GBase 8c

数据库管理指南

www.gbase.cn

GBase 8c 数据库管理指南，天津南大通用数据技术股份有限公司

GBase 版权所有©2024，保留所有权利

版权声明

本文档所涉及的软件著作权及其他知识产权已依法进行了相关注册、登记，由南大通用数据技术股份有限公司合法拥有，受《中华人民共和国著作权法》、《计算机软件保护条例》、《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的南大通用数据技术股份有限公司的版权信息由南大通用数据技术股份有限公司合法拥有，受法律的保护，南大通用数据技术股份有限公司对本文档可能涉及到的非南大通用数据技术股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅，并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经南大通用数据技术股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将视为侵权，南大通用数据技术股份有限公司具有依法追究其责任的权利。

本文档中包含的信息如有更新，恕不另行通知。您对本文档的任何问题，可直接向南大通用数据技术股份有限公司告知或查询。

未经本公司明确授予的任何权利均予保留。

通讯方式

天津南大通用数据技术股份有限公司

天津市高新区华苑产业园区工华道 2 号天百中心 3 层(300384)

电话：400-013-9696　　邮箱：info@gbase.cn

商标声明

GBASE® 是南大通用数据技术股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由南大通用数据技术股份有限公司合法拥有，受法律保护。未经南大通用数据技术股份有限公司书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯南大通用数据技术股份有限公司商标权的，南大通用数据技术股份有限公司将依法追究其法律责任。

目　录

1 文档简介

1.1 概述

本文档简介 GBase 8c 数据库相关概念、部署方案，以及常用的管理功能。

1.2 读者对象

本文档是为基于 GBase 8c 进行 C/Java 应用程序开发的程序员而写的，提供了必要的参考信息。作为应用程序开发人员，至少需要了解以下知识：

- 操作系统知识。这是一切的基础。
- C/Java 语言。这是做应用程序开发的基础。
- 熟悉 C/Java 的一种 IDE。这是高效完成开发任务的必备条件。
- SQL 语法。这是操作数据库的必备能力。

1.3 文档阅读约定

通用格式约定

表 1-1 通用格式约定

格式	说明
幼圆	正文采用幼圆表示。
Times New Roman	正文中的英文字母、数字采用 Times New Roman 表示。
粗体	表格编号及标题、图片编号及标题、多级标题、正文小节标题、注意说明等，采用幼圆/Times New Roman 粗体。
	引用 GBase 8c 安装及使用过程中部分显示内容
……	表示省略部分显示内容

命令行格式约定

表 1-2 命令行格式约定

格式	说明
幼圆	命令行语句采用幼圆表示。
粗体	执行 bash 命令或 SQL 语句，采用幼圆加粗表示。
大写英文（如 CREATE、SELECT）	表示 GBase 8c 数据库中关键字
斜体	表示 GBase 8c 数据库语法中，需要用户自定义的变量
……	表示省略部分显示内容
[]	用"[]”括起来的部分，表示在语法命令中为必选参数
<>	用"<>”括起来的部分，表示在语法命令中为可选参数
# 描述	由"#”开头的注释行
#bash 命令	表示以 root 超级用户执行此命令
$bash 命令	表示以普通用户执行此命令

2 数据库基础概念介绍

2.1 相关概念

数据库

数据库用于管理各类数据对象，与其他数据库隔离。创建数据对象时可以指定对应的表空间，如果不指定相应的表空间，相关的对象会默认保存在 PG_DEFAULT 空间中。数据库管理的对象可分布在多个表空间上。

表空间

在 GBase 8c 中，表空间是一个目录，可以存在多个，里面存储的是它所包含的数据库的各种物理文件。由于表空间是一个目录，仅是起到了物理隔离的作用，其管理功能依赖于文件系统。

模式

GBase 8c 的模式是对数据库做一个逻辑分割。所有的数据库对象都建立在模式下面。GBase 8c 的模式和用户是弱绑定的，所谓的弱绑定是指虽然创建用户的同时会自动创建一个同名模式，但用户也可以单独创建模式，并且为用户指定其他的模式。

用户和角色

GBase 8c 使用用户和角色来控制对数据库的访问。根据角色自身的设置不同，一个角色可以看做是一个数据库用户，或者一组数据库用户。在 GBase 8c 中角色和用户之间的区别只在于角色默认是没有 LOGIN 权限的。在 GBase 8c 中一个用户唯一对应一个角色，不过可以使用角色叠加来更灵活地进行管理。

事务管理

在事务管理上，GBase 8c 采取了 MVCC（多版本并发控制）结合两阶段锁的方式，其特点是读写之间不阻塞。GBase 8c 没有将历史版本数据统一存放，而是和当前元组的版本放在了一起。GBase 8c 没有回滚段的概念，但是为了定期清除历史版本数据引入了一个 VACUUM 线程。一般情况下用户不用关注它，除非要做性能调优。此外，GBase 8c 是自动提交事务。

3 数据库基础管理

3.1 数据存储管理

GBase 8c 数据库的 DN（Datanode）节点负责存储数据，其存储介质也是磁盘。本节主要从逻辑视角介绍数据库节点都有哪些对象，以及这些对象之间的关系。数据库逻辑结构如下图所示。

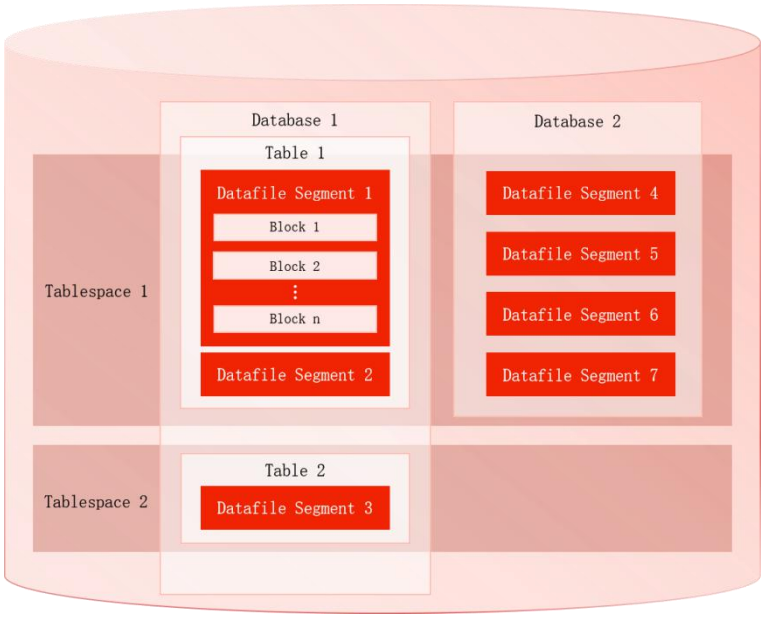


图 3-1 数据库逻辑结构图

说明

- Tablespace，即表空间，是一个目录，可以存在多个，里面存储的是它所包含的数据库的各种物理文件。每个表空间可以对应多个 Database。
- Database，即数据库，用于管理各类数据对象，各数据库间相互隔离。数据库管理的对象可分布在多个 Tablespace 上。
- Datafile Segment，即数据文件，通常每张表只对应一个数据文件。如果某张表的数据大于 1GB，则会分为多个数据文件存储。
- Table，即表，每张表只能属于一个数据库，也只能对应到一个 Tablespace。每张表对应的数据文件必须在同一个 Tablespace 中。
- Block，即数据块，是数据库管理的基本单位，默认大小为 8KB。

4 数据库部署方案

4.1 常见主备部署方案简介

本节介绍一些典型的部署方案，用户可以依据自身实际业务场景，对以下部署方案进行调整，比如增减备机数量、调整中心数量、适当安置同步备和异步备、适当使用级联备机等。

单数据中心

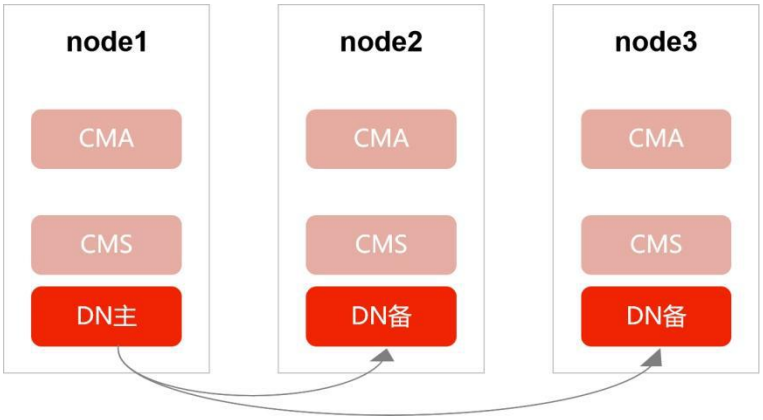


图 4-1 单中心部署图

组网特点：单数据中心部署，建议可以配置一个同步备、一个异步备。

优势：

- 三个节点完全等价，任一节点故障仍然可以提供服务；
- 成本低；

劣势：高可用能力较低，若发生机房级、城市级故障只能依赖节点恢复，且不具备异地容灾能力。

适用性：适用于对高可用性要求较低的业务系统。

同城两中心

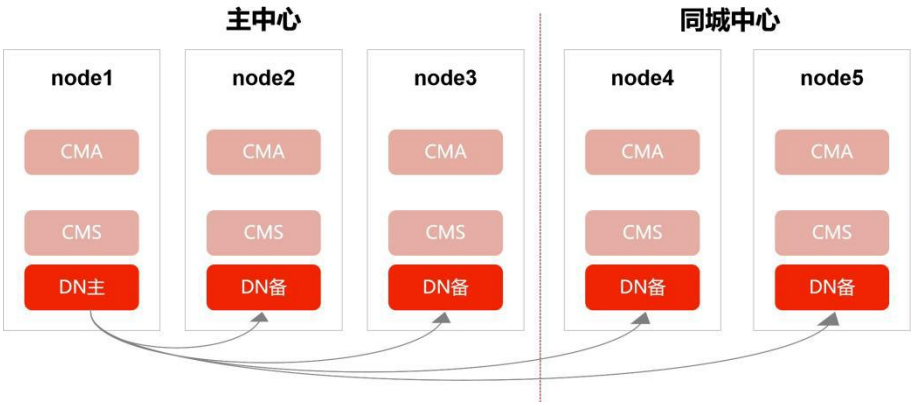


图 4-2 同城两中心部署图

组网特点：同城两中心相比单中心可靠性更强，两中心可以分别配置一个同步备机。

优势：

- 同城同步复制，任意一个中心故障，另一个中心还可以提供服务，数据不丢失，RPO=0；
- 成本适中；

劣势：

- 同城距离不宜太远，一般建议 70km 以内，业务设计要考虑读写次数过多导致的总延时；
- 不能抵抗城市级故障，不具备异地容灾能力；

适用性：适用于普通的业务系统。

两地三中心

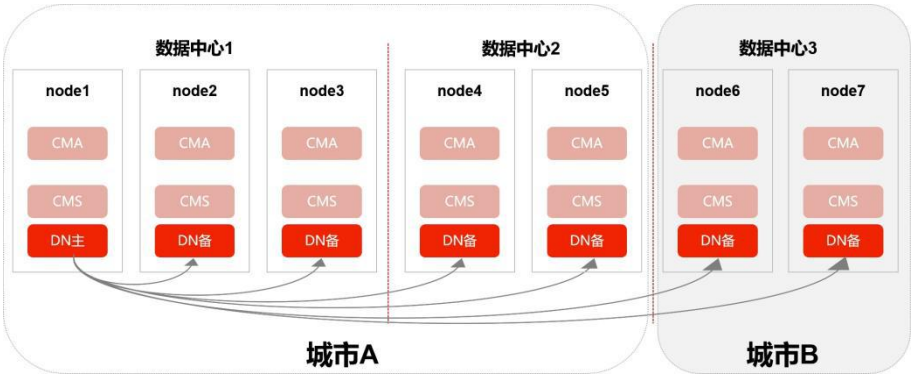


图 4-3 两地三中心部署图

组网特点：两地三中心，每个数据中心都保证至少有一个同步备，同时随着地点和中心

数的增加，集群的可靠性能够达到最高。

优势：具备异地容灾能力，并且能够保证异地容灾数据不丢失，RPO=0，可靠性最强。

劣势：

- 异地距离较远，若在异地中心配置了同步备，可能会影响性能；
- 成本较高；

适用性：适用于核心重要业务系统。

两地三中心流式容灾方案

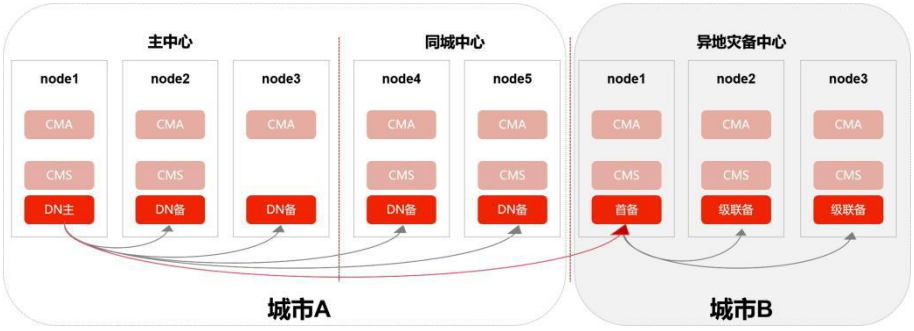


图 4-4 两地三中心流式容灾方案部署图

组网特点：双集群容灾方案，两个独立集群，主备集群组网方式可任意选择，备集群会选出首备连接主集群的主 DN，灾备集群内都以级联备方式连接首备。详见两地三中心跨 Region 容灾。

优势：

- 主集群具备单集群组网的优点，只有主集群彻底不可用后才需要手动切换为备集群；
- 跨集群(异地)复制链路无论是否发生容灾切换都只有一条，占用网络带宽相对较少；
- 组网更加灵活，主集群和灾备集群都可以选择不同的组网；

劣势：

- 需要增加灾备集群，相应增加成本；
- 异地灾备 RPO>0；

适用性：适用于核心重要业务系统。

4.2 两地三中心跨 Region 容灾

4.2.1概述

要实现跨 Region 容灾，需要部署两套数据库实例，一套主数据库实例，一套灾备数据库实例。主数据库实例和灾备数据库实例一般部署在相距较远的两个不同城市。数据库实例之间借助存储介质或者不借助存储介质，实现数据的全量和增量同步。当主数据库实例（即生产数据库实例）出现地域性故障，数据完全无法恢复时。可考虑启用将灾备数据库实例升主，以接管业务。

GBase 8c 提供基于流式复制的异地容灾解决方案。

4.2.2规格与约束

本节就该解决方案的特性规格与约束进行详细描述，管理人员需重点关注。

特性规格

- 主数据库实例或灾备数据库实例内网络时延要求<=10 毫秒，主备数据库实例之间异地网络时延要求<=100 毫秒。该时延范围内可保证容灾的正常运行，否则会导致主备数据库实例断链等情况出现。
- 在网络带宽非瓶颈、灾备数据库实例打开并行回放的前提下，不同硬件规格可支持主数据库实例日志产生速度如下表所示。在该日志产生速度下可以保证 RPO、RTO，否则无法保证。

表 4-1 典型配置下日志产生速率

典型配置	支持主数据库实例日志产生速率
96U/768G/SATA SSD	<=10MB/s
128U/2T/NVMe SSD	<=40MB/s

- 如果磁盘混合部署，应采用低配部分的规格（比如数据库实例内有 NVMe 和 SATA 盘，请参考 SATA 盘配置的规格）。
- 灾备数据库实例升主：
  - 灾备数据库实例升主允许丢失一定的数据，RPO<=10 秒 ；

- 灾备数据库实例处于 normal 态，灾备升主 RTO<=10 分钟，数据库实例处于 degraded 状态等叠加故障场景下，执行灾备数据库实例升主 RTO 一般在 20 分钟以内。
- 灾备数据库实例为独立数据库实例，无法感知主数据库实例的状态，因此不支持自动升主。第三方应用可通过接口调用触发灾备数据库实例升主，相关操作参见灾备数据库实例升主 failover。

- 演练特性：计划内主备数据库实例倒换，无数据丢失 RPO=0，RTO<=20 分钟(包含主数据库实例降为灾备实例，灾备数据库实例升主两个流程)。

  须知

  - 经过测试，SATA SSD 极限写入速率在 240MB/s 左右，SAS SSD 可以达到 500MB/s 以上的写入速度，NVMe SSD 表现则更为优异。如果硬件条件达不到如上标准，则可支持的主数据库实例单分片日志产生速度应下调，才可保证 RPO、RTO。
  - 主备数据库实例出现文件句柄，内存等资源耗尽时，无法保证 RPO，RTO。

  特性约束

- 搭建容灾关系前，主集群需创建具有流复制权限的容灾用户，用于容灾鉴权，主备集群必须使用相同的容灾用户名和密码，一次容灾搭建后，该用户密码不可修改。若需修改容灾用户名与密码，需要解除容灾，使用新的容灾用户重新进行搭建。容灾用户密码中不可包含以下字符"| ;&$<>`'"{}()[]~*?!"。
- 搭建容灾的主备集群版本号必须相同。
- 流式容灾搭建前不能存在首备及级联备。
- 搭建容灾关系时，如果集群副本数<=2，会设置 most_available_sync 为 on，在容灾解除或者 failover 后此参数不会恢复初始值，持续保证集群为最大可用模式。
- 搭建容灾关系时，会设置 synchronous_commit 为 on，解除容灾或 failover 升主时恢复初始值。
- 灾备集群可读不可写。
- 灾备集群通过 failover 命令升主后，和原主集群灾备关系将失效，需要重新搭建容灾关系。
- 在主数据库实例和灾备数据库实例处于 normal 状态时可进行容灾搭建；在主数据库实例处于 normal 态并且灾备数据库实例已经升主的情况下，主数据库实例可执行容灾解除，其他数据库实例状态不支持。在主数据库实例和灾备数据库实例处于 normal 状态

时，通过计划内 switchover 命令，主数据库实例可切换为灾备数据库实例，灾备数据库实例可切换为主数据库实例。灾备数据库实例处于非 Normal 且非 Degraded 状态时，无法升主，无法作为灾备数据库实例继续提供容灾服务，需要手动修复或重建灾备数据库实例。

