，因此需要按照 join key、group by key 上数据的 hash 值，让数据在各个 DN 之间进行重新分布，这个过程对应于计划中的 Redistribute 算子。当重分布列上的数据存在倾斜时，就会导致运行时的数据倾斜，即重分布后部分节点的数据远远大于其他。倾斜节点需要处理更多的数据，导致倾斜节点的计算性能远远低于其他节点。

如下例中，s 表和 t 表 join，join 条件中的 s.x 和 t.x 均不是表的分布列，因此需要重分布（REDISTRIBUTE 算子）。其中 s.x 列上存在倾斜值，t.x 上不存在倾斜。id=6 的 stream 算子在 datanode2 节点输出的结果集是其他 DN 的 3 倍，从而导致了计算倾斜。

```
select * from skew s,test t where s.x = t.x order by s.a limit 1;
 id |                  operation                   |       A-time
----+----------------------------------------------+----------------------
  1 | ->  Limit                                    | 52622.382
  2 |    ->  Streaming (type: GATHER)                 | 52622.374
  3 |      ->  Limit                               | [30138.494,52598.994]
  4 |        ->  Sort                              | [30138.486,52598.986]
  5 |          ->  Hash Join (6,8)                   | [30127.013,41483.275]
  6 |            ->  Streaming(type: REDISTRIBUTE)    | [11365.110,22024.845]
  7 |              ->  Seq Scan on public.skew s     | [2019.168,2175.369]
  8 |            ->  Hash                          | [2460.108,2499.850]
  9 |              ->  Streaming(type: REDISTRIBUTE) | [1056.214,1121.887]
 10 |                ->  Seq Scan on public.test t  | [310.848,325.569]
(10 rows)
 6 --Streaming(type: REDISTRIBUTE)
       datanode1 (rows=5050368)
       datanode2 (rows=15276032)
       datanode3 (rows=5174272)
       datanode4 (rows=5219328)
```

和存储倾斜相比，计算倾斜更难以提前识别，因此云数据库 GaussDB 提出了 RLBT(Runtime Load Balance Technology)方案，用以解决运行时的计算倾斜问题。RLBT 方案主要分为两个层面，第一步是计算倾斜识别，第二步是计算倾斜解决。下面分别进行介绍。

1. 倾斜识别

   计算倾斜的识别，即预先识别计算过程中的重分布列是否存在倾斜数据。RLBT 方案中给出了三个解决手段，统计信息识别，hint 方式指定以及规则识别：

   - 统计信息识别

     需要用户先执行 analyze 收集各表的统计信息，然后优化器能够自动利用统计信息对重分布键上的倾斜数据进行提前识别，对于存在倾斜的查询，生成相应的优化计划。在重分布键有多列的情况，只有所有列都属于同一个基表才能利用统计信息进行识别。

统计信息只能给出基表的倾斜情况，当基表某一列存在倾斜，其他列上带有过滤条件，或者经过和其他表的 join 之后，我们无法准确判断倾斜列上倾斜数据是否依旧存在。

- hint 方式指定

  统计信息有着一定的局限性，对于较为复杂的查询，其中间结果难以通过统计信息进行估算和识别倾斜数据。对于这种情况，我们设计了 hint 手段，通过用户手动指定的方式，给定倾斜信息。优化器根据用户给定的倾斜信息，来对查询进行优化。详细 hint 使用语法参见 3.2.3.9.8 运行倾斜的 hint。

- 规则识别

  现在 BI 系统往往会产生大量带有 outer join（left join、right join、full join）的 SQL，outer join 在匹配失败的情况下会补空产生大量 NULL 值，如果接下来在补空列上进行 join 或者 group by 操作，就会导致 NULL 值倾斜。当前 RLBT 技术会自动识别这种场景，并生成相应的 NULL 值倾斜优化计划。

2. 计算倾斜解决

在解决倾斜时，目前针对最常见的 join 和 agg 算子进行了优化。

- join 优化

基本思路是将倾斜数据和非倾斜数据进行隔离处理。主要分为以下三种情况：

a. join 两侧都需要做重分布：

对倾斜侧做 PART_REDISTRIBUTE_PART_ROUNDROBIN，其中对倾斜数据做 roundrobin，非倾斜数据做 redistribute；

对非倾斜侧做 PART_REDISTRIBUTE_PART_BROADCAST，其中对倾斜数据做 broadcast，非倾斜数据做 redistribute；

b. join 一侧需要重分布，另一侧不需要重分布：

对需要重分布的一侧做 PART_REDISTRIBUTE_PART_ROUNDROBIN；

对不需要重分布的一侧做 PART_LOCAL_PART_BROADCAST，其中对等于倾斜值的部分做 broadcast，其余数据保留在本地。

c. 对于有补 NULL 值的表：

对该表做 PART_REDISTERIBUTE_PART_LOCAL，其中将 NULL 值保留在本地，其余数据做 redistribute。

以前面的查询为例，s.x 列上存在倾斜数据，倾斜数据的值为 0。优化器通过统计信息，识别到了该倾斜数据，生成了倾斜优化计划如下：

```
 id |                          operation                          |       A-time
----+-------------------------------------------------------------+----------------------
  1 | -> Limit                                                    | 23642.049
  2 |    -> Streaming (type: GATHER)                              | 23642.041
  3 |       -> Limit                                              | [23310.768,23618.021]
  4 |          -> Sort                                            | [23310.761,23618.012]
  5 |             -> Hash Join (6,8)                              | [20898.341,21115.272]
  6 |                -> Streaming(type: PART REDISTRIBUTE PART ROUNDROBIN)   |
```

```
[7125.834,7472.111]
 7 |               ->  Seq Scan on public.skew s                     |
[1837.079,1911.025]
 8 |             ->  Hash                                          |
[2612.484,2640.572]
 9 |               ->  Streaming(type: PART REDISTRIBUTE PART BROADCAST) |
[1193.548,1297.894]
10 |                 ->  Seq Scan on public.test t                  |
[314.343,328.707]
(10 rows)
  5 --Vector Hash Join (6,8)
       Hash Cond: s.x = t.x
       Skew Join Optimizated by Statistic
  6 --Streaming(type: PART REDISTRIBUTE PART ROUNDROBIN)
       datanode1 (rows=7635968)
       datanode2 (rows=7517184)
       datanode3 (rows=7748608)
       datanode4 (rows=7818240)
```

上述执行计划中，可以看到 Skew Join Optimizated by Statistic 的字样，代表该计划为倾斜优化计划，其中 Statistic 关键字代表该倾斜优化来自于统计信息，除此之外还有 Hint 和 Rule，分别代表倾斜优化来自于 hint 语句和规则。对比前面的计划可以看到，这里对于非倾斜数据和倾斜数据做了分别处理。对于 s 表中的非倾斜数据，依旧按照原有的方案，根据数据的 hash 值进行重分布；而对于倾斜数据（即等于 0 的数据），则通过轮询发送的方式，均衡地发送到所有节点。通过这样的方式，解决了倾斜数据分布不均衡的问题。

同时，为了保证结果的正确性，需要对 t 表做相应的处理。对于 t 表中等于 0（s.x 表中的倾斜值）的数据做广播，对于其他数据，依旧根据数据的 hash 值进行重分布。

通过这样的方式，就解决了 join 操作中，数据倾斜的问题。从上面的结果来看，id=6 的 stream 算子各个 DN 的输出结果已经非常均衡，同时查询端到端性能提升了 1 倍。

- agg 优化

对于 agg 操作，解决倾斜的思路与 join 操作不同，这里是通过首先在本 DN 内按照 group by key 进行去重操作，然后再进行重分布。因为经过 DN 内部去重之后，从全局来看，每个值的数量都不会超过 DN 数，因此不会出现严重的数据倾斜问题。以如下 query 为例：

select c1, c2, c3, c4, c5, c6, c7, c8, c9, count(*) from t group by c1, c2, c3, c4, c5, c6, c7, c8, c9 limit 10;

原执行结果如下：

```
id |              operation               |        A-time        | A-rows    ---
-+---------------------------------------+----------------------+-------
---    1 | ->  Streaming (type: GATHER)            | 130621.783           |
12     2 |    ->  GroupAggregate                  | [85499.711,130432.341] |
12     3 |      ->  Sort                          | [85499.509,103145.632] |
36679237     4 |        ->  Streaming(type: REDISTRIBUTE) | [25668.897,85499.050]
| 36679237     5 |           ->  Seq Scan on public.t      | [9835.069,10416.388]
| 36679237   (5 rows)       4 --Streaming(type: REDISTRIBUTE)         datanode1
(rows=36678837)          datanode2 (rows=100)          datanode3 (rows=100)
datanode4 (rows=200)
```

其中存在大量倾斜数据，导致数据按照 group by key 进行重分布之后，datanode1 的数据量是其他节点的数十万倍。在倾斜优化之后，首先在本 DN 进行一次 group by 操作，达到数据去重的效果，然后再进行重分布，可以发现基本没有数据倾斜的问题出现。

```
id |              operation               |       A-time          ----+--------
-----------------------------------+----------------------    1 | ->
Streaming (type: GATHER)             | 10961.337              2 |    ->
HashAggregate                    | [10953.014,10953.705]     3 |      ->
HashAggregate                 | [10952.957,10953.632]     4 |         ->
Streaming(type: REDISTRIBUTE) | [10952.859,10953.502]     5 |           ->
HashAggregate            | [10084.280,10947.139]     6 |              ->  Seq
Scan on public.t    | [4757.031,5201.168]     (6 rows)     Predicate Information
(identified by plan id)  ------------------------------------------------    3
--HashAggregate         Skew Agg Optimized by Statistic  (2 rows)      4 --
Streaming(type: REDISTRIBUTE)         datanode1 (rows=17)         datanode2
(rows=8)         datanode3 (rows=8)         datanode4 (rows=14)
```

适用范围

- join 算子
  - 支持 nest loop，merge join，hash join 等 join 方式；
  - 当倾斜数据处于 join 的 left 侧时，支持 inner join，left join，semi join，anti join；当倾斜属于位于 join 的 right 侧时，支持 inner join，right join，right semi join，right anti join。
  - 通过统计信息得到的倾斜优化计划，优化器会根据代价判断该计划是否为最优计划。通过 hint 和规则会强制生成倾斜优化计划。
- agg 算子
  - array_agg、string_agg、subplan in agg qual 这几种场景不支持优化；
  - 通过统计信息识别到的倾斜优化计划会受到代价、plan_mode_seed 参数、best_agg_plan 参数影响，而 hint、规则识别到的不会。

3.2.3.7 经验总结：SQL 语句改写规则

根据数据库的 SQL 执行机制以及大量的实践，总结发现：通过一定的规则调整 SQL 语句，在保证结果正确的基础上，能够提高 SQL 执行效率。如果遵守这些规则，常常能够大幅度提升业务查询效率。

- **使用 union all 代替 union**

  union 在合并两个集合时会执行去重操作，而 union all 则直接将两个结果集合并、不执行去重。执行去重会消耗大量的时间，因此，在一些实际应用场景中，如果通过业务逻辑已确认两个集合不存在重叠，可用 union all 替代 union 以便提升性能。

- **join 列增加非空过滤条件**

  若 join 列上的 NULL 值较多，则可以加上 is not null 过滤条件，以实现数据的提前过滤，提高 join 效率。

- **not in 转 not exists**

  not in 语句需要使用 nestloop anti join 来实现，而 not exists 则可以通过 hash anti join 来实现。在 join 列不存在 null 值的情况下，not exists 和 not in 等价。因此在

确保没有 null 值时，可以通过将 not in 转换为 not exists，通过生成 hash join 来提升查询效率。

如下所示，如果 t2.d2 字段中没有 null 值(t2.d2 字段在表定义中 not null)查询可以修改为

SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);

产生的计划如下：

图3-10 not exists 执行计划

```
 id |                  operation
----+-----------------------------------------------
  1 | ->  Streaming (type: GATHER)
  2 |    ->  Hash Anti Join (3, 4)
  3 |       ->  Seq Scan on t1
  4 |       ->  Hash
  5 |          ->  Streaming(type: REDISTRIBUTE)
  6 |             ->  Seq Scan on t2
(6 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   2 --Hash Anti Join (3, 4)
         Hash Cond: (t1.c1 = t2.d2)
(2 rows)
```

- **选择 hashagg。**

  查询中 GROUP BY 语句如果生成了 groupagg+sort 的 plan 性能会比较差，可以通过加大 work_mem 的方法生成 hashagg 的 plan，因为不用排序而提高性能。

- **尝试将函数替换为 case 语句。**

  云数据库 GaussDB 函数调用性能较低，如果出现过多的函数调用导致性能下降很多，可以根据情况把可下推函数的函数改成 CASE 表达式。

- **避免对索引使用函数或表达式运算。**

  对索引使用函数或表达式运算会停止使用索引转而执行全表扫描。

- **尽量避免在 where 子句中使用!=或<>操作符、null 值判断、or 连接、参数隐式转换。**

- **对复杂 SQL 语句进行拆分。**

  对于过于复杂并且不易通过以上方法调整性能的 SQL 可以考虑拆分的方法，把 SQL 中某一部分拆分成独立的 SQL 并把执行结果存入临时表，拆分常见的场景包括但不限于：

  - 作业中多个 SQL 有同样的子查询，并且子查询数据量较大。
  - Plan cost 计算不准，导致子查询 hash bucket 太小，比如实际数据 1000W 行，hash bucket 只有 1000。
  - 函数（如 substr,to_number）导致大数据量子查询选择度计算不准。
  - 多 DN 环境下对大表做 broadcast 的子查询。

3.2.3.8 SQL 调优关键参数调整

本节将介绍影响云数据库 GaussDB SQL 调优性能的关键 CN 配置参数。

表3-2 CN 配置参数

参数/参考值	描述
enable_nestloop=on	控制查询优化器对嵌套循环连接（Nest Loop Join）类型的使用。当设置为“on”后，优化器优先使用 Nest Loop Join；当设置为“off”后，优化器在存在其他方法时将优先选择其他方法。 **说明** 如果只需要在当前数据库连接（即当前 Session）中临时更改该参数值，则只需要在 SQL 语句中执行如下命令： SET enable_nestloop to off; 实际调优中应根据情况选择是否关闭。一般情况下，在三种 join 方式（Nested Loop、Merge Join 和 Hash Join）里，Nested Loop 适合小数据量或者有索引的场景，Hash Join 适合大数据分析场景。
enable_bitmapscan=on	控制查询优化器对位图扫描规划类型的使用。设置为“on”，表示使用；设置为“off”，表示不使用。 **说明** 如果只需要在当前数据库连接（即当前 Session）中临时更改该参数值，则只需要在 SQL 语句中执行命令如下命令： SET enable_bitmapscan to off; bitmapscan 扫描方式适用于“where a > 1 and b > 1”且 a 列和 b 列都有索引这种查询条件，但有时其性能不如 indexscan。因此，现场调优如发现查询性能较差且计划中有 bitmapscan 算子，可以关闭 bitmapscan，看性能是否有提升。
enable_fast_query_shipping=on	控制查询优化器是否使用分布式框架，执行快速执行计划。设置为“on”，表示执行计划在 CN 和 DN 上各自生成；设置为“off”，表示使用分布式框架，即执行计划在 CN 上生成，然后发送到 DN 中执行。 **说明** 如果只需要在当前数据库连接（即当前 Session）中临时更改该参数值，则只需要在 SQL 语句中执行如下命令： SET enable_fast_query_shipping to off;
enable_hashagg=on	控制优化器对 Hash 聚集规划类型的使用。
enable_hashjoin=on	控制优化器对 Hash 连接规划类型的使用。
enable_mergejoin=on	控制优化器对融合连接规划类型的使用。
enable_indexscan=on	控制优化器对索引扫描规划类型的使用。
enable_indexonlyscan	控制优化器对仅索引扫描规划类型的使用。

参数/参考值	描述
=on	
enable_seqscan=on	控制优化器对顺序扫描规划类型的使用。完全消除顺序扫描是不可能的，但是关闭这个变量会让优化器在存在其他方法的时候优先选择其他方法。
enable_sort=on	控制优化器使用的排序步骤。该设置不可能完全消除明确的排序，但是关闭这个变量可以让优化器在存在其他方法的时候优先选择其他方法。
enable_broadcast=on	控制查询优化器对于 broadcast 广播模式数据传输的使用。此方式网络传输数据量较大，因此当网络传输节点（Stream）实际数据量较大而估算不准时，可以将该参数设置为 off，看性能是否有提升。
rewrite_rule	控制优化器是否启用 LAZYAGG\MAGICSET\PARTIALPUSH\UNIQUECHECK\DISABLEREP\INTARGETLIST\PREDPUSH 重写规则。

3.2.3.9 使用 Plan Hint 进行调优

3.2.3.9.1 Plan Hint 调优概述

Plan Hint 为用户提供了直接影响执行计划生成的手段，用户可以通过指定 join 顺序，join、stream、scan 方法，指定结果行数，指定重分布过程中的倾斜信息等多个手段来进行执行计划的调优，以提升查询的性能。

功能描述

Plan Hint 仅支持在 SELECT 关键字后通过如下形式指定：

```
/*+ <plan hint>*/
```

可以同时指定多个 hint，之间使用空格分隔。hint 只能 hint 当前层的计划，对于子查询计划的 hint，需要在子查询的 select 关键字后指定 hint。

例如：

```
select /*+ <plan hint1> <plan hint2> */ * from t1, (select /*+ <plan hint3> */ from t2) where 1=1;
```

其中<plan_hint1>，<plan_hint2>为外层查询的 hint，<plan_hint3>为内层子查询的 hint。

须知

如果在视图定义（CREATE VIEW）时指定 hint，则在该视图每次被应用时会使用该 hint。

说明

当使用 random plan 功能（参数 plan_mode_seed 不为 0）时，查询指定的 plan hint 不会被使用。

支持范围

当前版本 Plan Hint 支持的范围如下，后续版本会进行增强。

- 指定 Join 顺序的 Hint - leading hint
- 指定 Join 方式的 Hint，仅支持除 semi/anti join，unique plan 之外的常用 hint。
- 指定结果集行数的 Hint
- 指定 Stream 方式的 Hint
- 指定 Scan 方式的 Hint，仅支持常用的 tablescan，indexscan 和 indexonlyscan 的 hint。
- 指定子链接块名的 Hint
- 指定倾斜信息的 Hint，仅支持 Join 与 HashAgg 的重分布过程倾斜。

注意事项

- 不支持 Agg、Sort、Setop 和 Subplan 的 hint。
- 不支持 SMP 和 Node Group 场景下的 Hint。

示例

本章节使用同一个语句进行示例，便于 Plan Hint 支持的各方法作对比，示例语句及不带 hint 的原计划如下所示：

```
explain
select i_product_name product_name
,i_item_sk item_sk
,s_store_name store_name
,s_zip store_zip
,ad2.ca_street_number c_street_number
,ad2.ca_street_name c_street_name
,ad2.ca_city c_city
,ad2.ca_zip c_zip
,count(*) cnt
,sum(ss_wholesale_cost) s1
,sum(ss_list_price) s2
,sum(ss_coupon_amt) s3
FROM   store_sales
,store_returns
,store
,customer
,promotion
,customer address ad2
,item
WHERE  ss store sk = s store sk AND
ss customer sk = c customer sk AND
ss item sk = i item sk and
ss item sk = sr item sk and
ss_ticket_number = sr_ticket_number and
```

```
c_current_addr_sk = ad2.ca_address_sk and
ss_promo_sk = p_promo_sk and
i_color in ('maroon','burnished','dim','steel','navajo','chocolate') and
i_current_price between 35 and 35 + 10 and
i_current_price between 35 + 1 and 35 + 15
group by i_product_name
,i_item_sk
,s_store_name
,s_zip
,ad2.ca_street_number
,ad2.ca_street_name
,ad2.ca_city
,ad2.ca_zip
;
```

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |  E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+------------
  1 | ->  Row Adapter                                                          |          6 |          |     273 | 3401632.49
  2 |    ->  Vector Streaming (type: GATHER)                                   |          6 |          |     273 | 3401632.49
  3 |       ->  Vector Hash Aggregate                                          |          6 | 16MB     |     273 | 3401630.82
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |          6 | 1MB      |     169 | 3401630.78
  5 |             ->  Vector Hash Join (6,21)                                  |          6 | 16MB     |     169 | 3401630.42
  6 |                ->  Vector Hash Join (7,20)                               |          7 | 43MB     |     173 | 3400343.15
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)               |          7 | 1MB      |     123 | 3395775.64
  8 |                      ->  Vector Hash Join (9,19)                         |          7 | 27MB     |     123 | 3395775.48
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)         |          7 | 1MB      |     123 | 3386294.72
 10 |                            ->  Vector Hash Join (11,18)                  |          7 | 16MB     |     123 | 3386294.56
 11 |                               ->  Vector Hash Join (12,14)               |          7 | 19MB     |     112 | 3384018.02
 12 |                                  ->  Vector Partition Iterator           |  287999764 | 1MB      |      12 | 227383.99
 13 |                                     ->  Partitioned CStore Scan on store_returns |  287999764 | 1MB      |      12 | 227383.99
 14 |                                  ->  Vector Hash Join (15,17)            |    1516824 | 16MB     |     124 | 3065686.08
 15 |                                     ->  Vector Partition Iterator        | 2879987999 | 1MB      |      66 | 2756066.50
 16 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB      |      66 | 2756066.50
 17 |                                     ->  CStore Scan on item              |        158 | 1MB      |      58 | 4051.25
 18 |                               ->  CStore Scan on store                   |      24048 | 1MB      |      19 | 2264.00
 19 |                            ->  CStore Scan on customer                   |   12000000 | 1MB      |       8 | 12923.00
 20 |                         ->  CStore Scan on customer_address ad2          |    6000000 | 1MB      |      58 | 5770.00
 21 |                      ->  CStore Scan on promotion                        |      36000 | 1MB      |       4 | 1268.50
(21 rows)
```

3.2.3.9.2 Join 顺序的 Hint

功能描述

指明 join 的顺序，包括内外表顺序。

语法格式

- 仅指定 join 顺序，不指定内外表顺序。

```
leading(join_table_list)
```

- 同时指定 join 顺序和内外表顺序，内外表顺序仅在最外层生效。

```
leading((join_table_list))
```

参数说明

join_table_list 为表示表 join 顺序的 hint 字符串，可以包含当前层的任意个表（别名），或对于子查询提升的场景，也可以包含子查询的 hint 别名，同时任意表可以使用括号指定优先级，表之间使用空格分隔。

须知

表只能用单个字符串表示，不能带 schema。

说明

表如果存在别名，需要优先使用别名来表示该表。

join table list 中指定的表需要满足以下要求，否则会报语义错误。

- list 中的表必须在当前层或提升的子查询中存在。
- list 中的表在当前层或提升的子查询中必须是唯一的。如果不唯一，需要使用不同的别名进行区分。
- 同一个表只能在 list 里出现一次。
- 如果表存在别名，则 list 中的表需要使用别名。

例如：

leading(t1 t2 t3 t4 t5)表示：t1, t2, t3, t4, t5 先 join，五表 join 顺序及内外表不限。

leading((t1 t2 t3 t4 t5))表示：t1 和 t2 先 join，t2 做内表；再和 t3 join，t3 做内表；再和 t4 join，t4 做内表；再和 t5 join，t5 做内表。

leading(t1 (t2 t3 t4) t5)表示：t2, t3, t4 先 join，内外表不限；再和 t1, t5 join，内外表不限。

leading((t1 (t2 t3 t4) t5))表示：t2, t3, t4 先 join，内外表不限；在最外层，t1 再和 t2, t3, t4 的 join 表 join，t1 为外表，再和 t5 join，t5 为内表。

leading((t1 (t2 t3) t4 t5)) leading((t3 t2))表示：t2，t3 先做 join，t2 做内表；然后再和 t1 做 join，t2，t3 的 join 表做内表；然后再跟 t4 做 join，t4 做内表，最后和 t5 做 join，t5 做内表。

示例

对示例中原语句使用如下 hint:

```
explain
select /*+ leading((((((store sales store) promotion) item) customer) ad2)
store_returns) leading((store store_sales))*/ i_product_name product_name ...
```

该 hint 表示：表之间的 join 关系是：store_sales 和 store 先 join， store_sales 做内表，然后依次跟 promotion, item, customer, ad2, store_returns 做 join。生成计划如下所示：

```
WARNING:  Duplicated or conflict hint: Leading(store_sales store), will be discarded.
 id |                                    operation                                     |   E-rows   | E-memory | E-width |   E-costs
----+----------------------------------------------------------------------------------+------------+----------+---------+-------------
  1 | ->  Row Adapter                                                                  |          6 |          |     273 | 16308094.34
  2 |    ->  Vector Streaming (type: GATHER)                                           |          6 |          |     273 | 16308094.34
  3 |       ->  Vector Hash Aggregate                                                  |          6 | 16MB     |     273 | 16308092.67
  4 |          ->  Vector Hash Join (5,20)                                             |          6 | 585MB    |     169 | 16308092.63
  5 |             ->  Vector Streaming(type: REDISTRIBUTE)                             |    1320811 | 1MB      |     181 | 16069870.93
  6 |                ->  Vector Hash Join (7,19)                                       |    1320811 | 43MB     |     181 | 16061891.00
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)                       |    1320811 | 1MB      |     131 | 16056566.78
  8 |                      ->  Vector Hash Join (9,18)                                 |    1320811 | 27MB     |     131 | 16048586.85
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)                 |    1383248 | 1MB      |     131 | 16038321.62
 10 |                            ->  Vector Hash Join (11,17)                          |    1383248 | 16MB     |     131 | 16029664.50
 11 |                               ->  Vector Hash Join (12,16)                       | 2626366951 | 16MB     |      73 | 15751384.88
 12 |                                  ->  Vector Hash Join (13,14)                    | 2750085660 | 2156MB   |      77 | 14226077.19
 13 |                                     ->  CStore Scan on store                     |      24048 | 1MB      |      19 | 2264.00
 14 |                                     ->  Vector Partition Iterator                | 2879987999 | 1MB      |      66 | 2756066.50
 15 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB      |      66 | 2756066.50
 16 |                                  ->  CStore Scan on promotion                    |      36000 | 1MB      |       4 | 1268.50
 17 |                               ->  CStore Scan on item                            |        158 | 1MB      |      58 | 4051.25
 18 |                         ->  CStore Scan on customer                              |   12000000 | 1MB      |       8 | 12923.00
 19 |                      ->  CStore Scan on customer_address ad2                     |    6000000 | 1MB      |      58 | 5770.00
 20 |             ->  Vector Partition Iterator                                        |  287999764 | 1MB      |      12 | 227383.99
 21 |                ->  Partitioned CStore Scan on store_returns                      |  287999764 | 1MB      |      12 | 227383.99
(21 rows)
```

图中计划顶端 warning 的提示详见 3.2.3.9.9 Hint 的错误、冲突及告警的说明。

3.2.3.9.3 Join 方式的 Hint

功能描述

指明 Join 使用的方法，可以为 Nested Loop，Hash Join 和 Merge Join。

语法格式

```
[no] nestloop|hashjoin|mergejoin(table_list)
```

参数说明

- **no** 表示 hint 的 join 方式不使用。
- **table_list** 为表示 hint 表集合的字符串，该字符串中的表与 3.2.3.9.2 Join 顺序的 Hint 相同，只是中间不允许出现括号指定 join 的优先级。

例如：

no nestloop(t1 t2 t3)表示：生成 t1，t2，t3 三表连接计划时，不使用 nestloop。三表连接计划可能是 t2 t3 先 join，再跟 t1 join，或 t1 t2 先 join，再跟 t3 join。此 hint 只 hint 最后一次 join 的 join 方式，对于两表连接的方法不 hint。如果需要，可以单独指定，例如：任意表均不允许 nestloop 连接，且希望 t2 t3 先 join，则增加 hint：no nestloop(t2 t3)。

示例

对示例中原语句使用如下 hint:

```
explain
select /*+ nestloop(store_sales store_returns item) */ i_product_name
product_name ...
```

该 hint 表示：生成 store_sales，store_returns 和 item 三表的结果集时，最后的两表关联使用 nestloop。生成计划如下所示：

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |     E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+-----------------
  1 | ->  Row Adapter                                                          |          6 |          |     273 | 100061693161.06
  2 |    ->  Vector Streaming (type: GATHER)                                   |          6 |          |     273 | 100061693161.06
  3 |       ->  Vector Hash Aggregate                                          |          6 | 16MB     |     273 | 100061693159.40
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |          6 | 1MB      |     169 | 100061693159.36
  5 |             ->  Vector Hash Join (6,22)                                  |          6 | 43MB     |     169 | 100061693158.99
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  |          6 | 1MB      |     119 | 100061688591.48
  7 |                   ->  Vector Hash Join (8,21)                            |          6 | 16MB     |     119 | 100061688591.30
  8 |                      ->  Vector Hash Join (9,20)                         |          7 | 27MB     |     123 | 100061687304.04
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)         |          7 | 1MB      |     123 | 100061677823.27
 10 |                            ->  Vector Hash Join (11,19)                  |          7 | 16MB     |     123 | 100061677823.12
 11 |                               ->  Vector Nest Loop (12,17)               |          7 | 1MB      |     112 | 100061675546.57
 12 |                                  ->  Vector Hash Join (13,15)            |      13670 | 585MB    |      62 | 6163443.54
 13 |                                     ->  Vector Partition Iterator        | 2879987999 | 1MB      |      66 | 2756066.50
 14 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB |  66 | 2756066.50
 15 |                                     ->  Vector Partition Iterator        |  287999764 | 1MB      |      12 | 227383.99
 16 |                                        ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB | 12 | 227383.99
 17 |                                  ->  Vector Materialize                  |        158 | 16MB     |      58 | 4051.28
 18 |                                     ->  CStore Scan on item              |        158 | 1MB      |      58 | 4051.25
 19 |                               ->  CStore Scan on store                   |      24048 | 1MB      |      19 | 2264.00
 20 |                            ->  CStore Scan on customer                   |   12000000 | 1MB      |       8 | 12923.00
 21 |                         ->  CStore Scan on promotion                     |      36000 | 1MB      |       4 | 1268.50
 22 |                   ->  CStore Scan on customer_address ad2                |    6000000 | 1MB      |      58 | 5770.00
(22 rows)
```

3.2.3.9.4 行数的 Hint

功能描述

指明中间结果集的大小，支持绝对值和相对值的 hint。

语法格式

```
rows(table_list #|+|-|* const)
```

参数说明

- **#,+,-,*，**进行行数估算 hint 的四种操作符号。#表示直接使用后面的行数进行 hint。+,-,*表示对原来估算的行数进行加、减、乘操作，运算后的行数最小值为 1 行。table_list 为 hint 对应的单表或多表 join 结果集，与 3.2.3.9.3 Join 方式的 Hint 中 table_list 相同。
- **const** 可以是任意非负数，支持科学计数法。

例如：

rows(t1 #5)表示：指定 t1 表的结果集为 5 行。

rows(t1 t2 t3 *1000)表示：指定 t1, t2, t3 join 完的结果集的行数乘以 1000。

建议

- 推荐使用两个表*的 hint。对于两个表的采用*操作符的 hint，只要两个表出现在 join 的两端，都会触发 hint。例如：设置 hint 为 rows(t1 t2 * 3)，对于(t1 t3 t4)和(t2 t5 t6)join 时，由于 t1 和 t2 出现在 join 的两端，所以其 join 的结果集也会应用该 hint 规则乘以 3。
- rows hint 支持在单表、多表、function table 及 subquery scan table 的结果集上指定 hint。

示例

对 3.2.3.9.1 Plan Hint 调优概述中原语句使用如下 hint:

```
explain
select /*+ rows(store_sales store_returns *50) */ i_product_name product_name ...
```

该 hint 表示：store_sales，store_returns 关联的结果集估算行数在原估算行数基础上乘以 50。生成计划如下所示：

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |  E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+------------
  1 | ->  Row Adapter                                                          |        312 |          |     273 | 3401656.58
  2 |    ->  Vector Streaming (type: GATHER)                                   |        312 |          |     273 | 3401656.58
  3 |       ->  Vector Hash Aggregate                                          |        312 | 16MB     |     273 | 3401634.91
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |        313 | 1MB      |     169 | 3401634.39
  5 |             ->  Vector Hash Join (6,21)                                  |        313 | 43MB     |     169 | 3401633.06
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  |        313 | 1MB      |     119 | 3397065.38
  7 |                   ->  Vector Hash Join (8,20)                            |        313 | 27MB     |     119 | 3397064.31
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)            |        328 | 1MB      |     119 | 3387583.37
  9 |                         ->  Vector Hash Join (10,19)                     |        328 | 16MB     |     119 | 3387582.18
 10 |                            ->  Vector Hash Join (11,18)                  |        344 | 16MB     |     123 | 3386294.74
 11 |                               ->  Vector Hash Join (12,14)               |        360 | 19MB     |     112 | 3384018.02
 12 |                                  ->  Vector Partition Iterator           |  287999764 | 1MB      |      12 | 227383.99
 13 |                                     ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      12 | 227383.99
 14 |                                  ->  Vector Hash Join (15,17)            |    1516824 | 16MB     |     124 | 3065686.08
 15 |                                     ->  Vector Partition Iterator        | 2879987999 | 1MB      |      66 | 2756066.50
 16 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB |      66 | 2756066.50
 17 |                                     ->  CStore Scan on item              |        158 | 1MB      |      58 | 4051.25
 18 |                               ->  CStore Scan on store                   |      24048 | 1MB      |      19 | 2264.00
 19 |                            ->  CStore Scan on promotion                  |      36000 | 1MB      |       4 | 1268.50
 20 |                      ->  CStore Scan on customer                         |   12000000 | 1MB      |       8 | 12923.00
 21 |                ->  CStore Scan on customer_address ad2                   |    6000000 | 1MB      |      58 | 5770.00
(21 rows)
```

第 11 行算子的估算行数修正为 360 行，原估算行数为 7 行（四舍五入后取值）。

3.2.3.9.5 Stream 方式的 Hint

功能描述

指明 stream 使用的方法，可以为 broadcast 和 redistribute。

语法格式

```
[no] broadcast|redistribute(table_list)
```

参数说明

- **no** 表示 hint 的 stream 方式不使用。
- **table_list** 为进行 stream 操作的单表或多表 join 结果集，见参数说明。

示例

对示例中原语句使用如下 hint:

```
explain
select /*+ no redistribute(store sales store returns item store)
leading(((store sales store returns item store) customer)) */ i product name
product_name ...
```

原计划中，(store_sales store_returns item store)和 customer 做 join 时，前者做了重分布，此 hint 表示禁止前者混合表做重分布，但仍然保持 join 顺序，则生成计划如下所示：

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |  E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+------------
  1 | ->  Row Adapter                                                          |          6 |          |     273 | 5718448.94
  2 |    ->  Vector Streaming (type: GATHER)                                   |          6 |          |     273 | 5718448.94
  3 |       ->  Vector Hash Aggregate                                          |          6 | 16MB     |     273 | 5718447.27
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |          6 | 1MB      |     169 | 5718447.23
  5 |             ->  Vector Hash Join (6,21)                                  |          6 | 16MB     |     169 | 5718446.86
  6 |                ->  Vector Hash Join (7,20)                               |          7 | 43MB     |     173 | 5717159.60
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)               |          7 | 1MB      |     123 | 5712592.09
  8 |                      ->  Vector Hash Join (9,18)                         |          7 | 585MB    |     123 | 5712591.93
  9 |                         ->  Vector Hash Join (10,17)                     |          7 | 16MB     |     123 | 3386294.56
 10 |                            ->  Vector Hash Join (11,13)                  |          7 | 19MB     |     112 | 3384018.02
 11 |                               ->  Vector Partition Iterator              |  287999764 | 1MB      |      12 | 227383.99
 12 |                                  ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      12 | 227383.99
 13 |                               ->  Vector Hash Join (14,16)               |    1516824 | 16MB     |     124 | 3065686.08
 14 |                                  ->  Vector Partition Iterator           | 2879987999 | 1MB      |      66 | 2756066.50
 15 |                                     ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB |      66 | 2756066.50
 16 |                                  ->  CStore Scan on item                 |        158 | 1MB      |      58 | 4051.25
 17 |                            ->  CStore Scan on store                      |      24048 | 1MB      |      19 | 2264.00
 18 |                         ->  Vector Streaming(type: BROADCAST)            |  288000000 | 1MB      |       8 | 2176297.36
 19 |                            ->  CStore Scan on customer                   |   12000000 | 1MB      |       8 | 12923.00
 20 |                      ->  CStore Scan on customer_address ad2             |    6000000 | 1MB      |      58 | 5770.00
 21 |                   ->  CStore Scan on promotion                           |      36000 | 1MB      |       4 | 1268.50
(21 rows)
```

3.2.3.9.6 Scan 方式的 Hint

功能描述

指明 scan 使用的方法，可以是 tablescan、indexscan 和 indexonlyscan。

语法格式

```
[no] tablescan|indexscan|indexonlyscan(table [index])
```

参数说明

- **no** 表示 hint 的 scan 方式不使用。
- **table** 表示 hint 指定的表，只能指定一个表，如果表存在别名应优先使用别名进行 hint。
- **index** 表示使用 indexscan 或 indexonlyscan 的 hint 时，指定的索引名称，当前只能指定一个。

对于 indexscan 或 indexonlyscan，只有 hint 的索引属于 hint 的表时，才能使用该 hint。

scan hint 支持在行列存表、obs 表、子查询表上指定。

示例

为了 hint 使用索引扫描，需要首先在表 item 的 i_item_sk 列上创建索引，名称为 i。

```
create index i on item(i_item_sk);
```

对示例中原语句使用如下 hint:

```
explain
select /*+ indexscan(item i) */ i_product_name product_name ...
```

该 hint 表示：item 表使用索引 i 进行扫描。生成计划如下所示：

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |     E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+-----------------
  1 | ->  Row Adapter                                                          |          6 |          |     273 | 100061674938.26
  2 |    ->  Vector Streaming (type: GATHER)                                   |          6 |          |     273 | 100061674938.26
  3 |       ->  Vector Hash Aggregate                                          |          6 | 16MB     |     273 | 100061674936.59
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |          6 | 1MB      |     169 | 100061674936.55
  5 |             ->  Vector Hash Join (6,21)                                  |          6 | 43MB     |     169 | 100061674936.19
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  |          6 | 1MB      |     119 | 100061670368.67
  7 |                   ->  Vector Hash Join (8,20)                            |          6 | 16MB     |     119 | 100061670368.50
  8 |                      ->  Vector Hash Join (9,19)                         |          7 | 27MB     |     123 | 100061669081.23
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)         |          7 | 1MB      |     123 | 100061659600.47
 10 |                            ->  Vector Hash Join (11,18)                  |          7 | 16MB     |     123 | 100061659600.31
 11 |                               ->  Vector Nest Loop (12,17)               |          7 | 1MB      |     112 | 100061657323.77
 12 |                                  ->  Vector Hash Join (13,15)            |      13670 | 585MB    |      62 | 6163443.54
 13 |                                     ->  Vector Partition Iterator        | 2879987999 | 1MB      |      66 | 2756066.50
 14 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB |  66 | 2756066.50
 15 |                                     ->  Vector Partition Iterator        |  287999764 | 1MB      |      12 | 227383.99
 16 |                                        ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |  12 | 227383.99
 17 |                                  ->  CStore Index Scan using i on item   |          1 | 1MB      |      58 | 4.01
 18 |                               ->  CStore Scan on store                   |      24048 | 1MB      |      19 | 2264.00
 19 |                            ->  CStore Scan on customer                   |   12000000 | 1MB      |       8 | 12923.00
 20 |                         ->  CStore Scan on promotion                     |      36000 | 1MB      |       4 | 1268.50
 21 |                      ->  CStore Scan on customer_address ad2             |    6000000 | 1MB      |      58 | 5770.00
(21 rows)
```

3.2.3.9.7 子链接块名的 hint

功能描述

指明子链接块的名称。

语法格式

```
blockname (table)
```

参数说明

- **table** 表示为该子链接块 hint 的别名的名称。

说明

- **blockname hint** 仅在对应的子链接块提升时才会被上层查询使用。目前支持的子链接提升包括 IN 子链接提升、EXISTS 子链接提升和包含 Agg 等值相关子链接提升。该 hint 通常会和前面章节提到的 hint 联合使用。
- 对于 FROM 关键字后的子查询，则需要使用子查询的别名进行 hint，blockname hint 不会被用到。
- 如果子链接中含有多个表，则提升后这些表可与外层表以任意优化顺序连接，hint 也不会被用到。

示例

```
explain select /*+nestloop(store sales tt) */ * from store sales where ss item sk
in (select /*+blockname(tt)*/ i_item_sk from item group by 1);
```

该 hint 表示：子链接的别名为 tt，提升后与上层的 store_sales 表关联时使用 nestloop。生成计划如下所示：

```
 id |                      operation                       |   E-rows   | E-memory | E-width |     E-costs
----+------------------------------------------------------+------------+----------+---------+-----------------
  1 | ->  Row Adapter                                      | 1439994000 |          |     216 | 325105765847.91
  2 |    ->  Vector Streaming (type: GATHER)               | 1439994000 |          |     216 | 325105765847.91
  3 |       ->  Vector Nest Loop Semi Join (4, 6)          | 1439994000 | 1MB      |     216 | 325026664615.00
  4 |          ->  Vector Partition Iterator               | 2879987999 | 1MB      |     216 | 2756066.50
  5 |             ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB |  216 | 2756066.50
  6 |          ->  Vector Materialize                      |     300000 | 16MB     |       4 | 4176.25
  7 |             ->  Vector Hash Aggregate                |     300000 | 16MB     |       4 | 3988.75
  8 |                ->  CStore Scan on item               |     300000 | 1MB      |       4 | 3832.50
(8 rows)
```

3.2.3.9.8 运行倾斜的 hint

功能描述

指明查询运行时重分布过程中存在倾斜的重分布键和倾斜值，针对 Join 和 HashAgg 运算中的重分布进行优化。

语法格式

- 指定单表倾斜：

  skew(table (column) [(value)])

- 指定中间结果倾斜：

  skew((join_rel) (column) [(value)])

参数说明

- **table** 表示存在倾斜的单个表名。
- **join_rel** 表示参与 join 的两个或多个表，如（t1 t2）表示 t1 和 t2join 后的结果存在倾斜。
- **column** 表示倾斜表中存在倾斜的一个或多个列。
- **value** 表示倾斜的列中存在倾斜的一个或多个值。

说明

- skew hint 仅在需要重分布且指定的倾斜信息与查询执行过程中的重分布信息相匹配时才会被使用。
- skew hint 目前仅处理普通表和子查询类型的表关系，支持基表 hint、子查询 hint、with as 子句 hint。对于子查询，无论提升与否都支持在 skew hint 中使用，这点与其它 hint 不一样。
- 对于倾斜表，如果定义了别名，则在 hint 中必须使用别名。
- 对于倾斜列，在不产生歧义的情况下，可以使用原名也可以使用别名。skew hint 的 column 不支持表达式，如果需要指定采用分布键为表达式的重分布存在倾斜，需要将重分布键指定为新的列，以新的列进行 hint。
- 对于倾斜值，个数需为列数的整数倍并按列的顺序进行组合，组合的个数不能超过 10 个。如果各倾斜列的倾斜值的个数不一样，为了满足按列组合，值可以重复指定。如，表 t1 的 c1 和 c2 存在倾斜，c1 列的倾斜值只有 a1，而 c2 列的倾斜有 b1 和 b2，则 skew hint 如下：skew(t1 (c1 c2) ((a1 b1)(a1 b2)))。例中(a1 b1)为一个值组合，NULL 可以作为倾斜值出现，每个 hint 中的值组合不超过十个， 且需为列的整数倍。
- 在 Join 的重分布优化中，skew hint 中的 value 不可缺省，在 HashAgg 中可以缺省。
- 对于表、列、值中若指定多个，则同类间需以空格分离。
- 对于倾斜值，不支持在 hint 中进行类型强转；对于 string 类型，需要使用单引号。

例如：

- 指定单表倾斜

  每一个 skew hint 用来表示一个表关系存在的倾斜信息，如果想要指定在查询中的多个表关系存在的倾斜信息，则通过指定多个 skew hint 实现。

  在指定 skew 时，包括以下四个场景的用法：

  - 单列单值： skew(t (c1) (v1))

    说明：表关系 t 的 c1 列中的 v1 值在查询执行中存在倾斜。

- 单列多值：skew(t (c1) (v1 v2 v3 ...))

  说明：表关系 t 的 c1 列中的 v1、v2、v3…等值在查询执行中存在倾斜。
- 多列单值：skew(t (c1 c2) (v1 v2))

  说明：表关系 t 的 c1 列的 v1 值和 c2 列的 v2 值在查询执行中存在倾斜。
- 多列多值：skew(t (c1 c2) ((v1 v2) (v3 v4) (v5 v6) ...))

  说明：表关系 t 的 c1 列的 v1、v3、v5…值和 c2 列的 v2、v4、v6…值在查询执行中存在倾斜。

须知

多列多值时，各组倾斜值间也可以不使用括号，如：skew(t (c1 c2) (v1 v2 v3 v4 v5 v6 ...))。是否使用括号必须统一，不可混合，

说明

如：skew(t (c1 c2) (v1 v2 v3 v4 （v5 v6） ...)) 将会产生语法报错。

- 指定中间结果倾斜

  如果基表不存在倾斜，而是查询执行中的中间结果出现倾斜，则需要通过指定中间结果倾斜的 skew hint 来进行倾斜的调优。skew((t1 t2) (c1) (v1))

  说明：表关系 t1 和 t2 Join 后的结果存在倾斜，倾斜的是 t1 表的 c1 列，c1 列的倾斜值是 v1。

  为了避免产生歧义，“c1”只能存在于 join_rel 的一个表关系中，如果存在同名列则通过别名进行规避。

建议

- 如果查询具有多层，则哪一层出现倾斜，则将 hint 写在哪一层中。
- 对于提升的子查询，skew hint 支持直接使用子查询名进行 hint。如果明确子查询提升后的哪一个基表存在倾斜，则直接使用基表进行 hint 的可用性更高。
- 无论对于表或列，若存在别名，则优先使用别名进行 hint。

示例

指定单表倾斜

- 原 query 中进行 hint。

  采用如下查询进行 skew hint 倾斜调优的举例，查询语句及不带 hint 的原计划如下所示：

```
explain
with customer total return as
(select sr customer sk as ctr customer sk
,sr store sk as ctr store sk
,sum(SR FEE) as ctr total return
from store returns
,date dim
where sr returned date sk = d date sk
and d_year =2000
```

```
group by sr_customer_sk
,sr_store_sk)
 select  c_customer_id
from customer_total_return ctr1
,store
,customer
where ctr1.ctr_total_return > (select avg(ctr_total_return)*1.2
from customer_total_return ctr2
where ctr1.ctr_store_sk = ctr2.ctr_store_sk)
and s_store_sk = ctr1.ctr_store_sk
and s_state = 'NM'
and ctr1.ctr_customer_sk = c_customer_sk
order by c_customer_id
limit 100;
```

```
 id |                                  operation                                  |  E-rows   |    E-memory    | E-width |  E-costs
----+-----------------------------------------------------------------------------+-----------+----------------+---------+-----------
  1 | ->  Row Adapter                                                             |       100 |                |      20 | 911254.47
  2 |    ->  Vector Limit                                                         |       100 |                |      20 | 911254.47
  3 |       ->  Vector Streaming (type: GATHER)                                   |      2400 |                |      20 | 911325.75
  4 |          ->  Vector Limit                                                   |      2400 | 1MB            |      20 | 911247.62
  5 |             ->  Vector Sort                                                 |   3684816 | 16MB           |      20 | 911631.21
  6 |                ->  Vector Hash Join (7,29)                                  |   3684817 | 41MB(12374MB)  |      20 | 905379.41
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)                  |   3684817 | 384KB          |       4 | 883010.31
  8 |                      ->  Vector Hash Join (9,19)                            |   3684817 | 16MB           |       4 | 861302.05
  9 |                         ->  Vector Hash Join (10,18)                        |  11054450 | 16MB           |      44 | 427109.71
 10 |                            ->  Vector Hash Aggregate                        |  50247501 | 397MB(12671MB) |      54 | 395302.57
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)      |  50247501 | 384KB          |      22 | 358663.76
 12 |                                  ->  Vector Hash Join (13,15)               |  50247501 | 16MB           |      22 | 294300.51
 13 |                                     ->  Vector Partition Iterator           | 287999764 | 1MB            |      26 | 227383.99
 14 |                                        ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      26 | 227383.99
 15 |                                     ->  Vector Streaming(type: BROADCAST)   |      8712 | 384KB          |       4 | 975.56
 16 |                                        ->  Vector Partition Iterator        |       363 | 1MB            |       4 | 910.65
 17 |                                           ->  Partitioned CStore Scan on date_dim |       363 | 1MB |       4 | 910.65
 18 |                            ->  CStore Scan on store                         |        44 | 1MB            |       4 | 1006.39
 19 |                         ->  Vector Hash Aggregate                           |       192 | 16MB           |      68 | 426707.38
 20 |                            ->  Vector Subquery Scan on ctr2                 |  50247501 | 1MB            |      36 | 416239.03
 21 |                               ->  Vector Hash Aggregate                     |  50247501 | 397MB(12671MB) |      54 | 395302.57
 22 |                                  ->  Vector Streaming(type: REDISTRIBUTE)   |  50247501 | 384KB          |      22 | 358663.76
 23 |                                     ->  Vector Hash Join (24,26)            |  50247501 | 16MB           |      22 | 294300.51
 24 |                                        ->  Vector Partition Iterator        | 287999764 | 1MB            |      26 | 227383.99
 25 |                                           ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      26 | 227383.99
 26 |                                        ->  Vector Streaming(type: BROADCAST)|      8712 | 384KB          |       4 | 975.56
 27 |                                           ->  Vector Partition Iterator     |       363 | 1MB            |       4 | 910.65
 28 |                                              ->  Partitioned CStore Scan on date_dim |       363 | 1MB |       4 | 910.65
 29 |                   ->  CStore Scan on customer                               |  12000000 | 1MB            |      24 | 12923.00
(29 rows)
```

对内层 with 子句中的 HashAgg 和外层的 Hash Join 进行 hint 指定，带 hint 的查询如下：

```
explain
with customer total return as
(select /*+ skew(store returns(sr store sk sr customer sk)) */sr customer sk as
ctr customer sk
,sr store sk as ctr store sk
,sum(SR FEE) as ctr total return
from store returns
,date dim
where sr returned date sk = d date sk
and d year =2000
group by sr customer sk
,sr store sk)
 select  /*+ skew(ctr1(ctr customer sk)(11))*/  c customer id
from customer total return ctr1
,store
,customer
where ctr1.ctr total return > (select avg(ctr total return)*1.2
from customer total return ctr2
where ctr1.ctr store sk = ctr2.ctr store sk)
and s store sk = ctr1.ctr store sk
and s state = 'NM'
and ctr1.ctr_customer_sk = c_customer_sk
order by c_customer_id
limit 100;
```

该 hint 表示：内层 with 子句中的 group by 在做 HashAgg 中进行重分布时存在倾斜，对应原计划的 10 和 21 号 Hash Agg 算子；外层 ctr1 表的 ctr_customer_sk 列在做 Hash Join 中进行重分布时存在倾斜，对应原计划的 6 号算子。生成计划如下所示：

```
 id |                                                                operation                                                                |  E-rows  |   E-memory   | E-width |  E-costs
----+-----------------------------------------------------------------------------------------------------------------------------------------+----------+--------------+---------+-----------
  1 | ->  Row Adapter                                                                                                                         |      100 |              |      20 | 1061778.14
  2 |    ->  Vector Limit                                                                                                                     |      100 |              |      20 | 1061778.14
  3 |       ->  Vector Streaming (type: GATHER)                                                                                               |     2400 |              |      20 | 1061849.41
  4 |          ->  Vector Limit                                                                                                               |     2400 | 1MB          |      20 | 1061771.29
  5 |             ->  Vector Sort                                                                                                             |  3684816 | 16MB         |      20 | 1062154.87
  6 |                ->  Vector Hash Join (7,31)                                                                                              |  3684817 | 41MB(12344MB)|      20 | 1055903.08
  7 |                   ->  Vector Streaming(type: PART REDISTRIBUTE PART ROUNDROBIN)                                                         |  3684817 | 384KB        |       4 | 1013056.49
  8 |                      ->  Vector Hash Join (9,20)                                                                                        |  3684817 | 16MB         |       4 | 1000006.10
  9 |                         ->  Vector Hash Join (10,19)                                                                                    | 11054450 | 16MB         |      44 | 496461.73
 10 |                            ->  Vector Hash Aggregate                                                                                    | 50247501 | 397MB(12010MB)|      54 | 464654.59
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)                                                                  | 50247501 | 384KB        |      54 | 428015.79
 12 |                                  ->  Vector Hash Aggregate                                                                              | 50247501 | 397MB(12010MB)|      54 | 330939.31
 13 |                                     ->  Vector Hash Join (14,16)                                                                        | 50247501 | 16MB         |      22 | 294300.51
 14 |                                        ->  Vector Partition Iterator                                                                    |287999764 | 1MB          |      26 | 227383.99
 15 |                                           ->  Partitioned CStore Scan on store_returns                                                  |287999764 | 1MB          |      26 | 227383.99
 16 |                                        ->  Vector Streaming(type: BROADCAST)                                                            |     8712 | 384KB        |       4 | 975.56
 17 |                                           ->  Vector Partition Iterator                                                                 |      363 | 1MB          |       4 | 910.65
 18 |                                              ->  Partitioned CStore Scan on date_dim                                                    |      363 | 1MB          |       4 | 910.65
 19 |                            ->  CStore Scan on store                                                                                     |       44 | 1MB          |       4 | 1006.39
 20 |                         ->  Vector Hash Aggregate                                                                                       |      192 | 16MB         |      68 | 496059.40
 21 |                            ->  Vector Subquery Scan on ctr2                                                                             | 50247501 | 1MB          |      36 | 485591.05
 22 |                               ->  Vector Hash Aggregate                                                                                 | 50247501 | 397MB(12010MB)|      54 | 464654.59
 23 |                                  ->  Vector Streaming(type: REDISTRIBUTE)                                                               | 50247501 | 384KB        |      54 | 428015.79
 24 |                                     ->  Vector Hash Aggregate                                                                           | 50247501 | 397MB(12010MB)|      54 | 330939.31
 25 |                                        ->  Vector Hash Join (26,28)                                                                     | 50247501 | 16MB         |      22 | 294300.51
 26 |                                           ->  Vector Partition Iterator                                                                 |287999764 | 1MB          |      26 | 227383.99
 27 |                                              ->  Partitioned CStore Scan on store_returns                                               |287999764 | 1MB          |      26 | 227383.99
 28 |                                           ->  Vector Streaming(type: BROADCAST)                                                         |     8712 | 384KB        |       4 | 975.56
 29 |                                              ->  Vector Partition Iterator                                                              |      363 | 1MB          |       4 | 910.65
 30 |                                                 ->  Partitioned CStore Scan on date_dim                                                 |      363 | 1MB          |       4 | 910.65
 31 |                      ->  Vector Streaming(type: PART LOCAL PART BROADCAST)                                                              | 12000000 | 384KB        |      24 | 34485.50
 32 |                         ->  CStore Scan on customer                                                                                     | 12000000 | 1MB          |      24 | 12923.00
(32 rows)
```

从优化后的计划可以看出：①对于 Hash Agg，由于其重分布存在倾斜，所以优化为双层 Agg；②对于 Hash Join，同样由于其重分布存在倾斜，所以优化为采用新的重分布算子。

- 需要改写 query 后进行 hint

  不带 hint 的查询和计划如下：

```
explain select count(*) from store_sales_1 group by round(ss_list_price);
```

```
 id |                                                operation                                                 | E-rows | E-memory | E-width | E-costs
----+----------------------------------------------------------------------------------------------------------+--------+----------+---------+----------
  1 | ->  Row Adapter                                                                                          |  16672 |          |      14 | 62261.28
  2 |    ->  Vector Streaming (type: GATHER)                                                                   |  16672 |          |      14 | 62261.28
  3 |       ->  Vector Streaming(type: LOCAL GATHER dop: 1/2)                                                  |  16672 | 32KB     |      14 | 61479.78
  4 |          ->  Vector Hash Aggregate                                                                       |  16672 | 16MB     |      14 | 61452.00
  5 |             ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)                                      |3112836 | 128KB    |       6 | 57498.43
  6 |                ->  CStore Scan on store_sales_1                                                          |3112836 | 1MB      |       6 | 21810.25
(6 rows)
```

由于 hint 中列不支持表达式，在进行倾斜优化时需要借助 subquery 改写查询，改写后的查询和计划如下：

```
explain
select count(*)
from (select round(ss list price),ss hdemo sk
from store sales 1)tmp(a,ss hdemo sk)
group by a;
```

```
 id |                                                operation                                                 | E-rows | E-memory | E-width | E-costs
----+----------------------------------------------------------------------------------------------------------+--------+----------+---------+----------
  1 | ->  Row Adapter                                                                                          |  16672 |          |      14 | 62261.28
  2 |    ->  Vector Streaming (type: GATHER)                                                                   |  16672 |          |      14 | 62261.28
  3 |       ->  Vector Streaming(type: LOCAL GATHER dop: 1/2)                                                  |  16672 | 32KB     |      14 | 61479.78
  4 |          ->  Vector Hash Aggregate                                                                       |  16672 | 16MB     |      14 | 61452.00
  5 |             ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)                                      |3112836 | 128KB    |       6 | 57498.43
  6 |                ->  CStore Scan on store_sales_1                                                          |3112836 | 1MB      |       6 | 21810.25
(6 rows)
```

改写注意不要影响到业务逻辑。

采用改写后的查询进行 hint，带 hint 的查询和计划如下：

```
explain
select /*+ skew(tmp(a)) */ count(*)
from (select round(ss list price),ss hdemo sk
from store sales 1)tmp(a,ss hdemo sk)
group by a;
```

```
 id |                                operation                                | E-rows  | E-memory | E-width | E-costs
----+-------------------------------------------------------------------------+---------+----------+---------+----------
  1 | ->  Row Adapter                                                         |   16672 |          |      14 | 27771.82
  2 |    ->  Vector Streaming (type: GATHER)                                  |   16672 |          |      14 | 27771.82
  3 |       ->  Vector Streaming(type: LOCAL GATHER dop: 1/2)                 |   16672 | 32KB     |      14 | 26990.32
  4 |          ->  Vector Hash Aggregate                                      |   16671 | 16MB     |      14 | 26962.54
  5 |             ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)     |   66216 | 128KB    |      14 | 26838.09
  6 |                ->  Vector Hash Aggregate                                |   66216 | 16MB     |      14 | 25949.61
  7 |                   ->  CStore Scan on store_sales_1                      | 3112836 | 1MB      |       6 | 21810.25
(7 rows)
```

从计划可以看出，对 Hash Agg 进行倾斜优化后，采用了双层 agg 实现，大大过滤了进行重分布时的数据量，减少了重分布时间。

此外，需要说明的是，对于子查询，支持使用查询内部的列进行 hint，如：

```
explain
select /*+ skew(tmp(b)) */ count(*)
from (select round(ss list price) b,ss hdemo sk
from store sales 1)tmp(a,ss hdemo sk)
group by a;
```

3.2.3.9.9 Hint 的错误、冲突及告警

Plan Hint 的结果会体现在计划的变化上，可以通过 explain 来查看变化。

Hint 中的错误不会影响语句的执行，只是不能生效，该错误会根据语句类型以不同方式提示用户。对于 explain 语句，hint 的错误会以 warning 形式显示在界面上，对于非 explain 语句，会以 debug1 级别日志显示在日志中，关键字为 PLANHINT。

hint 的错误分为以下类型：

- 语法错误

  语法规则树归约失败，会报错，指出出错的位置。

  例如：hint 关键字错误，leading hint 或 join hint 指定 2 个表以下，其它 hint 未指定表等。一旦发现语法错误，则立即终止 hint 的解析，所以此时只有错误前面的解析完的 hint 有效。

  例如：

  leading((t1 t2)) nestloop(t1) rows(t1 t2 #10)

  nestloop(t1)存在语法错误，则终止解析，可用 hint 只有之前解析的 leading((t1 t2))。

- 语义错误
  - 表不存在，存在多个，或在 leading 或 join 中出现多次，均会报语义错误。
  - scanhint 中的 index 不存在，会报语义错误。
  - 另外，如果子查询提升后，同一层出现多个名称相同的表，且其中某个表需要被 hint，hint 会存在歧义，无法使用，需要为相同表增加别名规避。
- hint 重复或冲突

  如果存在 hint 重复或冲突，只有第一个 hint 生效，其它 hint 均会失效，会给出提示。
  - hint 重复是指，hint 的方法及表名均相同。例如：nestloop(t1 t2) nestloop(t1 t2)。
  - hint 冲突是指，table list 一样的 hint，存在不一样的 hint，hint 的冲突仅对于每一类 hint 方法检测冲突。

    例如：nestloop (t1 t2) hashjoin (t1 t2)，则后面与前面冲突，此时 hashjoin 的 hint 失效。注意：nestloop(t1 t2)和 no mergejoin(t1 t2)不冲突。

须知

leading hint 中的多个表会进行拆解。例如：leading ((t1 t2 t3))会拆解成：leading((t1 t2)) leading(((t1 t2) t3))，此时如果存在 leading((t2 t1))，则两者冲突，后面的会被丢弃。（例外：指定内外表的 hint 若与不指定内外表的 hint 重复，则始终丢弃不指定内外表的 hint。）

- 子链接提升后 hint 失效

  子链接提升后的 hint 失效，会给出提示。通常出现在子链接中存在多个表连接的场景。提升后，子链接中的多个表不再作为一个整体出现在 join 中。

- 列类型不支持重分布
  - 对于 skew hint 来说，目的是为了进行重分布时的调优，所以当 hint 列的类型不支持重分布时，hint 将无效。
- hint 未被使用
  - 非等值 join 使用 hashjoin hint 或 mergejoin hint
  - 不包含索引的表使用 indexscan hint 或 indexonlyscan hint
  - 通常只有在索引列上使用过滤条件才会生成相应的索引路径，全表扫描将不会使用索引，因此使用 indexscan hint 或 indexonlyscan hint 将不会使用
  - indexonlyscan 只有输出列仅包含索引列才会使用，否则指定时 hint 不会被使用
  - 多个表存在等值连接时，仅尝试有等值连接条件的表的连接，此时没有关联条件的表之间的路径将不会生成，所以指定相应的 leading，join，rows hint 将不使用，例如：t1 t2 t3 表 join，t1 和 t2, t2 和 t3 有等值连接条件，则 t1 和 t3 不会优先连接，leading(t1 t3)不会被使用。
  - 生成 stream 计划时，如果表的分布列与 join 列相同，则不会生成 redistribute 的计划；如果不同，且另一表分布列与 join 列相同，只能生成 redistribute 的计划，不会生成 broadcast 的计划，指定相应的 hint 则不会被使用。
  - 如果子链接未被提升，则 blockname hint 不会被使用。
  - 对于 skew hint，hint 未被使用可能由于：
    - 计划中不需要进行重分布。
    - hint 指定的列为包含分布键。
    - hint 指定倾斜信息有误或不完整，如对于 join 优化未指定值。
    - 倾斜优化的 GUC 参数处于关闭状态。

3.2.3.9.10 Plan Hint 实际调优案例

本节以 TPC-DS 标准测试的 Q24 的部分语句为例，在 1000X，24DN 环境上，说明使用 plan hint 进行实际调优的过程。示例如下：