- 灾备集群 DN 多数派故障或者 CMS、DN 全故障，无法启动容灾，灾备集群无法升主，无法作为灾备集群，需要重建灾备集群。
- 主集群如果进行了强切操作，需要重建灾备集群。
- 主集群和灾备集群都支持 gs_probackup 工具中的全备和增备。容灾状态下，主集群和灾备集群都不能做恢复。如果主数据库实例要做恢复，需要先解除容灾关系，在完成备份恢复后重新搭建容灾关系。
- 容灾关系搭建之后，不支持 DN 实例端口修改。
- 建立容灾关系的主数据库实例与灾备数据库实例之间不支持 GUC 参数的同步。
- 主备集群不支持节点替换、修复、升降副本，DCF 模式。
- 当灾备数据库实例为 2 副本时，灾备数据库实例在 1 个副本损坏时，仍可以升主对外提供服务，如果剩余的这个副本也损坏，将导致不可避免的数据丢失。
- 容灾状态下仅支持灰度升级，且继承原升级约束，容灾状态下升级需要遵循先升级主集群，再升级备集群，再提交备集群，再提交主集群的顺序。
- 建议对于流式容灾流复制 IP 的选择，应考虑尽量使集群内的网络平面与跨集群网络平面分离，便于压力分流并提高安全性。

4.2.3影响容灾性能指标的 GUC 参数设置

检查点相关参数设置的影响

以上描述的容灾性能指标均是在检查点相关参数设置为默认值情况下测得。

检查点相关参数详见《GBase 8c V5_5.0.0_数据库参考手册》中"GUC 参数说明 > 预写式日志 > 检查点”章节。

其中 enable_incremental_checkpoint 参数为 on 时，设置自动 WAL 检查点之间的最长时间将由 incremental_checkpoint_timeout 参数决定。如果调大数值，可能导致实例重启时会有大量日志需要回放，进而影响到容灾指标 RTO 变大，无法达到特性规格。

极致 RTO 相关参数设置的影响

极致 RTO 相关参数描述参见《GBase 8c V5_5.0.0_数据库参考手册》中"GUC 参数说明 > 预写式日志 > 日志回放"章节的 recovery_parse_workers 和 recovery_redo_workers 参数描述。

如果要开启极致 RTO，应至少满足每台机器上的逻辑 CPU 数大于打开极致 RTO 后额外启动的线程数（计算公式= (recovery_parse_workers * (recovery_redo_workers + 2) + 5) * 每台机器上的 DN 实例数），否则可能出现线程对 CPU 资源争抢的情况，导致容灾流程中部分操作耗时变长，无法达到容灾特性规格。

4.2.4基本操作

4.2.4.1 容灾搭建

容灾搭建前主数据库实例业务负载评估数据量

主数据库实例存储数据量，直接影响容灾搭建需要传输的数据量。该值结合异地网络带宽，直接影响容灾搭建时长，可在搭建容灾接口的 time-out 参数设置超时时间，当前默认值为 20min。超时时间的评估与主数据库实例搭容灾前的数据量与异地可使用带宽相关，计算公式为（数据量/传输速率 = 耗时）。

例如：主数据库实例有 100TB 数据，异地数据库实例间可用带宽为 512Mbps（传输速率为 64MB/s），搭建容灾传递这些数据需要时间为 1638400s（100*1024*1024/64，大约 19 天）。

日志产生速率

该值影响容灾搭建过程中主数据库实例需要保留在主数据库实例本地的日志量，灾备数据库实例在完成全量数据恢复后将与主数据库实例建立流式复制关系，如果主数据库实例未对日志进行保留，将可能导致流式复制关系建立失败。

例如：经过计算搭建过程要持续 2 天，那么这 2 天内的日志需要在搭建完成前保留在主数据库实例本地磁盘。

如果主数据库实例日志产生速率大于异地传输带宽；或者在带宽充足的情况下，主数据库实例日志产生速率大于灾备数据库实例的日志回放速率，即超规格场景搭建容灾后将导致 RPO/RTO 无法保持在特性规格水平。

容灾搭建调用接口

容灾搭建时需要对主备数据库实例发送搭建请求，参考《GBase 8c V5_5.0.0_工具与命令参考手册》中 gs_sdr 工具。

须知

- 容灾搭建时需要在主数据库实例和灾备数据库实例使用相同容灾用户名和密码用于数据库实例间鉴权，该用户的权限为 Replication（Replication 属性是特定的角色，仅用于复制）。
- 搭建容灾前需要在主集群创建容灾用户。
- 一次容灾搭建后，该用户密码不可修改，伴随整个容灾生命周期。若需修改容灾用户名与密码，需要解除容灾，使用新的容灾用户重新进行搭建。
- 容灾搭建过程可在"time_out”设置超时时间，当前默认值为 20min。超时时间的评估与主数据库实例搭容灾前的数据量与异地可使用带宽相关，计算公式为"数据量/传输速率 = 耗时”。 例如：主数据库实例有 100TB 数据，异地数据库实例间可用带宽为 512Mbps（传输速率为 64MB/s），搭建容灾传递这些数据需要时间为 1638400s（100*1024*1024/64，大约 19 天）。

4.2.4.2 灾备数据库实例升主 failover

对于向灾备数据库实例发送灾备数据库实例升主的请求，请参考《GBase 8c V5_5.0.0_工具与命令参考手册》中 gs_sdr 工具。

须知

- 灾备数据库实例升主后会进行容灾信息清除。
- 如果主数据库实例处于正常状态，正在处理业务，灾备数据库实例因要主动解除容灾可以执行该命令。在该命令下发后，灾备数据库实例将不会再接收主机的日志，会导致容灾指标 RPO 值持续增长，直到主备数据库实例中断联系，RPO 值为空。RPO 值查询参见查询主备数据库实例容灾状态。

4.2.4.3 主数据库实例容灾信息清除

对于向主数据库实例发送容灾信息清除的请求，参考《GBase 8c V5_5.0.0_工具与命令参考手册》中 gs_sdr 工具。

须知

- 该操作会对主数据库实例进行容灾信息清除。
- 该操作只能在灾备数据库实例升主后，对主数据库实例进行操作。在灾备数据库实例未升主条件下执行，将会导致容灾关系被破坏。

4.2.4.4 计划内倒换 switchover

向主备数据库实例发送计划内 switchover 的请求，参考《GBase 8c V5_5.0.0_工具与命令参考手册》中 gs_sdr 工具。

4.2.4.5 查询主备数据库实例容灾状态

向主备数据库实例发送容灾状态查询的请求，参考《GBase 8c V5_5.0.0_工具与命令参考手册》中 gs_sdr 工具。

4.2.4.6 容灾状态下主备数据库实例升级

(1) 主数据库实例先升级，主数据库实例升级完成后，备数据库实例升级。

(2) 升级完成后，灾备数据库实例先提交，主数据库实例后提交。

须知

- 备数据库实例提交前，主数据库实例需要升级完成。
- 备数据库实例已提交情况下，主数据库实例不可回滚。
- 主备数据库实例升级过程中，不可发生主备数据库实例的切换。

4.2.5故障处理

介绍使用基于流式复制的异地容灾解决方案可能遇到的常见问题，并提供故障处理下表列出了不同操作中问题现象、原因、解决方案。

4.2.5.1 容灾搭建异常

表 4-2 容灾搭建错误信息参考

故障描述	原因和解决方案
容灾搭建中主数据库实例执行容灾搭建返回如下错误，执行超时 Result exception error : Failed to do check main standby connection.	原因：在主数据库实例数据量较大，或者异地网络带宽较小时，可能会出现灾备数据库实例未完成数据拷贝，主数据库实例就已经超时退出容灾搭建流程的情况。 解决方案：

故障描述	原因和解决方案
Because Waiting timeout: XXs。	若灾备数据库实例处于搭建过程中或者搭建已完成，可直接重入主数据库实例容灾搭建流程，主数据库实例会重新进入等待灾备连接状态。若能重新设置超时参数，可根据主数据库实例数据量大小与异地网络带宽，重新估算超时时间后再执行重入。 若灾备数据库实例搭建过程也失败了，需要先针对灾备数据库实例进行故障处理，再重入数据库实例容灾搭建流程。
搭建容灾关系过程中，由于主集群内的主 dn 发生切换导致容灾搭建失败。	原因：主集群的主 dn 发生切换，灾备集群连接主集群进行数据 build 时断连导致搭建失败。 解决方案： 确认是否有人为进行主集群内主备切换的操作，如果有则停止该操作，如果没有则忽略。重新下发搭建命令。

4.2.5.2 灾备升主 failover 异常

表 4-3 灾备升主 failover 错误信息参考

故障描述	原因和解决方案

灾备数据库实例有故障节点未参与灾备数据库实例升主。	原因：因服务器宕机，网络中断等原因导致节点脱离灾备数据库实例，没有参与灾备数据库实例升主。
	解决方案：
	故障节点修复后重新加入数据库实例。
	修改 CMS 和 CMA 中关于数据库实例灾备模式的参数，切回主数据库实例配置。
	`gs_guc set -Z cmserver -N all -I all -c "backup_open = 0"`
	`gs_guc set -Z cmagent -N all -I all -c "agent_backup_open=0"`

故障描述	原因和解决方案
	gs_guc set -Z cmagent -N all -I all -c "disaster_recovery_type= 0" 接入故障节点，查询 CMS 和 CMA 的进程 ID，使用 kill -9 命令杀掉进程，然后进程会被 om_monitor 重启，完成 CMS 和 CMA 参数修改的生效。 手动修复改节点后使用 cm_ctl start-n NODEID -D DATADIR。

4.2.5.3 计划内倒换 switchover 异常

表 4-4 计划内 switchover 错误信息参考

故障描述	原因和解决方案
计划内 switchover 中主数据库实例执行命令返回如下错误，提示主数据库实例产生一致性点失败 Result exception error : Failed to generate switchover barrier before switchover	原因：在主数据库实例接收到计划内 switchover 命令，主数据库实例降为灾备数据库实例前会先产生一致性点 switchover barrier，这是执行 switchover 的前提，用于保证主备数据库实例所有 DN 分片的日志停止在一致性点。由于主数据库实例内网络抖动等原因导致主数据库实例内产生 switchover barrier 失败将放弃本次计划内 switchover。 解决方案： 等待灾备数据库实例执行 switchover 灾备升主命令超时退出后，计划内倒换 switchover 可在主数据库实例和灾备数据库实例重入执行。 若多次执行 switchover 均出现日志截断失败，需进一步分析流式容灾相关日志文件。
Result exception error : Failed to do check switchover_barrier on all main standby dn. Because	原因：在灾备数据库实例接收到计划内 switchover 命令，灾备数据库实例升为主数据库实例前会先在首备 DN 上查询是否收到一致性点 switchover barrier，这是执行 swichover

故障描述	原因和解决方案
check timeout: XXs	的前提，用于保证主备数据库实例 DN 的日志停止在一致性点。由于异地网络异常等原因，灾备数据库实例在超时时间内无法获得 switchover barrier 将放弃执行本次计划内 switchover。 解决方案： 等待主数据库实例执行 switchover 主降备命令超时退出后，计划内倒换 switchover 可在主数据库实例和灾备数据库实例重入执行。 若多次执行 switchover 灾备数据库实例均出现 switchover barrier 获取失败，需进一步分析流式容灾相关日志文件。

4.2.5.4 灾备集群数据库实例故障

表 4-5 灾备集群数据库实例错误信息参考故障描述

故障描述	原因和解决方案
备集群节点 CM_AGENT 故障。该节点上 DN 实例状态显示为 Unknown；部分首备显示 Main Standby Need repair(Connecting)。	原因：节点 CM_AGENT 发生故障 该节点上 DN 状态无法上报 CM_SERVER，DN 实例显示为 Unknown。 若该节点上存在首备实例(Main Standby)，则会触发首备切换。由于原首备实例并无异常，并与主数据库实例主 DN 存在正常流复制关系，而主数据库实例该分片主 DN 只允许一个首备的连接，导致新首备无法连接到主集群分片主 DN，实例状态显示为 Main Standby Need repair(Connecting)。 解决方案： 等观察灾备集群的 CM_AGENT 告警信息"ALM_AI_AbnormalCMSProcess”，并尝试修复发生故障的

故障描述	原因和解决方案
	CM_AGENT。故障排除后新首备的连接可恢复。 若如果故障的 CM_AGENT 短时间内无法修复，执行 gs_ctl stop -D DATADIR 命令或者 kill 命令手动停止该节点上的 DN 进程，可恢复。

5 访问数据库

5.1 使用 gsql 访问

gsql 是 GBase 8c 提供的在命令行下运行的数据库连接工具。此工具除了具备操作数据库的基本功能，还提供了若干高级特性，便于用户使用。本节只介绍如何使用 gsql 连接数据库。gsql 详细说明请参考《GBase 8c V5_5.0.0_工具与命令参考手册》中"客户端工具 > gsql”章节。

注意事项

- 缺省情况下，客户端连接数据库后处于空闲状态时会根据参数 session_timeout 的默认值自动断开连接。如果要关闭超时设置，设置参数 session_timeout 为 0 即可。

5.1.1本地连接

(1) 以 gbase 用户登录主节点并连接数据库。安装完成后，默认生成名称为 postgres 的数据库。首次连接时需连接到此数据库。例如：

```
$ gsql -d postgres -p 15400
```

其中 postgres 为需要连接的数据库名称，15400 为数据库主节点端口号。请根据实际情况修改。

连接成功后，系统显示类似如下信息：

```
gsql ((single_node GBase8cV5 S5.0.0BXX build b0e57b26) compiled at 2024-01-14 16:38:29
commit 0 last mr 443 )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

postgres=#
```

gbase 用户是管理员用户，因此系统显示"DBNAME=#”。若以普通用户连接数据库，系统显示"DBNAME=>”。

(2) 首次登录数据库后，建议修改密码以提高安全性。命令格式如下：

```
postgres=# ALTER ROLE user_name IDENTIFIED BY '新密码' REPLACE '旧密码';
```

首次执行时，旧密码为空。

(3) 退出数据库。

```
postgres=# \q
```

5.1.2密态数据库连接

以 gbase 用户登录主节点，并连接数据库。安装完成后，默认生成名称为 postgres 的数据库。首次连接时需连接到此数据库。例如：

```
$ gsql -d postgres -p 15400 -C
```

其中 postgres 为需要连接的数据库名称，15400 为数据库主节点端口号。请根据实际情况修改。-C 表示密态数据库开启，可以创建密钥和加密表。

连接成功后，系统显示类似如下信息：

```
gsql ((single_node GBase8cV5 S5.0.0BXX build b0e57b26) compiled at 2024-01-14 16:38:29
commit 0 last mr 443 )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

postgres=#
```

退出数据库。

```
postgres=# \q
```

5.1.3远程连接

将安装 GBase 8c 数据库的主机定义为服务端，将用于远程连接的主机定义为客户端。远程连接时，需要服务端配置后，用户才可以通过客户端远程连接。

服务端配置

(1) 以 gbase 登录数据库主节点。

(2) 配置 listen_addresses 参数，即指定远程客户端连接使用的数据库主节点 IP 或主机名。

查看数据库主节点目前的 listen_addresses 配置。

```
$ gs_guc check -I all -c "listen_addresses"
```

例如：查询到如下信息：

```
The gs_guc run with the following arguments: [gs_guc -I all -c listen_addresses check ].
expected guc information: gbase8c: listen_addresses=NULL:
[/opt/database/install/data/dn/postgresql.conf]
gs_guc check: plat1: listen_addresses='localhost,192.168.0.1':
[/opt/database/install/data/dn/postgresql.conf]
```

```
Total GUC values: 1. Failed GUC values: 0.
The value of parameter listen_addresses is same on all instances.
     listen_addresses='localhost,192.168.0.1'
```

(3) 把要添加的 IP 追加到 listen_addresses 后面，多个配置项之间用英文逗号分隔。例如，追加 IP 地址 10.11.12.34。

```
$ gs_guc set -I all -c "listen_addresses='localhost,192.168.0.1,10.11.12.34'"
```

(4) 配置 pg_hba.conf，添加数据库主节点 IP 和客户端 IP。详细说明参见配置客户端接入认证章节。

添加数据库主节点 IP 到 pg_hba.conf 配置文件中。

```
$ gs_guc reload [-Z datanode] -N all -I all -h "host all gbase 192.168.0.1/32 trust"
```

其中-Z datanode 参数可选，gbase 为数据库初始用户，192.168.0.1 为数据库主机 IP。

添加客户端 IP 到 pg_hba.conf。假设客户端 ip 为 10.11.12.34，认证方式为 sha256。

```
$ gs_guc reload [-Z datanode] -N all -I all -h "host all all 10.11.12.34/32 sha256"
```

其中-Z datanode 参数可选，10.11.12.34 为客户端主机 IP。

(5) 重启数据库。

```
$ gs_om -t stop && gs_om -t start
```

客户端连接

(1) 完成远程连接配置，操作步骤详见服务端配置。

(2) 在客户端机器，上传客户端工具并配置 gsql 的执行环境变量。以客户端 IP 为 10.11.12.34 为例。

以 root 用户身份创建安装目录，例如：

```
# mkdir -p /tmp/tools
```

(3) 获取软件安装包，并上传至安装路径（以/tmp/tools 为例）。GBase 8c 兼容 openGauss 开源工具，根据实际环境和连接方式获取 openGauss Connectors 安装包，官方链接：https://opengauss.org/zh/download/。

说明

- 软件包安装目录根据实际情况准备。
- 不同的操作系统，工具包文件名称会有差异。请根据实际的操作系统类型选择对应

的工具包。

(4) 解压安装包。

```
# cd /tmp/tools
# tar -zxvf openGauss-5.0.0-openEuler-64bit-Libpq.tar.gz
```

(5) 登录服务端，拷贝数据库运行目录下的 bin 和 lib 目录到客户端主机的安装路径下。例如：

```
# scp -r /opt/database/install/app/bin lib root@10.11.12.34:/tmp/tools
```

其中主机 IP 和路径，请按照实际情况填写。

(6) 登录客户端所在主机，设置环境变量。

```
# vi ~/.bashrc
```

添加如下内容：

```
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
```

(7) 连接数据库。

安装完成后，默认生成名称为 postgres 的数据库。首次连接时需连接到此数据库。例如：

```
# gsql -d postgres -h 192.168.0.1 -p 15400 -U jack -W Test@123
```

参数与本地连接章节描述相同。

说明

- 客户端不能直接通过数据库默认用户 gbase 进行远程连接。可在服务端执行 SQL 语句创建用户（用户所需权限根据实际情况而定）：

```
CREATE USER user_name password "password";
```

- 客户端登录时，需要数据库用户名及登录密码，请联系服务端获取。
- 执行连接数据库步骤可能报错，显示"gsql:error while loading shared libraries: libssl.so.1.1: cannot open shared object file: No such file or directory”，则请检查客户端目录下是否已拷贝 bin、lib 目录，以及环境变量中路径是否与实际一致。