```
select avg(netpaid) from
(select c last name
,c first name
,s store name
,ca state
,s_state
```

```
,i_color
,i_current_price
,i_manager_id
,i_units
,i_size
,sum(ss_sales_price) netpaid
from store_sales
,store_returns
,store
,item
,customer
,customer_address
where ss_ticket_number = sr_ticket_number
and ss_item_sk = sr_item_sk
and ss_customer_sk = c_customer_sk
and ss_item_sk = i_item_sk
and ss_store_sk = s_store_sk
and c_birth_country = upper(ca_country)
and s_zip = ca_zip
and s_market_id=7
group by c_last_name
,c_first_name
,s_store_name
,ca state
,s state
,i color
,i current price
,i manager id
,i units
,i_size);
```

1. 该语句的初始计划如下，运行时间 110s：

```
 id |                                                                operation                                                                 |        A-time         |  A-rows   |  E-rows
----+------------------------------------------------------------------------------------------------------------------------------------------+-----------------------+-----------+-----------+
  1 | ->  Row Adapter                                                                                                                          | 110324.107            |         1 |         1 |
  2 |    ->  Vector Aggregate                                                                                                                  | 110324.093            |         1 |         1 |
  3 |       ->  Vector Streaming (type: GATHER)                                                                                                | 110323.958            |        24 |        24 |
  4 |          ->  Vector Aggregate                                                                                                            | [110179.302,110309.653] |        24 |        24 |
  5 |             ->  Vector Hash Aggregate                                                                                                    | [110178.388,110308.515] |    647824 |     16656 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                                                                                  | [77616.177,96478.771]   | 666834733 |     16664 |
  7 |                   ->  Vector Hash Join (8,22)                                                                                            | [81727.257,84728.519]   | 666834733 |     16664 |
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)                                                                            | [78770.520,82021.087]   | 666834733 |     16664 |
  9 |                         ->  Vector Hash Join (10,21)                                                                                     | [88066.755,90701.860]   | 666834733 |     16664 |
 10 |                            ->  Vector Streaming(type: BROADCAST)                                                                         | [7940.962,21430.725]    | 591882336 |     51360 |
 11 |                               ->  Vector Hash Join (12,20)                                                                               | [2419.995,5319.606]     |  24661764 |      2140 |
 12 |                                  ->  Vector Streaming(type: REDISTRIBUTE)                                                                | [1750.448,4659.581]     |  25258268 |      2241 |
 13 |                                     ->  Vector Hash Join (14,18)                                                                         | [15240.666,17159.616]   |  25258268 |      2241 |
 14 |                                        ->  Vector Hash Join (15,17)                                                                      | [12112.913,13563.366]   | 252564412 |  472070592 |
 15 |                                           ->  Vector Partition Iterator                                                                  | [11148.731,12473.230]   | 2879987999 | 2879987999 |
 16 |                                              ->  Partitioned CStore Scan on public.store_sales                                           | [11097.921,12412.596]   | 2879987999 | 2879987999 |
 17 |                                           ->  CStore Scan on public.store                                                                | [0.447,0.689]           |      2064 |      2064 |
 18 |                                        ->  Vector Partition Iterator                                                                     | [296.805,319.014]       | 287999764 | 287999764 |
 19 |                                           ->  Partitioned CStore Scan on public.store_returns                                            | [292.938,314.787]       | 287999764 | 287999764 |
 20 |                                     ->  CStore Scan on public.customer                                                                   | [114.358,144.462]       |  12000000 |  12000000 |
 21 |                               ->  CStore Scan on public.customer_address                                                                 | [38.426,56.753]         |   6000000 |   6000000 |
 22 |                   ->  CStore Scan on public.item                                                                                         | [3.160,5.026]           |    300000 |    300000 |
(22 rows)
```

该计划中，第 10 层算子使用 broadcast 性能较差，由于第 11 层算子估算行数为 2140，比实际行数严重低估。错误行数估算主要来源于第 13 层算子的行数低估，根因是第 13 层 hashjoin 中，使用 store_sales 的(ss_ticket_number, ss_item_sk)列和 store_returns 的(sr_ticket_number, sr_item_sk)列进行关联，由于缺少多列相关性的估算导致行数严重低估。

2. 使用如下的 rows hint 进行调优后，计划如下，运行时间 318s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns * 11270)*/ c_last_name ...
```

id	operation	A-time	A-rows	E-rows
1	-> Row Adapter	318585.246	1	1
2	-> Vector Aggregate	318585.232	1	1
3	-> Vector Streaming (type: GATHER)	318585.082	24	24
4	-> Vector Aggregate	[318323.324,318499.290]	24	24
5	-> Vector Hash Aggregate	[318320.813,318497.054]	647824	187770504
6	-> Vector Streaming(type: REDISTRIBUTE)	[288074.860,305601.698]	666834733	187770507
7	-> Vector Hash Join (8,22)	[253642.468,315808.664]	666834733	187770507
8	-> Vector Hash Join (9,18)	[250904.317,315684.018]	666834733	187770507
9	-> Vector Streaming(type: REDISTRIBUTE)	[4552.500,310602.307]	275042158	147106999
10	-> Vector Hash Join (11,17)	[7658.951,14053.823]	275042158	147106999
11	-> Vector Streaming(type: REDISTRIBUTE)	[3953.255,10264.943]	287999764	154060900
12	-> Vector Hash Join (13,15)	[28196.188,32838.794]	287999764	154060900
13	-> Vector Partition Iterator	[11477.673,12324.583]	2879987999	2879987999
14	-> Partitioned CStore Scan on public.store_sales	[11411.382,12250.209]	2879987999	2879987999
15	-> Vector Partition Iterator	[304.188,403.205]	287999764	287999764
16	-> Partitioned CStore Scan on public.store_returns	[299.838,398.255]	287999764	287999764
17	-> CStore Scan on public.customer	[122.246,170.128]	12000000	12000000
18	-> Vector Streaming(type: REDISTRIBUTE)	[57.558,117.461]	492915	146467
19	-> Vector Hash Join (20,21)	[45.554,96.238]	492915	146467
20	-> CStore Scan on public.customer_address	[39.738,89.412]	6000000	6000000
21	-> CStore Scan on public.store	[0.361,1.095]	2064	2064
22	-> Vector Streaming(type: BROADCAST)	[48.986,91.170]	7200000	7200000
23	-> CStore Scan on public.item	[4.506,6.602]	300000	300000

(23 rows)

时间反而劣化了，原因是第 8 层 hashjoin 过慢引起第 9 层 redistribute 时间过慢导致，其中第 9 层 redistribute 并没有数据倾斜，hashjoin 慢的原因是由于第 18 层 redistribute 后数据倾斜导致。

3. 经过实际数据查证，customer_address 的两个 join 列的不同值数目较少，使用其进行 join 容易出现数据倾斜，故把 customer_address 放到最后进行 join。使用如下的 hint 进行调优后，计划如下，运行时间 116s：

```
select avg(netpaid) from
(select /*+rows(store sales store returns *11270)
leading((store sales store returns store item customer) customer address)*/
c_last_name ...
```

id	operation	A-time	A-rows	E-rows
1	-> Row Adapter	116326.597	1	1
2	-> Vector Aggregate	116326.590	1	1
3	-> Vector Streaming (type: GATHER)	116326.473	24	24
4	-> Vector Aggregate	[116157.161,116236.494]	24	24
5	-> Vector Hash Aggregate	[116155.328,116233.946]	647824	187770504
6	-> Vector Streaming(type: REDISTRIBUTE)	[84103.951,102052.326]	666834733	187770507
7	-> Vector Hash Join (8,10)	[23229.469,47484.697]	666834733	187770507
8	-> Vector Streaming(type: REDISTRIBUTE)	[38.367,74.930]	6000000	6000000
9	-> CStore Scan on public.customer_address	[69.877,121.460]	6000000	6000000
10	-> Vector Streaming(type: REDISTRIBUTE)	[17404.744,17567.550]	24661764	24112909
11	-> Vector Hash Join (12,22)	[16123.627,16397.246]	24661764	24112909
12	-> Vector Streaming(type: REDISTRIBUTE)	[15320.663,15741.646]	25258268	25252751
13	-> Vector Hash Join (14,21)	[14962.342,16375.458]	25258268	25252751
14	-> Vector Hash Join (15,19)	[14449.031,15825.949]	25258268	25252751
15	-> Vector Hash Join (16,18)	[11439.959,12510.065]	252564412	472070592
16	-> Vector Partition Iterator	[10531.986,11536.213]	2879987999	2879987999
17	-> Partitioned CStore Scan on public.store_sales	[10483.634,11474.944]	2879987999	2879987999
18	-> CStore Scan on public.store	[0.347,0.463]	2064	2064
19	-> Vector Partition Iterator	[293.977,365.021]	287999764	287999764
20	-> Partitioned CStore Scan on public.store_returns	[289.936,360.808]	287999764	287999764
21	-> CStore Scan on public.item	[3.109,5.245]	300000	300000
22	-> CStore Scan on public.customer	[113.871,141.791]	12000000	12000000

(22 rows)

发现时间基本花在了第 6 层 redistribute 算子上，需要进一步优化。

4. 由于最后一层 redistribute 包含倾斜，所以时间较长。为了避免倾斜，需要将 item 表放在最后 join，由于 item 表的 join 并不能使行数减少。修改 hint 如下并执行，计划如下，运行时间 120s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((customer_address (store_sales store_returns store customer) item))
c_last_name ...
```

id	operation	A-time	A-rows	E-rows
1	-> Row Adapter	120377.258	1	1
2	-> Vector Aggregate	120377.245	1	1
3	-> Vector Streaming (type: GATHER)	120377.091	24	24
4	-> Vector Aggregate	[120184.884,120301.704]	24	24
5	-> Vector Hash Aggregate	[120183.119,120297.845]	647824	187770504
6	-> Vector Streaming(type: REDISTRIBUTE)	[87775.682,106070.878]	666834733	187770507
7	-> Vector Hash Join (8,22)	[22323.764,49878.523]	666834733	187770507
8	-> Vector Hash Join (9,11)	[21129.236,45208.255]	666834733	187770507
9	-> Vector Streaming(type: REDISTRIBUTE)	[37.859,75.412]	6000000	6000000
10	-> CStore Scan on public.customer_address	[74.798,114.449]	6000000	6000000
11	-> Vector Streaming(type: REDISTRIBUTE)	[15714.458,15824.928]	24661764	24112909
12	-> Vector Hash Join (13,21)	[14637.516,14955.464]	24661764	24112909
13	-> Vector Streaming(type: REDISTRIBUTE)	[13898.593,14333.200]	25258268	25252751
14	-> Vector Hash Join (15,19)	[14166.917,15378.244]	25258268	25252751
15	-> Vector Hash Join (16,18)	[11272.239,12052.532]	252564412	472070592
16	-> Vector Partition Iterator	[10409.566,11127.981]	2879987999	2879987999
17	-> Partitioned CStore Scan on public.store_sales	[10365.838,11077.601]	2879987999	2879987999
18	-> CStore Scan on public.store	[0.431,0.609]	2064	2064
19	-> Vector Partition Iterator	[343.780,408.254]	287999764	287999764
20	-> Partitioned CStore Scan on public.store_returns	[339.844,403.923]	287999764	287999764
21	-> CStore Scan on public.customer	[117.234,163.598]	12000000	12000000
22	-> Vector Streaming(type: BROADCAST)	[44.571,130.129]	7200000	7200000
23	-> CStore Scan on public.item	[4.169,6.347]	300000	300000

(23 rows)

该计划中的 redistribute 问题并没有解决，因为第 22 层 item 表做了 broadcast，导致与 customer_address 表 join 后的倾斜并没有被消除掉。

5. 增加如下禁止 item 表做 broadcast 的 hint，使与 customer_address join 的表做 redistribute（也可以进行 join 表 redistribute 的 hint），计划如下，运行时间 105s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((customer_address (store_sales store_returns store customer) item))
no broadcast(item)*/
c_last_name ...
```

id	operation	A-time	A-rows	E-rows
1	-> Row Adapter	105854.957	1	1
2	-> Vector Aggregate	105854.948	1	1
3	-> Vector Streaming (type: GATHER)	105854.825	24	24
4	-> Vector Aggregate	[105706.709,105776.135]	24	24
5	-> Vector Hash Aggregate	[105705.061,105773.013]	647824	187770504
6	-> Vector Streaming(type: REDISTRIBUTE)	[70701.966,89973.672]	666834733	187770507
7	-> Vector Hash Join (8,23)	[71759.500,79018.433]	666834733	187770507
8	-> Vector Streaming(type: REDISTRIBUTE)	[69794.307,77269.178]	666834733	187770507
9	-> Vector Hash Join (10,12)	[21443.307,46714.378]	666834733	187770507
10	-> Vector Streaming(type: REDISTRIBUTE)	[41.295,83.419]	6000000	6000000
11	-> CStore Scan on public.customer_address	[70.405,166.072]	6000000	6000000
12	-> Vector Streaming(type: REDISTRIBUTE)	[15689.053,15788.475]	24661764	24112909
13	-> Vector Hash Join (14,22)	[14517.847,14712.929]	24661764	24112909
14	-> Vector Streaming(type: REDISTRIBUTE)	[13806.733,14089.770]	25258268	25252751
15	-> Vector Hash Join (16,20)	[13709.384,15095.449]	25258268	25252751
16	-> Vector Hash Join (17,19)	[10944.796,11827.285]	252564412	472070592
17	-> Vector Partition Iterator	[10070.316,10884.728]	2879987999	2879987999
18	-> Partitioned CStore Scan on public.store_sales	[10018.966,10828.990]	2879987999	2879987999
19	-> CStore Scan on public.store	[0.447,0.568]	2064	2064
20	-> Vector Partition Iterator	[293.042,329.056]	287999764	287999764
21	-> Partitioned CStore Scan on public.store_returns	[288.631,324.782]	287999764	287999764
22	-> CStore Scan on public.customer	[113.735,138.235]	12000000	12000000
23	-> CStore Scan on public.item	[3.127,5.357]	300000	300000

(23 rows)

6. 发现最后一层使用单层 Agg，但行数缩减较多。使用相同的 hint，同时结合参数 best_agg_plan=3 进行双层 Agg 调优，最终计划如下图所示，运行时间 94s，完成调优。

```
 id |                                  operation                                  |        A-time         |   A-rows   |   E-rows   |
----+-----------------------------------------------------------------------------+-----------------------+------------+------------+
  1 | ->  Row Adapter                                                             | 94004.670             |          1 |          1 |
  2 |    ->  Vector Aggregate                                                     | 94004.655             |          1 |          1 |
  3 |       ->  Vector Streaming (type: GATHER)                                   | 94004.504             |         24 |         24 |
  4 |          ->  Vector Aggregate                                               | [93833.832,93928.052] |         24 |         24 |
  5 |             ->  Vector Hash Aggregate                                       | [93832.460,93926.412] |     647824 |  187770507 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                     | [93640.866,93787.939] |     647824 |  183912384 |
  7 |                   ->  Vector Hash Aggregate                                 | [93687.544,93791.242] |     647824 |  183912384 |
  8 |                      ->  Vector Hash Join (9,24)                            | [70025.469,72773.161] |  666834733 |  187770507 |
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)            | [68242.223,71275.972] |  666834733 |  187770507 |
 10 |                            ->  Vector Hash Join (11,13)                     | [21421.136,44830.306] |  666834733 |  187770507 |
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)      | [35.444,71.328]       |    6000000 |    6000000 |
 12 |                                  ->  CStore Scan on public.customer_address | [67.246,119.224]      |    6000000 |    6000000 |
 13 |                               ->  Vector Streaming(type: REDISTRIBUTE)      | [16089.853,16212.570] |   24661764 |   24112909 |
 14 |                                  ->  Vector Hash Join (15,23)               | [14822.972,15188.942] |   24661764 |   24112909 |
 15 |                                     ->  Vector Streaming(type: REDISTRIBUTE) | [14061.867,14604.162] |   25258268 |   25252751 |
 16 |                                        ->  Vector Hash Join (17,21)         | [13949.756,15492.311] |   25258268 |   25252751 |
 17 |                                           ->  Vector Hash Join (18,20)      | [10935.742,12160.719] |  252564412 |  472070592 |
 18 |                                              ->  Vector Partition Iterator  | [10052.958,11194.962] | 2879987999 | 2879987999 |
 19 |                                                 ->  Partitioned CStore Scan on public.store_sales | [10008.415,11143.984] | 2879987999 | 2879987999 |
 20 |                                              ->  CStore Scan on public.store | [0.452,0.839]        |       2064 |       2064 |
 21 |                                           ->  Vector Partition Iterator     | [298.235,332.736]     |  287999764 |  287999764 |
 22 |                                              ->  Partitioned CStore Scan on public.store_returns | [294.067,327.629] |  287999764 |  287999764 |
 23 |                                     ->  CStore Scan on public.customer      | [114.377,145.156]     |   12000000 |   12000000 |
 24 |                         ->  CStore Scan on public.item                      | [3.150,3.530]         |     300000 |     300000 |
(24 rows)
```

如果有统计信息变更引起的查询劣化，可以考虑用 plan hint 来调整到之前的查询计划。这里以 TPCH-Q17 为例，在收集 default_statistics_target 设置为－2 的统计信息之后，计划相比于默认统计信息发生劣化。

1. 默认统计信息（default_statistics_target 设置为 100）的计划如下：

```
 id |                                 operation                                  |         A-time
----+----------------------------------------------------------------------------+-------------------------
  1 | ->  Row Adapter                                                            | 265006.779
  2 |    ->  Vector Aggregate                                                    | 265006.764
  3 |       ->  Vector Streaming (type: GATHER)                                  | 265006.071
  4 |          ->  Vector Aggregate                                              | [263699.512,264503.084]
  5 |             ->  Vector Hash Join (6,17)                                    | [263676.665,264477.932]
  6 |                ->  Vector Streaming(type: LOCAL GATHER dop: 1/4)           | [1.998,7.594]
  7 |                   ->  Vector Hash Aggregate                                | [201775.393,202432.672]
  8 |                      ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 4/4) | [201567.130,202231.524]
  9 |                         ->  Vector Hash Join (10,12)                       | [170675.231,199908.410]
 10 |                            ->  Vector Partition Iterator                   | [34847.797,51968.266]
 11 |                               ->  Partitioned CStore Scan on tpch10wx_col.lineitem | [33805.013,51137.657]
 12 |                            ->  Vector Hash Aggregate                       | [23283.387,25359.493]
 13 |                               ->  Vector Streaming(type: SPLIT BROADCAST dop: 4/4) | [12850.624,14608.515]
 14 |                                  ->  Vector Hash Aggregate                 | [2690.439,3616.623]
 15 |                                     ->  Vector Partition Iterator          | [2659.700,3579.390]
 16 |                                        ->  Partitioned CStore Scan on tpch10wx_col.part | [2642.213,3559.093]
 17 |                ->  Vector Streaming(type: REDISTRIBUTE dop: 1/4)           | [262300.732,262961.078]
 18 |                   ->  Vector Hash Join (19,21)                             | [225749.727,260990.322]
 19 |                      ->  Vector Partition Iterator                         | [40046.078,56220.694]
 20 |                         ->  Partitioned CStore Scan on tpch10wx_col.lineitem | [39204.414,55328.448]
 21 |                      ->  Vector Streaming(type: SPLIT BROADCAST dop: 4/4)  | [55748.177,61987.136]
 22 |                         ->  Vector Partition Iterator                      | [3042.864,3873.942]
 23 |                            ->  Partitioned CStore Scan on tpch10wx_col.part | [3027.023,3848.159]
(23 rows)
```

2. 统计信息变更（default_statistics_target 设置为－2）的计划如下：

```
id |                                 operation                                  |          A-time
----+----------------------------------------------------------------------------+---------------------------
  1 | ->  Row Adapter                                                            | 1440492.994
  2 |    ->  Vector Aggregate                                                    | 1440492.982
  3 |       ->  Vector Streaming (type: GATHER)                                  | 1440491.021
  4 |          ->  Vector Streaming(type: LOCAL GATHER dop: 1/6)                 | [1439737.284,1440008.568]
  5 |             ->  Vector Aggregate                                           | [1439008.369,1439854.148]
  6 |                ->  Vector Hash Join (7,18)                                 | [1439006.016,1439851.619]
  7 |                   ->  Vector Streaming(type: LOCAL BROADCAST dop: 6/6)     | [2.932,139.405]
  8 |                      ->  Vector Hash Aggregate                             | [190452.312,195910.748]
  9 |                         ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 6/6) | [190171.929,195653.119]
 10 |                            ->  Vector Hash Join (11,13)                    | [161076.195,178831.123]
 11 |                               ->  Vector Partition Iterator                | [27306.318,45564.565]
 12 |                                  ->  Partitioned CStore Scan on tpch10wx_col.lineitem | [26752.444,44912.020]
 13 |                               ->  Vector Hash Aggregate                    | [35601.624,39812.058]
 14 |                                  ->  Vector Streaming(type: SPLIT BROADCAST dop: 6/6) | [23096.460,27057.137]
 15 |                                     ->  Vector Hash Aggregate              | [2372.587,3052.445]
 16 |                                        ->  Vector Partition Iterator       | [2345.381,3012.732]
 17 |                                           ->  Partitioned CStore Scan on tpch10wx_col.part | [2329.874,2989.393]
 18 |                   ->  Vector Hash Join (19,22)                             | [1437388.414,1438470.781]
 19 |                      ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 6/6) | [1392693.529,1408571.859]
 20 |                         ->  Vector Partition Iterator                      | [29065.204,41264.514]
 21 |                            ->  Partitioned CStore Scan on tpch10wx_col.lineitem | [28212.219,40133.491]
 22 |                      ->  Vector Streaming(type: LOCAL REDISTRIBUTE dop: 6/6) | [2570.841,3438.567]
 23 |                         ->  Vector Partition Iterator                      | [2447.569,3276.369]
 24 |                            ->  Partitioned CStore Scan on tpch10wx_col.part | [2432.124,3263.641]
(24 rows)
```

3. 经过对比，劣化的原因主要为 lineitem 和 part 表 join 时 stream 类型由 BroadCast 变更为 Redistribute 导致。可以对语句进行 stream 方式的 hint 来调整到之前的计划，例如：

```
select /*+ no redistribute(part lineitem) */
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#23'
    and p_container = 'MED BOX'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

3.2.3.10 检查隐式转换的性能问题

在某些场景下，数据类型的隐式转换可能会导致潜在的性能问题。请看如下的场景：

```
SET enable fast query shipping = off;
CREATE TABLE t1(c1 VARCHAR, c2 VARCHAR);
CREATE INDEX on t1(c1);
EXPLAIN verbose SELECT * FROM t1 WHERE c1 = 10;
```

上述查询的执行计划如下：

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Streaming (type: GATHER)  (cost=0.06..13.29 rows=1 width=64)
   Output: c1, c2
   Node/s: All datanodes
   ->  Seq Scan on public.t1  (cost=0.00..13.20 rows=1 width=64)
         Output: c1, c2
         Distribute Key: c1
         Filter: ((t1.c1)::bigint = 10)
(7 rows)
```

c1 的数据类型是 varchar，当查询的过滤条件为 c1 = 10 时，优化器默认将 c1 隐式转换为 bigint 类型，导致两个后果：

- 不能进行 DN 裁剪，计划下发到所有 DN 上执行。
- 计划中不能使用 Index Scan 方式扫描数据。

这会引起潜在的性能问题。

当知道了问题原因后，我们可以做针对性的 SQL 改写。对于上面的场景，只要将过滤条件中的常量显示转换为 varchar 类型，结果如下：

```
EXPLAIN verbose SELECT * FROM t1 WHERE c1 = 10::varchar;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Streaming (type: GATHER)  (cost=0.06..8.36 rows=1 width=64)
   Output: c1, c2
   Node/s: datanode2
   ->  Index Scan using t1_c1_idx on public.t1  (cost=0.00..8.27 rows=1 width=64)
         Output: c1, c2
         Distribute Key: c1
         Index Cond: ((t1.c1)::text = '10'::text)
(7 rows)
```

为了提前识别隐式类型转换可能带来的性能影响，我们提供了一个 guc option：check_implicit_conversions。打开该参数后，对于查询中出现的隐式类型转换的索引列，在路径生成阶段进行检查，如果发现索引列没有生成候选的索引扫描路径，则会通过报错的形式提示给用户。举例如下：

```
SET check_implicit_conversions = on;
SELECT * FROM t1 WHERE c1 = 10;
ERROR:  There is no optional index path for index column: "t1"."c1".
Please check for potential performance problem.
```

说明

- 参数 check_implicit_conversions 只用于检查隐式类型转换引起的潜在性能问题，在正式生产环境中请关闭该参数（该参数默认关闭）。
- 在将 check_implicit_conversions 打开时，必须同时关闭 enable_fast_query_shipping 参数，否则由于后一个参数的作用，无法查看对隐式类型转换修复的结果。
- 一个表的候选路径可能包括 seq scan 和 index scan 等多个可能的数据扫描方式，最终执行计划使用的表扫描方式是由执行计划的代价来决定的，因此即使生成了索引扫描的候选路径，也可能生成的最终执行计划中使用其它扫描方式。

3.2.4 实际调优案例

3.2.4.1 案例：选择合适的分布列

现象描述

表定义如下：

```
CREATE TABLE t1 (a int, b int);
CREATE TABLE t2 (a int, b int);
```

执行如下查询：

```
SELECT * FROM t1, t2 WHERE t1.a = t2.b;
```

优化分析

如果将 a 作为 t1 和 t2 的分布列：

```
CREATE TABLE t1 (a int, b int) DISTRIBUTE BY HASH (a);
CREATE TABLE t2 (a int, b int) DISTRIBUTE BY HASH (a);
```

则执行计划将存在“Streaming”，导致 DN 之间存在较大通信数据量，如图 3-11 所示。

图3-11 选择合适的分布列案例（一）

```
postgres=> explain select * from t1, t2 where t1.a = t2.b;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Streaming (type: GATHER)  (cost=245.40..582.15 rows=240 width=16)
   Node/s: All datanodes
   ->  Hash Join  (cost=10.22..24.26 rows=10 width=16)
         Hash Cond: (t1.a = t2.b)
         ->  Seq Scan on t1  (cost=0.00..10.10 rows=10 width=8)
         ->  Hash  (cost=3.79..3.79 rows=10 width=8)
               ->  Streaming(type: REDISTRIBUTE)  (cost=0.00..3.79 rows=10 width=8)
                     Spawn on: All datanodes
                     ->  Seq Scan on t2  (cost=0.00..10.10 rows=10 width=8)
(9 rows)
```

如果将 a 作为 t1 的分布列，将 b 作为 t2 的分布列：

```
CREATE TABLE t1 (a int, b int) DISTRIBUTE BY HASH (a);
CREATE TABLE t2 (a int, b int) DISTRIBUTE BY HASH (b);
```

则执行计划将不包含“Streaming”，减少 DN 之间存在的通信数据量，从而提升查询性能，如图 3-12 所示。

图3-12 选择合适的分布列案例（二）

```
postgres=> explain select * from t1, t2 where t1.a = t2.b;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Streaming (type: GATHER)  (cost=245.40..491.10 rows=240 width=16)
   Node/s: All datanodes
   ->  Hash Join  (cost=10.22..20.46 rows=10 width=16)
         Hash Cond: (t1.a = t2.b)
         ->  Seq Scan on t1  (cost=0.00..10.10 rows=10 width=8)
         ->  Hash  (cost=10.10..10.10 rows=10 width=8)
               ->  Seq Scan on t2  (cost=0.00..10.10 rows=10 width=8)
(7 rows)
```

3.2.4.2 案例：建立合适的索引

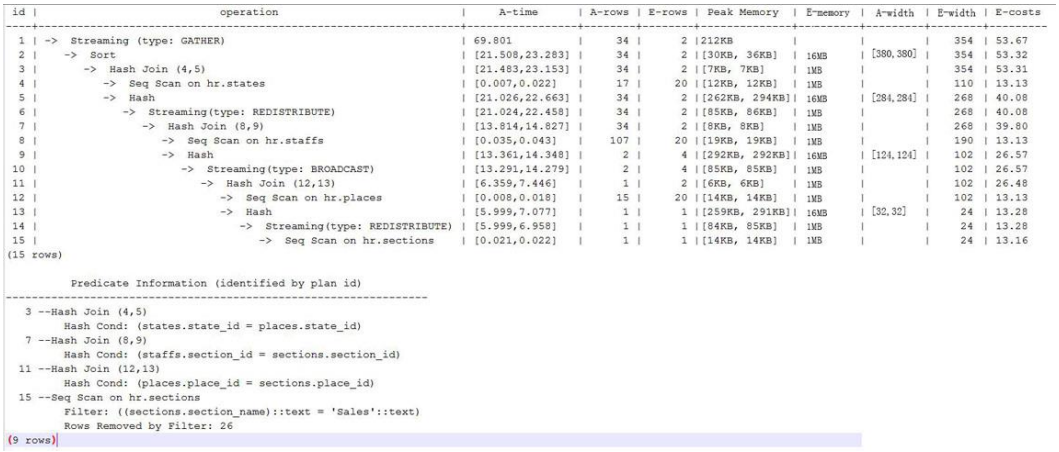
现象描述

查询所有员工的信息：

```
SELECT staff_id,first_name,last_name,employment_id,state_name,city
FROM staffs,sections,states,places
WHERE sections.section_name='Sales'
AND staffs.section_id = sections.section_id
AND sections.place id = places.place id
AND places.state id = states.state id
ORDER BY staff_id;
```