5.2 使用应用程序接口访问

5.2.1ODBC

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，增强了应用程序的可移植性、扩展性和可维护性。

ODBC 的系统结构参见图 5-1。

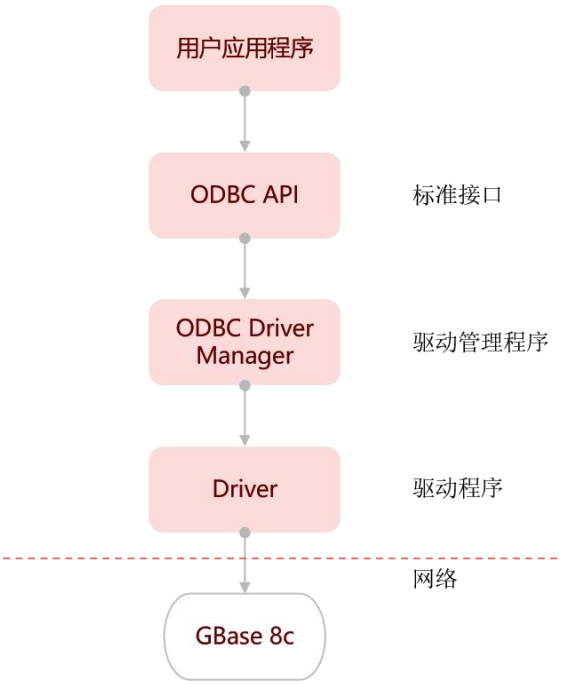


图 5-1ODBC 系统机构

GBase 8c 目前在以下环境中提供对 ODBC 的支持。

表 5-1 ODBC 支持平台

操作系统	平台
CentOS 6.4/6.5/6.6/6.7/6.8/6.9/7.0/7.1/7.2/7.3/7.4	x86_64 位
CentOS 7.6	ARM64 位
EulerOS 2.0 SP2/SP3	x86_64 位

EulerOS 2.0 SP8	ARM64 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-2.3.7 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。

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

5.2.1.1 ODBC 包及依赖的库和头文件

Linux 下的 ODBC 包

获取发布包。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。

5.2.1.2 Linux 下配置数据源

将 GBase 8c 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。

操作步骤

(1) 获取 unixODBC 源码包。

获取参考地址：https://sourceforge.net/projects/unixodbc/files/unixODBC/2.3.9/unixODBC-2.3.9pre.tar.gz/download

(2) 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。

(3) 替换客户端驱动程序。

① 解压 tar 包。解压后会得到两个文件夹：lib 与 odbc，在 odbc 文件夹中还会有一个 lib 文件夹。/odbc/lib 中会有"psqlodbca.la”， "psqlodbca.so”， "psqlodbcw.la”和"psqlodbcw.so”四个文件，将这四个文件拷贝到"/usr/local/lib”目录下。

② 将 lib 目录中的库拷贝到"/usr/local/lib”目录下。

(4) 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 5-2 所示。

表 5-2 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=/usr/local/lib/psqlodbcw.so
setup	驱动安装路径，与 Driver64 中动态库的路径一致。	setup=/usr/local/lib/psqlodbcw.so

配置数据源文件。

在"/usr/local/etc/odbc.ini”文件中追加以下内容。

```
[MPPODBC]
Driver=GaussMPP Servername=10.145.130.26（数据库 Server IP） Database=postgres （数据库名） Username=gbase （数据库用户名） Password= （数据库用户密码） Port=15432 （数据库侦听端口）
Sslmode=allow
```

odbc.ini 文件配置参数说明如表 5-3 所示。

表 5-3 odbc.ini 文件配置参数

参数	描述	示例
[DSN]	数据源的名称。	[MPPODBC]

参数	描述	示例
Driver	驱动名，对应 odbcinst.ini 中的 DriverName。	Driver=DRIVER_N
Servername	服务器的 IP 地址。可配置多个 IP 地址。	Servername=10.145.130.2 6
Database	要连接的数据库的名称。	Database=postgres
Username	数据库用户名称。	Username=gbase
Password	数据库用户密码。	Password= 说明 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=15432
Sslmode	开启 SSL 模式	Sslmode=allow
Debug	设置为 1 时，将会打印 psqlodbc 驱动的 mylog，日志生成目录为 /tmp/。设置为 0 时则不会生成。	Debug=1
UseServerSidePrep	是否开启数据库端扩展查询协	UseServerSidePrepare=1

参数	描述	示例
are	议。 可选值 0 或 1，默认为 1， 表示打开扩展查询协议。	
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
ForExtensionConnect or	这个开关控制着 savepoint 是否发送，savepoint 相关问题可以注意这个开关。	ForExtensionConnector=1
UnamedPrepStmtThr eshold	每次调用 SQLFreeHandle 释放 Stmt 时，ODBC 都会向 server 端发送一个 Deallocate plan_name 语句，业务中存在大量这类语句。为了减少这类语句的发送，我们将 stmt->plan_name 置空，从而使得数据库识别这个为 unamed stmt。增加这个参数对 unamed stmt 的阈值进行控制。	UnamedPrepStmtThresho ld=100
ConnectionExtraInfo	GUC 参数 connection_info（参见 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

参数	描述	示例
		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（参见 connection_info）里；同时可以在 20.3.72 PG_STAT_ACTIVITY 中查询到。
BoolAsChar	设置为 Yes 是，Bools 值将会映射为 SQL_CHAR。如不设置将会映射为 SQL_BIT。	BoolsAsChar = Yes
RowVersioning	当尝试更新一行数据时，设置为 Yes 会允许应用检测数据有没有被其他用户进行修改。	RowVersioning=Yes
ShowSystemTables	驱动将会默认系统表格为普通 SQL 表格。	ShowSystemTables=Yes

其中关于 Sslmode 的选项的允许值，具体信息见下表：

表 5-4 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，那么建议使用 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。GBase 8c 不支持此模式。

(5) （可选）生成 SSL 证书，具体请参见证书生成。在服务端与客户端通过 ssl 方式连接的情况下，需要执行步骤 5 或步骤 6。非 ssl 方式连接情况下可以跳过。

(6) （可选）替换 SSL 证书，具体请参见证书替换。

(7) 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：

```
退回根目录，创建.postgresql 目录，并将 root.crt，client.crt，client.key，client.key.cipher，
client.key.rand，
client.req，server.crt，server.key，server.key.cipher，server.key.rand，server.req 放在此路径
下。
Unix 系统下，server.crt、server.key 的权限设置必须禁止任何外部或组的访问，请执行如
下命令实现这一点。chmod 0600 server.key
将 root.crt 以及 server 开头的证书相关文件全部拷贝进数据库 install/data 目录下（与
postgresql.conf 文件在同一路径）。
修改 postgresql.conf 文件：
ssl = on
ssl_cert_file = 'server.crt' ssl_key_file = 'server.key' ssl_ca_file = 'root.crt'
修改完参数后需重启数据库。
修改配置文件 odbc.ini 中的 sslmode 参数（require 或 verify-ca）。
```

(8) 配置数据库服务器。

① 以操作系统用户 gbase 登录数据库主节点。

② 执行如下命令增加对外提供服务的网卡 IP 或者主机名（英文逗号分隔），NODETYPE 指定节点类型，NodeName 为当前节点名称：

```
gs_guc reload -Z datanode -N NodeName -I all -c "listen_addresses='XXX,XX'"
```

在 DR(Direct Routing, LVS 的直接路由 DR 模式)模式中需要将虚拟 IP 地址(10.11.12.13)加入到服务器的侦听地址列表中。

listen_addresses 也可以配置为"*”或"0.0.0.0”，此配置下将侦听所有网卡，但存在安全风

险，不推荐用户使用，推荐用户按照需要配置 IP 或者主机名，打开侦听。

执行如下命令在数据库主节点配置文件中增加一条认证规则。例如，假设客户端 IP 地址为 10.11.12.13，即远程连接的机器的 IP 地址。

```
gs_guc reload -Z datanode -N all -I all -h "host all jack 10.11.12.13/32 sha256"
```

- -N all 表示 GBase 8c 中的所有主机。
- -I all 表示主机中的所有实例。
- -h 表示指定需要在"pg_hba.conf”增加的语句。
- all 表示允许客户端连接到任意的数据库。
- jack 表示连接数据库的用户。
- 10.11.12.13/32 表示只允许 IP 地址为 10.11.12.13 的主机连接。在使用过程中，请根据用户的网络进行配置修改。32 表示子网掩码为 1 的位数，即 255.255.255.255
- sha256 表示连接时 jack 用户的密码使用 sha256 算法加密。

如果将 ODBC 客户端配置在和要连接的数据库主节点在同一台机器上，则可使用 local trust 认证方式，如下：

```
local all all trust
```

如果将 ODBC 客户端配置在和要连接的数据库主节点在不同机器上，则需要使用 sha256 认证方式，如下：

```
host all all xxx.xxx.xxx.xxx/32 sha256
```

重启数据库。

```
gs_om -t restart
```

(9) 在客户端配置环境变量。

```
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
```

(10) 执行如下命令使设置生效。

```
source ~/.bashrc
```

测试数据源配置

执行./isql -v MPPODBC（数据源名称）命令。

- 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。

常见问题处理

- [UnixODBC][Driver Manager]Can't open lib 'xxx/xxx/psqlodbcw.so' : file not found.

此问题的可能原因：

odbcinst.ini 文件中配置的路径不正确

确认的方法：'ls'一下错误信息中的路径，以确保该 psqlodbcw.so 文件存在， 同时具有执行权限。

psqlodbcw.so 的依赖库不存在，或者不在系统环境变量中

确认的办法：ldd 一下错误信息中的路径，如果是缺少 libodbc.so.1 等 UnixODBC 的库，那么按照"操作步骤”中的方法重新配置 UnixODBC，并确保它的安装路径下的 lib 目录添加到了 LD_LIBRARY_PATH 中；如果是缺少其他库，请将 ODBC 驱动包中的 lib 目录添加到 LD_LIBRARY_PATH 中。

- [UnixODBC]connect to server failed: no such file or directory

此问题可能的原因：

  - 配置了错误的/不可达的数据库地址，或者端口

请检查数据源配置中的 Servername 及 Port 配置项。

  - 服务器侦听不正确

如果确认 Servername 及 Port 配置正确，请根据"操作步骤”中数据库服务器的相关配置，确保数据库侦听了合适的网卡及端口。

- 防火墙及网闸设备
- 请确认防火墙设置，将数据库的通信端口添加到可信端口中。如果有网闸设备，请确认一下相关的设置。

- [unixODBC]The password-stored method is not supported.

  此问题可能原因：

  数据源中未配置 sslmode 配置项。

  解决办法：

  请配置该选项至 allow 或以上选项。此配置的更多信息，见表 6-10。

- Server common name "xxxx" does not match host name "xxxxx"

  此问题的原因：使用了 SSL 加密的"verify-full”选项，驱动程序会验证证书中的主机名与实际部署数据库的主机名是否一致。

  解决办法：碰到此问题可以使用"verify-ca”选项，不再校验主机名；或者重新生成一套与数据库所在主机名相同的 CA 证书。

- Driver's SQLAllocHandle on SQL_HANDLE_DBC failed

  此问题的可能原因：可执行文件（比如 UnixODBC 的 isql，以下都以 isql 为例）与数据库驱动（psqlodbcw.so）依赖于不同的 odbc 的库版本：libodbc.so.1 或者 libodbc.so.2。此 问题可以通过如下方式确认：

```
ldd `which isql` | grep odbc ldd psqlodbcw.so | grep odbc
```

  这时，如果输出的 libodbc.so 最后的后缀数字不同或者指向不同的磁盘物理文件， 那么基本就可以断定是此问题。isql 与 psqlodbcw.so 都会要求加载 libodbc.so，这时如果它们加载的是不同的物理文件，便会导致两套完全同名的函数列表，同时出现在同一个可见域里（UnixODBC 的 libodbc.so.*的函数导出列表完全一致）， 产生冲突，无法加载数据库驱动。

  解决办法：确定一个要使用的 UnixODBC，然后卸载另外一个（比如卸载库版本号为.so.2 的 UnixODBC），然后将剩下的.so.1 的库，新建一个同名但是后缀为.so.2 的软链接，便可解决此问题。

- FATAL: Forbid remote connection with trust method!

  由于安全原因，数据库主节点禁止 GBase 8c 内部其他节点无认证接入。

  如果要在 GBase 8c 内部访问数据库主节点，请将 ODBC 程序部署在数据库主节点所在

机器，服务器地址使用"127.0.0.1"。建议业务系统单独部署在 GBase 8c 外部，否则可能会影响数据库运行性能。

- [unixODBC][Driver Manager]Invalid attribute value

有可能是 unixODBC 的版本并非推荐版本，建议通过"odbcinst --version”命令排查环境中的 unixODBC 版本。

- authentication method 10 not supported.

使用开源客户端碰到此问题，可能原因：

数据库中存储的口令校验只存储了 SHA256 格式哈希，而开源客户端只识别 MD5 校验，双方校验方法不匹配报错。

说明

- 数据库并不存储用户口令，只存储用户口令的哈希码。
- 数据库当用户更新用户口令或者新建用户时，会同时存储两种格式的哈希码，这时将兼容开源的认证协议。
- 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。

要解决该问题，可以更新用户口令（参见《GBase 8c V5_5.0.0_SQL 参考手册》 ALTER USER 章节）；或者新建一个用户（参见《GBase 8c V5_5.0.0_SQL 参考手册》CREATE USER 章节），赋于同等权限，使用新用户连接数据库。

- unsupported frontend protocol 3.51: server supports 1.0 to 3.0

目标数据库版本过低，或者目标数据库为开源数据库。请使用对应版本的数据库驱动连接目标数据库。

- FATAL: GSS authentication method is not allowed because XXXX user password is not disabled.

目标数据库主节点的 pg_hba.conf 里配置了当前客户端 IP 使用"gss"方式来做认证， 该认证算法不支持用作客户端的身份认证，请修改到"sha256"后再试。配置方法见步骤 8。

5.2.1.3 开发流程

图 5-2 ODBC 开发应用程序的流程

开发流程中涉及的 API

表 5-5 相关 API 说明

功能	API
申请句柄资源	SQLAllocHandle：申请句柄资源，可替代如下函数： SQLAllocEnv：申请环境句柄 SQLAllocConnect：申请连接句柄 SQLAllocStmt：申请语句句柄
设置环境属性	SQLSetEnvAttr
设置连接属性	SQLSetConnectAttr
设置语句属性	SQLSetStmtAttr

连接数据源	SQLConnect
绑定缓冲区到结果集的列中	SQLBindCol
绑定 SQL 语句的参数标志和缓冲区	SQLBindParameter
查看最近一次操作错误信息	SQLGetDiagRec
为执行 SQL 语句做准备	SQLPrepare
执行一条准备好的 SQL 语句	SQLExecute
直接执行 SQL 语句	SQLExecDirect
结果集中取行集	SQLFetch
返回结果集中某一列的数据	SQLGetData
获取结果集中列的描述信息	SQLColAttribute
断开与数据源的连接	SQLDisconnect
释放句柄资源	SQLFreeHandle：释放句柄资源，可替代如下函数： SQLFreeEnv：释放环境句柄 SQLFreeConnect：释放连接句柄 SQLFreeStmt：释放语句句柄

说明

- 数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务，同时如果其中有一个语句失败，那么整个请求都将会被回滚。

警告

- ODBC 为应用程序与数据库的中心层，负责把应用程序发出的 SQL 指令传到数据库当中，自身并不解析 SQL 语法。故在应用程序中写入带有保密信息的 SQL 语句时（如明文密码），保密信息会被暴露在驱动日志中。

5.2.1.4 示例：常用功能和批量绑定

常用功能示例代码

```
// 此示例演示如何通过 ODBC 方式获取 GBase 8c 中的数据。
// DBtest.c (compile with: libodbc.so) #include <stdlib.h>
#include <stdio.h> #include <sqlext.h> #ifdef WIN32 #include <windows.h> #endif
SQLHENV V_OD_Env;      // Handle ODBC environment
SQLHSTMT    V_OD_hstmt    // Handle statement
SQLHDBC    V_OD_hdbc;    // Handle connection
char       typename[100];
SQLINTEGER value = 100;
SQLINTEGER V_OD_erg,V_OD_buffer,V_OD_err,V_OD_id; int main(int argc,char *argv[])
{
// 1. 申请环境句柄
V_OD_erg = SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&V_OD_Env);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
printf("Error AllocHandle\n"); exit(0);
}
// 2. 设置环境属性（版本信息）
SQLSetEnvAttr(V_OD_Env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
// 3. 申请连接句柄
V_OD_erg = SQLAllocHandle(SQL_HANDLE_DBC, V_OD_Env, &V_OD_hdbc);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env); exit(0);
}
// 4. 设置连接属性
SQLSetConnectAttr(V_OD_hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_ON,
0);
// 5. 连接数据源，这里的"userName”与"password”分别表示连接数据库的用户名和用户密码，请根据实际情况修改。
// 如果 odbc.ini 文件中已经配置了用户名密码，那么这里可以留空（""）；但是不建议这么做，因为一旦
odbc.ini 权限管理不善，将导致数据库用户密码泄露。
V_OD_erg = SQLConnect(V_OD_hdbc, (SQLCHAR*) "gaussdb", SQL_NTS,
(SQLCHAR*) "userName", SQL_NTS, (SQLCHAR*) "password", SQL_NTS); if
((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
```

```
printf("Error SQLConnect %d\n",V_OD_erg); SQLFreeHandle(SQL_HANDLE_ENV,
V_OD_Env); exit(0);
}
printf("Connected !\n");
// 6. 设置语句属性
SQLSetStmtAttr(V_OD_hstmt,SQL_ATTR_QUERY_TIMEOUT,(SQLPOINTER *)3,0);
// 7. 申请语句句柄
SQLAllocHandle(SQL_HANDLE_STMT, V_OD_hdbc, &V_OD_hstmt);
// 8. 直接执行 SQL 语句。
SQLExecDirect(V_OD_hstmt,"drop table IF EXISTS customer_t1",SQL_NTS);
SQLExecDirect(V_OD_hstmt,"CREATE TABLE customer_t1(c_customer_sk INTEGER,
c_customer_name
VARCHAR(32));",SQL_NTS);
SQLExecDirect(V_OD_hstmt,"insert into customer_t1 values(25,li)",SQL_NTS);
// 9. 准备执行
SQLPrepare(V_OD_hstmt,"insert into customer_t1 values(?)",SQL_NTS);
// 10. 绑定参数
SQLBindParameter(V_OD_hstmt,1,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_INTEGER,
0,0, &value,0,NULL);
// 11. 执行准备好的语句
SQLExecute(V_OD_hstmt);
SQLExecDirect(V_OD_hstmt,"select id from testtable",SQL_NTS);
// 12. 获取结果集某一列的属性
SQLColAttribute(V_OD_hstmt,1,SQL_DESC_TYPE,typename,100,NULL,NULL);
printf("SQLColAtrribute %s\n",typename);
// 13. 绑定结果集
SQLBindCol(V_OD_hstmt,1,SQL_C_SLONG, (SQLPOINTER)&V_OD_buffer,150,
(SQLLEN *)&V_OD_err);
// 14. 通过 SQLFetch 取结果集中数据
V_OD_erg=SQLFetch(V_OD_hstmt);
// 15. 通过 SQLGetData 获取并返回数据。
while(V_OD_erg != SQL_NO_DATA)
{
SQLGetData(V_OD_hstmt,1,SQL_C_SLONG,(SQLPOINTER)&V_OD_id,0,NULL);
printf("SQLGetData     ID = %d\n",V_OD_id);
V_OD_erg=SQLFetch(V_OD_hstmt);
};
printf("Done !\n");
// 16. 断开数据源连接并释放句柄资源
```

```
SQLFreeHandle(SQL_HANDLE_STMT,V_OD_hstmt); SQLDisconnect(V_OD_hdbc);
SQLFreeHandle(SQL_HANDLE_DBC,V_OD_hdbc); SQLFreeHandle(SQL_HANDLE_ENV,
V_OD_Env); return(0);
}
批量绑定示例代码
/*****************************************************************
请在数据源中打开 UseBatchProtocol，同时指定数据库中参数 support_batch_bind
为 on
CHECK_ERROR 的作用是检查并打印错误信息。
此示例将与用户交互式获取 DSN、模拟的数据量，忽略的数据量，并将最终数据入库到
test_odbc_batch_insert 中
*****************************************************************/
#include <stdio.h>
#include <stdlib.h> #include <sql.h> #include <sqlext.h> #include <string.h>
void Exec(SQLHDBC hdbc, SQLCHAR* sql)
{
SQLRETURN retcode; // Return status
SQLHSTMT hstmt = SQL_NULL_HSTMT; // Statement handle SQLCHAR  loginfo[2048];
// Allocate Statement Handle
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLAllocHandle(SQL_HANDLE_STMT) failed");
return;
}
// Prepare Statement
retcode = SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS); sprintf((char*)loginfo,
"SQLPrepare log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) {
printf("SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS) failed"); return;
}
// Execute Statement
retcode = SQLExecute(hstmt);
sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute(hstmt) failed"); return;
}
// Free Handle
retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt); sprintf((char*)loginfo,
"SQLFreeHandle stmt log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLFreeHandle(SQL_HANDLE_STMT, hstmt)
failed"); return;
}
}
```

```
int main ()
{
SQLHENV henv = SQL_NULL_HENV; SQLHDBC hdbc = SQL_NULL_HDBC;
int  batchCount = 1000; //  批量绑定的数据量
SQLLEN rowsCount = 0;
int  ignoreCount = 0; //  批量绑定的数据中，不要入库的数据量
SQLRETURN retcode; SQLCHAR     dsn[1024] = {'\0'};
SQLCHAR    loginfo[2048];
do
{
if (ignoreCount > batchCount)
{
printf("ignoreCount(%d) should be less than batchCount(%d)\n", ignoreCount, batchCount);
}
}while(ignoreCount > batchCount);
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv); if
(!SQL_SUCCEEDED(retcode)) {
printf("SQLAllocHandle failed"); goto exit;
}
// Set ODBC Verion
retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
(SQLPOINTER*)SQL_OV_ODBC3, 0);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetEnvAttr failed"); goto exit;
}
// Allocate Connection
retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLAllocHandle failed"); goto exit;
}
// Set Login Timeout
retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetConnectAttr failed"); goto exit;
}
// Set Auto Commit
retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT,
(SQLPOINTER)(1), 0);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetConnectAttr failed"); goto exit;
}
// Connect to DSN
// gaussdb 替换成用户所使用的数据源名称 sprintf(loginfo, "SQLConnect(DSN:%s)", dsn);
retcode = SQLConnect(hdbc, (SQLCHAR*) "gaussdb", SQL_NTS, (SQLCHAR*) NULL, 0,
NULL, 0);
```

```
if (!SQL_SUCCEEDED(retcode)) { printf("SQLConnect failed"); goto exit;
}
// init table info.
Exec(hdbc, "drop table if exists test_odbc_batch_insert");
Exec(hdbc, "create table test_odbc_batch_insert(id int primary key, col varchar2(50))");
// 下面的代码根据用户输入的数据量，构造出将要入库的数据:
{
SQLRETURN retcode;
SQLHSTMT hstmtinesrt = SQL_NULL_HSTMT; int       i;
SQLCHAR    *sql = NULL; SQLINTEGER *ids = NULL; SQLCHAR    *cols = NULL;
SQLLEN      *bufLenIds = NULL; SQLLEN     *bufLenCols = NULL; SQLUSMALLINT
*operptr = NULL; SQLUSMALLINT *statusptr = NULL; SQLULEN           process = 0;
// 这里是按列构造，每个字段的内存连续存放在一起。
ids = (SQLINTEGER*)malloc(sizeof(ids[0]) * batchCount); cols =
(SQLCHAR*)malloc(sizeof(cols[0]) * batchCount * 50);
// 这里是每个字段中，每一行数据的内存长度。
bufLenIds = (SQLLEN*)malloc(sizeof(bufLenIds[0]) * batchCount); bufLenCols =
(SQLLEN*)malloc(sizeof(bufLenCols[0]) * batchCount);
// 该行是否需要被处理，SQL_PARAM_IGNORE 或 SQL_PARAM_PROCEED operptr =
(SQLUSMALLINT*)malloc(sizeof(operptr[0]) * batchCount); memset(operptr, 0,
sizeof(operptr[0]) * batchCount);
// 该行的处理结果。
// 注：由于数据库中处理方式是同一语句隶属同一事务中，所以如果出错，那么待处理数
据都将是出错的， 并不会部分入库。
statusptr = (SQLUSMALLINT*)malloc(sizeof(statusptr[0]) * batchCount); memset(statusptr,
88, sizeof(statusptr[0]) * batchCount);
if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds)
{
fprintf(stderr, "FAILED:\tmalloc data memory failed\n"); goto exit;
}
for (int i = 0; i < batchCount; i++)
{
ids[i] = i;
sprintf(cols + 50 * i, "column test value %d", i); bufLenIds[i] = sizeof(ids[i]);
bufLenCols[i] = strlen(cols + 50 * i);
operptr[i] = (i < ignoreCount) ? SQL_PARAM_IGNORE : SQL_PARAM_PROCEED;
}
// Allocate Statement Handle
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtinesrt);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLAllocHandle failed"); goto exit;
}
```

```
// Prepare Statement
sql = (SQLCHAR*)"insert into test_odbc_batch_insert values(?, ?)"; retcode =
SQLPrepare(hstmtinesrt, (SQLCHAR*) sql, SQL_NTS); sprintf((char*)loginfo, "SQLPrepare
log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLPrepare failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMSET_SIZE,
(SQLPOINTER)batchCount, sizeof(batchCount));
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLBindParameter(hstmtinesrt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, sizeof(ids[0]), 0,&(ids[0]), 0, bufLenIds);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLBindParameter failed"); goto exit;
}
retcode = SQLBindParameter(hstmtinesrt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 50, 50,
cols, 50, bufLenCols);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLBindParameter failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMS_PROCESSED_PTR,
(SQLPOINTER)&process, sizeof(process));
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_STATUS_PTR,
(SQLPOINTER)statusptr, sizeof(statusptr[0]) * batchCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_OPERATION_PTR,
(SQLPOINTER)operptr, sizeof(operptr[0]) * batchCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLExecute(hstmtinesrt);
sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute(hstmtinesrt) failed"); goto exit;
retcode = SQLRowCount(hstmtinesrt, &rowsCount); if (!SQL_SUCCEEDED(retcode)) {
printf("SQLRowCount failed");
goto exit;
}
if (rowsCount != (batchCount - ignoreCount))
{
```

```
sprintf(loginfo, "(batchCount - ignoreCount)(%d) != rowsCount(%d)", (batchCount -
ignoreCount), rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
else
{
sprintf(loginfo, "(batchCount - ignoreCount)(%d) == rowsCount(%d)", (batchCount -
ignoreCount), rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
// check row number returned if (rowsCount != process)
{
sprintf(loginfo, "process(%d) != rowsCount(%d)", process, rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
else
{
sprintf(loginfo, "process(%d) == rowsCount(%d)", process, rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
for (int i = 0; i < batchCount; i++)
{
if (i < ignoreCount)
{
if (statusptr[i] != SQL_PARAM_UNUSED)
{
sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_UNUSED", i, statusptr[i]);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
}
else if (statusptr[i] != SQL_PARAM_SUCCESS)
{
sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_SUCCESS", i, statusptr[i]);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
```

```
}
retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmtinesrt); sprintf((char*)loginfo,
"SQLFreeHandle hstmtinesrt");
if (!SQL_SUCCEEDED(retcode)) { printf("SQLFreeHandle failed"); goto exit;
}
}
exit:
(void) printf ("\nComplete.\n");
// Connection
if (hdbc != SQL_NULL_HDBC) { SQLDisconnect(hdbc);
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
}
// Environment
if (henv != SQL_NULL_HENV) SQLFreeHandle(SQL_HANDLE_ENV, henv);
return 0;
}
```

5.2.1.5 典型应用场景配置

日志诊断场景

ODBC 日志分为 unixODBC 驱动管理器日志和 psqlODBC 驱动端日志。前者可以用于追溯应用程序 API 的执行是否成功，后者是底层实现过程中的一些 DFX 日志，用来帮助定位问题。

unixODBC 日志需要在 odbcinst.ini 文件中配置：

```
[ODBC]
Trace=Yes TraceFile=/path/to/odbctrace.log
[GaussMPP] Driver64=/usr/local/lib/psqlodbcw.so setup=/usr/local/lib/psqlodbcw.so
```

psqlODBC 日志只需要在 odbc.ini 加上：

```
[gaussdb] Driver=GaussMPP
Servername=10.10.0.13（数据库 Server IP）
...
Debug=1（打开驱动端 debug 日志）
```

说明

- unixODBC 日志将会生成在 TraceFile 配置的路径下，psqlODBC 会在系统/tmp/下生成 mylog_xxx.log。

高性能场景

进行大量数据插入时，建议如下：

- 需要设置批量绑定：odbc.ini 配置文件中设置 UseBatchProtocol=1、数据库设置 support_batch_bind=on。
- ODBC 程序绑定类型要和数据库中类型一致。
- 客户端字符集和数据库字符集一致。
- 事务改成手动提交。

odbc.ini 配置文件：

```
[gaussdb] Driver=GaussMPP
Servername=10.10.0.13（数据库 Server IP）
...
UseBatchProtocol=1 （默认打开）
ConnSettings=set client_encoding=UTF8 （设置客户端字符编码，保证和 server 端一致）
```

绑定类型用例：

```
#include <stdio.h> #include <stdlib.h> #include <sql.h> #include <sqlext.h> #include
<string.h> #include <sys/time.h>
#define MESSAGE_BUFFER_LEN 128 SQLHANDLE h_env = NULL; SQLHANDLE
h_conn = NULL; SQLHANDLE h_stmt = NULL;
void print_error()
{
SQLCHAR Sqlstate[SQL_SQLSTATE_SIZE+1];
SQLINTEGER NativeError;
SQLCHAR MessageText[MESSAGE_BUFFER_LEN];
SQLSMALLINT TextLength; SQLRETURN ret = SQL_ERROR;
ret = SQLGetDiagRec(SQL_HANDLE_STMT, h_stmt, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
if ( SQL_SUCCESS == ret)
{
printf("\n STMT ERROR-%05d %s", NativeError, MessageText); return;
}
ret = SQLGetDiagRec(SQL_HANDLE_DBC, h_conn, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
if ( SQL_SUCCESS == ret)
{
printf("\n CONN ERROR-%05d %s", NativeError, MessageText); return;
}
```

```
ret = SQLGetDiagRec(SQL_HANDLE_ENV, h_env, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
if ( SQL_SUCCESS == ret)
{
printf("\n ENV ERROR-%05d %s", NativeError, MessageText); return;
}
return;
}
/* 期盼函数返回 SQL_SUCCESS */
#define RETURN_IF_NOT_SUCCESS(func) \
{\
SQLRETURN ret_value = (func);\ if (SQL_SUCCESS != ret_value)\
{\
print_error();\
printf("\n failed line = %u: expect SQL_SUCCESS, but ret = %d", LINE , ret_value);\ return
SQL_ERROR; \
}\
}
/* 期盼函数返回 SQL_SUCCESS */
#define RETURN_IF_NOT_SUCCESS_I(i, func) \
{\
SQLRETURN ret_value = (func);\ if (SQL_SUCCESS != ret_value)\
{\
print_error();\
printf("\n failed line = %u (i=%d): : expect SQL_SUCCESS, but ret = %d", LINE , (i),
ret_value);\ return SQL_ERROR; \
}\
}
/* 期盼函数返回 SQL_SUCCESS_WITH_INFO */ #define
RETURN_IF_NOT_SUCCESS_INFO(func) \
{\
SQLRETURN ret_value = (func);\
if (SQL_SUCCESS_WITH_INFO != ret_value)\
{\
print_error();\
printf("\n failed line = %u: expect SQL_SUCCESS_WITH_INFO, but ret = %d", LINE ,
ret_value);\ return SQL_ERROR; \
}\
}
/* 期盼数值相等 */
#define RETURN_IF_NOT(expect, value) \ if ((expect) != (value))\
```

```
{\
printf("\n failed line = %u: expect = %u, but value = %u", LINE , (expect), (value)); \ return
SQL_ERROR;\
}
/* 期盼字符串相同 */
#define RETURN_IF_NOT_STRCMP_I(i, expect, value) \ if (( NULL == (expect) ) || (NULL
== (value)))\
{\
printf("\n failed line = %u (i=%u): input NULL pointer !", LINE , (i)); \ return SQL_ERROR; \
}\
else if (0 != strcmp((expect), (value)))\
{\
printf("\n failed line = %u (i=%u): expect = %s, but value = %s", LINE , (i), (expect), (value)); \
return SQL_ERROR;\
}
// prepare + execute SQL 语句 int execute_cmd(SQLCHAR *sql)
{
if ( NULL == sql )
{
return SQL_ERROR;
}
if ( SQL_SUCCESS != SQLPrepare(h_stmt, sql, SQL_NTS))
{
return SQL_ERROR;
}
if ( SQL_SUCCESS != SQLExecute(h_stmt))
{
return SQL_ERROR;
}
return SQL_SUCCESS;
}
// execute + commit 句柄
int commit_exec()
{
if ( SQL_SUCCESS != SQLExecute(h_stmt))
{
return SQL_ERROR;
}
// 手动提交
if ( SQL_SUCCESS != SQLEndTran(SQL_HANDLE_DBC, h_conn, SQL_COMMIT))
{
```

```
return SQL_ERROR;
}
return SQL_SUCCESS;
}
int begin_unit_test()
{
SQLINTEGER ret;
/* 申请环境句柄 */
ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &h_env); if
((SQL_SUCCESS != ret) && (SQL_SUCCESS_WITH_INFO != ret))
{
printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_ENV failed ! ret = %d", ret);
return SQL_ERROR;
}
/* 进行连接前必须要先设置版本号 */
if (SQL_SUCCESS != SQLSetEnvAttr(h_env, SQL_ATTR_ODBC_VERSION,
(SQLPOINTER)SQL_OV_ODBC3, 0))
{
print_error();
printf("\n begin_unit_test::SQLSetEnvAttr SQL_ATTR_ODBC_VERSION failed ! ret = %d",
ret); SQLFreeHandle(SQL_HANDLE_ENV, h_env);
return SQL_ERROR;
}
/* 申请连接句柄 */
ret = SQLAllocHandle(SQL_HANDLE_DBC, h_env, &h_conn); if (SQL_SUCCESS != ret)
{
print_error();
printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_DBC failed ! ret = %d", ret);
SQLFreeHandle(SQL_HANDLE_ENV, h_env);
return SQL_ERROR;
}
/* 建立连接 */
ret = SQLConnect(h_conn, (SQLCHAR*) "gaussdb", SQL_NTS,
(SQLCHAR*) NULL, 0, NULL, 0); if (SQL_SUCCESS != ret)
{
print_error();
printf("\n begin_unit_test::SQLConnect failed ! ret = %d", ret);
SQLFreeHandle(SQL_HANDLE_DBC, h_conn); SQLFreeHandle(SQL_HANDLE_ENV,
h_env);
return SQL_ERROR;
}
```

```
/* 申请语句句柄 */
ret = SQLAllocHandle(SQL_HANDLE_STMT, h_conn, &h_stmt); if (SQL_SUCCESS != ret)
{
print_error();
printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_STMT failed ! ret = %d", ret);
SQLFreeHandle(SQL_HANDLE_DBC, h_conn);
SQLFreeHandle(SQL_HANDLE_ENV, h_env); return SQL_ERROR;
}
return SQL_SUCCESS;
}
void end_unit_test()
{
/* 释放语句句柄 */ if (NULL != h_stmt)
{
SQLFreeHandle(SQL_HANDLE_STMT, h_stmt);
}
/* 释放连接句柄 */ if (NULL != h_conn)
{
SQLDisconnect(h_conn); SQLFreeHandle(SQL_HANDLE_DBC, h_conn);
}
/* 释放环境句柄 */ if (NULL != h_env)
{
SQLFreeHandle(SQL_HANDLE_ENV, h_env);
}
return;
}
int main()
{
// begin test
if (begin_unit_test() != SQL_SUCCESS)
{
printf("\n begin_test_unit failed."); return SQL_ERROR;
}
// 句柄配置同前面用例
int  i = 0;
SQLCHAR* sql_drop = "drop table if exists test_bindnumber_001"; SQLCHAR* sql_create =
"create table test_bindnumber_001("
"f4 number, f5 number(10, 2)" ")";
SQLCHAR* sql_insert = "insert into test_bindnumber_001 values(?, ?)"; SQLCHAR*
sql_select = "select * from test_bindnumber_001";
SQLLEN      RowCount; SQL_NUMERIC_STRUCT st_number;
```

```
SQLCHAR   getValue[2][MESSAGE_BUFFER_LEN];
/* step 1. 建 表 */ RETURN_IF_NOT_SUCCESS(execute_cmd(sql_drop));
RETURN_IF_NOT_SUCCESS(execute_cmd(sql_create));
/* step 2.1 通 过 SQL_NUMERIC_STRUCT 结 构 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
//第一行: 1234.5678
memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
st_number.precision = 8;
st_number.scale = 4;
st_number.sign = 1; st_number.val[0] = 0x4E; st_number.val[1] = 0x61; st_number.val[2] =
0xBC;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
// 关闭自动提交
SQLSetConnectAttr(h_conn, SQL_ATTR_AUTOCOMMIT,
(SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0);
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
//第二行: 12345678
memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
st_number.precision = 8;
st_number.scale = 0;
st_number.sign = 1; st_number.val[0] = 0x4E; st_number.val[1] = 0x61; st_number.val[2] =
0xBC;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 0, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 0, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
//第三行: 12345678
memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
st_number.precision = 0;
```

```
st_number.scale = 4;
st_number.sign = 1; st_number.val[0] = 0x4E; st_number.val[1] = 0x61; st_number.val[2] =
0xBC;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.2 第 四 行 通 过 SQL_C_CHAR 字 符 串 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS)); SQLCHAR*
szNumber = "1234.5678";
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_NUMERIC, strlen(szNumber), 0, szNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_NUMERIC, strlen(szNumber), 0, szNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.3 第 五 行 通 过 SQL_C_FLOAT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS)); SQLREAL
fNumber = 1234.5678;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_FLOAT, SQL_NUMERIC, sizeof(fNumber), 4, &fNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_FLOAT, SQL_NUMERIC, sizeof(fNumber), 4, &fNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.4 第 六 行 通 过 SQL_C_DOUBLE 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS)); SQLDOUBLE
dNumber = 1234.5678;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_DOUBLE, SQL_NUMERIC, sizeof(dNumber), 4, &dNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_DOUBLE, SQL_NUMERIC, sizeof(dNumber), 4, &dNumber, 0, NULL));
```

```
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
SQLBIGINT bNumber1 = 0xFFFFFFFFFFFFFFFF; SQLBIGINT bNumber2 = 12345;
/* step 2.5 第 七 行 通 过 SQL_C_SBIGINT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_SBIGINT, SQL_NUMERIC, sizeof(bNumber1), 4, &bNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_SBIGINT, SQL_NUMERIC, sizeof(bNumber2), 4, &bNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.6 第 八 行 通 过 SQL_C_UBIGINT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_UBIGINT,
SQL_NUMERIC, sizeof(bNumber1), 4, &bNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_UBIGINT,
SQL_NUMERIC, sizeof(bNumber2), 4, &bNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NO_UCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
SQLLEN lNumber1 = 0xFFFFFFFFFFFFFFFF; SQLLEN lNumber2 = 12345;
/* step 2.7 第 九 行 通 过 SQL_C_LONG 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_LONG,
SQL_NUMERIC, sizeof(lNumber1), 0, &lNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_LONG,
SQL_NUMERIC, sizeof(lNumber2), 0, &lNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.8 第 十 行 通 过 SQL_C_ULONG 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_ULONG,
```

```
SQL_NUMERIC, sizeof(lNumber1), 0, &lNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_ULONG,
SQL_NUMERIC, sizeof(lNumber2), 0, &lNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
SQLSMALLINT sNumber = 0xFFFF;
/* step 2.9 第十一行通过 SQL_C_SHORT 绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_SHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_SHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.10 第十二行通过 SQL_C_USHORT 绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_USHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_USHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
SQLCHAR cNumber = 0xFF;
/* step 2.11 第十三行通过 SQL_C_TINYINT 绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_TINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_TINYINT, SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
```

```
/* step 2.12 第 十 四 行 通 过 SQL_C_UTINYINT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_UTINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_UTINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* 用字符串类型统一进行期盼 */
SQLCHAR*       expectValue[14][2] = {{"1234.5678",    "1234.57"},
{"12345678","12345678"},
{"0", "0"},
{"1234.5678","1234.57"},
{"1234.5677","1234.57"},
{"1234.5678","1234.57"},
{"-1","12345"},
{"-1","12345"},
{"4294967295","12345"},
{"18446744073709551615", "12345"},
RETURN_IF_NOT_SUCCESS(execute_cmd(sql_select)); while ( SQL_NO_DATA !=
SQLFetch(h_stmt))
{
RETURN_IF_NOT_SUCCESS_I(i, SQLGetData(h_stmt, 1, SQL_C_CHAR, &getValue[0],
MESSAGE_BUFFER_LEN, NULL));
RETURN_IF_NOT_SUCCESS_I(i, SQLGetData(h_stmt, 2, SQL_C_CHAR, &getValue[1],
MESSAGE_BUFFER_LEN, NULL));
//RETURN_IF_NOT_STRCMP_I(i, expectValue[i][0], getValue[0]);
//RETURN_IF_NOT_STRCMP_I(i, expectValue[i][1], getValue[1]); i++;
}
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(i,
RowCount);
SQLCloseCursor(h_stmt);
/* step final. 删 除 表 还 原 环 境 */
RETURN_IF_NOT_SUCCESS(execute_cmd(sql_drop));
end_unit_test();
}
```