优化分析

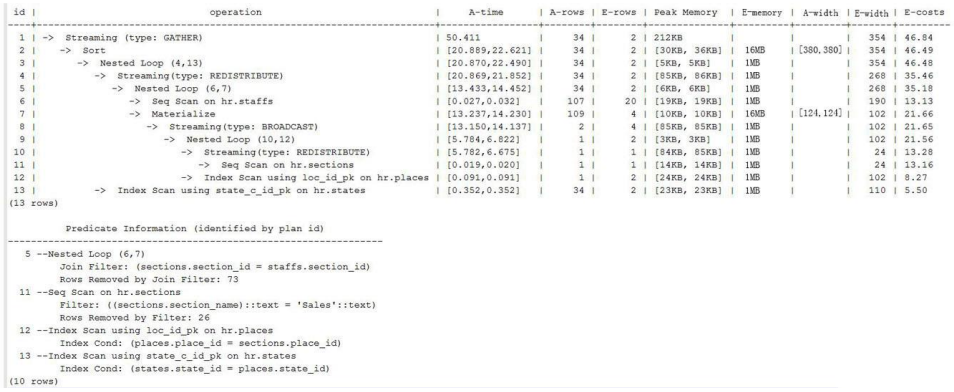
在优化前，没有创建 places.place_id 和 states.state_id 索引，执行计划如下：



```
 id |                          operation                          |     A-time      | A-rows | E-rows | Peak Memory  | E-memory | A-width | E-width | E-costs
----+-------------------------------------------------------------+-----------------+--------+--------+--------------+----------+---------+---------+---------
  1 | ->  Streaming (type: GATHER)                                | 69.801          |     34 |      2 |212KB         |          |         |     354 | 53.67
  2 |   ->  Sort                                                  | [21.508,23.283] |     34 |      2 |[30KB, 36KB]  | 16MB     | [380,380] |     354 | 53.32
  3 |     ->  Hash Join (4,5)                                     | [21.483,23.153] |     34 |      2 |[7KB, 7KB]    | 1MB      |         |     354 | 53.31
  4 |       ->  Seq Scan on hr.states                             | [0.007,0.022]   |     17 |     20 |[12KB, 12KB]  | 1MB      |         |     110 | 13.13
  5 |       ->  Hash                                              | [21.026,22.663] |     34 |      2 |[262KB, 294KB]| 16MB     | [284,284] |     268 | 40.08
  6 |         ->  Streaming(type: REDISTRIBUTE)                   | [21.024,22.458] |     34 |      2 |[85KB, 86KB]  | 1MB      |         |     268 | 40.08
  7 |           ->  Hash Join (8,9)                               | [13.814,14.827] |     34 |      2 |[8KB, 8KB]    | 1MB      |         |     268 | 39.80
  8 |             ->  Seq Scan on hr.staffs                       | [0.035,0.043]   |    107 |     20 |[19KB, 19KB]  | 1MB      |         |     190 | 13.13
  9 |             ->  Hash                                        | [13.361,14.348] |      2 |      4 |[292KB, 292KB]| 16MB     | [124,124] |     102 | 26.57
 10 |               ->  Streaming(type: BROADCAST)                | [13.291,14.279] |      2 |      4 |[85KB, 85KB]  | 1MB      |         |     102 | 26.57
 11 |                 ->  Hash Join (12,13)                       | [6.359,7.446]   |      1 |      2 |[6KB, 6KB]    | 1MB      |         |     102 | 26.48
 12 |                   ->  Seq Scan on hr.places                 | [0.008,0.018]   |     15 |     20 |[14KB, 14KB]  | 1MB      |         |     102 | 13.13
 13 |                   ->  Hash                                  | [5.999,7.077]   |      1 |      1 |[259KB, 291KB]| 16MB     | [32,32] |      24 | 13.28
 14 |                     ->  Streaming(type: REDISTRIBUTE)       | [5.999,6.958]   |      1 |      1 |[84KB, 85KB]  | 1MB      |         |      24 | 13.28
 15 |                       ->  Seq Scan on hr.sections           | [0.021,0.022]   |      1 |      1 |[14KB, 14KB]  | 1MB      |         |      24 | 13.16
(15 rows)

          Predicate Information (identified by plan id)
-------------------------------------------------------------------
   3 --Hash Join (4,5)
         Hash Cond: (states.state_id = places.state_id)
   7 --Hash Join (8,9)
         Hash Cond: (staffs.section_id = sections.section_id)
  11 --Hash Join (12,13)
         Hash Cond: (places.place_id = sections.place_id)
  15 --Seq Scan on hr.sections
         Filter: ((sections.section_name)::text = 'Sales'::text)
         Rows Removed by Filter: 26
(9 rows)
```

建议在 places.place_id 和 states.state_id 列上建立 2 个索引，执行计划如下：



```
 id |                          operation                          |     A-time      | A-rows | E-rows | Peak Memory  | E-memory | A-width | E-width | E-costs
----+-------------------------------------------------------------+-----------------+--------+--------+--------------+----------+---------+---------+---------
  1 | ->  Streaming (type: GATHER)                                | 50.411          |     34 |      2 | 212KB        |          |         |     354 | 46.84
  2 |   ->  Sort                                                  | [20.889,22.621] |     34 |      2 | [30KB, 36KB] | 16MB     | [380,380] |     354 | 46.49
  3 |     ->  Nested Loop (4,13)                                  | [20.870,22.490] |     34 |      2 | [5KB, 5KB]   | 1MB      |         |     354 | 46.48
  4 |       ->  Streaming(type: REDISTRIBUTE)                     | [20.869,21.852] |     34 |      2 | [85KB, 86KB] | 1MB      |         |     268 | 35.46
  5 |         ->  Nested Loop (6,7)                               | [13.433,14.452] |     34 |      2 | [6KB, 6KB]   | 1MB      |         |     268 | 35.18
  6 |           ->  Seq Scan on hr.staffs                         | [0.027,0.032]   |    107 |     20 | [19KB, 19KB] | 1MB      |         |     190 | 13.13
  7 |           ->  Materialize                                   | [13.237,14.230] |    109 |      4 | [10KB, 10KB] | 16MB     | [124,124] |     102 | 21.66
  8 |             ->  Streaming(type: BROADCAST)                  | [13.150,14.137] |      2 |      4 | [85KB, 85KB] | 1MB      |         |     102 | 21.65
  9 |               ->  Nested Loop (10,12)                       | [5.784,6.822]   |      1 |      2 | [3KB, 3KB]   | 1MB      |         |     102 | 21.56
 10 |                 ->  Streaming(type: REDISTRIBUTE)           | [5.782,6.675]   |      1 |      1 | [84KB, 85KB] | 1MB      |         |      24 | 13.28
 11 |                   ->  Seq Scan on hr.sections               | [0.019,0.020]   |      1 |      1 | [14KB, 14KB] | 1MB      |         |      24 | 13.16
 12 |                 ->  Index Scan using loc_id_pk on hr.places | [0.091,0.091]   |      1 |      2 | [24KB, 24KB] | 1MB      |         |     102 | 8.27
 13 |       ->  Index Scan using state_c_id_pk on hr.states       | [0.352,0.352]   |     34 |      2 | [23KB, 23KB] | 1MB      |         |     110 | 5.50
(13 rows)

          Predicate Information (identified by plan id)
-------------------------------------------------------------------
   5 --Nested Loop (6,7)
         Join Filter: (sections.section_id = staffs.section_id)
         Rows Removed by Join Filter: 73
  11 --Seq Scan on hr.sections
         Filter: ((sections.section_name)::text = 'Sales'::text)
         Rows Removed by Filter: 26
  12 --Index Scan using loc_id_pk on hr.places
         Index Cond: (places.place_id = sections.place_id)
  13 --Index Scan using state_c_id_pk on hr.states
         Index Cond: (states.state_id = places.state_id)
(10 rows)
```

3.2.4.3 案例：增加 JOIN 列非空条件

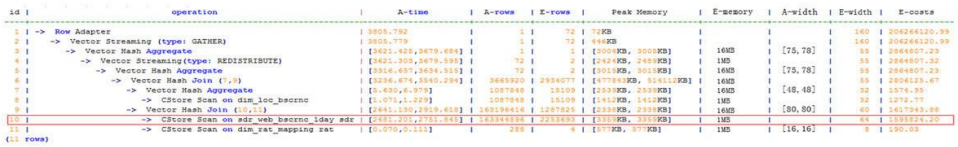
现象描述

```
SELECT
 *
FROM
( ( SELECT
  STARTTIME STTIME,
  SUM(NVL(PAGE DELAY MSEL,0)) PAGE DELAY MSEL,
  SUM(NVL(PAGE SUCCEED TIMES,0)) PAGE SUCCEED TIMES,
  SUM(NVL(FST PAGE REQ NUM,0)) FST PAGE REQ NUM,
  SUM(NVL(PAGE AVG SIZE,0)) PAGE AVG SIZE,
  SUM(NVL(FST PAGE ACK NUM,0)) FST PAGE ACK NUM,
  SUM(NVL(DATATRANS DW DURATION,0)) DATATRANS DW DURATION,
  SUM(NVL(PAGE SR DELAY MSEL,0)) PAGE SR DELAY MSEL
 FROM
  PS.SDR WEB BSCRNC 1DAY SDR
  INNER JOIN (SELECT
      BSCRNC ID,
      BSCRNC NAME,
      ACCESS TYPE,
      ACCESS TYPE ID
     FROM
      nethouse.DIM_LOC_BSCRNC
```

```
    GROUP BY
     BSCRNC_ID,
     BSCRNC_NAME,
     ACCESS_TYPE,
     ACCESS_TYPE_ID) DIM
 ON SDR.BSCRNC_ID = DIM.BSCRNC_ID
 AND DIM.ACCESS_TYPE_ID IN (0,1,2)
 INNER JOIN nethouse.DIM_RAT_MAPPING RAT
 ON (RAT.RAT = SDR.RAT)
WHERE
 ( (STARTTIME >= 1461340800
 AND STARTTIME < 1461427200) )
 AND RAT.ACCESS_TYPE_ID IN (0,1,2)
 --and SDR.BSCRNC_ID is not null
GROUP BY
 STTIME ) ) ;
```

执行计划如图 3-13 所示。

图3-13 增加 JOIN 列非空条件（一）



优化分析

1. 分析执行计划图 3-13 可知，在顺序扫描阶段耗时较多。
2. 多表 JOIN 中，由于表 PS.SDR_WEB_BSCRNC_1DAY 的 JOIN 列“BSCRNC_ID”存在大量空值，JOIN 性能差。

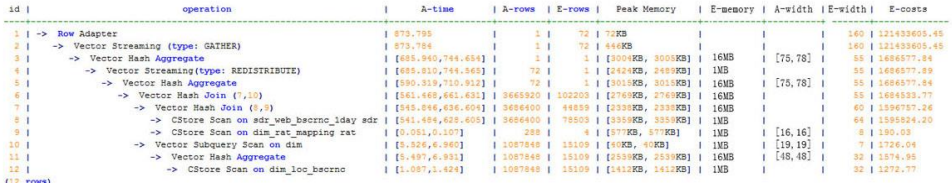
   建议在语句中手动添加 JOIN 列的非空判断，修改后的语句如下所示。

```
SELECT
 *
FROM
( ( SELECT
  STARTTIME STTIME,
  SUM(NVL(PAGE DELAY MSEL,0)) PAGE DELAY MSEL,
  SUM(NVL(PAGE SUCCEED TIMES,0)) PAGE SUCCEED TIMES,
  SUM(NVL(FST PAGE REQ NUM,0)) FST PAGE REQ NUM,
  SUM(NVL(PAGE AVG SIZE,0)) PAGE AVG SIZE,
  SUM(NVL(FST PAGE ACK NUM,0)) FST PAGE ACK NUM,
  SUM(NVL(DATATRANS DW DURATION,0)) DATATRANS DW DURATION,
  SUM(NVL(PAGE SR DELAY MSEL,0)) PAGE SR DELAY MSEL
 FROM
  PS.SDR WEB BSCRNC 1DAY SDR
  INNER JOIN (SELECT
     BSCRNC ID,
     BSCRNC NAME,
     ACCESS TYPE,
     ACCESS TYPE ID
    FROM
```

```
    nethouse.DIM_LOC_BSCRNC
   GROUP BY
    BSCRNC_ID,
    BSCRNC_NAME,
    ACCESS_TYPE,
    ACCESS_TYPE_ID) DIM
 ON SDR.BSCRNC_ID = DIM.BSCRNC_ID
 AND DIM.ACCESS_TYPE_ID IN (0,1,2)
 INNER JOIN nethouse.DIM_RAT_MAPPING RAT
 ON (RAT.RAT = SDR.RAT)
WHERE
 ( (STARTTIME >= 1461340800
 AND STARTTIME < 1461427200) )
 AND RAT.ACCESS_TYPE_ID IN (0,1,2)
 and SDR.BSCRNC_ID is not null
GROUP BY
 STTIME ) ) A;
```

执行计划如图 3-14 所示。

图3-14 增加 JOIN 列非空条件（二）



3.2.4.4 案例：使排序下推

现象描述

在做场景性能测试时，发现某场景大部分时间是 CN 端在做 window agg，占到总执行时间 95%以上，系统资源不能充分利用。研究发现该场景的特点是：将两列分别求 sum 作为一个子查询，外层对两列的和再求和后做 trunc，然后排序。

表结构如下所示：

```
CREATE TABLE public.test(imsi int,L4 DW THROUGHPUT int,L4 UL THROUGHPUT int)
with (orientation = column) DISTRIBUTE BY hash(imsi);
```

查询语句如下所示：

```
SELECT COUNT(1) over() AS DATACNT,
IMSI AS IMSI IMSI,
CAST(TRUNC(((SUM(L4 UL THROUGHPUT) + SUM(L4 DW THROUGHPUT))), 0) AS
DECIMAL(20)) AS TOTAL VOLOME KPIID
FROM public.test AS test
GROUP BY IMSI
order by TOTAL_VOLOME_KPIID DESC;
```

执行计划如下：

```
Row Adapter  (cost=10.70..10.70 rows=10 width=12)
   ->  Vector Sort  (cost=10.68..10.70 rows=10 width=12)
         Sort Key: ((trunc((((sum(l4_ul_throughput)) +
(sum(l4_dw_throughput))))::numeric, 0))::numeric(20,0))
         ->  Vector WindowAgg  (cost=10.09..10.51 rows=10 width=12)
               ->  Vector Streaming (type: GATHER)  (cost=242.04..246.84 rows=240
width=12)
                     Node/s: All datanodes
                     ->  Vector Hash Aggregate  (cost=10.09..10.29 rows=10 width=12)
                           Group By Key: imsi
                           ->  CStore Scan on test  (cost=0.00..10.01 rows=10 width=12)
```

可以看到 window agg 和 sort 全部在 CN 端执行，耗时非常严重。

优化分析

尝试将语句改写为子查询。

```
SELECT COUNT(1) over() AS DATACNT, IMSI IMSI, TOTAL VOLOME KPIID
FROM (SELECT IMSI AS IMSI IMSI,
CAST(TRUNC(((SUM(L4 UL THROUGHPUT) + SUM(L4 DW THROUGHPUT))),
0) AS DECIMAL(20)) AS TOTAL VOLOME KPIID
FROM public.test AS test
GROUP BY IMSI
ORDER BY TOTAL_VOLOME_KPIID DESC);
```

将 trunc 两列的和作为一个子查询，然后在子查询的外面做 window agg，这样排序就可以下推了，执行计划如下：

```
Row Adapter  (cost=10.70..10.70 rows=10 width=24)
   ->  Vector WindowAgg  (cost=10.45..10.70 rows=10 width=24)
         ->  Vector Streaming (type: GATHER)  (cost=250.83..253.83 rows=240 width=24)
               Node/s: All datanodes
               ->  Vector Sort  (cost=10.45..10.48 rows=10 width=12)
                     Sort Key: ((trunc(((sum(test.l4_ul_throughput) +
sum(test.l4_dw_throughput)))::numeric, 0))::numeric(20,0))
                     ->  Vector Hash Aggregate  (cost=10.09..10.29 rows=10 width=12)
                           Group By Key: test.imsi
                           ->  CStore Scan on test  (cost=0.00..10.01 rows=10 width=12)
```

经过 SQL 改写，性能由 120s 提升到 7s，优化效果明显。

3.2.4.5 案例：设置 cost_param 对查询性能优化

现象描述 1

cost_param 的 bit0(set cost_param=1)值为 1 时，表示对于求由不等式（!=）条件连接的选择率时选择一种改良机制，此方法在自连接（两个相同的表之间连接）的估算中更加准确。下面查询的例子是 cost_param 的 bit0 为 1 时的优化场景。当前版本已弃用 cost_param & 1 不为 0 时的路径，默认选择已优化的估算公式。

注：选择率是两表 join 时，满足 join 条件的行数在 join 结果集中所占的比率。

表结构如下所示：

```
CREATE TABLE LINEITEM
(
L_ORDERKEY BIGINT NOT NULL
, L_PARTKEY BIGINT NOT NULL
, L_SUPPKEY BIGINT NOT NULL
, L_LINENUMBER BIGINT NOT NULL
, L_QUANTITY DECIMAL(15,2) NOT NULL
, L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL
, L_DISCOUNT DECIMAL(15,2) NOT NULL
, L_TAX DECIMAL(15,2) NOT NULL
, L_RETURNFLAG CHAR(1) NOT NULL
, L_LINESTATUS CHAR(1) NOT NULL
, L_SHIPDATE DATE NOT NULL
, L_COMMITDATE DATE NOT NULL
, L_RECEIPTDATE DATE NOT NULL
, L_SHIPINSTRUCT CHAR(25) NOT NULL
, L_SHIPMODE CHAR(10) NOT NULL
, L_COMMENT VARCHAR(44) NOT NULL
) with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(L_ORDERKEY);

CREATE TABLE ORDERS
(
O_ORDERKEY BIGINT NOT NULL
, O CUSTKEY BIGINT NOT NULL
, O ORDERSTATUS CHAR(1) NOT NULL
, O TOTALPRICE DECIMAL(15,2) NOT NULL
, O ORDERDATE DATE NOT NULL
, O ORDERPRIORITY CHAR(15) NOT NULL
, O CLERK CHAR(15) NOT NULL
, O SHIPPRIORITY BIGINT NOT NULL
, O COMMENT VARCHAR(79) NOT NULL
)with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(O_ORDERKEY);
```

查询语句如下所示：

```
explain verbose select
count(*) as numwait
from
lineitem l1,
orders
where
o orderkey = l1.l orderkey
and o orderstatus = 'F'
and l1.l receiptdate > l1.l commitdate
and not exists (
select
*
from
lineitem l3
where
l3.l orderkey = l1.l orderkey
and l3.l suppkey <> l1.l suppkey
and l3.l receiptdate > l3.l commitdate
)
order by
numwait desc;
```

执行计划如下图所示：（verbose 条件下，新增 distinct 列，受 cost off/on 控制，hashjoin 行显示内外表的 distinct 估值，其他行为空）

```
 id |                          operation                          | E-rows | E-distinct | E-width | E-costs
----+-------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Row Adapter                                             |      1 |            |       8 | 39.36
  2 |    ->  Vector Sort                                          |      1 |            |       8 | 39.36
  3 |       ->  Vector Aggregate                                  |      1 |            |       8 | 39.34
  4 |          ->  Vector Streaming (type: GATHER)                |      2 |            |       8 | 39.34
  5 |             ->  Vector Aggregate                            |      2 |            |       8 | 39.25
  6 |                ->  Vector Hash Anti Join (7, 10)            |      2 | 4, 5       |       0 | 39.24
  7 |                   ->  Vector Hash Join (8,9)                |      2 | 200, 1     |      16 | 26.12
  8 |                      ->  CStore Scan on public.lineitem 11  |      7 |            |      16 | 13.05
  9 |                      ->  CStore Scan on public.orders       |      1 |            |       8 | 13.05
 10 |                   ->  CStore Scan on public.lineitem 13     |      7 |            |      16 | 13.05
(10 rows)
```

优化分析 1

以上查询为 lineitem 表自连接的 Anti Join，当使用 cost_param 的 bit0 为 0 时，估算 Anti Join 的行数与实际行数相差很大，导致查询性能下降。可以通过设置 cost_param 的 bit0 为 1 时，使 Anti Join 的行数估算更准确，从而提高查询性能。优化后的执行计划如下：

```
id |                          operation                          |  E-rows  | E-memory | E-width |  E-costs
---+-------------------------------------------------------------+----------+----------+---------+-----------
 1 |  -> Row Adapter                                             |        1 |          |       0 | 9104892.37 9
 2 |    -> Vector Sort                                           |        1 |          |       0 | 9104892.37 9
 3 |      -> Vector Aggregate                                    |        1 |          |       0 | 9104892.35 8
 4 |        -> Vector Streaming (type: GATHER)                   |       48 |          |       0 | 9104892.35 8
 5 |          -> Vector Aggregate                                |       48 | 1MB      |       0 | 9104890.82 5
 6 |            -> Vector Hash Join (7.12)                       | 2526630903 | 929MB  |       0 | 8973295.45 4
 7 |              -> Vector Hash Anti Join (8. 10)               | 1999996587 | 3178MB |       8 | 7198231.14
 8 |                -> Vector Partition Iterator                 | 1999996587 | 1MB    |      16 | 3000158.25
 9 |                  -> Partitioned CStore Scan on public.lineitem 11 | 1999996587 | 1MB | 16 | 3000158.25 1
10 |                -> Vector Partition Iterator                 | 1999996587 | 1MB    |      16 | 3000158.25
11 |                  -> Partitioned CStore Scan on public.lineitem 13 | 1999996587 | 1MB | 16 | 3000158.25
12 |                -> Vector Partition Iterator                 |  730839014 | 1MB    |       8 | 589611.00
13 |                  -> Partitioned CStore Scan on public.orders |  730839014 | 1MB    |       8 | 589611.00
(13 rows)
```

现象描述 2

当 cost_param 的 bit1(set cost_param=2)为 1 时，表示求多个过滤条件（Filter）的选择率时，选择最小的作为总的选择率，而非两者乘积，此方法在过滤条件的列之间关联性较强时估算更加准确。下面查询的例子是 cost_param 的 bit1 为 1 时的优化场景。

表结构如下所示：

```
CREATE TABLE NATION
(
  N_NATIONKEY INT NOT NULL
, N NAME      CHAR(25) NOT NULL
, N REGIONKEY INT NOT NULL
, N COMMENT   VARCHAR(152)
) distribute by replication;
CREATE TABLE SUPPLIER
(
  S SUPPKEY   BIGINT NOT NULL
, S_NAME      CHAR(25) NOT NULL
```

```
, S_ADDRESS   VARCHAR(40) NOT NULL
, S_NATIONKEY INT NOT NULL
, S_PHONE     CHAR(15) NOT NULL
, S_ACCTBAL   DECIMAL(15,2) NOT NULL
, S_COMMENT   VARCHAR(101) NOT NULL
) distribute by hash(S_SUPPKEY);
CREATE TABLE PARTSUPP
(
  PS_PARTKEY    BIGINT NOT NULL
, PS_SUPPKEY    BIGINT NOT NULL
, PS_AVAILQTY   BIGINT NOT NULL
, PS_SUPPLYCOST DECIMAL(15,2)NOT NULL
, PS_COMMENT    VARCHAR(199) NOT NULL
)distribute by hash(PS_PARTKEY);
```

查询语句如下所示：

```
set cost_param=2;
explain verbose select
nation,
sum(amount) as sum profit
from
(
select
n name as nation,
l extendedprice * (1 - l discount) - ps supplycost * l quantity as amount
from
supplier,
lineitem,
partsupp,
nation
where
s suppkey = l suppkey
and ps suppkey = l suppkey
and ps partkey = l partkey
and s nationkey = n nationkey
) as profit
group by nation
order by nation;
```

当 cost_param 的 bit1 为 0 时，执行计划如下图所示：

```
 id |                           operation                           | E-rows | E-distinct | E-width | E-costs
----+---------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Sort                                                      |      1 |            |     208 | 61.52
  2 |    ->  HashAggregate                                          |      1 |            |     208 | 61.51
  3 |       ->  Streaming (type: GATHER)                            |      2 |            |     208 | 61.51
  4 |          ->  HashAggregate                                    |      2 |            |     208 | 61.36
  5 |             ->  Hash Join (6,7)                               |      2 | 20, 15     |     176 | 61.33
  6 |                ->  Seq Scan on public.nation                  |     40 |            |     108 | 20.20
  7 |                ->  Hash                                       |      2 |            |      76 | 41.04
  8 |                   ->  Hash Join (9,16)                        |      2 | 10, 13     |      76 | 41.04
  9 |                      ->  Streaming(type: REDISTRIBUTE)        |      2 |            |      88 | 27.73
 10 |                         ->  Hash Join (11,14)                 |      2 | 10, 13     |      88 | 27.62
 11 |                            ->  Streaming(type: REDISTRIBUTE)  |     20 |            |      70 | 14.19
 12 |                               ->  Row Adapter                 |     21 |            |      70 | 13.01
 13 |                                  ->  CStore Scan on public.lineitem |     20 |            |      70 | 13.01
 14 |                            ->  Hash                           |     21 |            |      34 | 13.13
 15 |                               ->  Seq Scan on public.partsupp |     20 |            |      34 | 13.13
 16 |                      ->  Hash                                 |     21 |            |      12 | 13.13
 17 |                         ->  Seq Scan on public.supplier       |     20 |            |      12 | 13.13
(17 rows)
```

优化分析 2

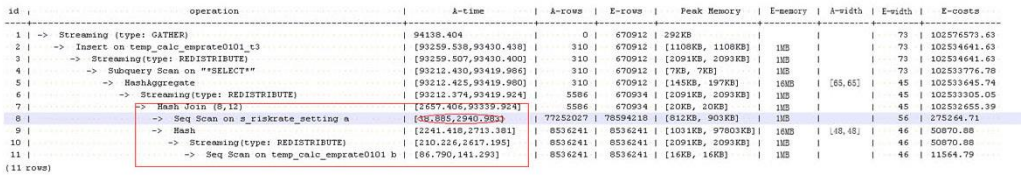
在以上查询中，supplier、lineitem、partsupp 三表做 hashjoin 的条件为 (lineitem.l_suppkey = supplier.s_suppkey) AND (lineitem.l_partkey = partsupp.ps_partkey)，此 hashjoin 条件中存在两个过滤条件，这前一个过滤条件中的 lineitem.l_suppkey 和后一个过滤条件中的 lineitem.l_partkey 同为 lineitem 表的两列，这两列存在强相关的关联关系。在这种情况，估算 hashjoin 条件的选择率时，如果使用 cost_param 的 bit1 为 0 时，实际是将 AND 的两个过滤条件分别计算的 2 个选择率的值相乘来得到 hashjoin 条件的选择率，导致行数估算不准确，查询性能较差。所以需要将 cost_param 的 bit1 为 1 时，选择最小的选择率作为总的选择率估算行数比较准确，查询性能较好，优化后的计划如下图所示：

```
 id |                               operation                               | E-rows | E-distinct | E-width | E-costs
----+-----------------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Sort                                                              |     10 |            |     208 | 64.42
  2 |    ->  HashAggregate                                                  |     10 |            |     208 | 64.23
  3 |       ->  Streaming (type: GATHER)                                    |     20 |            |     208 | 64.23
  4 |          ->  HashAggregate                                            |     20 |            |     208 | 62.71
  5 |             ->  Hash Join (6,7)                                       |     20 | 20, 10     |     176 | 62.46
  6 |                ->  Seq Scan on public.nation                          |     40 |            |     108 | 20.20
  7 |                ->  Hash                                               |     20 |            |      76 | 41.97
  8 |                   ->  Hash Join (9,16)                                |     20 | 10, 13     |      76 | 41.97
  9 |                      ->  Streaming(type: REDISTRIBUTE)                |     20 |            |      82 | 28.54
 10 |                         ->  Hash Join (11,14)                         |     20 | 10, 13     |      82 | 27.63
 11 |                            ->  Streaming(type: REDISTRIBUTE)          |     20 |            |      70 | 14.19
 12 |                               ->  Row Adapter                         |     21 |            |      70 | 13.01
 13 |                                  ->  CStore Scan on public.lineitem   |     20 |            |      70 | 13.01
 14 |                            ->  Hash                                   |     21 |            |      12 | 13.13
 15 |                               ->  Seq Scan on public.supplier         |     20 |            |      12 | 13.13
 16 |                      ->  Hash                                         |     21 |            |      34 | 13.13
 17 |                         ->  Seq Scan on public.partsupp               |     20 |            |      34 | 13.13
(17 rows)
```

3.2.4.6 案例：调整分布键

现象描述

某局点测试过程中 EXPLAIN ANALYZE 后有如下情况：



```
 id |                                operation                                |          A-time          | A-rows | E-rows |       Peak Memory       | E-memory | A-width | E-width |   E-costs
----+-------------------------------------------------------------------------+--------------------------+--------+--------+-------------------------+----------+---------+---------+-------------
  1 | ->  Streaming (type: GATHER)                                            | 94138.404                |      0 |  670912 | 292KB                   |          |         |      73 | 102576573.63
  2 |    ->  Insert on temp_calc_emprate0101_t3                               | [93259.538,93430.438]    |    310 |  670912 | [1108KB, 1108KB]        | 1MB      |         |      73 | 102534641.63
  3 |       ->  Streaming(type: REDISTRIBUTE)                                 | [93259.507,93430.400]    |    310 |  670912 | [2091KB, 2093KB]        | 1MB      |         |      73 | 102534641.63
  4 |          ->  Subquery Scan on "*SELECT*"                                | [93212.430,93419.986]    |    310 |  670912 | [7KB, 7KB]              | 1MB      |         |      73 | 102533776.78
  5 |             ->  HashAggregate                                           | [93212.425,93419.980]    |    310 |  670912 | [145KB, 197KB]          | 16MB     | [65,65] |      45 | 102533645.74
  6 |                ->  Streaming(type: REDISTRIBUTE)                        | [93212.374,93419.924]    |   5586 |  670934 | [2091KB, 2093KB]        | 1MB      |         |      45 | 102533305.05
  7 |                   ->  Hash Join (8,12)                                  | [2657.406,93339.924]     |   5586 |  670934 | [20KB, 20KB]            | 1MB      |         |      45 | 102532655.39
  8 |                      ->  Seq Scan on s_riskrate_setting a               | [48.885,2940.983]        |  77252027 | 78594218 | [812KB, 903KB]        | 1MB      |         |      56 | 275264.71
  9 |                      ->  Hash                                           | [2241.418,2713.381]      |  8536241 |  8536241 | [1031KB, 97803KB]     | 16MB     | [48,48] |      46 | 50870.88
 10 |                         ->  Streaming(type: REDISTRIBUTE)               | [210.226,2617.195]       |  8536241 |  8536241 | [2091KB, 2093KB]      | 1MB      |         |      46 | 50870.88
 11 |                            ->  Seq Scan on temp_calc_emprate0101_b       | [86.790,141.293]         |  8536241 |  8536241 | [16KB, 16KB]          | 1MB      |         |      46 | 11564.79
(11 rows)
```