说明

- 上述用例中定义了 number 列，调用 SQLBindParameter 接口时，绑定 SQL_NUMERIC 会比 SQL_LONG 性能高一些。因为如果是 char，在数据库服务端插入数据时需要进行数据类型转换，从而引发性能瓶颈。

5.2.1.6 ODBC 接口参考

请参见《GBase 8c V5_5.0.0_应用开发指南》ODBC。

5.2.2基于 libpq 开发

libpq 是 GBase 8c 的 C 应用程序接口。libpq 是一套允许客户程序向服务器服务进程发送查询并且获得查询返回的库函数。同时也是其他几个 GBase 8c 应用接口下面的引擎，如 ODBC 等依赖的库文件。本章给出两个示例，显示如何利用 libpq 编写代码。

5.2.2.1 libpq 使用依赖的头文件

使用 libpq 的前端程序必须包括头文件 libpq-fe.h，并且必须与 libpq 库链接。

5.2.2.2 开发流程

编译并且链接一个 libpq 的源程序，需要做下面的一些事情：

解压相应的发布包文件。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 makefile 命令，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
```

5.2.2.3 常用功能示例代码

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='test_1234'";
/*  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');
*
期望的输出如下
*
*
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='test_1234'";
/* 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;
}
```

5.2.2.4 libpq 接口参数

请参见《GBase 8c V5_5.0.0_应用开发指南》libpq。

5.2.2.5 链接参数

表 5-6 链接参数

参数	描述
host	要链接的主机名。如果主机名以斜杠开头，则它声明使用 Unix 域套接字通讯而不是 TCP/IP 通讯；该值就是套接字文件所存储的目录。如果 没有声明 host，那么默认是与位于/tmp 目录（或者安装数据库的时候声明的套接字目录）里面的 Unix-域套接字链接。在没有 Unix 域套接字的机器上，默认与 localhost 链接。

参数	描述
	接受以’,’分割的字符串来指定多个主机名，支持指定多个主机名。
hostaddr	与之链接的主机的 IP 地址，是标准的 IPv4 地址格式，比如， 172.28.40.9。如果机器支持 IPv6，那么也可以使用 IPv6 的地址。如果声明了一个非空的字符串，那么使用 TCP/IP 通讯机制。 接受以’,’分割的字符串来指定多个 IP 地址，支持指定多个 IP 地址。 使用 hostaddr 取代 host 可以让应用避免一次主机名查找，这一点对于那些有时间约束的应用来说可能是非常重要的。不过，GSSAPI 或 SSPI 认证方法要求主机名（host）。因此，应用下面的规则： 如果声明了不带 hostaddr 的 host 那么就强制进行主机名查找。 如果声明中没有 host，hostaddr 的值给出服务器网络地址；如果认证方法要求主机名，那么链接尝试将失败。 如果同时声明了 host 和 hostaddr，那么 hostaddr 的值作为服务器网络地址。host 的值将被忽略，除非认证方法需要它，在这种情况下它将被用作主机名。 须知 ➢ 要注意如果 host 不是网络地址 hostaddr 处的服务器名，那么认证很有可能失败。 ➢ 如果主机名（host）和主机地址都没有，那么 libpq 将使用一个本地的 Unix 域套接字进行链接；或者是在没有 Unix 域套接字的机器上，它将尝试与 localhost 链接。
port	主机服务器的端口号，或者在 Unix 域套接字链接时的套接字扩展文件名。 接受以’,’分割的字符串来指定多个端口号，支持指定多个端口号。
user	要链接的用户名，缺省是与运行该应用的用户操作系统名同名的用户。
dbname	数据库名，缺省和用户名相同。
password	如果服务器要求口令认证，所用的口令。

参数	描述
connect_timeout	链接的最大等待时间，以秒计（用十进制整数字符串书写），0 或者不声明表示无穷。不建议把链接超时的值设置得小于 2 秒。
client_encoding	为这个链接设置 client_encoding 配置参数。除了对应的服务器选项接 受的值，你可以使用 auto 从客户端中的当前环境中确定正确的编码（Unix 系统上是 LC_CTYPE 环境变量）。
tty	忽略（以前，该参数指定了发送服务器调试输出的位置）。
options	添加命令行选项以在运行时发送到服务器。
application_name	为 application_name 配置参数指定一个值，表明当前用户身份。
fallback_application_name	为 application_name 配置参数指定一个后补值。如果通过一个连接参数或 PGAPPNAME 环境变量没有为 application_name 给定一个值， 将使用这个值。在希望设置一个默认应用名但不希望它被用户覆盖的一般工具程序中指定一个后补值很有用。
keepalives	控制客户端侧的 TCP 保持激活是否使用。缺省值是 1，意思为打开， 但是如果不想要保持激活，你可以更改为 0，意思为关闭。通过 Unix 域套接字做的链接忽略这个参数。
keepalives_idle	在 TCP 应该发送一个保持激活的信息给服务器之后，控制不活动的秒数。0 值表示使用系统缺省。通过 Unix 域套接字做的链接或者如果禁用了保持激活则忽略这个参数。
keepalives_interval	在 TCP 保持激活信息没有被应该传播的服务器承认之后，控制秒数。0 值表示使用系统缺省。通过 Unix 域套接字做的链接或者如果禁用了保持激活则忽略这个参数。
keepalives_count	添加命令行选项以在运行时发送到服务器。例如， 设置为 -c comm_debug_mode=off 设置 guc 参数 comm_debug_mode 参数的会话的值为 off。

参数	描述
rw_timeout	设置客户端连接读写超时时间。
sslmode	启用 SSL 加密的方式： ● disable：不使用 SSL 安全连接。 ● allow：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：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：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：必须使用 SSL 安全连接，当前 windows odbc 不支持 cert ● 方式认证。 ● verify-full：必须使用 SSL 安全连接，当前 windows odbc 不支持 cert 方式认证。
sslcompression	如果设置为 1（默认），SSL 连接之上传送的数据将被压缩（这要求 OpenSSL 版本为 0.9.8 或更高）。如果设置为 0，压缩将被禁用（这要求 OpenSSL 版本为 1.0.0 或更高）。如果建立的是一个没有 SSL 的连接，这个参数会被忽略。如果使用的 OpenSSL 版本不支持该参数，它也会被忽略。压缩会占用 CUP 时间，但是当瓶颈为网络时可以提高吞吐量。如果 CPU 性能是限制因素，禁用压缩能够改进响应时间和吞吐量。
sslcert	这个参数指定客户端 SSL 证书的文件名，它替换默认的~/.postgresql/postgresql.crt。如果没有建立 SSL 连接，这个参数会被忽略。
sslkey	这个参数指定用于客户端证书的密钥位置。它能指定一个会被用来替代默认的~/.postgresql/postgresql.key 的文件名，或者它能够指定一个从外部"引擎”（引擎是 OpenSSL 的可载入模块）得到的密钥。一个外部引擎说明应该由一个冒号分隔的引擎名称以及一个引擎相关的关键标识符组成。如果没有建立 SSL 连接，这个参数会被忽略。

参数	描述
sslrootcert	这个参数指定一个包含 SSL 证书机构（CA）证书的文件名称。如果该文件存在，服务器的证书将被验证是由这些机构之一签发。默认值是~/.postgresql/root.crt。
sslcrl	这个参数指定 SSL 证书撤销列表（CRL）的文件名。列在这个文件中的证书如果存在，在尝试认证该服务器证书时会被拒绝。默认值是~/.postgresql/root.crl。
requirepeer	这个参数指定服务器的操作系统用户，例如 requirepeer=postgres。当建立一个 Unix 域套接字连接时，如果设置了这个参数，客户端在连接开始时检查服务器进程是否运行在指定的用户名之下。如果发现不是，该连接会被一个错误中断。这个参数能被用来提供与 TCP/IP 连接上 SSL 证书相似的服务器认证（注意，如果 Unix 域套接字在/tmp 或另一个公共可写的位置，任何用户能启动一个在那里侦听的服务器。使用这个参数来保证你连接的是一个由可信用户运行的服务器）。这个选项只在实现了 peer 认证方法的平台上受支持。
krbsrvname	当用 GSSAPI 认证时，要使用的 Kerberos 服务名。为了让 Kerberos 认证成功，这必须匹配在服务器配置中指定的服务名。
gsslib	用于 GSSAPI 认证的 GSS 库。只用在 Windows 上。设置为 gssapi 可强制 libpq 用 GSSAPI 库来代替默认的 SSPI 进行认证。
service	用于附加参数的服务名。它指定保持附加连接参数的 pg_service.conf 中的一个服务名。这允许应用只指定一个服务名，这样连接参数能被集中维护。
authtype	不再使用"authtype”，因此将其标记为"不显示”。我们将其保留在数组中，以免拒绝旧应用程序中的 conninfo 字符串，这些应用程序可能仍在尝试设置它。
remote_nodename	指定连接本地节点的远端节点名称。
localhost	指定在一个连接通道中的本地地址。

参数	描述
localport	指定在一个连接通道中的本地端口。
fencedUdfRPCMode	控制 fenced UDF RPC 协议是使用 unix 域套接字或特殊套接字文件名。缺省值是 0，意思为关闭，使用 unix domain socket 模式，文件类型为 ".s.PGSQL.%d ”，但是要使用 fenced udf，文件类型为.s.fencedMaster_unixdomain，可以更改为 1，意思为开启。
replication	这个选项决定是否该连接应该使用复制协议而不是普通协议。这是 PostgreSQL 的复制连接以及 pg_basebackup 之类的工具在内部使用的协议，但也可以被第三方应用使用。支持下列值，大小写无关： true、on、yes、1：连接进入到物理复制模式。 database：连接进入到逻辑复制模式，连接到 dbname 参数中指定的数据库。 false、off、no、0：该连接是一个常规连接，这是默认行为。在物理或者逻辑复制模式中，仅能使用简单查询协议。
backend_version	传递到远端的后端版本号。
prototype	设置当前协议级别，默认：PROTO_TCP。
enable_ce	控制是否允许客户端连接全密态数据库。默认 0，如果需要开启，则修改为 1。
connection_info	Connection_info 是一个包含 driver_name、driver_version、driver_path 和 os_user 的 json 字符串。 如果不为 NULL，使用 connection_info 忽略 connectionExtraInf 如果为 NULL，生成与 libpq 相关的连接信息字符串，当 connectionExtraInf 为 false 时 connection_info 只有 driver_name 和 driver_version。
connectionExtraInf	设置 connection_info 是否存在扩展信息，默认值为 0，如果包含其他信息，则需要设置为 1。

参数	描述
target_session_attrs	设定连接的主机的类型。主机的类型和设定的值一致时才能连接成功。target_session_attrs 的设置规则如下： any(默认值)：可以对所有类型的主机进行连接。 read-write：当连接的主机允许可读可写时，才进行连接。 read-only：仅对可读的主机进行连接。 primary：仅对主备系统中的主机能进行连接。 standby：仅对主备系统中的备机进行连接。 prefer-standby：首先尝试找到一个备机进行连接。如果对 hosts 列表的所有机器都连接失败，那么尝试"any”模式进行连接。

5.2.3 Java

JDBC（Java Database Connectivity，Java 数据库连接）是一种用于执行 SQL 语句的 Java API，可以为多种关系数据库提供统一访问接口，应用程序可基于它操作数据。GBase 8c 数据库提供了对 JDBC 4.0 特性的支持，需要使用 JDK1.8 版本编译程序代码，不支持 JDBC 桥接 ODBC 方式。

5.2.3.1 JDBC 包、驱动类和环境类

JDBC 包

在 linux 服务器端源代码目录下执行 build.sh，获得驱动 jar 包 postgresql.jar。

驱动包与 PostgreSQL 保持兼容，其中类名、类结构与 PostgreSQL 驱动完全一致，曾经运行于 PostgreSQL 的应用程序可以直接移植到当前系统使用。

驱动类

在创建数据库连接之前，需要加载数据库驱动类"org.postgresql.Driver”。

由于 GBase 8c 在 JDBC 的使用上与 PG 的使用方法保持兼容，所以同时在同一进程内使用两个 JDBC 驱动的时候，可能会类名冲突。

相比于 PG 驱动，GBase 8c 的 JDBC 驱动主要做了以下特性的增强：

- 支持 SHA256 加密方式登录。

- 支持对接实现 sf4j 接口的第三方日志框架。
- 支持容灾切换。

环境类

客户端需配置 JDK1.8，配置方法如下：

(1) DOS 窗口输入"java -version”，查看 JDK 版本，确认为 JDK1.8 版本。如果未安装 JDK，请从官方网站下载安装包并安装。

(2) 根据如下步骤配置系统环境变量。

右键单击"我的电脑”，选择"属性”。

① 在"系统”页面左侧导航栏单击"高级系统设置”。

② 在"系统属性”页面，"高级”页签上单击"环境变量”。

③ 在"环境变量”页面上，"系统变量”区域单击"新建”或"编辑”配置系统变量。变量说明请参见下表。

表 5-7 变量说明

变量名	操作	变量值
JAVA_HO ME	若存在，则单击"编辑”。 若不存在，则单击"新建”。	JAVA 的安装目录。 例如：C:\Program Files\Java\jdk1.8.0_131
Path	编辑	若配置了 JAVA_HOME，则在变量值的最前面加上：%JAVA_HOME%\bin; 若未配置 JAVA_HOME，则在变量值的最前面加上 JAVA 安装的全路径：C:\Program Files\Java\jdk1.8.0_131\bin;
CLASSPAT H	新建	.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar;