从执行信息上比较明确的可以看出 HashJoin 是整个计划的性能瓶颈点，并且从 HashJoin 的执行时间信息[2657.406,93339.924](数值的具体含义请参见 3.2.3.2.2 详解)，上可以看出 HashJoin 在不同的 DN 上存在严重的计算偏斜。

同时在 Memory Information(如下图)中可以看出各个节点的内存资源消耗也存在极为严重的偏斜。

```
                Memory Information (identified by plan id)
--------------------------------------------------------------------------------
Coordinator:
   Query Peak Memory: 4MB
Datanode:
   Max Query Peak Memory: 118MB
   Min Query Peak Memory: 24MB
  12 --Hash
         Max Buckets: 131072  Max Batches: 1  Max Memory Usage: 91857kB
         Min Buckets: 131072  Min Batches: 1  Min Memory Usage: 0kB
(8 rows)
```

优化分析

上述两个特征表明了此 SQL 语句存在极为严重的计算倾斜。进一步向 HashJoin 算子的下层分析发现 Seq Scan on s_riskrate_setting 也存在极为严重的计算倾斜 [38.885,2940.983]。根据 Scan 的含义推测此计划性能问题的根源在于表 s_riskrate_setting 数据的分布倾斜。实际分析之后确实发现表 s_riskrate_setting 存在严重的数据倾斜。整改之后性能从 94s 提升为 50s。

3.2.4.7 案例：改建分区表

现象描述

如下简单 SQL 语句查询， 性能瓶颈点在 dwcjk 的 Scan 上。

```
postgres=# explain performance select zqdh, count(1) from dwcjk where cjrq = '2015-05-02 00:00:00' group by zqdh;
 id |                    operation                    |       A-time        | A-rows  | E-rows  |    Peak Memory     | E-memory | A-width | E-width | E-costs
----+-------------------------------------------------+---------------------+---------+---------+--------------------+----------+---------+---------+-----------
  1 | ->  Row Adapter                                 | 1599.794            |      58 |      12 | 19KB               |          |         |       7 | 771106.83
  2 |    ->  Vector Streaming (type: GATHER)          | 1599.781            |      58 |      12 | 210KB              |          |         |       7 | 771106.83
  3 |       ->  Vector Hash Aggregate                 | [1445.092,1446.332] |      58 |       2 | [2315KB, 2315KB]   | 16MB     | [16,16] |       7 | 128517.80
  4 |          ->  Vector Streaming(type: REDISTRIBUTE) | [1444.996,1446.259] |     340 |      12 | [247KB, 247KB]     | 1MB      |         |       7 | 128518.09
  5 |             ->  Vector Hash Aggregate           | [573.150,1261.354]  |     340 |      12 | [2297KB, 2297KB]   | 16MB     | [16,16] |       7 | 128517.80
  6 |                ->  CStore Scan on public.dwcjk  | [330.178,1021.695]  | 10000000 | 1623137 | [786KB, 786KB]     | 1MB      |         |       7 | 120402.00
(6 rows)
```

优化分析

从业务层确认表数据(在 cjrq 字段上)有明显的日期特征，符合分区表的特征。重新规划 dwcjk 表的表定义：字段 cjrq 为分区键、天为间隔单位定义分区表 dwcjk_part。修改后结果如下，性能提升近 1 倍。

```
postgres=# explain performance select zqdh, count(1) from dwcjk_part where cjrq = '2015-05-02 00:00:00' group by zqdh;
 id |                          operation                          |       A-time        | A-rows  | E-rows  |    Peak Memory     | E-memory | A-width | E-width | E-costs
----+-------------------------------------------------------------+---------------------+---------+---------+--------------------+----------+---------+---------+-----------
  1 | ->  Row Adapter                                             | 977.457             |      58 |      14 | 19KB               |          |         |       7 | 773142.84
  2 |    ->  Vector Streaming (type: GATHER)                      | 977.437             |      58 |      14 | 210KB              |          |         |       7 | 773142.84
  3 |       ->  Vector Hash Aggregate                             | [651.238,734.931]   |      58 |       2 | [2316KB, 2316KB]   | 16MB     | [16,16] |       7 | 128857.14
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)           | [651.137,734.834]   |     340 |      14 | [247KB, 247KB]     | 1MB      |         |       7 | 128857.47
  5 |             ->  Vector Hash Aggregate                       | [402.145,515.752]   |     340 |      14 | [2297KB, 2297KB]   | 16MB     | [16,16] |       7 | 128857.14
  6 |                ->  Vector Partition Iterator                | [162.630,275.990]   | 10000000 | 1691000 | [312BYTE, 312BYTE] | 1MB      |         |       7 | 120402.00
  7 |                   ->  Partitioned CStore Scan on public.dwcjk_part | [161.746,275.207] | 10000000 | 1691000 | [795KB, 795KB]     | 1MB      |         |       7 | 120402.00
(7 rows)
```

3.3 权限管理

3.3.1 创建用户并授权使用云数据库 GaussDB

如果您需要对您所拥有的云数据库 GaussDB 进行精细的权限管理，您可以使用（Identity and Access Management，简称 IAM），通过 IAM，您可以：

- 根据企业的业务组织，在您的帐号中，给企业中不同职能部门的员工创建 IAM 用户，让员工拥有唯一安全凭证，并使用云数据库 GaussDB 资源。
- 根据企业用户的职能，设置不同的访问权限，以达到用户之间的权限隔离。
- 将云数据库 GaussDB 资源委托给更专业、高效的其他帐号或者云服务，这些帐号或者云服务可以根据权限进行代运维。

如果帐号已经能满足您的要求，不需要创建独立的 IAM 用户，您可以跳过本章节，不影响您使用云数据库 GaussDB 服务的其它功能。

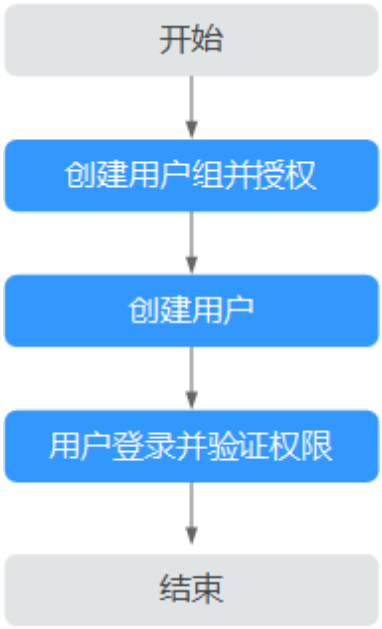
本章节为您介绍对用户授权的方法，操作流程如图 3-15 所示。

前提条件

给用户组授权之前，请您了解用户组可以添加的云数据库 GaussDB 系统策略，并结合实际需求进行选择。云数据库 GaussDB 支持的系统权限，请参见：表 1-4。

示例流程

图3-15 给用户授权云数据库 GaussDB 权限流程



1. 在 IAM 控制台创建用户组，并授予关系型数据库只读权限“云数据库 GaussDB ReadOnlyAccess”。
2. 在 IAM 控制台创建用户，并将其加入 1 中创建的用户组。
3. 并验证权限

   新创建的用户登录控制台，切换至授权区域，验证权限：

   - 在“服务列表”中选择云数据库 云数据库 GaussDB，进入云数据库 GaussDB 主界面，在左侧导航栏选择云数据库 GaussDB > 实例管理。单击右上角“购买数据库实例”，尝试购买数据库实例，如果无法购买（假设当前权限仅包含云数据库 GaussDB ReadOnlyAccess），表示“云数据库 GaussDB ReadOnlyAccess”已生效。
   - 在“服务列表”中选择除云数据库 云数据库 GaussDB 外（假设当前策略仅包含云数据库 GaussDB ReadOnlyAccess）的任一服务，若提示权限不足，表示“云数据库 GaussDB ReadOnlyAccess”已生效。

3.3.2 自定义策略

如果系统预置的云数据库 GaussDB 权限，不满足您的授权要求，可以创建自定义策略。自定义策略中可以添加的授权项（Action）请参考《云数据库 云数据库 GaussDB 2.22.10.100 API 参考》的“策略及授权项说明”章节。

本章为您介绍常用的云数据库 GaussDB 自定义策略样例。

自定义策略样例

- 示例 1：授权用户创建云数据库 GaussDB 实例

```
{
    "Version": "1.1",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["gaussdb:instance:create"]
        }]
}
```

- 示例 2：拒绝用户删除云数据库 GaussDB 实例

  拒绝策略需同时配合其他策略使用，否则没有实际作用。用户被授予的策略中，一个授权项的作用如果同时存在 Allow 和 Deny，则遵循 Deny 优先。

  如果您给用户授予云数据库 GaussDB FullAccess 的系统策略，但不希望用户拥有云数据库 GaussDB FullAccess 中定义的删除云数据库 GaussDB 实例，您可以创建一条拒绝删除云服务的自定义策略，然后同时将云数据库 GaussDB FullAccess 和拒绝策略授予用户，根据 Deny 优先原则，则用户可以对云数据库 GaussDB 实例执行除了删除实例外的所有操作。拒绝策略示例如下：

```
{
    "Version": "1.1",
    "Statement": [{
        "Action": ["gaussdb:instance:delete"],
        "Effect": "Deny"
    }]
}
```

3.4 实例管理

3.4.1 修改实例名称

操作场景

云数据库 GaussDB 支持修改实例名称，以方便用户识别。

约束

实例名称修改中，以下操作不可进行：

- 删除实例。
- 创建备份。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，单击目标实例名称后的，编辑实例名称，单击“确认”，即可修改实例名称。

您也可以单击目标实例名称，进入实例的“基本信息”页面，在“数据库信息”模块实例名称处，单击，修改实例名称。

实例名称长度在 4 个到 64 个字符之间，必须以字母开头，可包含大写字母、小写字母、数字、中划线或下划线，不能包含其他特殊字符。

- 单击，提交修改。
- 单击，取消修改。

步骤 3 在实例的“基本信息”页面，查看修改结果。

----结束

3.4.2 重启实例

操作场景

当修改的实例参数需要重启生效时，可以重启实例。

须知

- 如果数据库实例未处于“正常”状态，则无法重启该实例。您的数据库可能会由于几个原因而不可用，例如，正在进行以前请求的修改操作。
- 重启数据库实例将导致数据库业务短暂中断，在此期间，数据库实例状态将显示为“重启中”。
- 重启过程中，实例将不可用。重启后实例会自动释放内存中的缓存，请注意对业务进行预热，避免业务高峰期出现阻塞。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击“操作”列的“更多 > 重启实例”。

您也可以在“实例管理”页面，单击目标实例名称，进入实例基本信息页面，在页面右上角，单击“重启实例”。

步骤 3 在“重启实例”弹框中，单击“是”，重启实例。

数据库实例状态将显示为“重启中”，则说明重启指令下发成功。

步骤 4 稍后刷新实例列表，查看重启结果。如果实例状态为“正常”，说明实例重启成功。

----结束

3.4.3 删除实例

操作场景

- 用户需要删除不需要的数据库实例。
- 用户需要删除创建失败的数据库实例。

须知

- 实例删除后，不可恢复，请谨慎操作。如需保留数据，请务必确认完成数据备份后再删除实例。
- 执行操作中的实例不能手动删除，只有在实例操作完成后，才可删除实例。
- “按需计费”类型的实例删除后手动备份会继续保留。

删除按需实例

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面的实例列表中，选择需要删除的实例，在“操作”列，单击“更多 > 删除实例”。

步骤 3 在“删除实例”弹框，单击“是”下发请求，稍后刷新“实例管理”页面，查看删除结果。

----结束

3.4.4 重置管理员密码

操作场景

在使用云数据库 GaussDB 过程中，如果忘记数据库 root 帐号密码，可以重新设置密码。

注意事项

- 如果数据库实例处于“异常”状态，则无法重置管理员密码。
- 重置密码生效时间取决于该实例当前执行的业务数据量。
- 请定期修改用户密码，以提高系统安全性，防止出现密码被暴力破解等安全风险。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击“更多 > 重置密码”。

您也可以在“基本信息”页签，在“数据库信息”模块的“管理员帐户名”处，单击“重置密码”。

步骤 3 在“重置密码”弹框，输入新密码及确认密码。

须知

重置的密码需满足以下几个条件：

- 8 到 32 个字符。
- 至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（限定为~!@#%^*-_=+?,）四类字符中的三类字符。
- 不能与旧密码重复。

- 单击“确定”，提交重置。
- 单击“取消”，取消本次重置。

----结束

3.4.5 绑定和解绑弹性公网 IP

操作场景

云数据库 GaussDB 实例创建成功后，支持用户绑定弹性公网 IP，在公共网络来访问数据库实例，绑定后也可根据需要解绑。

须知

为保证数据库可正常访问，请确保数据库使用的安全组开通了相关端口的访问权限，假设数据库的访问端口是 1611，那么需确保安全组开通了 1611 端口的访问。

注意事项

对于已绑定弹性公网 IP 的实例，需解绑后，才可重新绑定其他弹性公网 IP。

绑定弹性公网 IP

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例基本信息页面。

步骤 3 在“连接信息”模块处，单击内网地址后的“绑定”。

步骤 4 在弹出框的 EIP 地址列表中，显示“未绑定”状态的 EIP，选择所需绑定的 EIP，单击“是”，提交绑定任务。如果没有可用的 EIP，单击“查看弹性公网 IP”，获取 EIP。

步骤 5 在“连接信息”模块的内网地址处，查看结果。

如需关闭，请参见解绑弹性公网 IP。

说明

绑定成功后，您还可以单击内网 IP 前的 查看弹性公网 IP 详细信息。

----结束

解绑弹性公网 IP

步骤 1 3.1 登录管理控制台。

步骤 2 对于已绑定 EIP 的实例，在“实例管理”页面，选择指定实例，单击实例名称，进入实例基本信息页面。

步骤 3 在“连接信息”模块，单击 IP 地址后面的“解绑”，在弹出框中单击“是”，解绑 EIP。

步骤 4 在“连接信息”模块的内网地址处，查看结果。

如需重新绑定，请参见绑定弹性公网 IP。

----结束

3.4.6 扩容实例

操作场景

随着实例部署时间及业务的增长，数据库在运行性能及存储上逐渐会达到瓶颈。此时，需要通过增加主机来提升实例的性能及存储能力。云数据库 GaussDB 支持扩容节点操作。

须知

云数据库 GaussDB 支持灵活选择扩容 CN 或分片，但是要确保扩容后，实例中 CN 节点的数量必须小于或等于两倍的分片数量。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例的名称。

步骤 3 在“基本信息”页面，进行扩容操作。

分片数量扩容

1. 单击“分片数量”后的“添加”，
2. 选择新增的分片数量。单击“下一步”。

3. 确认无误后，单击“提交”进行分片数量扩容。

说明

一个分片中默认包含三个 DN 副本，因此每增加一个分片会新增三个 DN 副本。

协调节点数量扩容

1. 单击“协调节点数量”后的“添加”。
2. 选择新增的协调节点数量以及可用区。
   若创建实例时指定的可用区为 1 个，协调节点横向扩容需选择同一可用区。
3. 单击“下一步”，进入“规格确认”界面。
4. 确认无误后，单击“提交”进行协调节点扩容。

----结束

3.4.7 协调节点缩容

操作场景

随着业务下降，数据库协调节点利用率低，资源浪费严重。为提高资源利用率，需要对协调节点进行缩容。云数据库 GaussDB 支持协调节点缩容操作。

注意事项

- 协调节点缩容操作不会中断业务。
- 仅支持独立部署实例。
- 协调节点至少需要保留一个。
- 缩容前请确认该节点不在 JDBC 连接配置中，避免影响 JDBC 连接高可用性能。
- 缩容过程中执行的 DDL 操作将被回滚。
- 缩容过程中 PITR 备份暂停，缩容结束后自动恢复。
- 缩容完成后将自动进行一次全量备份。
- 缩容前须保证实例状态正常，所有协调节点状态正常，否则无法进行缩容。
- 当选择首节点缩容时，将随机替换为其他协调节点进行缩容。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例的名称。

步骤 3 在“基本信息”页面，进行缩容操作。

1. 单击“协调节点数量”后的“删除”。
2. 选择需要删除的协调节点。
3. 单击“下一步”，进入“删除协调节点”确认界面。
4. 确认要删除的协调节点正确无误后，单击“提交”进行协调节点缩容

----结束

3.4.8 磁盘扩容

操作场景

云数据库 GaussDB 实例使用一段时间后业务攀升，原申请磁盘空间大小不足以支撑储存完整业务量。内核监测到磁盘使用量超过 85%会将实例设置为只读，无法再写入数据，实例进入盘满只读状态。您可以通过磁盘扩容功能扩容数据库实例的磁盘。

注意事项

- 在最大值的允许范围内，扩容磁盘后磁盘使用率不能仍然高于 85%。
- 节点状态需要为正常，否则应先联系运维人员修复节点。
- 扩容磁盘的大小必须是（40GB*分片数量）的整数倍。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择目标实例，单击“操作”列的“磁盘扩容”，进入“磁盘扩容”页面。

您也可以通过单击目标实例名称，进入“基本信息”页面，在“存储/备份空间”模块的“存储空间”处，单击“磁盘扩容”，进入“磁盘扩容”页面。

步骤 3 在“磁盘扩容”页面，选择空间大小，单击“下一步”。

选择空间大小时，需要考虑扩容后的磁盘使用率应该小于 85%。只有低于 85%时，才可以让实例从只读状态恢复为可读写状态。

步骤 4 规格确认。

- 如果需要重新选择，单击“上一步”，回到上个页面，修改新增大小。
- 如果确认无误，单击“提交”，提交扩容。

步骤 5 查看扩容结果。

在实例管理页面，可看到实例状态为“扩容中”，稍后单击实例名称，在“基本信息”页面，查看磁盘大小，检查扩容是否成功。此过程需要 3～5 分钟。

----结束

3.4.9 查看和修改实例参数

您可以实时修改云数据库 GaussDB 数据库实例参数，也可以通过该功能查看当前实例所使用的参数值。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例基本信息页面。

步骤 3 在左侧导航栏单击“参数修改”，进入参数修改页面。

- 您可以在该页面对实例应用的参数进行修改和查询。修改参数后，可以预览修改结果或取消修改，确认修改无误后，保存修改内容。

  说明

  根据参数列表中“是否需要重启”提示，进行相应操作：

  - 是：在实例列表中，查看“运行状态”，如果显示参数变更，等待重启，则需重启实例使之生效。
  - 否：无需重启，立即生效。
- 您可以单击“复制”，将当前实例应用的参数另存为参数模板，可在 3.5 参数模板管理的自定义页签中查看。
- 您还可以单击“导出”，将当前实例的参数下载到本地。
- 您可以单击“比较参数”，将已有的参数模板与当前实例应用的参数模板进行比较。

----结束

3.4.10 规格变更

操作场景

随着业务的不断增加，实例的 CPU 和内存资源可成会为实例性能的瓶颈，无法满足业务要求时，云数据库 GaussDB 提供了规格变更功能来提升实例的 CPU 和内存。

注意事项

- 不支持将规格参数变小。
- 规格变更前，须确保实例状态正常。实例异常，节点异常，磁盘满均不允许进行规格变更。
- 修改 CPU|内存后，将会重启数据库实例。请选择业务低峰期，避免业务异常中断。重启后实例会自动释放内存中的缓存，请在业务低峰期进行重启，避免对高峰期业务造成影响。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择目标实例，单击“操作”列的“更多 > 规格变更”，进入“规格变更”页面。

您也可以通过单击目标实例名称，进入“基本信息”页面，在“数据库信息”模块的“性能规格”处，单击“规格变更”，进入“规格变更”页面。

步骤 3 在“规格变更”页面，选择所需修改的性能规格，单击“下一步”。

步骤 4 进行规格确认，单击“提交”。

步骤 5 查看变更结果。

任务提交成功后，单击“返回实例列表”，在实例管理页面，可以看到实例状态为“规格变更中”。稍后在对应的“基本信息”页面，查看实例规格，检查修改是否成功。

----结束

3.4.11 导出实例列表

操作场景

您可以导出实例列表，查看并分析实例信息。

导出所有实例

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，单击实例列表右上角 ，默认选择所有的数据库实例。

步骤 3 在导出弹框勾选所需导出信息，单击“导出”。

步骤 4 导出任务执行完成后，您可在本地查看到一个“.csv”文件。

----结束

3.4.12 设置安全组规则

操作场景

安全组是一个逻辑上的分组，为同一个虚拟私有云内具有相同安全保护需求，并相互信任的弹性云主机和云数据库 GaussDB 实例提供访问策略。

创建分布式版实例时，如果需要修改内网安全组，请确保入方向规则 TCP 协议端口包含：40000-60480,20050,5000-5001,2379-2380,6000,6500，<database port> - (<database port> + 100)。（例如设置的数据库端口为 8000，则安全组中需要包含 8000-8100）。

创建主备版实例时，如果需要修改内网安全组，请确保入方向规则 TCP 协议端口包含：20050，5000-5001，2379-2380，6000，6500，<database port> - (<database port> + 100)。（例如设置的数据库端口为 8000，则安全组中需要包含 8000-8100）。

为了保障数据库的安全性和稳定性，在使用云数据库 GaussDB 实例之前，您需要设置安全组，开通需访问数据库的 IP 地址和端口。

内网连接云数据库 GaussDB 实例时需要为云数据库 GaussDB 和 ECS 分别设置安全组规则。

- 设置云数据库 GaussDB 安全组规则：为云数据库 GaussDB 所在安全组配置相应的**入方向规则**。
- 设置 ECS 安全组规则：安全组默认规则为出方向上数据报文全部放行，此时，无需对 ECS 配置安全组规则。当在 ECS 所在安全组为非默认安全组且出方向规则**非全放通**时，需要为 ECS 所在安全组配置相应的**出方向**规则。

注意事项

因为安全组的默认规则是在出方向上的数据报文全部放行，同一个安全组内的弹性云主机和实例可互相访问。安全组创建后，您可以在安全组中定义各种访问规则，当云数据库 GaussDB 实例加入该安全组后，即受到这些访问规则的保护。

- 默认情况下，一个租户可以创建 500 条安全组规则。
- 建议一个安全组内的安全组规则不超过 50 条。
- 当需要从安全组外访问安全组内的云数据库 GaussDB 实例时，需要为安全组添加相应的**入方向规则**。

说明

源地址默认的 IP 地址 0.0.0.0/0 是指允许所有 IP 地址访问安全组内的云数据库 GaussDB 实例。

3.4.13 变更副本数量

操作场景

云数据库 GaussDB 提供了将分片中的三副本降低为两副本功能，从原先的 1 主 2 备，降低为 1 主 1 备。其他类型的节点，例如 CN，CMS/ETCD，GTM 的副本数保持不变。

注意事项

- 降副本操作仅支持部署在单 AZ 上的实例。
- 实例状态异常，或者存在节点状态异常时，不允许执行降副本操作。
- 降副本操作仅支持按需实例，包周期实例由于涉及到订单退费，不支持降副本操作。
- 副本集数量变更过程中，实例将出现多次短暂中断，请在业务低峰期进行降副本操作，避免业务异常中断。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择目标实例，单击实例名称，进入实例的“基本信息”页面。

步骤 3 在实例基本信息页面的“副本数量”处，单击“变更”。进入副本选择页面。

步骤 4 在副本选择页面，选择变更后的副本数量，单击“下一步”，下发变更操作。

任务提交成功后，在实例管理页面，可以看到实例状态为“变更副本集数量中”。

稍后在对应的“基本信息”页面，查看实例 DN 副本数，检查变更是否成功。

----结束

3.5 参数模板管理

3.5.1 创建参数模板

您可以使用数据库参数模板中的参数来管理数据库引擎配置，数据库参数模板可应用于一个或多个数据库实例。

如果您在创建数据库实例时未指定客户创建的数据库参数模板，系统将会为您的数据库实例适配默认的数据库参数模板。该默认组包含数据库引擎默认值和系统默认值，具体根据引擎、计算规格及实例的分配存储空间而定。您无法修改默认数据库参数模板的参数设置，您必须创建自己的数据库参数模板才能更改参数设置的默认值。

须知

并非所有数据库引擎参数都可在客户创建的数据库参数模板中进行更改。

如果您想使用您自己的数据库参数模板，只需创建一个新的数据库参数模板，创建实例的时候选择该参数模板，如果是在创建实例后有这个需求，可以重新应用该参数模板，请参见 3.5.8 应用参数模板。

若您已成功创建数据库参数模板，并且想在新的数据库参数模板中包含该组中的大部分自定义参数和值时，复制参数模板是一个方便的解决方案，请参见 3.5.6 复制参数模板。

以下是您在使用数据库参数模板中的参数时应了解的几个要点：

- 某些参数需要重启才能生效，这些参数更改将在您手动重启数据库实例后生效。
- 在数据库参数模板内设置参数不恰当可能会产生意外的不利影响，包括性能降低和系统不稳定。修改数据库参数时应始终保持谨慎，禁止对参数组进行边界测试，否则会导致实例异常，且修改数据库参数模板前要备份数据。将参数模板更改应用于生产数据库实例前，您应当在测试数据库实例上试用这些参数模板设置更改。

说明

每个项目最多可以创建 100 个云数据库 GaussDB 数据库参数模板，各云数据库 GaussDB 引擎共享该配额。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 单击左侧导航栏的“参数模板管理”。

步骤 3 在“参数模板管理”页面，单击“创建参数模板”。

步骤 4 选择数据库引擎版本，命名并添加对该参数模板的描述，单击“确定”，创建参数模板。

- 选择该数据库引擎参数模板所需应用的参数模板类型。

- 参数模板名称长度在 1~64 个字符之间，区分大小写，可包含字母、数字、中划线、下划线或句点，不能包含其他特殊字符。
- 参数模板的描述长度不能超过 256 个字符，且不能包含回车和! < " = ' > &特殊字符。

----结束

3.5.2 编辑参数模板

为确保云数据库 GaussDB 发挥出最优性能，用户可根据业务需求对用户创建的参数模板里边的参数进行调整。

您可以修改用户创建的数据库参数模板中的参数值，但不能更改默认数据库参数模板中的参数值。

如果您更改一个参数值，则所做更改的应用时间将由该参数的类型决定，可以根据界面提示确认参数是否重启生效。

说明

系统提供的默认参数模板不允许修改，只可单击参数模板名进行查看。当用户参数设置不合理导致数据库无法启动时，可参考默认参数模板重新配置。

批量修改参数

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面的“自定义”页签，选择目标参数模板，单击参数模板名称。

步骤 3 根据需要修改相关参数值。

- 单击“保存”，在弹出框中单击“是”，保存修改。
- 单击“取消”，放弃本次设置。
- 单击“预览”，可对比参数修改前和修改后的值。

步骤 4 参数修改完成后，您可在“参数模板管理”页面单击目标参数模板名称，然后在左侧导航栏中，单击“参数修改历史”查看参数的修改详情。

须知

参数模板修改后，不会立即应用到当前使用的实例，您需要进行应用操作才可生效，具体操作请参见 3.5.8 应用参数模板。

----结束

3.5.3 导出参数

操作场景

您可以将该实例对应的参数模板信息（参数名称，值，描述）导出到 CSV 中，方便查看并分析。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例的基本信息页面。

步骤 3 在左侧导航栏中选择“参数修改”，在“参数”页签单击“导出”。

导出到文件。将该实例对应的参数模板信息（参数名称，值，描述）导出到 CSV 表中，方便用户查看并分析。

步骤 4 在弹出框中，填写文件名称，单击“确定”。

说明

文件名称在 4 位到 81 位之间，必须以字母开头，可以包含字母、数字、中划线或下划线，不能包含其他特殊字符。

----结束

3.5.4 比较参数模板

操作场景

- 您可以将当前实例应用的参数与参数模板进行比较，以了解当前实例参数的差异项。
- 您可以比较云数据库 GaussDB 默认参数模板，以了解默认参数模板与其他参数模板的配置差异。
- 你还可以比较自定义的参数模板，以区分不同参数模板之间的差别。

比较当前实例参数模板

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，单击实例名称，进入实例的“基本信息”页签。

步骤 3 在左侧导航栏中选择“参数修改”。

步骤 4 在“参数”子页签中单击“比较参数”，比较当前实例参数。

步骤 5 在弹出框中选择与当前实例同数据库类型的参数模板，单击“确定”，比较两个参数的差异项。

- 有差异项，则会显示差异参数的如下信息：参数名称、当前实例参数模板的参数值和被比较参数模板的参数值。

- 无差异项，则不显示。

----结束

比较目标参数模板

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面的“系统默认”或者“自定义”页签，选择一个参数模板，单击“比较”。

步骤 3 选择同一数据库引擎的不同参数模板，单击“确定”，比较两个参数模板之间的配置参数差异项。

- 有差异项，则会显示差异参数模板的如下信息：参数名称、当前参数模板的参数值和被比较参数模板的参数值。
- 无差异项，则不显示。

----结束

3.5.5 查看参数修改历史

操作场景

您可以查看当前实例所使用参数模板的修改历史，以满足业务需要。

您也可以查看自定义参数模板的修改历史，以满足业务需要。

说明

用户创建或复制的新参数模板，在未进行参数修改前，无修改历史。

当前仅显示 7 天之内的参数修改历史。

查看当前实例的参数修改历史

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例的基本信息页面。

步骤 3 在左侧导航栏，单击“参数修改”。

步骤 4 在弹出的页签中，单击“参数修改历史”。

您可查看参数对应的参数名称、修改前参数值、修改后参数值、修改状态、修改时间、是否应用以及应用时间。

如修改后参数模板未应用，请根据业务需要，参考 3.5.8 应用参数模板，将其应用到对应实例。

----结束

查看目标参数模板的参数修改历史

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面的“自定义”页签，单击目标参数模板名称。

步骤 3 单击“参数修改历史”。

您可查看参数对应的参数名称、修改前参数值、修改后参数值、修改状态和修改时间。

----结束

3.5.6 复制参数模板

操作场景

您可以复制您创建的自定义数据库参数模板。当您已创建一个数据库参数模板，并且想在新的数据库参数模板中包含该组中的大部分自定义参数和值时，复制参数模板是一个方便的解决方案。

复制数据库参数模板之后，新参数模板可能不会立即显示，建议您等待至少 5 分钟再使用。

您无法复制默认参数模板。不过，您可以创建基于默认参数模板的新参数模板。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面的“自定义”页签，选择需要复制的参数模板，单击“复制”。

步骤 3 在弹出框中，填写新参数模板名称和描述，单击“确定”。

- 参数模板名称长度在 1~64 个字符之间，区分大小写，可包含字母、数字、中划线、下划线或句点，不能包含其他特殊字符。
- 参数模板的描述长度不能超过 256 字符，且不能包含回车和>!<"&'=特殊字符。