5.2.3.2 开发流程

图 5-3 采用 JDBC 开发应用程序的流程

5.2.3.3 加载驱动

在创建数据库连接之前，需要先加载数据库驱动程序。

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。

5.2.3.4 连接数据库

在创建数据库连接之后，才能使用它来执行 SQL 语句操作数据。

函数原型

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);

参数

表 5-8 数据库连接参数

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。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。 ● enable_ce: String 类型。其中 enable_ce=1 表示 JDBC 支持密态等值查询。 ● refreshClientEncryption:String 类型。其中 refreshClientEncryption=1 表示密态数据库支持客户端缓存刷新（默认值为 1）。 ● 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。此参数已废弃， 不再生效，如需使用可通过 java.util.logging 属性文件或系统属性进行配置。 ● 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；characterEncoding 取值范围为 UTF8、GBK、LATIN1。 ● 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：

NonValidatingFactory 通道需要配置用户名和密码，同时将 SSL 设置为 true。

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。
- 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、url 连接配置信息上报到数据库中，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 1 和 3。注意：设置 1 时仅代表连接的是 V1 服务端。设置 3 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：String 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 true，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。
- uppercaseAttributeName：Boolean 类型，默认值为 false 不开启，为 true 时开启。该参数开启后会将获取元数据的接口的查询结果转为大写。适用场景为数据库中存储元数据全为小写，但要使用大写的元数据作为出参和入参。

涉及到的接口：java.sql.DatabaseMetaData、java.sql.ResultSetMetaData

- 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。
- priorityServers：Integer 类型。此值用于指定 url 上配置的前 n 个节点作为主数据库实例被优先连接。默认值为 null。该值为数字，大于 0， 且小于 url 上配置的 DN 数量。

	例 如 ： jdbc:postgresql:// host1:port1,host2:port2,host3:port3,host4:port4,/database?priorityServers=2。即表示 host1 与 host2 为主数据库实例节点，host3 与 host4 为容灾数据库实例节点。 ● forceTargetServerSlave：Boolean 类型。此值用于控制是否开启强制连接备机功能，并在数据库实例发生主备切换时，禁止已存在的连接在升主备机上继续使用。默认值为 false，表示不开启强制连接备机功能。true，表示开启强制连接备机功能。 ● traceInterfaceClass：String 类型。默认值为 null，用于获取 traceId 的实现类。值是实现获取 traceId 方法的接口 org.postgresql.log.Tracer 的 实现类的完整限定类名。 ● use_boolean：Boolean 类型。用于设置 extended 模式下 setBoolean 方法绑定的 oid 类型，默认为 false，绑定 int2 类型；设置为 true 则绑定 bool 类型。 ● allowReadOnly：Boolean 类型。用于设置是否允许只读模式，默认为 true，允许设置只读模式；设置为 false 则禁用只读模式。 ● TLSCiphersSupperted：String 类型。用于设置支持的 TLS 加密套件， 默认为 TLS_DHE_RSA_WITH_AES_128_GCM_SHA256,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
user	数据库用户。
password	数据库用户的密码。

说明

uppercaseAttributeName 参数开启后，如果数据库中有小写、大写和大小写混合的元数据，只能查询出小写部分的元数据，并以大写的形式输出，使用前请务必确认元数据的存储是否全为小写以避免数据出错。

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://10.10.0.13:15432/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://10.10.0.13:15432/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;
};
```

5.2.3.5 连接数据库（以 SSL 方式）

用户通过 JDBC 连接数据库服务器时，可以通过开启 SSL 加密客户端和服务器之间 的通讯，为敏感数据在 Internet 上的传输提供了一种安全保障手段。本小节主要介绍应用程序通过 JDBC 如何采用 SSL 的方式连接 GBase 8c 数据库。在使用本小节所描述的方法前，默认用户已经获取了服务端和客户端所需要的证书和私钥文件。

服务端配置

当开启 SSL 模式后，必须提供根证书、服务器证书和私钥。

配置步骤（假设用户的证书文件放在数据目录/opt/database/install/data/dn/下，且采用默认文件名）：

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 生成并配置证书。

生成 SSL 证书，具体请参见证书生成。将生成出的文件 server.crt，server.key，cacert.pem 拷贝到服务端数据目录下。

使用如下命令可以查询数据库节点的数据目录，instance 列为数据目录。

```
gs_om -t status --detail
```

在 Unix 系统上，server.crt、server.key 的权限设置必须禁止任何外部或组的访问，请执行如下命令实现这一点。

```
chmod 0600 server.key
```

(3) 开启 SSL 认证模式。

```
gs_guc set -D /opt/database/install/data/dn -c "ssl=on"
```

(4) 配置客户端接入认证参数，IP 为所要连接的主机 IP。

```
gs_guc reload -D /opt/database/install/data/dn -h "hostssl all     all    127.0.0.1/32        cert"
gs_guc reload -D /opt/database/install/data/dn -h "hostssl all     all    IP/32       cert"
```

表示允许 127.0.0.1/32 网段的客户端以 ssl 认证方式连接到数据库服务器。

须知

如果服务端 pg_hba.conf 文件中 METHOD 配置为 cert，则只有客户端使用证书（client.crt）中所设置的用户名（common name）才能够成功连接数据库。如果设置为 md5、sm3 或 sha256 则对连接数据库的用户没有限制。

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。

(5) 配置 SSL 认证相关的数字证书参数。各命令后所附为设置成功的回显。

```
gs_guc set -D /opt/database/install/data/dn -c "ssl_cert_file='server.crt'"
gs_guc set: ssl_cert_file='server.crt'
gs_guc set -D /opt/database/install/data/dn -c "ssl_key_file='server.key'"
gs_guc set: ssl_key_file='server.key'
gs_guc set -D /opt/database/install/data/dn -c "ssl_ca_file='cacert.pem'"
gs_guc set: ssl_ca_file='cacert.pem'
```

(6) 重启数据库。

```
gs_om -t restart
```

客户端配置

上传证书文件，将在服务端配置章节生成出的文件 client.key.pk8，client.crt，cacert.pem 放置在客户端。

示例

注：示例 1 和示例 2 选择其一。

```
public class SSL{
public static void main(String[] args) { Properties urlProps = new Properties();
String urls = "jdbc:postgresql://10.29.37.136:15432/postgres";
/**
* ================== 示例 1 使用 NonValidatingFactory 通道
```

```
*/ urlProps.setProperty("sslfactory","org.postgresql.ssl.NonValidatingFactory");
urlProps.setProperty("user", "world");
urlProps.setProperty("password", "test@123"); urlProps.setProperty("ssl", "true");
/**
* ================== 示例 2 使用证书
*/
urlProps.setProperty("sslcert", "client.crt"); urlProps.setProperty("sslkey", "client.key.pk8");
urlProps.setProperty("sslrootcert", "cacert.pem"); urlProps.setProperty("user", "world");
urlProps.setProperty("ssl", "true");
/* sslmode 可配置为：require、verify-ca、verify-full，以下三个示例选择其一*/
/* ================== 示例 2.1 设置 sslmode 为 require，使用证书 */
urlProps.setProperty("sslmode", "require");
/* ================== 示例 2.2 设置 sslmode 为 verify-ca，使用证书 */
urlProps.setProperty("sslmode", "verify-ca");
/* ================== 示例 2.3 设置 sslmode 为 verify-full，使用证书（Linux 下验证）
*/ urls = "jdbc:postgresql://world:15432/postgres";
urlProps.setProperty("sslmode", "verify-full"); try {
Class.forName("org.postgresql.Driver").newInstance();
} catch (Exception e) { e.printStackTrace();
}
try {
Connection conn;
conn = DriverManager.getConnection(urls,urlProps); conn.close();
} catch (Exception e) { e.printStackTrace();
}
}
}
/**
注：将客户端密钥转化为 DER 格式:
openssl pkcs8 -topk8 -outform DER -in client.key -out client.key.pk8 -nocrypt
openssl pkcs8 -topk8 -inform PEM -in client.key -outform DER -out client.key.der -v1
PBE-MD5-DES
openssl pkcs8 -topk8 -inform PEM -in client.key -outform DER -out client.key.der -v1
PBE-SHA1-3DES
以上算法由于安全级别较低，不推荐使用。
如果客户需要采用更高级别的私钥加密算法，启用 bouncycastle 或者其他第三方私钥解密
密码包后可以使用的私钥加密算法如下：
openssl pkcs8 -in client.key -topk8 -outform DER -out client.key.der -v2 AES128
openssl pkcs8 -in client.key -topk8 -outform DER -out client.key.der -v2 aes-256-cbc -iter
1000000
```

```
openssl pkcs8 -in client.key -topk8 -out client.key.der -outform Der -v2 aes-256-cbc -v2prf hmacWithSHA512
启用 bouncycastle：使用 jdbc 的项目引入依赖：bcpkix-jdk15on.jar 包，版本建议：1.65 以上。
*/
```

5.2.3.6 执行 SQL 语句

执行普通 SQL

应用程序通过执行 SQL 语句来操作数据库的数据（不用传递参数的语句），需要按以下步骤执行：

```
Connection conn = DriverManager.getConnection("url","user","password"); Statement stmt = conn.createStatement();
```

(1) 调用 Connection 的 createStatement 方法创建语句对象。

```
int rc = stmt.executeUpdate("CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name VARCHAR(32));");
```

(2) 调用 Statement 的 executeUpdate 方法执行 SQL 语句。

- 数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务，事务块中不支持 vacuum 操作。如果其中有一个语句失败，那么整个请求都将会被回滚。
- 使用 Statement 执行多语句时应以";”作为各语句间的分隔符，存储过程、函数、匿名块不支持多语句执行。
- "/”可用作创建单个存储过程、函数、匿名块的结束符。

(3) 关闭语句对象。

```
stmt.close();
```

执行预编译 SQL 语句

预编译语句是只编译和优化一次，然后可以通过设置不同的参数值多次使用。由于已经预先编译好，后续使用会减少执行时间。因此，如果多次执行一条语句，请选择使用预编译语句。可以按以下步骤执行：

(1) 调用 Connection 的 prepareStatement 方法创建预编译语句对象。

(2) 调用 PreparedStatement 的 setShort 设置参数。

```
pstmt.setShort(1, (short)2);
```

(3) 调用 PreparedStatement 的 executeUpdate 方法执行预编译 SQL 语句。

```
int rowcount = pstmt.executeUpdate();
```

(4) 调用 PreparedStatement 的 close 方法关闭预编译语句对象。

```
pstmt.close();
```

调用存储过程

GBase 8c 支持通过 JDBC 直接调用事先创建的存储过程，步骤如下：

(1) 调用 Connection 的 prepareCall 方法创建调用语句对象。

```
Connection myConn = DriverManager.getConnection("url","user","password");
CallableStatement cstmt = myConn.prepareCall("{? = CALL TESTPROC(?,?,?)}");
```

(2) 调用 CallableStatement 的 setInt 方法设置参数。

```
cstmt.setInt(2, 50);
cstmt.setInt(1, 20);
cstmt.setInt(3, 90);
```

(3) 调用 CallableStatement 的 registerOutParameter 方法注册输出参数。

```
cstmt.registerOutParameter(4, Types.INTEGER); //注册 out 类型的参数，类型为整型。
```

(4) 调用 CallableStatement 的 execute 执行方法调用。

```
cstmt.execute();
```

(5) 调用 CallableStatement 的 getInt 方法获取输出参数。

```
int out = cstmt.getInt(4); //获取 out 参数
```

示例：

```
//在数据库中已创建了如下存储过程，它带有 out 参数。
create or replace procedure testproc (
psv_in1 in integer, psv_in2 in integer, psv_inout in out integer
)
as begin
psv_inout := psv_in1 + psv_in2 + psv_inout; end;
/
```

(6) 调用 CallableStatement 的 close 方法关闭调用语句。

```
cstmt.close();
```

说明

很多的数据库类如 Connection、Statement 和 ResultSet 都有 close()方法，在使用完对象

后应把它们关闭。要注意的是，Connection 的关闭将间接关闭所有与它关联的 Statement，Statement 的关闭间接关闭了 ResultSet。

一些 JDBC 驱动程序还提供命名参数的方法来设置参数。命名参数的方法允许根据名称而不是顺序来设置参数，若参数有默认值，则可以不用指定参数值就可以使用此参数的默认值。即使存储过程中参数的顺序发生了变更，也不必修改应用程序。目前 GBase 8c 数据库的 JDBC 驱动程序不支持此方法。

GBase 8c 数据库不支持带有输出参数的函数，也不支持存储过程和函数参数默认值。

须知

- 当游标作为存储过程的返回值时，如果使用 JDBC 调用该存储过程，返回的游标将不可用。
- 存储过程不能和普通 SQL 在同一条语句中执行。
- 存储过程中 inout 类型参数必需注册出参。

Oracle 兼容模式启用重载时，调用存储过程

打开参数 behavior_compat_options='proc_outparam_override'后，JDBC 调用事先创建的存储过程。

(1) 调用 Connection 的 prepareCall 方法创建调用语句对象。

```
Connection conn = DriverManager.getConnection("url","user","password"); CallableStatement cs = conn.prepareCall("{ CALL TEST_PROC(?,?,?) }");
```

(2) 调用 CallableStatement 的 setInt 方法设置参数。

```
PGobject pGobject = new PGobject();
```

pGobject.setType("public.compfoo"); // 设置复合类型名，格式为"schema.typename”。pGobject.setValue("(1,demo)"); // 绑定复合类型值，格式为"(value1,value2)”。cs.setObject(1, pGobject);

(3) 调用 CallableStatement 的 registerOutParameter 方法注册输出参数。

// 注册 out 类型的参数，类型为复合类型,格式为"schema.typename”。

```
cs.registerOutParameter(2, Types.STRUCT, "public.compfoo");
```

(4) 调用 CallableStatement 的 execute 执行方法调用。

```
cs.execute();
```

(5) 调用 CallableStatement 的 getObject 方法获取输出参数。

```
PGobject result = (PGobject)cs.getObject(2);   // 获取 out 参数
result.getValue(); // 获取复合类型字符串形式值。
result.getArrayValue(); //获取复合类型数组形式值，以复合数据类型字段顺序排序。
result.getStruct(); //获取复合类型子类型名，按创建顺序排序。
```

(6) 调用 CallableStatement 的 close 方法关闭调用语句。

```
cs.close();
```

说明

oracle 兼容模式开启参数后，调用存储过程必须使用{call proc_name(?,?,?)}形式调用，调用函数必须使用{? = call func_name(?,?)}形式调用（等号左侧的" ? "为函数返回值的占位符，用于注册函数返回值）。

参数 behavior_compat_options='proc_outparam_override'行为变更后，业务需要重新建立连接，否则无法正确调用存储过程和函数。

函数和存储过程中包含复合类型时，参数的绑定与注册需要使用 schema.typename 形式。

```
// 在数据库创建复合数据类型。
CREATE TYPE compfoo AS (f1 int, f3 text);
// 在数据库中已创建了如下存储过程，它带有 out 参数。
create or replace procedure test_proc (
psv_in in compfoo, psv_out out compfoo
)
as begin
psv_out := psv_in; end;
/
```

执行批处理

用一条预处理语句处理多条相似的数据，数据库只创建一次执行计划，节省了语句的编译和优化时间。可以按如下步骤执行：

(1) 调用 Connection 的 prepareStatement 方法创建预编译语句对象。

```
Connection conn = DriverManager.getConnection("url","user","password"); PreparedStatement
pstmt = conn.prepareStatement("INSERT INTO customer_t1 VALUES (?)");
```

针对每条数据都要调用 setShort 设置参数，以及调用 addBatch 确认该条设置完毕。

```
pstmt.setShort(1, (short)2);
pstmt.addBatch();
```

(2) 调用 PreparedStatement 的 executeBatch 方法执行批处理。

```
int[] rowcount = pstmt.executeBatch();
```

(3) 调用 PreparedStatement 的 close 方法关闭预编译语句对象。

```
pstmt.close();
```

说明

- 在实际的批处理过程中，通常不终止批处理程序的执行，否则会降低数据库的性能。因此在批处理程序时，应该关闭自动提交功能，每几行提交一次。关闭自动提交功能的语句为： conn.setAutoCommit(false);

5.2.3.7 处理结果集

SQL 兼容性检查

GBase 8c 数据库支持接收外部发出的 SQL，并对其执行语法解析，无需重写或执行（因为数据库内当前不存在相应元数据）。若失败，返回失败原因。建议配合 GBase 8c 迁移工具使用，用于 SQL 兼容性检查。

设置结果集类型

不同类型的结果集有各自的应用场景，应用程序需要根据实际情况选择相应的结果集类型。在执行 SQL 语句过程中，都需要先创建相应的语句对象，而部分创建语句对象的方法提供了设置结果集类型的功能。具体的参数设置如表 6-3 所示。涉及的 Connection 的方法如下：

```
//创建一个 Statement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。
createStatement(int resultSetType, int resultSetConcurrency);
//创建一个 PreparedStatement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。
prepareStatement(String sql, int resultSetType, int resultSetConcurrency);
//创建一个 CallableStatement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。
prepareCall(String sql, int resultSetType, int resultSetConcurrency);
```

表 5-9 结果集类型

参数	描述
resultSetType	表示结果集的类型，具体有三种类型： ResultSet.TYPE_FORWARD_ONLY：ResultSet 只能向前移动。是缺省值。 ResultSet.TYPE_SCROLL_SENSITIVE：在修改后重新滚动到修改所

	在行，可以看到修改后的结果。 ResultSet.TYPE_SCROLL_INSENSITIVE：对可修改例程所做的编辑不进行显示。 说明： 结果集从数据库中读取了数据之后，即使类型是 ResultSet.TYPE_SCROLL_SENSITIVE，也不会看到由其他事务在这之后引起的改变。调用 ResultSet 的 refreshRow()方法，可进入数据库并从其中取得当前游标所指记录的最新数据。
resultSetConcurrenc y	表示结果集的并发，具体有两种类型： ResultSet.CONCUR_READ_ONLY：如果不从结果集中的数据建立一个新的更新语句，不能对结果集中的数据进行更新。 ResultSet.CONCUR_UPDATEABLE：可改变的结果集。对于可滚动的结果集，可对结果集进行适当的改变。

在结果集中定位

ResultSet 对象具有指向其当前数据行的光标。最初，光标被置于第一行之前。next 方法将光标移动到下一行；因为该方法在 ResultSet 对象没有下一行时返回 false，所以可以在 while 循环中使用它来迭代结果集。但对于可滚动的结果集，JDBC 驱动程序提供更多的定位方法，使 ResultSet 指向特定的行。定位方法如下表所示。

表 5-10 在结果集中定位的方法

方法	描述
next()	把 ResultSet 向下移动一行。
previous()	把 ResultSet 向上移动一行。
beforeFirst()	把 ResultSet 定位到第一行之前。
afterLast()	把 ResultSet 定位到最后一行之后。
first()	把 ResultSet 定位到第一行。

last()	把 ResultSet 定位到最后一行。
absolute(int)	把 ResultSet 移动到参数指定的行数。
relative(int)	通过设置为 1 向前（设置为 1，相当于 next()）或者向后（设置为-1，相当于 previous()）移动参数指定的行。

获取结果集中光标的位置

对于可滚动的结果集，可能会调用定位方法来改变光标的位置。JDBC 驱动程序提供了获取结果集中光标所处位置的方法。获取光标位置的方法如下表所示。

表 5-11 获取结果集光标的位置

方法	描述
isFirst()	是否在一行。
isLast()	是否在最后一行。
isBeforeFirst()	是否在第一行之前。
isAfterLast()	是否在最后一行之后。
getRow()	获取当前在第几行。

获取结果集中的数据

ResultSet 对象提供了丰富的方法，以获取结果集中的数据。获取数据常用的方法如下表所示，其他方法请参考 JDK 官方文档。

表 5-12 ResultSet 对象的常用方法

方法	描述
int getInt(int columnIndex)	按列标获取 int 型数据。
int getInt(String columnLabel)	按列名获取 int 型数据。
String getString(int columnIndex)	按列标获取 String 型数据。

String getString(String columnLabel)	按列名获取 String 型数据。
Date getDate(int columnIndex)	按列标获取 Date 型数据
Date getDate(String columnLabel)	按列名获取 Date 型数据。

5.2.3.8 关闭连接

在使用数据库连接完成相应的数据操作后，需要关闭数据库连接。

关闭数据库连接可以直接调用其 close 方法即可。如：Connection conn = DriverManager.getConnection("url","user","password") ; conn.close() ;

5.2.3.9 日志管理

GBase 8c JDBC 驱动程序支持使用日志记录，来帮助解决在应用程序中使用 JDBC 驱动程序时的问题。GBase 8c 的 JDBC 支持如下三种日志管理方式：

- 对接应用程序使用的 SLF4J 日志框架。
- 对接应用程序使用的 JdkLogger 日志框架。
- SLF4J 和 JdkLogger 是业界 Java 应用程序日志管理的主流框架，描述应用程序如何使用这些框架超出了本文范围，用户请参考对应的官方文档（SLF4J：http://www.slf4j.org/manual.html，JdkLogger：https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html）。

  方式一：对接应用程序的 SLF4J 日志框架。

  在建立连接时，url 配置 logger=Slf4JLogger。

  可采用 Log4j 或 Log4j2 来实现 SLF4J。当采用 Log4j 实现 SLF4J，需要添加如下 jar 包：

  log4j-*.jar、slf4j-api-*.jar、slf4j-log4*-*.jar，（*区分版本），和配置文件：

  log4j.properties。若采用 Log4j2 实现 SLF4J，需要添加如下 jar 包：log4j-api-*.jar、log4j-core-*.jar、log4j-slf4j18-impl-*.jar、slf4j-api-*-alpha1.jar（*区分版本），和配置文件：log4j2.xml。

  此方式支持日志管控。SLF4J 可通过文件中的相关配置实现强大的日志管控功能，建议使用此方式进行日志管理。

  注意

此方式依赖 slf4j 的通用 API 接口，如 org.slf4j.LoggerFactory.getLogger(String name)、org.slf4j.Logger.debug(String var1)、org.slf4j.Logger.info(String var1)、org.slf4j.Logger.warn(String warn)、org.slf4j.Logger.warn(String warn)等，若以上接口发生变更，日志将无法打印。

示例：

```
public static Connection GetConnection(String username, String passwd){
String sourceURL = "jdbc:postgresql://10.10.0.13:15432/postgres?logger=Slf4JLogger";
Connection conn = 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;
}
```

log4j.properties 示例：

```
log4j.logger.org.postgresql=ALL, log_gsjdbc
# 默认文件输出配置 log4j.appender.log_gsjdbc=org.apache.log4j.RollingFileAppender
log4j.appender.log_gsjdbc.Append=true log4j.appender.log_gsjdbc.File=gsjdbc.log
log4j.appender.log_gsjdbc.Threshold=TRACE log4j.appender.log_gsjdbc.MaxFileSize=10MB
log4j.appender.log_gsjdbc.MaxBackupIndex=5
log4j.appender.log_gsjdbc.layout=org.apache.log4j.PatternLayout
log4j.appender.log_gsjdbc.layout.ConversionPattern=%d %p %t %c - %m%n
log4j.appender.log_gsjdbc.File.Encoding = UTF-8
```

log4j2.xml 示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="OFF">
<appenders>
<Console name="Console" target="SYSTEM_OUT">
<PatternLayout pattern="%d %p %t %c - %m%n"/>
</Console>
<File name="FileTest" fileName="test.log">
<PatternLayout pattern="%d %p %t %c - %m%n"/>
</File>
<!--JDBC Driver 日志文件输出配置，支持日志回卷，设定日志大小超过 10MB 时，创建新的文件，新文件的命名格式为：年-月-日-文件编号-->
```

```
<RollingFile name="RollingFileJdbc" fileName="gsjdbc.log"
filePattern="%d{yyyy-MM-dd}-%i.log">
<PatternLayout pattern="%d %p %t %c - %m%n"/>
<Policies>
<SizeBasedTriggeringPolicy size="10 MB"/>
</Policies>
</RollingFile>
</appenders>
<loggers>
<root level="all">
<appender-ref ref="Console"/>
<appender-ref ref="FileTest"/>
</root>
<!--指定 JDBC Driver 日志，级别为：all，可查看所有日志，输出到 gsjdbc.log 文件中-->
<logger name="org.postgresql" level="all" additivity="false">
<appender-ref ref="RollingFileJdbc"/>
</logger>
</loggers>
</configuration>
```

方式二：对接应用程序使用的 JdkLogger 日志框架。

默认的 Java 日志记录框架将其配置存储在名为 logging.properties 的文件中。Java 会在 Java 安装目录的文件夹中安装全局配置文件。logging.properties 文件也可以创建并与单个项目一起存储。

logging.properties 配置示例：

```
# 指定处理程序为文件。
handlers= java.util.logging.FileHandler
# 指定默认全局日志级别
.level= ALL
# 指定日志输出管控标准 java.util.logging.FileHandler.level=ALL
java.util.logging.FileHandler.pattern = gsjdbc.log java.util.logging.FileHandler.limit = 500000
java.util.logging.FileHandler.count = 30 java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter java.util.logging.FileHandler.append=false
```

代码中使用示例：

```
System.setProperty("java.util.logging.FileHandler.pattern","jdbc.log");
FileHandler fileHandler = new
FileHandler(System.getProperty("java.util.logging.FileHandler.pattern")); Formatter formatter
= new SimpleFormatter();
fileHandler.setFormatter(formatter);
```

```
Logger logger = Logger.getLogger("org.postgresql"); logger.addHandler(fileHandler);
logger.setLevel(Level.ALL); logger.setUseParentHandlers(false);
```

链路跟踪功能

JDBC 驱动程序提供了应用到数据库的链路跟踪功能，用于将数据库端离散的 SQL 和应用程序的请求关联起来。该功能需要应用开发者实现 org.postgresql.log.Tracer 接口类，并在 url 中指定接口实现类的全限定名。

url 示例：

```
String URL = "jdbc:postgresql://127.0.0.1:15432/postgres?
traceInterfaceClass=xxx.xxx.xxx.OpenGaussTraceImpl";
```

org.postgresql.log.Tracer 接口类定义如下：

```
public interface Tracer {
// Retrieves the value of traceId. String getTraceId();
}
```

org.postgresql.log.Tracer 接口实现类示例：

```
import org.postgresql.log.Tracer;
public class OpenGaussTraceImpl implements Tracer {
private static MDC mdc = new MDC(); private final String TRACE_ID_KEY = "traceId";
public void set(String traceId) { mdc.put(TRACE_ID_KEY, traceId);
}
public void reset() { mdc.clear();
}
@Override
public String getTraceId() {
return mdc.get(TRACE_ID_KEY);
}
}
```

上下文映射示例，用于存放不同请求的生成的 traceId。

```
import java.util.HashMap;
public class MDC {
static final private ThreadLocal<HashMap<String, String>> threadLocal = new
ThreadLocal<>();
public void put(String key, String val) { if (key == null || val == null) {
throw new IllegalArgumentException("key or val cannot be null");
} else {
if (threadLocal.get() == null) { threadLocal.set(new HashMap<>());
}
```

```
threadLocal.get().put(key, val);
}
}
public String get(String key) { if (key == null) {
throw new IllegalArgumentException("key cannot be null");
} else if (threadLocal.get() == null) { return null;
} else {
return threadLocal.get().get(key);
}
}
public void clear() {
if (threadLocal.get() == null) { return;
} else {
threadLocal.get().clear();
}
}
}
```

业务使用 traceId 示例。

```
String traceId = UUID.randomUUID().toString().replaceAll("-", "");
openGaussTrace.set(traceId);
pstm = con.prepareStatement("select * from test_trace_id where id = ?"); pstm.setInt(1, 1);
pstm.execute();
pstm = con.prepareStatement("insert into test_trace_id values(?,?)"); pstm.setInt(1, 2);
pstm.setString(2, "test"); pstm.execute(); openGaussTrace.reset();
```

5.2.3.10 JDBC 接口参考

请参见《GBase 8c V5_5.0.0_应用开发指南》JDBC。

5.2.4Python

Psycopg 是一种用于执行 SQL 语句的 Python API，可以为数据库提供统一访问接口。基于此，应用程序可进行数据操作。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；可以通过灵活的对象适配系统进行扩展和定制适配。Psycopg2 兼容 Unicode 和 Python 3。

GBase 8c 数据库支持 Psycopg2 特性，并且支持通过 SSL 模式链接到 Psycopg2。

5.2.4.1 Psycopg 包

获取发布包，并解压。解压后有两个文件夹：

- psycopg2：psycopg2 库文件。
- lib：lib 库文件。

5.2.4.2 开发流程

图 5-4 采用 Psycopg2 开发应用程序的流程

5.2.4.3 加载驱动

在使用驱动之前，需要做如下操作：

- 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。
- 修改 psycopg2 目录权限为 755。
- 将 psycopg2 目录添加到环境变量$PYTHONPATH，并使之生效。
- 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
```

5.2.4.4 连接数据库

- 使用 psycopg2.connect 函数获得 connection 对象。
- 使用 connection 对象创建 cursor 对象。

5.2.4.5 执行 SQL 语句

- 构造操作语句，使用%s 作为占位符，执行时 psycopg2 会用参数值智能替换掉占位符。可以添加 RETURNING 子句，来得到自动生成的字段值。
- 使用 cursor.execute 方法来操作一行，使用 cursor.executemany 方法来操作多行。

5.2.4.6 处理结果集

- cursor.fetchone()：这种方法提取的查询结果集的下一行，返回一个序列，没有数 据可用时则返回空。
- cursor.fetchall()：这个例程获取所有查询结果(剩余)行，返回一个列表。空行时 则返回空列表。

5.2.4.7 关闭连接

- 在使用数据库连接完成相应的数据操作后，需要关闭数据库连接。关闭数据库连接可以直接调用其 close 方法，如 connection.close()。

  注意

  - 此方法关闭数据库连接，并不自动调用 commit()。如果只是关闭数据库连接而不调用 commit()方法，那么所有更改将会丢失。

5.2.4.8 连接数据库（SSL 方式）

用户通过 psycopy2 连接 Kernel 服务器时，可以通过开启 SSL 加密客户端和服务器之间的通讯。在使用 SSL 时，默认用户已经获取了服务端和客户端所需要的证书和私钥文件，关于证书等文件的获取请参考 Openssl 相关文档和命令。

- 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。

- 在连接选项中添加 SSL 连接相关参数：sslmode，sslcert，sslkey，sslrootcert。
  - sslmode：可选项见表 5-13。
  - sslcert：客户端证书路径。
  - sslkey：客户端密钥路径。
  - sslrootcert：根证书路径。
- 使用 psycopg2.connect 函数获得 connection 对象。
- 使用 connection 对象创建 cursor 对象。

表 5-13 sslmode 的可选项及其描述

sslmode	是否会启用 SSL 加密	描述
disable	否	不适用 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 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。
verify-full	是	必须使用 SSL 安全连接，目前暂不支持。

5.2.4.9 示例：常用操作

```
import psycopg2
#创建连接对象
conn=psycopg2.connect(database="postgres",user="user",password="password",host="localhost",port=port) cur=conn.cursor() #创建指针对象
#创建连接对象（SSl 连接）
```

```
conn = psycopg2.connect(dbname="postgres", user="user", password="password",
host="localhost", port=port,
sslmode="verify-ca", sslcert="client.crt",sslkey="client.key",sslrootcert="cacert.pem")
注意： 如果 sslcert, sslkey,sslrootcert 没有填写，默认取当前用户.postgresql 目录下对应的
client.crt， client.key， root.crt
# 创建表
cur.execute("CREATE TABLE student(id integer,name varchar,sex varchar);")
#插入数据
cur.execute("INSERT INTO student(id,name,sex) VALUES(%s,%s,%s)",(1,'Aspirin','M'))
cur.execute("INSERT INTO student(id,name,sex) VALUES(%s,%s,%s)",(2,'Taxol','F'))
cur.execute("INSERT INTO student(id,name,sex) VALUES(%s,%s,%s)",(3,'Dixheral','M'))
# 获取结果
cur.execute('SELECT * FROM student') results=cur.fetchall()
print (results)
# 关闭连接 conn.commit() cur.close() conn.close()
psycopg2 常用链接方式
conn = psycopg2.connect(dbname="postgres", user="user", password="password",
host="localhost", port=port)
conn = psycopg2.connect("dbname=postgres user=user password=password host=localhost
port=port")
使用日志 import logging import psycopg2
from psycopg2.extras import LoggingConnection
logging.basicConfig(level=logging.DEBUG) # 日志级别
logger = logging.getLogger( name )
db_settings = { "user": "user",
"password": "password", "host": "localhost", "database": "postgres", "port": port
}
conn = psycopg2.connect(connection_factory=LoggingConnection, **db_settings)
conn.initialize(logger)
```

5.2.4.10 Psycopg 接口参考

请参见《GBase 8c V5_5.0.0_应用开发指南》Psycopg。

6 创建和管理数据库

前提条件

系统管理员或具有创建数据库权限的用户才能创建数据库。赋予创建数据库的权限参见管理用户及权限。

背景信息

- GBase 8c 数据库默认包含两个初始模板数据库 template0、template1，以及用户数据库 postgres。postgres 默认兼容类型为 O（即 DBCOMPATIBILITY = A ），该兼容类型下将空字符串作为 NULL 处理。
- CREATE DATABASE 实际上是通过拷贝模板数据库来创建新数据库。默认情况下，拷贝 template0。

  说明

  - 模板数据库中没有用户表，可通过系统表 PG_DATABASE 查看模板数据库属性。
  - 模板数据库 template0 不允许用户连接；模板数据库 template1 只允许数据库初始用户和系统管理员连接，普通用户无法连接。
- 数据库系统中会有多个数据库，但是客户端程序同时只能连接一个数据库。也不能在不同的数据库之间相互查询。当存在多个数据库时，需要通过-d 参数指定相应的数据库实例进行连接。

注意事项

数据库的编码为 SQL_ASCII 的前提下（可以通过"show server_encoding;”命令查看当前数据库存储编码），在创建数据库对象时，如果对象名中含有多字节字符（例如中文），超过数据库对象名长度限制（63 字节）的时候，数据库将会将最后一个字节（而不是字符）截断，可能造成出现半个字符的情况。

针对这种情况，请遵循以下条件：

- 保证数据对象的名称不超过限定长度。
- 修改数据库的默认存储编码集（server_encoding）为 utf-8 编码集。
- 不要使用多字节字符做为对象名。
- 创建的数据库总数目建议不超过 128 个。
- 如果出现因为误操作导致在多字节字符的中间截断而无法删除数据库对象的现象，请使

用截断前的数据库对象名进行删除操作，或将该对象从各个数据库节点的相应系统表中依次删掉。

操作步骤

(1) 创建一个新数据库，例如 db_tpcc：

```
postgres=# CREATE DATABASE db_tpcc;
CREATE DATABASE
```

说明

- 数据库名称遵循 SQL 标识符的一般规则。当前角色自动成为此新数据库的所有者。
- 如果一个数据库系统用于承载多个相互独立的用户和项目，建议把这些用户和项目放在不同的数据库里。
- 如果项目或者用户是相互关联的，并且可以相互使用对方的资源，则应该把它们放在同一个数据库里，但可以规划在不同的模式中。模式只是一个纯粹的逻辑结构，某个模式的访问权限由权限系统模块控制。
- 创建数据库时，若数据库名称长度超过 63 字节，server 端会对数据库名称进行截断，保留前 63 个字节，因此建议数据库名称长度不要超过 63 个字节。

(2) 查看数据库信息

- 查看数据库系统的数据库列表：

```
postgres=# \l
```

- 通过系统表 pg_database 查询数据库列表：

```
postgres=# SELECT datname FROM pg_database;
```

(3) 修改数据库

用户可以使用 ALTER DATABASE 命令，修改数据库属性（比如：owner、名称和默认的配置属性）。

- 为数据库设置默认的模式搜索路径，例如：

```
postgres=# ALTER DATABASE db_tpcc SET search_path TO pa_catalog,public;
ALTER DATABASE
```

- 重命名数据库，例如：

```
postgres=# ALTER DATABASE db_tpcc RENAME TO human_tpcds;
ALTER DATABASE
```

(4) 删除数据库

用户可以使用 DROP DATABASE 命令删除数据库。这个命令可以删除数据库中的系统目录，并且删除了磁盘上带有数据的数据库目录。必须是系统管理员或数据库 owner 用户才能删除数据库。并且当有人还在连接该数据库，删除操作会失败。删除数据库前请先连接到其他的数据库。

使用如下命令删除数据库：