创建完成后，会生成一个新的参数模板，您可在参数模板列表中对其进行管理。

----结束

3.5.7 重置参数模板

操作场景

您可根据自己的业务需求，重置自己创建的参数模板对应的所有参数，使其恢复到默认值。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面的“自定义”页签，选择需要设置的参数模板，单击“更多 > 重置”。

步骤 3 单击“是”，重置所有参数为其默认值。

----结束

3.5.8 应用参数模板

操作场景

参数模板编辑修改后，不会立即应用到实例，您可以根据业务需要应用到实例中，参数模板只能应用于相同版本的实例中。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面，根据参数模板类型不同进行如下操作。

- 若需要将默认参数模板应用到实例，在“系统默认”页签的目标参数模板操作列，单击“应用”。
- 若需要将用户自己创建的参数模板应用到实例，在“自定义”页签的目标参数模板操作列，单击“更多 > 应用”。

一个参数模板可被应用到一个或多个实例。

步骤 3 在弹出框中，选择或输入所需应用的实例，单击“确定”。

参数模板应用成功后，您可参考 3.5.9 查看参数模板应用记录，查看应用状态是否为“成功”。如果应用状态为“应用中”，则再次应用该模板时，将无法选择正在应用参数模板的实例。如果需要再次应用参数模板到同一实例，请确保应用状态为“成功”。

说明

参数模板应用成功后，如果重启生效的参数有修改，实例将会变为“等待重启”状态，此时需要重启实例参数修改才能生效。如果所有重启生效的参数都没有修改，则实例状态不会发生改变。

----结束

3.5.9 查看参数模板应用记录

操作场景

参数模板编辑修改后，您可根据业务需要将其应用到对应实例中，云数据库 GaussDB 支持查看参数模板所应用到实例的记录。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面，“系统默认”页签或“自定义”页签，选择目标参数模板，单击操作列的“应用记录”，查看应用记录。

步骤 3 您可查看参数模板所应用到的实例名称/ID、应用状态、应用时间、失败原因。

----结束

3.5.10 修改参数模板描述

操作场景

参数模板创建成功后，用户可根据需要对自己创建的参数模板描述进行修改。

说明

默认参数模板的描述不可修改。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面的“自定义”页签，选择一个用户创建的参数模板，单击“描述”列 。

步骤 3 输入新的描述信息，单击 ，提交修改，单击 ，取消修改。

- 修改成功后，新的新描述信息，可在参数模板列表的“描述”列查看。
- 参数模板的描述长度不能超过 256 字符，且不能包含>!<"&'=特殊字符。

----结束

3.5.11 删除参数模板

操作场景

每个用户最多可以创建 100 个数据库参数模板，如果参数模板已满或者一些参数模板已经没有使用价值，您可以参考本章内容删除已有的参数模板。

须知

- 参数模板删除后，不可恢复，请谨慎操作。
- 默认参数模板不可被删除。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“参数模板管理”页面的“自定义”页签，选择需要删除的参数模板，单击“更多 > 删除”。

步骤 3 单击“是”，删除。

----结束

3.6 数据备份

3.6.1 备份概述

云数据库 GaussDB 支持数据库实例的备份和恢复，以保证数据可靠性。备份目前将以未加密的方式存储。

备份的作用

当数据库或表被恶意或误删除，虽然云数据库 GaussDB 支持高可用，但备机数据库会被同步删除且无法还原。因此，数据被删除后只能依赖于实例的备份保障数据安全。

全量备份

全量备份（Full Backup）表示对所有目标数据进行备份，包含备份时刻点上数据库的全量数据，耗时时间长（和数据库数据总量成正比），自身即可恢复出完整的数据库。全量备份总是备份所有选择的目标，即使从上次备份后数据没有变化。

差量备份

差量备份（Differential Backup）只包含从指定时刻点之后的增量修改数据，耗时时间短（和增量数据成正比，和数据总量无关），但是必须要和全量备份数据一起才能恢复出完整的数据库。云数据库 GaussDB 默认自动每 30 分钟对上一次自动备份后更新的数据进行备份，支持修改备份周期为最小 15 分钟，最大 1440 分钟。

自动备份

云数据库 GaussDB 会在数据库实例的备份时段中创建数据库实例的自动备份。系统根据您指定的备份保留期保存数据库实例的自动备份。如果需要，您可以将保存的备份恢复。

手动备份

用户还可以创建手动备份对数据库进行备份，手动备份是由用户启动的数据库实例的全量备份，会一直保存，直到用户手动删除。

3.6.2 设置自动备份策略

操作场景

创建云数据库 GaussDB 实例时，系统默认开启自动备份策略。实例创建成功后，您可根据业务需要设置自动备份策略。云数据库 GaussDB 按照用户设置的自动备份策略对数据库进行备份。

云数据库 GaussDB 默认开启的自动备份策略设置如下：

- 备份时间段：默认为 24 小时中，间隔一小时的随机的一个时间段 ，例如 01:00～02:00，12:00～13:00 等。备份时间段以 UTC 时区保存。如果碰到夏令时/冬令时切换，备份时间段会因时区变化而改变。
- 备份周期：默认为一周内的每一天。
- 差量备份策略：默认每 30 分钟保存一次。

说明

为了满足时间点恢复的需求，超出备份保留天数最近的一次全量备份不会被立即删除。示例：设置自动备份策略为每天备份 1 次，保留天数为 1 天，即 11.1 号生成备份 1，11.2 号生成备份 2 并保留备份 1；11.3 号生成备份 3，并保留备份 2 及删除备份 1。

修改自动备份策略

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称。

步骤 3 在左侧导航栏，选择“备份恢复”，然后单击“修改备份策略”。

步骤 4 按照界面提示修改备份策略。

- 全量备份策略

  保留天数是指自动备份可保留的时间，增加保留天数可提升数据可靠性，请根据需要设置。

  备份周期请根据需要进行选择。并且最少需要选择一天。

  说明

  减少保留天数的情况下，该备份策略对已有备份文件同时生效，即超出备份保留天数的已有备份文件会被删除。

  保留天数范围为 1～732 天。

  备份时间段为间隔 1 小时，建议根据业务情况，选择业务低峰时段，备份周期默认全选，可修改，且至少选择一周中的 1 天。

  实例创建完成后，会立即触发一次全量备份，之后会按照策略中的备份时间段和备份周期进行全量备份和差量备份策略。备份时间段请选择为业务峰值较低的时间段。全量备份会在此时间段进行。

- 差量备份策略

需要选择差量备份的周期，即每隔多长时间进行一次差量备份。默认 30 分钟一次。

步骤 5 单击“确定”，确认修改。

----结束

3.6.3 创建手动备份

操作场景

云数据库 GaussDB 支持对运行正常的实例创建手动备份。

注意事项

- 备份操作需要在实例状态为正常时才可以进行。
- 同一用户在一个实例上，同一时间只能进行一次备份操作。

方式一

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，在操作列选择“更多 > 创建备份”。

步骤 3 在创建备份弹出框中，命名该备份，并添加描述，单击“确定”，提交备份创建，单击“取消”，取消创建。

- 备份名称的长度在 4~64 个字符之间，必须以字母开头，区分大小写，可以包含字母、数字、中划线或者下划线，不能包含其他特殊字符。
- 备份描述不能超过 256 字符，且不能包含回车和>!<"&'=特殊字符。
- 手动备份创建过程中，状态显示为“备份中”，此过程所需时间由数据量大小决定。

步骤 4 手动备份创建成功后，用户可在“备份恢复管理”页面，对其进行查看并管理。

也可在“实例管理”页面，单击实例名称，在左侧导航栏，单击“备份恢复”，对其进行查看并管理。

----结束

方式二

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称。

步骤 3 在左侧导航栏中选择“备份恢复”，单击“创建备份”。

步骤 4 命名该备份，并添加描述，单击“确定”，提交备份创建，单击“取消”，取消创建。

- 备份名称的长度在 4~64 个字符之间，必须以字母开头，区分大小写，可以包含字母、数字、中划线或者下划线，不能包含其他特殊字符。
- 备份描述不能超过 256 字符，且不能包含回车和>!<"&'=特殊字符。
- 手动备份创建过程中，状态显示为“备份中”，此过程所需时间由数据量大小决定。

步骤 5 手动备份创建成功后，用户可在“实例管理”页面，单击实例名称，在左侧导航栏中选择“备份恢复”，对其进行查看并管理。

也可在“备份恢复管理”页面，对其进行查看并管理。

----结束

3.6.4 导出备份信息

操作场景

云数据库 GaussDB 支持导出备份，用户可以通过导出备份功能将备份信息（ID，备份名称，实例名称，实例 ID，引擎，备份类型，备份开始时间，备份结束时间，备份状态，备份大小，备份描述）导出到 csv 并下载，方便用户查看并分析备份信息。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在左侧导航栏，单击“备份恢复管理”，在“备份恢复管理”页面，勾选需要导出的备份，单击  ，导出备份信息。

您也可以在“实例管理”页面，单击实例名称，进入“基本信息”页面，在左侧导航栏，单击“备份恢复”，勾选需要导出的备份，单击“导出”，导出备份信息。

导出的备份信息列表为 csv 文件，您可以对其进行分析，以满足业务需求。

步骤 3 查看导出的数据库备份。

----结束

3.6.5 删除手动备份

操作场景

云数据库 GaussDB 支持对手动备份进行删除，从而释放相关存储空间。

须知

- 手动备份删除后，不可恢复。
- 自动备份的文件不可手动删除。
- 恢复中的备份不允许删除。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在左侧导航栏，单击“备份恢复管理”，在“备份恢复管理”页面，选择目标备份，单击操作列中的“删除”。

您也可以在“实例管理”页面，单击实例名称，进入“基本信息”页面，在左侧导航栏，单击“备份恢复”，勾选需要删除的备份，单击操作列的“删除”。

步骤 3 单击“是”，删除手动备份。

备份删除后，将不会在备份恢复管理界面展示。

----结束

3.7 数据恢复

3.7.1 通过备份文件恢复实例

操作场景

云数据库 GaussDB 支持使用已有的自动备份和手动备份，将实例数据恢复到备份被创建时的状态。该操作恢复的为整个实例的数据。

限制条件

- 恢复时目标实例异常、实例磁盘满将会导致恢复失败。
- 不支持跨大版本恢复。例如：1.4.x 的实例仅可以恢复到 1.4.y 版本的实例。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在左侧导航栏单击“备份恢复管理”，选择需要恢复的备份，单击操作列的“恢复”。

您也可在“实例管理”页面，单击指定的实例名称，在左侧导航栏单击“备份恢复”，在“全量备份”页签下单击目标备份对应的操作列中的“恢复”。

步骤 3 单击“确定”，恢复实例。

说明

- 如果打开并行恢复功能，那么恢复过程中，所有主、备副本会同时从 OBS 服务器下载备份数据，与默认的串行恢复相比，OBS 带宽消耗量增加到 N 倍（N 等于每个分片的副本个数）。因此，为了防止 OBS 带宽达到上限导致恢复速度反而下降的情况，当待恢复集群的分片个数大于 5 个时，建议先咨询运维当前 OBS 服务器空闲带宽，然后再决定是否开启并行恢复功能。
- 主备版实例只支持并行恢复。
- 数据库内核版本小于 1.4 时，不支持开启并行恢复。
- 全量备份和增量备份除了备份数据文件之外，也会备份这个过程中的增量日志文件，用于保证该备份集恢复以后数据的一致性。由于增量日志文件的备份和上传需要一定时间（受网络、OBS 存储介质流控等影响），因此，需要注意的是，备份结束时间并不代表该备份集恢复后的数据一致性时间点（该恢复一致性点一般在备份结束时刻之前的几分钟以内）。如果用户对于恢复后数据的一致性时刻点有严格要求，请使用指定时间点恢复。

- 恢复到新实例：
  - 数据库大版本与原实例相同。例如：1.4.x 的实例仅可以恢复到 1.4.y 版本的实例。
  - 存储空间大小默认和备份时实例磁盘空间相同，且必须大于或等于备份时实例存储空间大小。
  - 数据库密码需重新设置。
  - 新实例的规格默认和原实例相同，如果需要修改规格，新实例的规格必须大于或等于原实例的规格。
  - 新实例的节点配置需要与备份时保持一致。

步骤 4 查看恢复结果。

- 恢复到新实例

  为用户重新创建一个和该备份数据相同的实例。可看到实例由“创建中”变为“正常”，说明恢复成功。

  恢复成功的新实例是一个独立的实例，与原有实例没有关联。

----结束

3.7.2 恢复实例到指定时间点

操作场景

云数据库 GaussDB 支持使用已有的自动备份，恢复实例数据到指定时间点。

注意事项

- 恢复到任意时间点仅支持 2.1 版本以上实例。
- 节点扩容，版本升级，恢复自身期间，对应时间点无法恢复。
- 实例故障，发生 CN 剔除等场景无法产生归档日志，对应时间点无法恢复。
- 如果您要将数据库备份恢复到新实例：
  - 数据库引擎、数据库大版本，与原实例相同，不可修改。
  - 数据库密码需重新设置。

- 恢复到当前实例会将当前实例上的数据全部覆盖，并且恢复过程中数据库不可用，且立即停止归档。恢复完成后会出现数据确认按钮，在单击数据确认前，可多次进行恢复。数据确认后会删除本次恢复时间点后的归档日志，并重新开启日志归档。
- 删除实例会默认删除所有归档日志，不支持选择保留。重建后不支持恢复任意时间点。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称。

步骤 3 在左侧导航栏中选择“备份恢复”页签，单击“恢复到指定时间点”。

步骤 4 单击“确定”，恢复实例。

说明

- 如果打开并行恢复功能，那么恢复过程中，所有主、备副本会同时从 OBS 服务器下载备份数据，与默认的串行恢复相比，OBS 带宽消耗量增加到 N 倍（N 等于每个分片的副本个数）。因此，为了防止 OBS 带宽达到上限导致恢复速度反而下降的情况，当待恢复集群的分片个数大于 5 个时，建议先咨询运维当前 OBS 服务器空闲带宽，然后再决定是否开启并行恢复功能。
- 数据库内核版本小于 1.4 时，不支持开启并行恢复。

- 恢复到新实例：
  - 数据库大版本与原实例相同。例如：1.4.x 的实例仅可以恢复到 1.4.y 版本的实例。
  - 存储空间大小默认和备份时实例磁盘空间相同，且必须大于或等于备份时实例存储空间大小。
  - 数据库密码需重新设置。
  - 新实例的规格默认和原实例相同，如果需要修改规格，新实例的规格必须大于或等于原实例的规格。
  - 新实例的节点配置需要与备份时保持一致。

步骤 5 查看恢复结果。

- 恢复到新实例

  为用户重新创建一个和该备份数据相同的实例。可看到实例由“创建中”变为“正常”，说明恢复成功。

  恢复成功的新实例是一个独立的实例，与原有实例没有关联。

----结束

3.8 监控与告警

3.8.1 监控指标

功能说明

本节定义了云数据库 GaussDB 上报云监控的监控指标的命名空间，监控指标列表和维度定义。

命名空间

SYS.GAUSSDBV5

支持的监控指标

云数据库 GaussDB 数据库性能监控指标，如下表所示。

表3-3 云数据库 GaussDB 支持的监控指标

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期(原始指标)
rds001_cpu_util	CPU 使用率	该指标用于统计测量对象的 CPU 使用率。	（0，60）	节点	60 秒
rds002_mem_util	内存使用率	该指标用于统计测量对象的内存使用率。	（0，60）	节点	60 秒
rds003_bytes_in	数据写入量	该指标用于统计测量对象对应 VM 的网络发送字节数，取时间段的平均值	暂无	节点	60 秒
rds004_bytes_out	数据传出量	该指标用于统计测量对象对应 VM 的网络接受字节数，取时间段的平均值	暂无	节点	60 秒
rds014_iops	数据磁盘每秒读写次数	该指标用于统计测量对象的节点数据磁盘每秒读写次数，该值为实时值。	暂无	节点	60 秒
rds016_disk_write_throughput	数据磁盘写吞吐量	该指标用于统计测量对象的节点数据磁盘每秒写吞吐量，该值为实时值。	暂无	节点	60 秒
rds017_disk_read_t	数据磁盘读吞吐量	该指标用于统计测量对象的节点数据磁盘每秒	暂无	节点	60 秒

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期(原始指标)
hroughput		读吞吐量，该值为实时值。			
rds020_avg_disk_ms_per_write	数据磁盘单次写入花费的时间	该指标用于统计测量对象的节点数据磁盘单次写入花费的时间，取时间段的平均值。	暂无	节点	60 秒
rds021_avg_disk_ms_per_read	数据磁盘单次读取花费的时间	该指标用于统计测量对象的节点数据磁盘单次读取花费的时间，取时间段的平均值。	暂无	节点	60 秒
io_bandwidth_usage	磁盘 io 带宽占用率	当前磁盘 io 带宽与磁盘最大带宽比值	NA	节点	60 秒
iops_usage	IOPS 使用率	当前 iops 与磁盘最大 iops 比值	NA	节点	60 秒
rds005_instance_disk_used_size	实例数据磁盘已使用大小	该指标用于统计测量对象的实例数据磁盘已使用大小，该值为实时值。	暂无	实例	60 秒
rds006_instance_disk_total_size	实例数据磁盘总大小	该指标用于统计测量对象的实例数据磁盘总大小，该值为实时值。	暂无	实例	60 秒
rds007_instance_disk_usage	实例数据磁盘已使用百分比	该指标用于统计测量对象的实例数据磁盘使用率，该值为实时值。	(0, 60%)	实例	60 秒
rds035_buffer_hit_ratio	buffer 命中率	该指标用于统计数据库 buffer 命中率。	（95，100）	实例	60 秒
rds036_deadlocks	死锁次数	该指标用于统计数据库发生事务死锁的次数，取该时间段的增量值。	(0，1)	实例	60 秒
rds048_P80	80% SQL 的响应时间	该指标用于统计数据库 80% SQL 的响应时间，该值为实时值。	（0，150000）	实例	60 秒
rds049_P95	95% SQL 的响应时间	该指标用于统计数据库 95% SQL 的响应时间，该值为实时值。	（0，200000）	实例	60 秒

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期(原始指标)
rds008_disk_used_size	磁盘已使用大小	该指标用于统计测量对象的节点数据磁盘使用值，该值为实时值。	NA	组件	60 秒
rds009_disk_total_size	磁盘总大小	该指标用于统计测量对象的节点数据磁盘总大小，该值为实时值。	NA	组件	60 秒
rds010_disk_usage	磁盘已使用百分比	该指标用于统计测量对象的节点数据磁盘使用率，该值为实时值。	80	组件	60 秒
rds024_current_sleep_time	主机流控时间	该指标用于统计测量对象的主机流控时间，该值为实时值。	1	组件	60 秒
rds025_current_rto	备机 RTO 时间	该指标用于统计测量对象的主备复制的 RTO，该值为实时值。	60	组件	60 秒
rds026_login_counter	用户登入次数/秒	该指标用于统计 CN 节点上每秒的登入次数，取时间段的平均值。	1000	组件	60 秒
rds027_logout_counter	用户登出次数/秒	该指标用于统计 CN 节点上每秒的登出次数，取时间段的平均值。	1000	组件	60 秒
rds028_standby_delay	备机 redo 进度	该指标用于统计分片内备机 redo 进度，表示备机和主机的差距，该值为实时值。	200MB	组件	60 秒
rds030_wait_ratio	锁等待状态会话比率	该指标用于统计当前处于锁等待状态会话占活跃工作状态下会话比率，该值为实时值。	50	组件	60 秒
rds031_active_ratio	活跃会话率	该指标用于统计当前处于活跃工作状态会话占总会话数比率，该值为实时值。	20	组件	60 秒
rds034_inuse_counter	CN 连接数	该指标用于统计 CN 连接池中正在使用的连接数，该值为实时值。	80	组件	60 秒
rds037_commit_co	用户提交事	该指标用于统计用户每秒提交的事务数，取时	50000	组件	60 秒

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期(原始指标)
unter	务数/秒	间段的平均值。			
rds038_rollback_counter	用户回滚事务数/秒	该指标用于统计用户每秒回滚的事务数，取时间段的平均值。	0	组件	60 秒
rds039_bg_commit_counter	后台提交事务数/秒	该指标用于统计后台每秒提交的事务数，取时间段的平均值。	5000	组件	60 秒
rds040_bg_rollback_counter	后台回滚事务数/秒	该指标用于统计后台每秒回滚的事务数，取时间段的平均值。	0	组件	60 秒
rds041_resp_avg	用户事务平均响应时间	该指标用于统计用户事务的平均响应时间。	10ms	组件	60 秒
rds042_rollback_ratio	用户事务回滚率	该指标用于统计用户事务回滚事务占用户提交、回滚事务之和的比率，取时间段的平均值。	0	组件	60 秒
rds043_bg_rollback_ratio	后台事务回滚率	该指标用于统计后台事务回滚事务占用户提交、回滚事务之和的比率，取时间段的平均值。	0	组件	60 秒
rds044_ddl_count	data definition language	该指标用于统计用户负载在 query 层的 DDL 数量，取时间段的平均值。	10	组件	60 秒
rds045_dml_count	data manipulation language/min	该指标用于统计用户负载在 query 层的 DML 数量，取时间段的平均值。	50000	组件	60 秒
rds046_dcl_count	data Control Language/min	该指标用于统计用户负载在 query 层的 DCL 数量，取时间段的平均值。	10	组件	60 秒
rds047_ddl_dcl_ratio	DDL+DCL 比率	该指标用于统计用户负载在 query 层的 DDL+DCL 占 DDL+DCL+DML 的比	5	组件	60 秒

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期(原始指标)
		率，取时间段的平均值。			
rds050_ckpt_delay	待落盘的数据量	该指标用于统计信息同步到磁盘过程中待落盘的数据量，该值为实时值。	200MB/s	组件	60 秒
rds051_phyrds	读物理文件的 IO 次数/秒	该指标用于统计数据库每秒读物理物件的 IO 次数，取时间段的平均值。	20000	组件	60 秒
rds052_phywrts	写物理文件的 IO 次数/秒	该指标用于统计数据库每秒写物理物件的 IO 次数，取时间段的平均值。	10000	组件	60 秒
rds053_online_session	在线会话数量	该指标用于统计当前在线的 session 个数，该值为实时值。	NA	组件	60 秒
rds054_active_session	活跃会话数量	该指标用于统计当前所有活跃工作状态下会话个数，该值为实时值。	NA	组件	60 秒
rds055_online_ratio	在线会话率	该指标用于统计 CN（分布式）/主 DN（主备版）上的在线会话比例，该值为实时值	NA	组件	60 秒
rds060_long_running_transaction_exectime	数据库最长事务的执行时长	该指标用于统计测量对象的数据库最长事务的执行时长，该值为实时值。	NA	组件	60 秒
rds066_replication_slot_wal_log_size	复制槽保留的 WAL 日志大小	该指标用于统计主 DN 上复制槽中保留的 WAL 日志的大小，该值为实时值。	NA	组件	60 秒
rds067_xlog_lsn	xlog 速率	该指标用于统计 CN 或者 DN 上 xlog 的速率，该值为实时值。	NA	组件	60 秒
rds068_swap_used_ratio	交换内存使用率	该指标用于描述操作系统交换内存使用率，该值为实时值。	NA	节点	60 秒

指标 ID	指标名称	指标含义	取值范围	测量对象	监控周期(原始指标)
rds069_swap_total_size	交换内存总大小	该指标用于描述操作系统交换内存总大小，该值为实时值。	NA	节点	60 秒

维度

表3-4 云数据库 GaussDB 涉及的维度

Key	Value
gaussdbv5_instance_id	云数据库 GaussDB 实例
gaussdbv5_node_id	云数据库 GaussDB 节点
gaussdbv5_component_id	云数据库 GaussDB 组件

3.8.2 创建告警规则

操作场景

通过设置数据库告警规则，用户可自定义监控目标与通知策略，及时了解数据库运行状况，从而起到预警作用。

设置的告警规则包括设置告警规则名称、资源类型、维度、监控对象、监控指标、告警阈值、监控周期和是否发送通知等参数。

设置告警规则

步骤 1 登录管理控制台。

步骤 2 选择“管理与监管 > 云监控服务 CES”。

步骤 3 在左侧导航树栏，选择“告警 > 告警规则”。

步骤 4 在“告警规则”界面，单击“创建告警规则”进行添加。

----结束

3.8.3 查看监控指标

操作场景

云服务平台提供的云监控，可以对数据库的运行状态进行日常监控。您可以通过管理控制台，直观地查看数据库的各项监控指标。您可以查看实例监控。

由于监控数据的获取与传输会花费一定时间，因此，云监控显示的是当前时间 5～10 分钟前的数据库状态。如果您的数据库刚创建完成，请等待 5～10 分钟后查看监控数据。

前提条件

- 数据库正常运行。

  故障、删除状态的数据库，无法在云监控中查看其监控指标。当数据库再次启动或恢复后，即可正常查看。

  说明

  故障 24 小时的数据库，云监控将默认该数据库不存在，并在监控列表中删除，不再对其进行监控，但告警规则需要用户手动清理。

- 数据库已正常运行一段时间（约 10 分钟）。

  对于新创建的数据库，需要等待一段时间，才能查看上报的监控数据和监控视图。

查看实例监控

步骤 1 登录管理控制台。

步骤 2 单击管理控制台左上角的 ，选择区域和项目。

步骤 3 在“所有服务”或“服务列表”中选择“管理与监管 > 云监控”，进入“云监控”服务信息页面。

步骤 4 在左侧导航栏选择“云服务监控 > 云数据库 云数据库 GaussDB”。

步骤 5 在云监控页面，单击需要查看监控的实例，可以查看指定实例的监控信息。

云监控支持的性能指标监控时间窗包括：近 1 小时、近 3 小时、近 12 小时、1 天、7 天。

----结束

3.9 CTS 审计

3.9.1 支持审计的关键操作列表

通过云审计服务，您可以记录与云数据库 GaussDB 实例相关的操作事件，便于日后的查询、审计和回溯。

表3-5 云审计服务支持的操作列表

操作名称	资源类型	事件名称
创建实例、恢复到新实例	instance	createInstance
删除实例	instance	deleteInstance
数据库实例规格变更	instance	resizeFlavor
实例版本升级	instance	upgradeVersion
密码重置	instance	resetPassword
实例重启	instance	instanceRestart
修改资源标签	instance	modifyTag
删除资源标签	instance	deleteTag
添加资源标签	instance	createTag
重命名实例	instance	instanceRename
实例扩容	instance	instanceAction
删除任务记录	instance	deleteTaskRecord4OpenGauss
减少副本	instance	reduceReplica
协调节点缩容	instance	reduceCoordinatorNode
设置回收站策略	backup	setRecyclePolicy
创建手动备份	backup	createManualSnapshot
删除手动备份	backup	deleteManualSnapshot
修改备份策略	backup	setBackupPolicy
实例还原	backup	restoreInstance

3.9.2 查看追踪事件

操作场景

在您开通了云审计服务后，系统开始记录云服务资源的操作。云审计服务管理控制台保存最近 7 天的操作记录。

本节介绍如何在云审计服务管理控制台查看最近 7 天的操作记录。

操作步骤

步骤 1 登录管理控制台。

步骤 2 单击左侧导航树的“事件列表”，进入事件列表信息页面。

步骤 3 事件列表支持通过筛选来查询对应的操作事件。详细信息如下：

- 事件来源、资源类型和筛选类型：在下拉框中选择查询条件。
  其中筛选类型选择资源 ID 时，还需选择或者手动输入某个具体的资源 ID。
- 操作用户：在下拉框中选择某一具体的操作用户。
- 事件级别：可选项为“所有事件级别”、“normal”、“warning”、“incident”，只可选择其中一项。
- 时间范围：可通过选择时间段查询操作事件。

步骤 4 选择查询条件后，单击“查询”。

步骤 5 在需要查看的记录左侧，单击展开该记录的详细信息。

步骤 6 在需要查看的记录右侧，单击“查看事件”，在弹出框中显示该操作事件结构的详细信息。

步骤 7 单击右侧的“导出”，将查询结果以 CSV 格式的文件导出，该 CSV 文件包含了云审计服务记录的七天以内的操作事件的所有信息。

关于事件结构的关键字段详解，请参见《云审计服务用户指南》的“事件结构”和“事件样例”章节。

----结束

3.10 LTS 日志

操作场景

配置访问日志后，云数据库 GaussDB 实例新生成的审计日志记录会上传到云日志服务（Log Tank Service，简称 LTS）进行管理。您可以查看云数据库 GaussDB 实例审计日志的详细信息，包括搜索日志、日志可视化、下载日志和查看实时日志等功能。

- 配置单个实例访问日志：添加单个实例的 LTS 配置。
- 解除单个实例访问日志：解除单个实例的 LTS 配置。

使用须知

- 当前只提供主备版实例， 并且数据库引擎需要为 2.1.0 及以上版本。
- 访问日志提供了实例所请求的所有详细日志，日志存在 LTS 云日志服务中。
- 配置完成后，日志不会立即上传，需要等待 10 分钟左右才可以在 LTS 服务上查询审计日志。
- 在您进行 LTS 审计日志配置后，会默认上传当前实例的所有审计策略到 LTS 服务。

配置单个实例访问日志

步骤 1　3.1 登录管理控制台。

步骤 2　单击管理控制台左上方的，选择区域和项目。

步骤 3　在页面左上角单击，选择“数据库 > 云数据库 云数据库 GaussDB”，进入云数据库 GaussDB 页面。

步骤 4　在左侧导航树，单击“实例管理”。

步骤 5　在右边列表点击实例名称，进入实例详情。

步骤 6　在实例详情的左侧导航树里，单击“审计日志”。

步骤 7　在右边页面中找到“开启上传审计日志到 LTS”，单击相邻右侧。

步骤 8　在弹框中，选择“日志组”和“日志流”。

图3-16 配置 LTS 参数

说明

步骤 9　单击“保存”。

----结束

解除单个实例访问日志

步骤 1　3.1 登录管理控制台。

步骤 2　单击管理控制台左上方的，选择区域和项目。

步骤 3 在页面左上角单击☰，选择“数据库 > 云数据库 云数据库 GaussDB”，进入云数据库 GaussDB 页面。

步骤 4 在左侧导航树，单击“实例管理”。

步骤 5 在右边列表点击实例名称，进入实例详情。

步骤 6 在实例详情的左侧导航树里，单击“审计日志”。

步骤 7 在右边页面中找到“开启上传审计日志到 LTS”，单击相邻右侧  。

步骤 8 在弹框中，确认解除 LTS 的实例信息。

步骤 9 在弹框中，单击“是”。

----结束

3.11 配额管理

云数据库 GaussDB 在管理控制台提供配额管理的功能，可以对租户下的企业项目进行配额管理。

只有账号为企业账号，并配置白名单后，才可以使用配额管理功能。

管理配额

步骤 1 3.1 登录管理控制台。

步骤 2 在左侧导航栏，单击“配额管理”，进入“配额管理”界面。

可以在此界面查看每个项目的实例使用情况、CPU 使用情况、内存使用情况、存储空间信息。

步骤 3 选择需要管理的企业项目，单击“操作”列的“编辑”。

说明

首次进入配额管理页面时，此处显示为“设置”。

步骤 4 在弹出的对话框中填写需要变更的配额数量。单击“确认”。

----结束

3.12 任务中心

3.12.1 查看任务

您可以通过“任务中心”查看任务执行进度和结果，并进行管理。

说明

云数据库 GaussDB 支持查看和管理以下任务：

- 创建云数据库 GaussDB 实例
- 手动创建备份
- 恢复到新实例
- 分片扩容
- 协调节点扩容

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“任务中心”页面，选择目标任务，单击任务名称左侧的，在展开信息中查看目标任务的执行进度和结果。

- 通过任务名称/任务 ID、实例名称/ID 确定目标任务，或通过右上角的搜索框选择任务类型来确定目标任务。
- 单击页面右上角的，查看某一时间段内的任务执行进度和状态，默认时长为一周。

  任务保留时长最多为 30 天。
- 系统支持查看以下状态的任务：
  - 执行中
  - 完成
  - 失败

----结束

3.12.2 删除任务

对于不再需要展示的任务，您可以通过“任务中心”进行任务记录的删除。删除任务仅删除记录，不会删除数据库实例或者停止正在执行中的任务。

须知

删除任务将无法恢复，请谨慎操作。

操作步骤

步骤 1 3.1 登录管理控制台。

步骤 2 在“任务中心”页面，选择目标任务，单击操作列的“删除”，在弹出框中单击“是”，删除任务。

云数据库 GaussDB 服务支持删除以下状态的任务：

- 完成
- 失败

----结束

3.13 计费管理

3.13.1 实例续费

操作场景

您可根据业务需要，对单个“包年/包月”实例进行续费，暂不支持批量续费。

说明

“按需付费”的实例暂不支持续费。

运行状态为“正常”或“异常”的实例才可进行续费。

对当前实例续费

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择目标实例，单击“操作”列的“续费”。

您也可以单击目标实例名称，进入实例的“基本信息”页面，在“计费信息”模块的“计费模式”处，单击“续费”。

步骤 3 进入续费页面，对实例进行续费。

----结束

3.13.2 按需实例转包周期

操作场景

云数据库 GaussDB 服务支持按需实例转为包周期（包年/包月）实例。由于按需资源费用较高，需要长期使用资源的按需用户可以选择对按需资源进行转包周期，继续使用这些资源的同时，享受包周期的优惠资费。

说明

单个按需实例转包周期

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择目标实例，单击“操作”列的“转包周期”，进入“按需转包周期”页面。

步骤 3 选择续费规格，以月为单位，最小包周期时长为一个月。

- 如果订单确认无误，单击“提交”，进入“支付”页面。
- 如果暂不确定实例规格，单击“确认订单，暂不付款”，系统将保留您的订单，稍后可在“费用 > 我的订单”中支付或取消订单。

步骤 4 选择支付方式，单击“确认付款”。

步骤 5 按需转包周期创建成功后，用户可以在“实例管理”页面对其进行查看和管理。

在实例列表的右上角，单击  刷新列表，可查看到按需转包周期完成后，实例状态显示为“正常”。“计费模式”显示为“包年/包月”。

----结束

按需实例批量转包周期

说明

仅“按需计费”模式的实例支持转包周期。

运行状态为“正常”或“异常”的实例才可转包周期。

批量转换属于白名单特性，如需配置白名单权限，您可以在管理控制台右上角，选择，提交开通白名单的申请。

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，勾选目标实例，单击实例列表上方的“转包周期”，进入“按需转包周期”页面。

步骤 3 选择续费规格，以月为单位，最小包周期时长为一个月。

- 如果订单确认无误，单击“提交”，进入“支付”页面。
- 如果暂不确定实例规格，单击“确认订单，暂不付款”，系统将保留您的订单，稍后可在“费用 > 我的订单”中支付或取消订单。

步骤 4 选择支付方式，单击“确认付款”。

步骤 5 按需转包周期创建成功后，用户可以在“实例管理”页面对其进行查看和管理。

在实例列表的右上角，单击  刷新列表，可查看到按需转包周期完成后，实例状态显示为“正常”。“计费模式”显示为“包年/包月”。

----结束

3.13.3 包周期实例转按需

操作场景

云数据库 GaussDB 支持单个包周期（包年/包月）实例转为按需实例，方便用户灵活使用该资费的实例。

须知

实例的按需计费方式需要等包周期到期后才会生效，且自动续费功能会同步失效。

单个包周期实例转按需

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择目标包周期实例，单击“操作”列的“更多 > 转按需”，进入“包周期转按需”页面。

步骤 3 进入计费模式变更页面，对实例进行计费模式的变更。

步骤 4 包周期转按需创建成功后，用户可以在“实例管理”页面对其进行查看和管理。

在实例列表的右上角，单击  刷新列表，可查看到按需转包周期完成后，实例状态显示为“正常”。“计费模式”显示为“按需”。

----结束

包周期实例批量转按需

说明

仅“包年/包月”模式的实例支持转按需。

运行状态为“正常”或“异常”的实例才可转按需。

实例的按需计费方式需要等包周期到期后才会生效，且自动续费功能会同步失效。

批量转换属于白名单特性，如需配置白名单权限，您可以在管理控制台右上角，选择，提交开通白名单的申请。

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，勾选目标实例，单击实例列表上方的“转按需”，进入“包周期转按需”页面。

步骤 3 进入计费模式变更页面，对实例进行计费模式的变更。

步骤 4 包周期转按需创建成功后，用户可以在“实例管理”页面对其进行查看和管理。

在实例列表的右上角，单击  刷新列表，可查看到按需转包周期完成后，实例状态显示为“正常”。“计费模式”显示为“按需”。

----结束

3.13.4 退订包周期实例

操作场景

对于“包年/包月”模式的数据库实例，您需要退订订单，从而删除数据库实例资源。目前仅支持退订单个实例，具体退订操作请参考退订单个实例。

对于“按需计费”模式的实例，您需要在“实例管理”页面对其进行删除，更多操作请参见 3.4.3 删除实例。

退订单个实例

您可在“实例管理”页面的实例列表中，退订包周期实例。

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择目标实例，单击“操作”列的“更多 > 退订”。

步骤 3 在“退订资源”页面，确认待退订实例信息，并选择退订原因，单击“退订”。

步骤 4 在弹出框中确认是否退订该资源，单击“是”，提交退订申请。

须知

- 提交退订后，资源和数据将会被删除并无法找回。
- 如需保留数据，请务必确认完成数据备份后再提交退订。

步骤 5 查看退订结果。数据库实例订单退订成功后，实例将会被删除，即“实例管理”页面将不再显示该订单对应的数据库实例。

----结束

3.14 标签

操作场景

标签管理服务（Tag Management Service，TMS）用于用户在云平台，通过统一的标签管理各种资源。TMS 服务与各服务共同实现标签管理能力，TMS 提供全局标签管理能力，各服务维护自身标签管理 。

- 建议您先在 TMS 系统中设置预定义标签。
- 标签由“键”和“值”组成，每个标签中的一个“键”只能对应一个“值”。
- 每个实例最多支持 20 个标签配额。

添加标签

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例的“基本信息”页签。

步骤 3 在左侧导航栏，单击“标签”，单击“添加标签”，在“添加标签”弹出框中，输入标签的键和值，单击“确定”。

- 标签的键不能为空且必须唯一，长度为 1~36 个字符，只能包含字母、数字、中划线、下划线和@符号。
- 标签的值可以为空字符串，长度为 0~43 个字符，只能包含字母、数字、中划线、下划线、 英文句点和@符号。

步骤 4 添加成功后，您可在当前实例的所有关联的标签集合中，查询并管理自己的标签。

----结束

编辑标签

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称。

步骤 3 在左侧导航栏，单击“标签”，选择需要编辑的标签，单击“编辑”。

步骤 4 在“编辑标签”弹出框中修改标签值，单击“确定”。

- 编辑标签时，不能修改标签的键，只能修改标签的值。
- 标签的值可以为空字符串，长度为 0~43 个字符，只能包含字母、数字、中划线、下划线、英文句点和@符号。

步骤 5 编辑成功后，您可在当前实例的所有关联的标签集合中，查询并管理自己的标签。

----结束

删除标签

步骤 1 3.1 登录管理控制台。

步骤 2 在“实例管理”页面，选择指定的实例，单击实例名称。

步骤 3 在左侧导航栏，单击“标签”，选择需要删除的标签，单击“删除”。

步骤 4 在“删除标签”弹出框中单击“是”。

步骤 5 删除成功后，该标签将不再显示在实例的所有关联的标签集合中。

----结束

3.15 回收站

云数据库 GaussDB 支持将删除的实例，加入回收站管理。您可以在回收站中重建实例恢复数据。

回收站策略机制默认开启，默认保留时间为 7 天，且不可关闭。

设置回收站策略

须知

修改回收站保留天数，仅对修改后新进入回收站的实例生效，对于修改前已经存在的实例，仍保持原来的回收策略，请您谨慎操作。

步骤 1 3.1 登录管理控制台。

步骤 2 在左侧导航栏，单击“回收站”。

步骤 3 在“回收站”页面，单击“回收站策略”，设置已删除实例保留天数，可设置范围为 1~7 天。

步骤 4 单击“确定”，完成设置。

----结束

重建实例

在回收站保留期限内的实例可以通过重建实例恢复数据。

步骤 1 3.1 登录管理控制台。

步骤 2 在左侧导航栏，单击“回收站”。

步骤 3 在“回收站”页面，在实例列表中找到需要恢复的目标实例，单击操作列的“重建”。

步骤 4 在“重建新实例”页面，选填配置后，提交重建任务。

----结束

3.16 使用规范建议

3.16.1 概述

简介

本规范以产品生命周期为主线，详细描述产品设计和开发流程过程中与数据库相关的设计和开发规范。

规范以提高可读性、代码质量为原则，强调实用性、可操作性，对云数据库 GaussDB 数据库开发容易产生问题的地方做出了明确的规定。主要包括下列内容：

设计规范包括：数据库设计、性能设计。

编程规范包括：排版、命名、注释、语法、脚本、数据库编程、安装部署和安全规范。

同时，在必要时，对部分规范给出细则及具体的示例。

术语约定

本规范采用以下的术语描述：

- **规格**：数据库规格，编程、设计必须遵守的原则，否则数据库将报错。
- **规则**：编程、设计强制必须考虑遵守的原则。
- **建议**：编程、设计必须推荐加以考虑的原则。
- **说明**：对规则或建议进行必要的解释。
- **示例**：对规则或建议进行或正或反方面的举例。
- **分片**：将一个表中的数据按照指定的策略拆分并存储至多个 DN 上的同名表中，这些表中存储的数据互不重叠，可以理解为“表的水平分库”，举例：t_user 表按主键 userId Hash 拆分到云数据库 GaussDB 的 4 个不同 DN 中就是 4 个分片，每个分片内都有一张同名的 t_user 表。

适用范围

本规范适用于云数据库 GaussDB 1.x 及以上版本。

本规范适合人群包括：设计人员、开发人员、开发 DBA、运维 DBA、运维人员。

3.16.2 数据库设计规范

3.16.2.1 基本规范

数据库特性规范

表3-6 数据库价值特性推荐

特性分类	特性列表	说明
表类型	HASH 分布表	自动分片的表，建议数据量大的表使用（如交易记录）。
	REPLICATE 分布表	不分片的普通表，建议数据少的表使用（如国家名称表）。
事务	分布式事务（弱一致）	GTM Free 模式，在 sharding 场景下可保证强一致，不保证跨 DN 分片读一致性。建议完美 sharding 业务使用。
	分布式事务（强一致）	GTM Lite 模式，保证跨 DN 读写一致性，建议非完美 sharding 业务使用。
扩容	在线平滑扩容	在线业务的最大阻塞小于 5s，主要是为了追增切换期间写入日志，扩容速度 100G/小时。
数据类型	整数类型	TINYINT, SMALLINT, INTEGER, BIGINT
	任意精度类型	NUMERIC/DEMICAL
	浮点类型	REAL/FLOAT4,DOUBLE PRECISION/FLOAT8,FLOAT
	布尔类型	BOOLEAN
	定长字符	CHAR(n)
	变长字符	VARCHAR(n),NVARCHAR2(n), TEXT
	时间类型	DATE, TIME, TIMETZ, TIMESTAMP, TIMESTAMPTZ, SMALLDATETIME, INTERVAL, REALTIME
	二进制类型	BYTEA（变长二进制类型）
	位串类型	BIT(n), VARBIT(n)
函数	字符处理函数	字符类数据类型处理函数
	二进制字符串函数	二进制字符类型处理函数
	数字操作函数	数值类型处理函数
	时间和日期处理函数	时间和日期类型处理函数

特性分类	特性列表	说明
索引	主键/唯一索引	单列或多列主键/唯一索引
	BTREE 索引	索引类型

说明

未包含在价值列表中的特性（包括但不限于自定义，UUID 等数据类型，触发器等特性），如需使用建议联系云数据库 GaussDB 数据库技术人员进行评估。

数据库指标规范

表3-7 数据库指标规范列表

指标	推荐值
集群最佳分片数（主 DN 数）	<256
集群最佳长连接数	<10w
单 DN 数据量最大值	2TB
单表最佳字段个数	<50
单表最佳索引个数	<5
单表最佳复合索引个数	<3
单复合索引包含最佳列数	<5
单行最佳行宽	<2k
单个字段建议最大值	10MB
SQL 语句最佳长度	<5k
磁盘可用空间不足提示（考虑扩容极端情况）	45%
磁盘可用空间不足告警（考虑扩容极端情况）	55%

3.16.2.2 部署规范

资源评估规范

基于表 3-8 模板估算台账，资源利用率尽量控制在 40%-70%间，低 40%建议缩容，高于 70%建议扩容。

表3-8 云数据库 GaussDB 上线台账

组件	规格一	规格二	规格三	规格四	规格五	MEM/CPU	文件	磁盘容量	备注
CN	8C64G	16C128G	32C256G	ARM：60C480G X86：64C512G	/	8	/var/chroot/var/lib/log	64G	日志盘
							/var/chroot/usr/local	256G	数据盘
DN	8C64G	16C128G	32C256G	ARM：60C480G X86：64C512G	/	8	/var/chroot/var/lib/log	64G	日志盘
							/var/chroot/var/lib/engine/data	3788G	数据盘
							/var/chroot/var/lib/log/backup	20G	备份会把元数据信息放一份到这个盘上
GTM	4C32G	8C64G	16C128G	32C256G	ARM：60C480G X86：64C512G	8	/var/chroot/var/lib/log	32G	日志盘
	1-9分片	9-16分片	16-32分片	32-64分片	64分片以上		/var/chroot/usr/local	32G	数据盘
CMS+ETCD	4C32G	8C64G	16C128G	32C256G	ARM：60C480G X86：64C512G	8	/var/chroot/var/lib/log	32G	日志盘
	1-9分片	9-16分片	16-32分片	32-64分片	64分片以上		/var/chroot/usr/local	64G	数据盘

数据库参数规范

安装数据库后，DBA 应根据环境特性，合理配置数据库 GUC 参数，当前云数据库 GaussDB 已提供默认参数模板，如果业务模型较为特殊的，需要联系技术支持进行调参。

3.16.2.3 数据库对象命名规范

- 数据库对象命名需要满足约束：长度不超过 63 个字符，以字母或下划线开头， 中间字符可以是字母、数字、下划线、$、#。

说明

也可以使用如下指令查看关键字：

SELECT * **FROM** pg_get_keywords();

- 避免使用双引号括起来的字符串来定义数据库对象名称，除非必须限制数据库对象名称的大小写。

说明

云数据库 GaussDB 默认不区分 SQL 中对象名称的大小写，假如同一数据库中同时存在“t_Table”“t_table”两张不同的表，这种情况下应使用“”双引号，如果不存在该情况，禁止避免使用“”双引号。数据库对象名称大小写敏感会使定位问题难度增加。

- 数据库对象命名风格务必保持统一，建议使用小写。

  增量开发的业务系统或进行业务迁移的系统，建议遵守历史的命名风格。

  建议使用多个单词组成，以下划线分割。

  数据库对象名称建议能够望文知意，尽量避免使用自定义缩写（可以使用通用的术语缩写进行命名）。例如，在命名中可以使用具有实际业务含义的英文词汇或汉语拼音，但规则应该在集群范围内保持一致。

  变量名的关键是要具有描述性，即变量名称要有一定的意义，变量名要有前缀标明该变量的类型。

表3-9 数据库对象命名规则

对象类型	前缀	范例	长度约束 (单位：字节)	备注
数据库名	db__	db_businessname	<=63	/
普通表	t_	t_tablename	<=63	/
临时表	tmp_	tmp_tablename	<=63	例如：运营人员临时用作备份或临时进行数据采集用的中间表。 建议使用命名规则：tmp_表名缩写_创建人账号缩写_创建日期，例如：tmp_user_ytw_160505

对象类型	前缀	范例	长度约束(单位：字节)	备注
主键	pk_	pk_tablename	<=63	如果表名过长，则用表名的缩写表示，尽量使用通用缩写或去元音的缩写方式。
唯一性约束	uk_	uk_tablename_columnname	<=63	唯一索引，用 uk_表示。 如果表名或字段名过长，则用表名和字段名的缩写表示，尽量使用通用缩写或去元音的缩写方式。
函数	f_	f_functionname	<=63	/
表字段	/	/	<=63	字段命名建议使用实际含义的英文单词或简写。 例如表示 bool 类型的字段，命名规则：“is_”+描述。如 member 表上表示为 enabled 的会员的列命名为 is_enabled；

3.16.2.4 数据库设计规范

- 使用 JDBC 客户端连接数据库时必须指明数据库名，具体格式为：

```
jdbc:postgresql://host:port/database?param1=value1&param2=value2
```

- JDBC 实例一旦创建，无法进行数据库切换。
- 数据库目前不支持不区分大小写的排序方式。
- 目前仅支持对数据库定义字符集，不支持对表、字段等其他对象定义字符集。
- 业务使用前必须先创建业务数据库。

说明

不应使用数据库安装后默认创建的 postgres 数据库存储业务数据。

- 创建数据库时必须指定字符集为 UTF8，创建数据库时必须选择与客户端统一的编码字符集。

  为了使用全球化需求，数据库编码应能够存储与标识绝大多数的字符，因此推荐使用 UTF8。云数据库 GaussDB 中的 UTF8 字符集与 MySQL 的 UTF8MB4 等价，能够支持 emoji 表情字符。

  如果客户端的编码方式与数据库的编码方式不统一，会带来转码性能，同时，针对同编码的内核优化无法触发，影响查询效率。

  客户端的编码字符集需通过以下方式修改：

  - 设置客户端连接参数，例如 JDBC 连接参数可通过在 URL 中追加 characterEncoding 和 allowEncodingChanges 参数

```
jdbc:postgresql://ip:port/database_name?characterEncoding=utf8&allowEncodingChanges=true
```

- 修改数据库 GUC 参数

```
SET client_encoding = 'UTF8';
```

- 数据库的编码在 CREATE DATABASE 时进行设置。

```
CREATE DATABASE tester WITH ENCODING = 'UTF8';
```

- 数据库一旦创建无法更改字符集。
- 从便捷性和资源共享效率上考虑，建议使用 SCHEMA 进行业务隔离。

说明

云数据库 GaussDB 可以使用 DATABASE 和 SCHEMA 两种方式实现业务的隔离。

区别在于 DATABASE 的隔离更加彻底，各个 DATABASE 之间共享资源极少，可实现连接隔离、权限隔离等。

但 DATABASE 之间无法互相访问，JDBC 建连时必须指明 DATABASE，连接后无法切换 DATABASE。

SCHEMA 隔离的方式共用资源较多，可以通过 GRANT 与 REVOKE 语法便捷地控制不同用户对各 SCHEMA 及其下属对象的权限。

- 创建数据库时建议指定 LC_COLLATE 和 LC_CTYPE 和存放的数据内容语言（中文\英文\等等）一致，该参数将影响数据的排序顺序。默认会用系统当前环境变量的默认设置。

  示例：

```
CREATE DATABASE tester WITH ENCODING = 'UTF8' LC_COLLATE = 'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8';
```

LC_COLLATE：用于明确字符排序规则。

```
LC_COLLATE=C
1
2
3
A
B
C
a  --注：小写在大写后面，按 ASCII 码排序
b
c

en_US.UTF-8
1
2
3
a  --注：按字符排序
A
b
B
c
C

zh_CN.UTF-8
1
2
3
```

```
a
A
b
B
c
C
```

LC_CTYPE：用于判断哪些是字符 is_alpha，是大写 is_upper 还是小写 is_lower。

3.16.2.5 权限设计规范

- 业务使用前必须由 root 用户为业务创建 DATABASE、SCHEMA 和 USER，然后再赋予相关用户对应对象的权限。

说明

如果该用户不是该 schema 的 owner，要访问 schema 下的对象，需要同时给用户赋予 schema 的 usage 权限和对象的相应权限。

- DATABASE、SCHEMA 和 USER 名使用小写。

说明

由于数据库默认会把对象名称转为小写，连接串里面如果出现大写的对象名无法连接到数据库。如 create user MyUser;创建的用户，无法使用 user=MyUser 连接到数据库，需要用 user=myuser 才能连接到数据库。为了避免混淆，一律使用小写。

- 合理对角色和用户赋权，应使用最小化权限原则。

表3-10 数据库对象权限及说明

对象	权限	说明
数据库 DATABASE	CONNECT	允许用户连接到指定的数据库
	TEMP	-
	CREATE	允许在数据库里创建新的模式
模式 SCHEMA	CREATE	允许在模式中创建新的对象
	USAGE	允许访问包含在指定模式中的对象，若没有该权限，则只能看到这些对象的名字。
函数 FUNCTION	EXECUTE	允许使用指定的函数，以及利用这些函数实现的操作符
表空间 TABLESPACE	CREATE	允许在表空间中创建表，允许在创建数据库和模式的时候把该表空间指定为缺省表空间。
表 TABLE	INSERT，DELETE UPDATE，SELECT	允许用户对指定表进行增删改查操作
	TRUNCATE	允许执行 TRUNCATE 语句删除指定表中的所有记录。
	REFERENCES	创建一个外键约束，必须拥有参考

对象	权限	说明
		表和被参考表的 REFERENCES 权限

- 通过角色而不是用户来管理权限。

  使用角色管理权限，即在角色中配置权限，再将角色赋予用户。

  通过角色管理权限，更便于多用户、用户变更等场景下的权限管理。例如：

  - 角色和用户为多对多关系，一个角色可以赋予多个用户，修改角色中的权限，被赋予角色的用户权限就可以同时更新。
  - 删除用户时，不会影响到角色。
  - 新建用户后可以通过赋予角色快速获取所需权限。
- 在删除指定数据库时，应回收用户对该数据库的 CONNECT 权限，避免删除时仍然存在活跃的数据库连接而失败。

3.16.2.6 表设计规范

- 必须指定表分布（DISTRIBUTE BY），表分布策略选择的原则如下：

  目前提供 REPLICATION 和 HASH 两种表分布策略。REPLICATION 分布会在每个节点保留一份相同的完整的数据表。HASH 分布会根据所提供的分布键值将表数据分布到多个节点中。

  - 对于系统配置表、数据字典表等数据规模小于 2000w 且插入更新十分低频的表，要求建议采用 REPLICATION 分布。

须知

慎用 REPLICATION 分布，该分布表会造成空间膨胀、DML 性能下降等负面影响。

  - 对于数据量较大，更新频率较高的表，必须进行数据分片，要求采用 HASH 分布策略，分布键必须建议是主键中的一个或多个字段。
- 合理设计分布键，既考虑查询开发的便利性，又要考虑数据的均匀存储，避免数据倾斜和读热点。

  Hash 表的分布键选取至关重要，如果分布键选择不当，可能会导致数据倾斜，从而导致查询时，I/O 负载集中在部分 DN 上，影响整体查询性能。因此，在确定 Hash 表的分布策略之后，需要对表数据进行倾斜性检查，以确保数据的均匀分布。

  a. 应使用取值较为离散的字段作为分布键，以便数据能够均匀分布到各个 DN 中。

  b. 在满足条件 1 情况下，存在常量过滤的字段不建议成为分布键，否则会使得所有的查询任务都会分发到唯一固定的 DN 上。

  c. 在满足条件 1 和 2 原则下，尽量选择查询中的关联条件作为分布键，这样可保证 JOIN 任务的相关数据分布在相同的 DN 上，减少 DN 间数据的流动代价。

说明

尽量避免数据 shuffle。shuffle，是指在物理上，数据从一个节点，传输到另一个节点。shuffle 占用了大量宝贵的网络资源，减小不必要的数据 shuffle，可以减少网络压力，使数据的处理本地化，提高集群的性能和可支持的并发度。通过对关联条件和分组条件的仔细设计，能够尽可能的减少不必要的数据 shuffle。

d. 由于数据库规格要求 HASH 分布表的主键必须包含其分布列，因此在选择分布列时，也可以考虑选择表的主键作为分布键。

表3-11 **常见的分布键及效果**

分布键值	分布键分布均匀性
用户 ID，应用程序中有许多用户。	好
状态代码，只有几个可用的状态代码。	差
项目创建日期，四舍五入至最近的时间段（例如，天、小时或分钟）。	差
设备 ID，每个设备以相对类似的间隔访问数据.	好

- 分布键使用的列长度不易超过 128，过长会带来较高的计算开销。
- 分布键值一旦插入不允许更新（UPDATE），如需更新需删除后插入。
- 视图不允许嵌套。

  一方面，如果视图编写时使用了通配符，当被调用的视图新增或删除列时，视图将发生错误。

  另一方面，视图嵌套可能因无法使用索引而执行效率低下，尽量使用带有索引的基表而不是视图上做关联操作。
- 分布键不建议超过 3 列，列数过多将带来较高的计算开销。
- 视图定义中尽量避免排序操作。

  ORDER BY 子句在顶层视图上无效，如果必须对输出数据排序，请考虑在调用视图中使用 ORDER BY。

3.16.2.7 字段设计规范

- 字段设计应使用推荐类型。

  字段设计需使用推荐字段，如果需要使用禁用、不推荐的字段类型，建议联系技术支持进行评估。

  这些数据类型不推荐或禁止的原因是业务使用场景较少，未大规模商用。

  对于业务上有迫切字段类型要求的，联系技术支持，提交需求。

表3-12 **数据库数据类型最佳实践**

数据类型	说明	是否推荐
UUID	不同集群可能产生相同 UUID	禁止，建议业务直接采用中间件

数据类型	说明	是否推荐
		平台提供的分布式 ID
序列整型	即自增列，包括 SMALLSERIAL,SERIAL,BIGSERIAL	禁止
整数类型	TINYINT, SMALLINT, INTEGER, BIGINT	推荐
任意精度类型	NUMERIC/DEMICAL	推荐
浮点类型	REAL/FLOAT4,DOUBLE PRECISION/FLOAT8,FLOAT	推荐
布尔类型	BOOLEAN	推荐
定长字符	CHAR(n)	推荐
变长字符	VARCHAR(n),NVARCHAR2(n) VARCHAR/TEXT	推荐
时间类型	DATE, TIME, TIMESTAMP, SMALLDATETIME, INTERVAL, REALTIME	推荐
	TIMETZ，TIMESTAMPTZ	不推荐
二进制类型	BYTEA（变长二进制类型）	推荐
	CLOB（字符大对象）,BLOB（二进制大对象），RAW（变长十六进制）	禁止
位串类型	BIT(n), VARBIT(n)	推荐
特殊字符类型	NAME,"CHAR"，通常供数据库系统内部使用	禁止
JSON 类型	JSON 类型目前不支持操作符	禁止
自定义类型	可用于定义枚举 EMU 等类型	禁止
HLL 数据类型	建议直接使用 HLL 相关函数，减少性能影响	禁止
货币类型	MONEY 存储带有固定小数精度的货币金额	禁止
几何类型	POINT, LSEG, BOX, PATH, POLYGON, CIRCLE	禁止
网络地址类型	存储 IPV4 IPV6 MAC 地址数据类型	禁止
文本搜索类型	用于支持全文检索	禁止

- 合理选用字符串数据类型。优先使用变长字符类 VARCHAR。只有该字段输入确定为固定字符则使用定长字符类型，或需要自动补充空格，才使用 CHAR(n)。

说明

典型的定长字段类型，例如“sex”字段，仅允许输入“f”或“m”一个字节长度的字符。这类字段建议使用定长数据类型（如 CHAR(n)）。

如果不存在此特点，或者后续可能扩展需要输入更长的字符，请优先使用变长字符类型（如 VARCHAR, TEXT），且**不建议指定变长类型的长度**。

原因如下：

- 定长字段会对不够长度的输入数据补充空格，然后存入数据库中，产生不必要的存储空间浪费。
- 如果定义为定长字符类型，后续扩展长度，需要对全表进行扫描重写，性能开销大，影响在线业务。

对于指定固定长度的变长字段，每次插入时会检查是否长度越界，带来性能开销。

- 字符类型字段不应存储数字类型的数据。

  如果对存储在字符类型字段中的数据进行数值计算，或者与数值进行比较操作（如置于过滤条件中），会带来不必要的数据类型转换的开销，同时该字段上的索引可能失效，影响查询性能。

- 字符类型字段不应存储时间或日期类数据。

  如果对存储在字符类型字段中的数据与日期类数据进行计算或比较操作（如置于过滤条件中），会带来不必要的数据类型转换的开销，同时该字段上的索引可能失效，影响查询性能。

- 对于明确不存在 NULL 值的字段加上 NOT NULL 约束。

  对于 NOT NULL 字段，优化器在某些场景下会进行特殊优化，可较大提升查询性能。

- 相关联字段的数据类型应保持一致。

  在进行关联操作时，如果字段类型不一致，会带来数据类型转换开销。

- 大字段（例如 varchar（1000）、varchar(4000)）不建议超过 8 个。
- 字段定义时建议同时创建 COMMENT 注释信息，以便于未来维护。
- 用于 WHERE 条件过滤和关联的字段都应设置 NOT NULL 约束。

  对于 NOT NULL 字段，优化器在某些场景下会进行特殊优化，可较大提升查询性能。

- 不建议对表预留字段。大部分场景下可支持快速新增、删除表字段，或者修改字段的 DEFAULT 值。

说明

新增列必须符合以下要求，否则会带来全表更新开销，影响在线业务。

- 数据类型为以下类型中的一种：BOOL, BYTEA, SMALLINT, BIGINT, SMALLINT, INTEGER, NUMERIC, FLOAT, DOUBLE PRECISION, CHAR, VARCHAR, TEXT, TIMESTAMPTZ, TIMESTAMP, DATE, TIME, TIMETZ, INTERVAL；
- 新增列的 DEFAULT 值长度不超过 128 个字节；
- 新增列 DEFAULT 值不包含 volatile 函数；
- 新增列设置有 DEFAULT 值，且 DEFAULT 值不为 NULL。

如果不确定是否满足条件，请联系数据库技术人员进行评估。

- 尽量使用高效的数值类数据类型。在满足业务精度的情况下，选择的优先级从高到低依次为整数、浮点数、NUMERIC。
- 合理设置数值字段的数据类型，根据取值范围选择合适的数值类型，尽量少用NUMERIC/DECIMAL 类型。

  NUMERIC 和 DECIMAL 等价，NUMERIC(或 DECIMAL）数据类型操作对 CPU 消耗较高。

表3-13 **数值类数据类型存储空间及取值范围**

类型	存储空间	最小值	最大值
TINYINT	1	0	255
SMALLINT	2	-32768	32767
INTEGER	4	-2,147,483,648	2,147,483,647
BIGINT	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
REAL/FLOAT4	4	6 位十进制数字精度	
DOUBLE PRECISION/FLOAT8	8	15 位十进制数字精度	

3.16.2.8 索引设计规范

- 使用数据库索引实践推荐的索引类型。

  索引设计建议使用推荐类型，如果需要使用禁用、不推荐、限制使用的索引类型，建议联系云数据库 GaussDB 数据库专家进行评估。

表3-14 **数据库索引实践推荐**

索引类型	说明	是否推荐
主键/唯一索引	单列或多列主键/唯一索引	推荐
表达式索引	索引列为表的一列或多列计算而来的一个函数或者标量表达式	限制使用

- 对于 HASH 分布表，主键和唯一索引必须包含分布键。
- 合理设计组合索引，避免冗余。

  例如已对(a,b,c)创建索引，则不应再单独对 (a)、（b）、（c）、(a,b)、(b,c)创建索引。

  当查询时如果只带有 a 字段上的过滤条件，一般也会利用组合索引进行查询。

- 不建议单表创建多个唯一索引。

  同时维护多个唯一索引的开销远大于维护一个多列唯一索引，如果业务逻辑上多个唯一索引，与一个多列唯一索引等价，应使用多列唯一索引。
- 组合索引字段个数不超过 5 个。
- 禁止组合索引组合字符串的总长度超过 200。
- 索引（包括单列索引和复合索引）字段应为 NOT NULL 字段。
- 同字段上创建索引的维护效率不同。数值类型字段优于字符类型及其他数据类型，因此对于考虑创建索引的 ID、时间等字段，建议使用数值类型进行存储。
- 建议在关联列上创建索引。

  云数据库 GaussDB 支持 HASH JOIN，但是当内表较小等 RESCAN 代价较低的情况下，仍然可能选择 NESTLOOP JOIN 来完成关联。如果通过 EXPLAIN 可以查看到 NESTLOOP JOIN 计划，则可以通过在关联列上创建索引，提高 NESTLOOP JOIN 效率。

3.16.2.9 函数/存储过程设计规范

- 禁止避免使用存储过程、触发器等实现业务逻辑，应该将这些逻辑都放到业务服务器上处理，避免对数据库产生逻辑依赖。
- 业务的数据库升级脚本中，禁止使用存储过程实现升级逻辑。
- 仅创建对固定入参有固定返回值的函数，函数必须设为 IMMUTABLE 和 SHIPPABLE 类型。

  目前数据库支持三种类型的函数，分别是 IMMUTABLE, STABLE, VOLATILE。

  对于 IMMUTABLE 函数且设置为 SHIPPABLE 的函数，会允许函数在 DN 上执行。在大部分场景下，该函数的执行效率较高。

  但是此类函数要求对于固定的入参得到固定的返回值，来保证函数在 DN 上执行的正确性。如果函数的结果依赖对数据表的扫描结果（例如获取某个表中列的 max 值）或依赖时间（如获取当前时间），那么函数应设置为 STABLE 或者 VOLATILE，且 NOT SHIPPABLE，以保证函数执行的正确性。在此种场景下，所有 DN 上的数据将发送至某一个 CN 上进行计算，导致查询执行效率低下。

3.16.3 数据库编程规范

3.16.3.1 GUC 参数编程规范

客户端（如 JDBC）应使用默认（全局）参数执行查询，禁用会话级别的 GUC 参数。

通过 ODBC 或 JDBC 修改 GUC 参数时，需注意 GUC 参数仅会在当前 connection 中生效，特别是在连接池场景下，容易发生问题，且导致问题定位困难。

如果在连接中必须进行 GUC 参数设置，那么在将连接归还给连接池之前，必须使用

SET SESSION AUTHORIZATION DEFAULT;

RESET ALL;

将连接的状态清空。

3.16.3.2 对象访问编程规范

访问对象（表，函数等）时建议带上 SCHEMA 名称，即使用 schemaname.tablename 进行访问。

如果不追加 SCHEMA 名称前缀，会根据当前 search_path 中表空间列表，依次搜索所有表空间直到找到匹配的表作为目标表，带来不必要的性能开销。

3.16.3.3 WHERE

- 表查询时，WHERE 条件中应包含所有分布键字段等值查询条件，否则将在多个节点上进行查询，影响系统并发度和性能。
- 禁止在 WHERE 条件相同表字段进行相互比较。

  例如如下语句应考虑合理性：

  SELECT * **FROM** t1 **WHERE** col1 = col1;

  应考虑修改为：

  SELECT * **FROM** t1 **WHERE** col1 IS NOT NULL;
- 禁止 WHERE 条件涉及隐式数据类型转换。

  数据库中进行隐式转换后可能导致无法使用所创建的索引，导致潜在的性能问题。

  强烈建议在开发过程中开启 GUC 参数 check_implicit_conversions，并关闭 enable_fast_query_shipping，以便检查询语句中是否存在可能带来不良性能影响的隐式数据类型。

  SET enable_fast_query_shipping = off;

  SET check_implicit_conversions = true;

  由于隐式数据类型转换检测存在额外的开销，一旦查询语句开发完成后，请关闭 check_implicit_conversions 参数，并重置 enable_fast_query_shipping。

  示例：

  当不带如下代码不符合规范：

  t_tablename 表的 phonenumber 字段为 VARCHAR 类型（而不是数值类型），以下语句利用 phonenumber 进行条件过滤时，优化器会将 phonenumber 隐式转化为 bigint 类型。

  SELECT column1

  INTO i_l_variable1

  FROM t_tablename

  WHERE phonenumber = 13512345678;

  导致两个后果：

  a. 不能进行 DN 裁剪，计划下发到所有的 DN 上执行。

  b. 计划中不能使用 index scan 方式扫描数据。

  建议修改 t_tablename 表的 phonenumber 字段为 VARCHAR 类型（而不是数值类型）

  SELECT column1

  INTO i_l_variable1

  FROM t_tablename

WHERE phonenumber = '13512345678';

- 禁止 WHERE 条件字段使用表达式或是函数。

  对条件字段使用表达式或函数时，索引会失效，同时会对每一行数据进行计算，产生不必要的性能消耗。主要因为非常量的表达式在预处理阶段不能转化为 Const 值，因此不能用来剪枝，导致查询语句扫描所有的数据。

  示例：

  如下代码不符合规范：

  SELECT income **FROM** table **WHERE** abs(income) > ?;

  SELECT income **FROM** table **WHERE** income * 10 > ?;

  SELECT create_time

  FROM table

  WHERE date_format(create_time, '%Y-%m-%d %H:%i:%s') = '2009-01-01 00:00:0';

  应修改为：

  SELECT income **FROM** table **WHERE** income > ? **OR** income < (-1) * ?;

  SELECT income **FROM** table **WHERE** income > ?/10;

  SELECT create_time

  FROM table

  WHERE create_time = str_to_date('2009-01-01 00:00:0', '%Y-%m-%d %H:%i:%s');

- 查询条件中与 NULL 做比较时，禁止使用“！=”比较符，应使用 IS NULL 或 IS NOT NULL。

  不能写 expression=NULL 或 expression ！= NULL，因为 NULL 代表一个未知的值，不能通过表达式判断两个未知值是否相等。

- 查询条件中禁止对索引字段使用“!= ”比较符，避免索引失效。
- 在 where 子句中，应当对过滤条件进行排序，把选择读较小（筛选出的记录数较少）的条件排在前面。
- where 子句中的过滤条件，尽量符合单边规则。即把字段名放在比较条件的一边，优化器在某些场景下会自动进行剪枝优化。形如 col op expression，其中 col 为表的一个列， op 为‘ =’、‘ >’的等比较操作符， expression 为不含列名的表达式。

  示例：

  如下代码不推荐使用，根据 time 列进行筛选

  SELECT id, from_image_id, from_person_id, from_video_id FROM face_data WHERECurrent_timestamp(6) - time < '1 days'::interval;

  建议修改为：

  SELECT id, from_image_id, from_person_id, from_video_id FROM face_data where time >current_timestamp(6) - '1 days'::interval;

- 查询条件的索引字段上禁止避免与 NULL（IS NULL 和 IS NOT NULL）进行比较。
- 查询条件的索引字段上避免使用 NOT。
- 查询条件的索引字段上避免使用 NOT IN。
- 模糊查询 LIKE 语句，非必要情况下，%不应放在首字符位置。

  如果%放在首字符位置，将无法使用索引，会导致全表扫描。

- WHERE 条件中 IN 的候选子集不易过大，建议不超过 500。

说明

查询时，会对 IN 中每一条数据进行等值比较，开销较大。

如果包含的值为较为固定的值，应考虑创建 REPLICATION 表，并将候选数据写入表中，然后通过 INNER JOIN 来实现包含查询。

- WHERE 条件中 IN 的候选子集不为常量，而是表中的列时，建议改写为子查询。

说明

在这种情况下，实际上是一个不等值的 JOIN，会通过 nestloop 计划执行。在表过大时执行效率低下，建议修改为等值 JOIN 的子查询。

示例

如下代码不推荐使用：

```
SELECT col1, COALESCE(max(col2 - 1), 0)
FROM t1, t2
WHERE t1.col1 = ANY(VALUES(id1), (id2))
GROUP BY col1;
```

建议修改为：

```
SELECT col1, COALESCE(max(tmp), 0) FROM
(
(
SELECT col1, (col2-1) AS tmp
FROM t1, t2
WHERE t1.col1 = t2.id1 AND t1.col1 != t2.id2
) UNION ALL (
SELECT col1, (col2-1) AS tmp
FROM t1, t2
WHERE t1.col1 = t2.id2
)
) GROUP BY col1;
```

- 多使用等值操作，少使用非等值操作。

  WHERE 条件中的非等值条件（IN、BETWEEN、<、<=、>、>=）会导致后面的条件使用不了索引，因为不能同时用到两个范围条件。

3.16.3.4 SELECT

- SELECT 语句中禁用慎用通配符字段“*”。

  使用通配符字段查询表时，如果因业务或数据库升级导致表结构发生变化，可能出现与业务语句不兼容的情况。

  因此业务应指明所需查询的表字段名称，避免使用通配符。

- 带有 LIMIT 的查询语句中必须带有 ORDER BY 保证有序。

说明

云数据库 GaussDB 是一种分布式数据库，表数据将分布在多个 DN 上。

如果 SQL 语句中只带有 LIMIT，而不带有 ORDER BY 子句，数据库将会把网络传输较快的 DN 所发送的（符合查询要求的）结果作为最终结果输出到客户端。

由于网络传输效率不同时刻可能发生改变，因此导致多次执行该 SQL 语句时，返回结果表现出不一致的情况。

- 避免对大字段（如 VARCHAR(2000)）执行 ORDER BY、DISTINCT、GROUP BY、UNION 等会引起排序的操作。

  此类操作将消耗大量的 CPU 和内存资源，执行效率低下。

- 禁止使用慎用 LOCK TABLE 语句加锁，仅允许应考虑使用 SELECT .. FOR UPDATE 语句。

  LOCK TABLE 提供多种锁级别，但如果对数据库原理和业务理解不足，误用表锁可能触发死锁，导致集群不可用。

- 避免在 SELECT 目标列中使用子查询，可能导致计划无法下推到 DN 执行，影响执行性能。

- 考虑使用 UNION ALL，少使用 UNION，注意考虑去重。

  UNION ALL 不去重，少了排序操作，速度相对 UNION 更快。

  如果没有去重的需求，优先使用 UNION ALL。

- 需要统计表中所有记录数时，不要使用 count(col)来替代 count(*)。count(*)会统计 NULL 值（真实行数），而 count(col)不会统计。

- 在执行 count(col)时，将“值为 NULL”的记录行计数为 0。在执行 sum(col)时，当所有记录都为 NULL 时，最终将返回 NULL；当不全为 NULL 时，“值为 NULL”的记录行将被计数为 0。

- count(多个字段)时，多个字段名必须用圆括号括起来。例如，count((col1,col2,col3))。注意：通过多字段统计行数时，即使所选字段都为 NULL，该行也被计数，效果与 count(*)一致。

- count(distinct col)用来计算该列不重复的非 NULL 的数量， NULL 将不被计数。

- count(distinct (col1,col2,...))用来统计多列的唯一值数量，当所有统计字段都为 NULL 时，也会被计数，同时这些记录被认为是相同的。

- 使用连接操作符“ ||”替换 concat 函数进行字符串连接。因为 concat 函数生成的执行计划不能下推，导致查询性能严重劣化。

- 当 in(val1, val2, val3…)表达式中字段较多时，建议使用 in (values(va11), (val2),(val3)…)语句进行替换。优化器会自动把 in 约束转换为非关联子查询，从而提升查询性能。

- 避免频繁使用下使用 count()获取大表行数，该操作资源消耗较大，影响并行作业执行效率。

  如果不需要实时的行数统计信息，可以尝试使用如下语句来获取表行数。

  SELECT reltules **FROM** pg_class **WHERE** relname = 'tablename';

须知

pg_class 中所记录的表行数信息只会在对该表执行 ANALYZE 以后才会更新。

目前 ANALYZE 有两种触发条件:

- 业务主动发送 ANALYZE 语句，例如:

  --分析连接库中所有表

  ANALYZE;

--分析指定表

ANALYZE tablename;

- 助 AUTO VACCUUM 机制，在每间隔一定时间或表的增删达到一定行数时触发。间隔时间和增删比例可通过 GUC 参数设置。

3.16.3.5 INSERT

- INSERT ON DUPLICATE KEY UPDATE 不支持对主键或唯一约束的列上执行 UPDATE。

  INSERT ON DUPLICATE KEY UPDATE 的语义是对唯一约束冲突的行进行更新，这个过程中不应对约束的值进行更新。

- INSERT ON DUPLICATE KEY UPDATE 如果插入多条数据，这些数据之间不允许存在主键/唯一约束冲突。

  与 MySQL 行为存在一定差异，MYSQL 允许此行为，例如如下语句：

  INSERT INTO t1 VALUES(1, 1), (1, 2) **ON DUPLICATE KEY UPDATE** col2 = VALUES(col2);

  不符合 SQL 标准，SQL 标准不允许在同一条 SQL COMMAND 中，对插入行同时进行修改，使得结果无可预期。例如上例中，假设 col1 为主键，那么此时会在同一个 COMMAND 中，既插入一条主键为 1 的，同时对已插入进行修改。由于事务串行化无法确认哪一个操作先执行，因此可能导致结果不稳定。

  如果插入的数据本身存在冲突，在分布式场景下更新应以哪一条记录为准难以确定。

- 禁止对存在多个唯一约束的表执行 INSERT ON DUPLICATE KEY UPDATE。

说明

表中存在多个唯一约束包括存在多个唯一索引，或既存在主键（PRIMARY KEY），又存在唯一索引(UNQUE INDEX)两种情况。

当存在多个唯一约束时，会默认检查所有的唯一约束条件，只要任何一个约束存在冲突，就会对冲突行进行更新，即可能更新多条记录，与业务预期不相符。业务应给予更加明确的插入更新条件。

- 对于批量插入的情况，建议使用 INSERT INTO TABLE1 VALUES (),(),()，执行效率将高于执行多条 INSERT INTO VALUES()

说明

由于目前无法识别多个值是否属于一个 shard，所以要使用/*+ multinode */ 来允许此语句执行。

无论单条和多条插入，关键字均需是 VALUES。

不支持 MySQL 的 INSERT INTO mytable VALUE()的用法。

3.16.3.6 UPDATE

- 不支持 UPDATE 语句中直接使用 LIMIT，应使用 WHERE 条件明确需要更新的目标行。
- 在 GTM-FREE 模式下，不允许跨节点事务，因此更新 HASH 分布中数据表时 WHERE 条件中必须指定分布列等值过滤条件。
- 不支持多表更新。

  多表更新即在单条 SQL 语句中，对多个表进行更新。

- UPDATE 语句中必须有 WHERE 子句，避免全表扫描。
- 不允许在 UPDATE 子句同时更新多个列时，被更新列同样是更新源。

  同时更新多列，且更新源相同，在不同的数据库下行为不同，为了避免带来兼容性问题，业务层应避免上述操作。

  示例：

  UPDATE table **SET** col1 = col2, col3 = col1 **WHERE** col1 = 1;

  该语句在云数据库 GaussDB 中，col3 的值为原 col1 的值；而 MySQL 中，col3 的值为 col2 的值（因为 col2 的值被赋予给了 col1）。
- UPDATE 语句中禁止使用 ORDER BY、GROUP BY 子句，避免不必要的排序。
- 有主键/索引的表，更新时 WHERE 条件应结合主键/索引。

3.16.3.7 DELETE

- 不支持 DELETE 语句中使用 LIMIT。应使用 WHERE 条件明确需要更新的目标行。
- 在 GMT-FREE 模式下，不允许跨节点事务，因此删除 HASH 分布表中数据时，必须在 WHERE 条件中指定分布列等值过滤条件。
- 不支持多表删除。

  多表删除即在单条 SQL 语句中，对多个表进行删除。
- DELETE 语句中必须有 WHERE 子句，避免全表扫描。
- DELETE 语句中禁止不应使用 ORDER BY、GROUP BY 子句，避免不必要的排序。
- 如果需要清空一张表，建议使用 TRUNCATE，而不是 DELETE。

  TRUNCATE 会创建新的物理文件，并在事务结束时将原文件物理删除，清空磁盘空间。而 DELETE 会将表中数据进行标记，直到 VACCUUM FULL 阶段才会真正清理磁盘空间。
- DELETE 有主键或索引的表，WHERE 条件应结合主键或索引，提高执行效率。

3.16.3.8 关联查询

- 多表关联嵌套深度必须小于 8。

  关联嵌套过深，容易产生慢 sql，应从业务层考虑优化。
- 表关联查询必须明确指定各表的连接条件（ON），以避免产生笛卡尔积。

  例如在 MySQL 中，JOIN 与 CROSS JOIN 和 INNER JOIN 等价，但是在 SQL 标准中，JOIN 仅与 INNER JOIN 等价，必须配合使用 ON 连接条件。
- 关联时，应该根据 SQL 标准指明连接方式，避免直接使用 JOIN 关键词，而是使用 CROSS JOIN, INNER JOIN, LEFT JOIN, RIGHT JOIN 等。
- 多表关联查询时，必须应对表添加使用别名，保证语句逻辑清晰，便于维护。
- 不同字段的比较开销不同，关联字段应尽量使用比较效率高的字段类型。

  数值类型的比较效率远高于字符串类型。

  在数值类型中，整型效率高于 NUMERIC 和浮点类型。
- 关联字段应为相同数据类型，避免存在隐式类型转换影响执行效率。

- 少用嵌套子查询，尽量使用表关联，因为子查询会产生临时表，对 SQL 性能影响较大。
- 对于关联列上存在大量 NULL 值的情况，建议在 WHERE 条件中增加关联列 IS NOT NULL 的过滤条件，能够提升执行效率。

3.16.3.9 子查询

- 禁止一条 SQL 语句中，出现重复子查询语句。
- 少用标量子查询。

  标量子查询指结果为 1 个值，并且条件表达式为等值的子查询。

  示例：不符合规范的语句

  SELECT * FROM t1 WHERE id = (SELECT id FROM t2 LIMIT 1));

  上述语句建议业务拆分为两条 SQL 语句，先执行子查询。
- 避免在 SELECT 目标列中使用子查询，可能导致计划无法下推影响执行性能。
- 子查询嵌套深度建议不超过 2 层。

  由于子查询会带来临时表开销，过于复杂的查询应考虑从业务逻辑上进行优化。

3.16.3.10 事务

- 在 GTM-FREE 模式下，不允许执行跨节点事务。

  在 GTM-FREE 模式下，如果所执行的 SQL 语句包含跨节点事务，会报错处理。

  a. 如果语句拆分多条会报错：

  ```
  INSERT/UPDATE/DELETE/MERGE contains multiple remote queries under GTM-free modeUnsupport DML two phase commit under gtm free mode. modify your SQL to generate light-proxy or fast-query-shipping plan。
  ```

  此时需要修改语句，来单节点执行。

  b. 如果语句涉及多节点会报错：

  ```
  Your SQL needs more than one datanode to be involved in.
  ```

  建议对语句进行修改，使得能够单节点执行。如果需要此种语句多节点执行，需要添加一个 hint 来允许，例如：**insert /*+ multinode */ into t values(3,3),(1,1);**

  建议开发阶段在 jdbc 连接串内设置 application_type=perfect_sharding_type，这样所有跨节点读写操作的 SQL 都会报错，用来提示开发人员尽早优化语句。
- 大对象操作不支持事务。

  说明

  大对象操作包括：创建删除 DATABASE, ANAYLIZE, VACCUM。
- 通过 JDBC 接入数据库时，避免拼接多条 SQL 为一条语句发送执行。

  当多条语句拼接为一条语句，且其中包含对象操作时，如果中间对象操作失败，会重新开启新事务执行后续语句。

  示例：不符合规则语句

  ```
  Connection conn = ....
  try {
      Statement stmt = null;
      try {
  ```

```
        stmt = conn.createStatement();
        stmt.executeUpdate("CREATE TABLE t1 (a int); DROP TABLE t1");
    } finally {
        stmt.close();
    }
    conn.commit();
} catch(Exception e) {
   conn.rollback();
} finally {
   conn.close();
}
```

上述执行语句，如果**“CREATE TABLE t1;”**失败，会重新开启新事务执行**“DROP TABLE t1;”**导致执行失败。应拆分成两条语句分别发送:

```
Connection conn = ....
try {
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate("CREATE TABLE t1 (a int)");
        stmt.executeUpdate("DROP TABLE t1");
    } finally {
        stmt.close();
    }
    conn.commit();
} catch(Exception e) {
   conn.rollback();
} finally {
   conn.close();
}
```

3.16.4 客户端编程规范

3.16.4.1 JDBC

- JDBC 实例必须指定数据库，一旦实例创建，无法切换数据库。
- 单条 SQL 语句的长度不允许超过 2G 字节，业务应考虑通信成本，建议单条 SQL 语句不超过 5K。
- 不支持对 DDL 使用 Prepare Execute 执行方式。
- fetchsize 必须要在 autocommit 关闭情况下使用，否则 fetchsize 配置无效。
- 使用默认 GUC 参数，避免通过 JDBC 发送 SET 请求修改 GUC 参数。

说明

更多说明请参考 3.16.3.1 GUC 参数编程规范。

- 必须推荐使用 Prepare Execute 方式执行查询语句，提高执行效率。
- JDBC 客户端所在主机时区、数据库集群所在主机时区和集群配置过程中的时区，三者应保持一致。
- 如果在连接中创建了临时表，那么在将连接归还给连接池之前，必须将临时表删除，避免业务出错。
- 合理设置 prepareThreshold，如果 query 语句十分固定，建议设置为 1。

- 建议设置连接参数 autobalance=true，开启 CN 负载均衡功能，并中设置多个 CN 连接地址（使用逗号分隔）。

  一旦开启 autobalance，JDBC DRIVER 会尝试将 JDBC connection 分配到不同的 CN 节点上。

  设置多个 CN 连接地址的目的是避免 JDBC DRIVER 在首次获取集群 CN 列表时，因所设置的 CN 节点故障而失败。

  一旦首次成功获取，便不会再依赖连接参数中指定的 CN 列表，而是根据实时获取的集群 CN 列表，每隔一段时间，选取连接其中有效的 CN 获取最新的 CN 列表。

- 应根据业务上层请求超时时间合理设置 JDBC 连接超时时间，避免作业完成或常超作业持续占用数据库资源

说明

超时参数包括 loginTimeout、connectTimeout、socketTimeout 等。

- loginTimeout：Integer 类型。指建立数据库连接的等待时间。超时时间单位为秒。
- connectTimeout：Integer 类型。用于连接 CN 操作的超时值。如果连接到 CN 花费的时间超过此值，则连接断开。超时时间单位为秒，默认值为 0，表示已禁用，timeout 不发生。
- socketTimeout：Integer 类型。用于 socket 读取操作的超时值。如果从 CN 读取所花费的时间超过此值，则连接关闭。超时时间单位为秒，默认值为 0，表示已禁用，timeout 不发生。
- cancelSignalTimeout：Integer 类型。发送取消消息本身可能会阻塞，此属性控制用于取消命令的“connect 超时”和“socket 超时”。超时时间单位为秒，默认值为 10 秒。
- tcpKeepAlive：Boolean 类型。启用或禁用 TCP 保活探测功能。默认为 false。

以上参数可以在 JDBC 连接串或者 property 连接属性中配置，例如：

1. 在连接串中配置：

```
jdbc:postgresql://host:port/postgres?tcpKeepAlive=true
```

2. 在 property 中配置：

```
Properties info = new Properties();
Info.setProperty("tcpKeepAlive", true);
```

3.16.5 参数配置规范

3.16.5.1 云数据库 GaussDB 参数配置标准

数据库提供了许多运行参数，配置这些参数可以影响数据库系统的行为。在修改这些参数时请确保用户理解了这些参数对数据库的影响，否则可能会导致无法预料的结果。

4 常见问题

4.1 云数据库 GaussDB 是否支持磁盘缩容

云数据库 GaussDB 暂时不支持磁盘缩容。

4.2 云数据库 GaussDB 是否支持 SSL 连接？

支持，具体请参见“2.4.2 通过内网连接实例”处的 SSL 连接方式。

4.3 云数据库 GaussDB 如何赋予用户 SUPER 权限？

云数据库 GaussDB 不能赋予用户 SUPER 权限。

如果无法导入存储过程，是因为存储过程语句中有部分需要 super 权限的语句，去掉这些语句后，即可正常导入存储过程。

4.4 云数据库 GaussDB 冷备份和热备份都支持吗？

云数据库 GaussDB 仅支持热备份。

4.5 将根证书导入 Windows/Linux 操作系统

导入 Windows 操作系统

1. 单击“开始”，运行框输入“MMC”，回车。
2. 在 MMC 控制台菜单栏中单击“文件”，选择“添加/删除管理单元”。
3. 在“添加或删除管理单元”对话框，选择“可用管理单元”区域的“证书”。单击“添加”添加证书。
4. 在“证书管理”对话框，选择“计算机账户”，单击“下一步”。

5. 在“选择计算机”对话框，单击“完成”。
6. 在“添加或删除管理单元”对话框，单击“确定”。
7. 在 MMC 控制台，双击“证书”。
8. 右键单击“受信任的根证书颁发机构”，选择“所有任务”，单击“导入”。
9. 单击“下一步”。
10. 单击“浏览”，将文件类型更改为“所有文件 (*.*)”。
11. 找到下载的根证书 ca.pem 文件，单击“打开”，然后在向导中单击“下一步”。

须知

您必须在浏览窗口中将文件类型更改为“所有文件 (*.*)”才能执行此操作，因为“.pem”不是标准证书扩展名。

12. 单击“下一步”。
13. 单击“完成”。
14. 单击“确定”，完成根证书导入。

导入 Linux 操作系统

您可以使用任何终端连接工具（如 WinSCP、PuTTY 等工具）将证书上传至 Linux 系统任一目录下。

4.6 数据库实例被锁怎么处理？

步骤 1 在“实例管理”页面，选择指定的实例，单击实例名称，进入实例基本信息页面。

步骤 2 在左侧导航栏单击“参数修改”，进入参数修改页面。

步骤 3 修改 password_lock_time=0、failed_login_attempts=0，解除数据库锁定。

为了数据库安全，密码重置完成后，请将参数修改为默认值。

步骤 4 在“基本信息”页签，在“数据库信息”模块的“管理员帐户名”处，单击“重置密码”。

步骤 5 在“重置密码”弹框，输入新密码及确认密码。

步骤 6 修改完成后 password_lock_time 和 failed_login_attempts 修改为默认值。

----结束

4.7 当业务压力过大时，备机的回放速度跟不上主机的速度如何处理？

问题描述

当业务压力过大时，备机的回放速度跟不上主机的速度。在系统长时间的运行后，备机上会出现日志累积。当主机故障后，数据恢复需要很长时间，数据库不可用，严重影响系统可用性。

解决方案

云数据库 GaussDB 提供极致 RTO 能力，开启极致 RTO（Recovery Time Object，恢复时间目标），可以减少主机故障后数据的恢复时间，提高了可用性。

如需使用极致 RTO 能力，您可以在管理控制台右上角，选择，提交开通申请。

注意事项

- 极致 RTO 只关注同步备机的 RTO 是否满足需求。
- 开启极致 RTO 会消耗备机更多 CPU 和内存。
- 1.4 及之前版本开启极致 RTO 会有流控效果。
- 极致 RTO 不支持备机读。如果查询备机可能导致备机无法提供服务。

4.8 资源冻结/解冻/释放/删除/退订

按需计费实例不用时也会计费吗？

按需计费实例是从“创建成功”开启计费，到“删除”结束计费。即使中间不使用实例，但实例也仍会占用资源，所以仍然会按实际购买时长计费。

资源为什么被释放了？

客户购买产品后，如果没有及时的进行续费或充值，将进入宽限期。如宽限期满仍未续费或充值，将进入保留期。在保留期内资源将停止服务。保留期满仍未续费或充值，存储在云服务中的数据将被删除、云服务资源将被释放。

资源为什么被冻结了？

资源冻结的类型有多种，最常见类型为欠费冻结。

实例被冻结了，还可以备份数据吗？

不支持，如果是欠费冻结，需要您先续费解冻实例后才能备份数据。

怎样将资源解冻？

欠费冻结：用户可通过续费或充值来解冻资源，恢复实例正常使用。欠费冻结的实例允许续费、释放或删除；已经到期的包周期实例不能发起退订，未到期的包周期实例可以退订。

冻结、解冻、释放资源时对业务的影响

- 资源冻结时：
  - 资源将被限制访问和使用，会导致您的业务中断。例如实例被冻结时，会使得用户无法再连接至数据库。
  - 包周期资源被冻结后，将被限制进行变更操作。
  - 资源被冻结后，可以手动进行退订/删除。
- 资源解冻时：资源将被解除限制，用户可以连接至数据库。
- 资源释放时：资源将被释放，实例将被删除，删除前将依据用户策略决定是否执行 3.15 回收站。

怎样续费？

资源被释放了能否恢复？/退订错了可以找回吗？

退订资源前请一定要仔细确认资源信息。如果退订错了建议重新购买使用。

怎样删除实例？

- 按需实例，请参见 3.4.3 删除实例。
- 包周期实例，请参见 3.13.4 退订包周期实例。