```
postgres=# DROP DATABASE human_tpcds;
DROP DATABASE
```

7 创建和管理表空间

背景信息

通过使用表空间，管理员可以控制数据库所在磁盘的布局。这样有以下优点：

- 如果初始化数据库所在的分区或者卷空间已满，又不能逻辑上扩展更多空间，可以在不同的分区上创建和使用表空间，直到系统重新配置空间。
- 表空间允许管理员根据数据库对象的使用模式安排数据位置，从而提高性能。
  - 一个频繁使用的索引可以放在性能稳定且运算速度较快的磁盘上，比如固态设备。
  - 一个存储归档的数据，很少使用的或者对性能要求不高的表可以存储在一个运算速度较慢的磁盘上。
- 管理员通过表空间可以设置占用的磁盘空间。在和其他数据共用分区的时候，防止表空间占用相同分区上的其他空间。
- 表空间对应于一个文件系统目录，假定数据库节点数据目录/pg_location/ mount1/path1是用户拥有读写权限的空目录。

  使用表空间配额管理会使性能有 30%左右的影响，MAXSIZE 指定每个数据库节点的配额大小，误差范围在 500MB 以内。根据实际情况，确认是否需要设置表空间的最大值。

  GBase 8c 自带两个表空间：pg_default 和 pg_global。
  - 默认表空间 pg_default：用来存储非共享系统表、用户表、用户表 index、临时表、临时表 index、内部临时表的默认表空间。对应存储目录为实例数据目录下的 base 目录。
  - 共享表空间 pg_global：用来存放共享系统表的表空间。对应存储目录为实例数据目录下的 global 目录。

注意事项

用户自定义表空间通常配合主存（即默认表空间所在的存储设备，如磁盘）以外的其它存储介质使用，以隔离不同业务可以使用的 IO 资源。自定义表空间使用不当不利于系统长稳运行以及影响整体性能。因此建议使用默认表空间即可。

操作步骤

(1) 创建表空间

使用 CREATE DATABASE 创建表空间：

```
postgres=# CREATE TABLESPACE fastspace RELATIVE LOCATION
'tablespace/tablespace_1';
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLESPACE
```

其中，fastspace 为新创建的表空间，tablespace/tablespace_1 是用户拥有读写权限的空目录。

以系统管理员身份将表空间 fastspace 的访问权限赋予普通用户，例如：

```
postgres=# GRANT CREATE ON TABLESPACE fastspace TO jack;
```

当结果显示为如下信息，则表示赋予成功。

```
GRANT
```

(2) 在表空间中创建对象

如果用户拥有表空间的 CREATE 权限，就可以在表空间上创建数据库对象。以创建表为例。

- 直接在指定表空间创建表。例如在 fastspace 表空间中创建表 foo：

```
postgres=# CREATE TABLE foo(i int) TABLESPACE fastspace;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

- 使用 set default_tablespace 命令设置默认表空间，再创建表.例如，设置 fastspace 为默认表空间，创建表 foo2：

```
postgres=# SET default_tablespace = 'fastspace';
SET
postgres=# CREATE TABLE foo2(i int);
CREATE TABLE
```

(3) 查询表空间

- 通过 pg_tablespace 系统表查询系统和用户定义的全部表空间。

```
postgres=# SELECT spcname FROM pg_tablespace;
```

- 使用 gsql 元命令查询表空间。

```
postgres=# \db
```

(4) 查询表空间使用率

查询表空间的当前使用情况。

```
postgres=# SELECT PG_TABLESPACE_SIZE('fastspace');
```

返回如下信息：

```
pg_tablespace_size
--------------------
2146304
(1 row)
```

其中 2146304 表示表空间的大小，单位为字节。

计算表空间使用率公式为（PG_TABLESPACE_SIZE/表空间所在目录的磁盘大小）。

(5) 修改表空间

例如，将表空间 fastspace 重命名为 fspace。

```
postgres=# ALTER TABLESPACE fastspace RENAME TO fspace;
ALTER TABLESPACE
```

(6) 删除表空间

- 删除表空间 owner 用户：

```
postgres=# DROP USER jack CASCADE;
DROP ROLE
```

- 删除表空间中的表：

```
postgres=# DROP TABLE foo;
postgres=# DROP TABLE foo2;
```

当结果显示为如下信息，则表示删除成功。

```
DROP TABLE
```

- 删除表空间：

```
postgres=# DROP TABLESPACE fspace;
DROP TABLESPACE
```

说明

➢ 用户必须是表空间的 owner 或者系统管理员才能删除表空间。

8 数据库对象管理

8.1 创建和管理 schema

背景信息

schema 又称作模式。通过管理 schema，允许多个用户使用同一数据库而不相互干扰，可以将数据库对象组织成易于管理的逻辑组，同时便于将第三方应用添加到相应的 schema 下而不引起冲突。支持创建 schema、使用 schema、删除 schema、设置 schema 的搜索路径以及 schema 的权限控制等管理操作。

注意事项

- GBase 8c 包含一个或多个已命名数据库。用户和用户组在 GBase 8c 范围内是共享的，但是其数据并不共享。任何与服务器连接的用户都只能访问连接请求里声明的那个数据库。
- 一个数据库可以包含一个或多个已命名的 schema，schema 又包含表及其他数据库对象，包括数据类型、函数、操作符等。同一对象名可以在不同的 schema 中使用而不会引起冲突。例如，schema1 和 schema2 都可以包含一个名为 mytable 的表。
- 和数据库不同，schema 不是严格分离的。用户根据其对 schema 的权限，可以访问所连接数据库的 schema 中的对象。进行 schema 权限管理首先需要对数据库的权限控制进行了解。
- 不能创建以 PG_为前缀的 schema 名，该类 schema 为数据库系统预留的。
- 在每次创建新用户时，系统会在当前登录的数据库中为新用户创建一个同名 Schema。对于其他数据库，若需要同名 Schema，则需要用户手动创建。
- 通过未修饰的表名（名称中只含有表名，没有 schema 名）引用表时，系统会通过 search_path（搜索路径）来判断该表是哪个 schema 下的表。pg_temp 和 pg_catalog 始终会作为搜索路径顺序中的前两位，无论二者是否出现在 search_path 中，或者出现在 search_path 中的任何位置。search_path（搜索路径）是一个 schema 名列表，在其中找到的第一个表就是目标表，如果没有找到则报错。某个表即使存在，如果它的 schema 不在 search_path 中，依然会查找失败。在搜索路径中的第一个 schema 叫做"当前 schema"。它是搜索时查询的第一个 schema，同时在没有声明 schema 名时，新创建的数据库对象会默认存放在该 schema 下。
- 每个数据库都包含一个 pg_catalog schema，它包含系统表和所有内置数据类型、函数、

操作符。pg_catalog 是搜索路径中的一部分，始终在临时表所属的模式后面，并在 search_path 中所有模式的前面，即具有第二搜索优先级。这样确保可以搜索到数据库内置对象。如果用户需要使用和系统内置对象重名的自定义对象时，可以在操作自定义对象时带上自己的模式。

操作步骤

(1) 创建 schema。例如：

```
postgres=# CREATE SCHEMA myschema AUTHORIZATION gbase;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE SCHEMA
```

(2) 使用 schema

在特定 schema 下创建对象或者访问特定 schema 下的对象，其完整的对象名称由模式名称和具体的对象名称组成。中间由符号"."隔开。例如：myschema.table。

① 创建特定模式下的表，例如，创建 myschema 模式下的表 mytable：

```
postgres=# CREATE TABLE myschema.mytable(id int, name varchar(20));
CREATE TABLE
```

② 查看特定 schema 下的表，例如，查看 myschema 下 mytable 表的所有数据：

```
postgres=# SELECT * FROM myschema.mytable;
id | name
----+------
(0 rows)
```

(3) schema 的搜索路径

可以设置 search_path 配置参数指定寻找对象可用 schema 的顺序。在搜索路径列出的第一个 schema 会变成默认的 schema。如果在创建对象时不指定 schema，则会创建在默认的 schema 中。

① 查看搜索路径。

```
postgres=# SHOW SEARCH_PATH;
search_path
----------------
"$user",public
(1 row)
```

② 将搜索路径设置为 myschema、public，首先搜索 myschema：

```
postgres=# SET SEARCH_PATH TO myschema, public;
SET
```

③ 再次查看搜索路径，已变更。

```
postgres=# SHOW SEARCH_PATH;
search_path
----------------
myschema, public
(1 row)
```

(4) schema 的权限控制

默认情况下，用户只能访问属于自己的 schema 中的数据库对象。如果需要访问其他 schema 的对象，则该 schema 的所有者应该赋予他对该 schema 的 usage 权限。

通过将模式的 CREATE 权限授予某用户，被授权用户就可以在此模式中创建对象。注意默认情况下，所有角色都拥有在 public 模式上的 USAGE 权限，但是普通用户没有在 public 模式上的 CREATE 权限。普通用户能够连接到一个指定数据库并在它的 public 模式中创建对象是不安全的，如果普通用户具有在 public 模式上的 CREATE 权限则建议通过如下语句撤销该权限。

- 撤销 PUBLIC 在 public 模式下创建对象的权限，下面语句中第一个"public”是模式，第二个"PUBLIC”指的是所有角色。

```
postgres=# REVOKE CREATE ON SCHEMA public FROM PUBLIC;
REVOKE
```

- 查看现有的 schema：

```
postgres=# SELECT current_schema();
current_schema
----------------
myschema
 (1 row)
```

- 创建用户 jack，并将 myschema 的 usage 权限赋给用户 jack，例如：

```
postgres=# CREATE USER jack IDENTIFIED BY 'Gbase,123';
CREATE ROLE
postgres=# GRANT USAGE ON schema myschema TO jack;
GRANT
```

- 将用户 jack 对于 myschema 的 usage 权限收回。

```
postgres=# REVOKE USAGE ON schema myschema FROM jack;
```

```
REVOKE
```

(5) 删除 schema

- 删除 schema
  - 当 schema 为空时，即该 schema 下没有数据库对象，使用 DROP SCHEMA 命令进行删除。例如删除名为 nullschema 的空 schema。

```
postgres=# DROP SCHEMA IF EXISTS nullschema;
DROP SCHEMA
```

  - 当 schema 非空时，如果要删除一个 schema 及其包含的所有对象，需要使用 CASCADE 关键字。例如删除 myschema 及该 schema 下的所有对象。

```
postgres=# DROP SCHEMA myschema CASCADE;
DROP SCHEMA
```

- 删除用户 jack。

```
postgres=# DROP USER jack;
DROP ROLE
```

8.2 创建和管理普通表

8.2.1创建表

背景信息

表是建立在数据库中的，在不同的数据库中可以存放相同的表。甚至可以通过使用模式在同一个数据库中创建相同名称的表。

使用 CREATE TABLE 语句创建表，例如创建表 customer_t1 :

```
postgres=# CREATE TABLE customer_t1
(
c_customer_sk integer,
c_customer_id char(5),
c_first_name char(6),
c_last_name  char(8)
);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

其中 c_customer_sk 、c_customer_id、c_first_name 和 c_last_name 是表的字段名，integer、

char(5)、char(6)和 char(8)分别是这四个字段名称的类型。

8.2.2向表中插入数据

在创建一个表后，表中并没有数据，在使用这个表之前，需要向表中插入数据。本小节介绍如何使用 INSERT 命令插入一行或多行数据，及从指定表插入数据。如果有大量数据需要批量导入表中，请参考《GBase 8c V5_5.0.0_数据库运维指南》中的“导入数据”章节。

背景信息

服务端与客户端使用不同的字符集时，两者字符集中单个字符的长度也会不同，客户端输入的字符串会以服务端字符集的格式进行处理，所以产生的最终结果可能会与预期不一致。

表 8-1 客户端和服务端设置字符集的输出结果对比

操作过程	服务端和客户端编码一致	服务端和客户端编码不一致
存入和取出过程中没有对字符串进行操作	输出预期结果	输出预期结果(输入与显示的客户端编码必须一致）。
存入取出过程对字符串有做一定的操作(如字符串函数操作)	输出预期结果	根据对字符串具体操作可能产生非预期结果。
存入过程中对超长字符串有截断处理	输出预期结果	字符集中字符编码长度是否一致，如果不一致可能会产生非预期的结果。

上述字符串函数操作和自动截断产生的效果会有叠加效果，例如：在客户端与服务端字符集不一致的场景下，如果既有字符串操作，又有字符串截断，在字符串被处理完以后的情况下继续截断，这样也会产生非预期的效果。详细的示例请参见下表。

说明

- 数据库 DBCOMPATIBILITY 设为兼容 C 模式，且 td_compatible_truncation 参数设置为 on 的情况下，才会对超长字符串进行截断。

  执行如下命令建立示例中需要使用的表 table1、table2。

```
postgres=# CREATE TABLE table1(id int, a char(6), b varchar(6),c varchar(6));
postgres=# CREATE TABLE table2(id int, a char(20), b varchar(20),c varchar(20));
```

表 8-2 示例

编号	服务端字符集	客户端字符集	是否启用自动截断	示例	结果	说明
1	SQL_ASCII	UTF8	是	postgres=# INSERT INTO table1 VALUES(1,reverse('123 A A 78'),reverse('123 A A 78'),reverse('123 A A 78'));	id |a|b|c --+--+--+-- 1 | 87| 87| 87	字符串在服务端翻转后，并进行截断，由于服务端和客户端的字符集不一致，字符 A 在客户端由多个字节表示，结果产生异常。
2	SQL_ASCII	UTF8	是	postgres=# INSERT INTO table1 VALUES(2,reverse('123 A 78'),reverse('123 A 78'),reverse('123 A 78'));	id |a|b|c --+--+--+-- 2 |873|873|873	字符串翻转后，又进行了自动截断，所以产生了非预期的效果。
3	SQL_ASCII	UTF8	是	postgres=# INSERT INTO table1 VALUES(3,'87 A 123','87 A 123','87 A 123');	id | a | b| c --+--+--+--- 3 | 87 A 1 | 87 A 1 | 87 A 1	字符串类型的字段长度是客户端字符编码长度的整数倍，所以截断后产生结果正常。
4	SQL_ASCII	UTF8	否	postgres=# INSERT INTO table2 VALUES(1,reverse('123 A A 78'),reverse('123 A A 78'),reverse('123 A A 78')); postgres=# INSERT INTO table2 VALUES(2,reverse('123 A 78'),reverse('123 A	id |a|b|c ---+-+--+-- 1 |87|321| 87 321 | 87 321 2 | 87321| 87321| 87321	多字节字符翻转之后不再表示原来的字符。

				78'),reverse('123 A 78'));		

操作步骤

(1) 向表中插入数据前，意味着表已创建成功，参考创建表。

(2) 向表 customer_t1 中插入一行：数据值是按照这些字段在表中出现的顺序列出的，并且用逗号分隔。通常数据值是文本（常量），但也允许使用标量表达式。

```
postgres=# INSERT INTO customer_t1(c_customer_sk, c_customer_id, c_first_name)
VALUES (3769, 'hello', 'Grace');
```

如果用户已经知道表中字段的顺序，也可无需列出表中的字段。例如以下命令上面的命令效果相同。

```
postgres=# INSERT INTO customer_t1 VALUES (3769, 'hello', 'Grace');
```

如果用户不知道所有字段的数值，可以忽略其中的一些。没有数值的字段将被填充为字段的缺省值。例如：

```
postgres=# INSERT INTO customer_t1 (c_customer_sk, c_first_name) VALUES (3769,
'Grace');
postgres=# INSERT INTO customer_t1 VALUES (3769, 'hello');
```

用户也可以对独立的字段或者整个行明确缺省值：

```
postgres=# INSERT INTO customer_t1 (c_customer_sk, c_customer_id, c_first_name)
VALUES (3769, 'hello', DEFAULT);
postgres=# INSERT INTO customer_t1 DEFAULT VALUES;
```

如果需要在表中插入多行，请使用以下命令：

```
postgres=# INSERT INTO customer_t1 (c_customer_sk, c_customer_id, c_first_name)
VALUES (6885, 'maps', 'Joes'), (4321, 'tpcds', 'Lily'), (9527, 'world', 'James');
```

如果需要向表中插入多条数据，除此命令外，也可以多次执行插入一行数据命令实现。但是建议使用此命令可以提升效率。

如果从指定表插入数据到当前表，例如在数据库中创建了一个表 customer_t1 的备份表 customer_t2，现在需要将表 customer_t1 中的数据插入到表 customer_t2 中，则可以执行如下命令。

```
postgres=# CREATE TABLE customer_t2 (
c_customer_sk     integer,
c_customer_id     char(5),
c_first_name char(6),
```

```
c_last_name  char(8)
);
postgres=# INSERT INTO customer_t2 SELECT * FROM customer_t1;
```

说明

- 从指定表插入数据到当前表时，若指定表与当前表对应的字段数据类型之间不存在隐式转换，则这两种数据类型必须相同。

(3) 删除备份表。

```
postgres=# DROP TABLE customer_t2 CASCADE;
```

说明

- 在删除表的时候，若当前需删除的表与其他表有依赖关系，需先删除关联的表，然后再删除当前表。

更新表中数据

修改已经存储在数据库中数据的行为叫做更新。用户可以更新单独一行、所有行或者指定的部分行。还可以独立更新每个字段，而其他字段则不受影响。

使用 UPDATE 命令更新现有行，需要提供以下三种信息：

- 表的名称和要更新的字段名
- 字段的新值
- 要更新哪些行

SQL 通常不会为数据行提供唯一标识，因此无法直接声明需要更新哪一行。但是可以通过声明一个被更新的行必须满足的条件。只有在表里存在主键的时候，才可以通过主键指定一个独立的行。

(1) 建立表和插入数据的参考创建表和向表中插入数据。

(2) 需要将表 customer_t1 中 c_customer_sk 为 9527 的地域重新定义为 9876：

```
postgres=# UPDATE customer_t1 SET c_customer_sk = 9876 WHERE c_customer_sk = 9527;
```

这里的表名称也可以使用模式名修饰，否则会从默认的模式路径找到这个表。SET 后面紧跟字段和新的字段值。新的字段值不仅可以是常量，也可以是变量表达式。

比如，把所有 c_customer_sk 的值增加 100：

```
postgres=# UPDATE customer_t1 SET c_customer_sk = c_customer_sk + 100;
```

在这里省略了 WHERE 子句，表示表中的所有行都要被更新。如果出现了 WHERE 子句，那么只有匹配其条件的行才会被更新。

在 SET 子句中的等号是一个赋值，而在 WHERE 子句中的等号是比较。WHERE 条件不一定是相等测试，许多其他的操作符也可以使用。

用户可以在一个 UPDATE 命令中更新更多的字段，方法是在 SET 子句中列出更多赋值，比如：

```
postgres=# UPDATE customer_t1 SET c_customer_id = 'Admin', c_first_name = 'Local'
WHERE c_customer_sk = 4421;
```

批量更新或删除数据后，会在数据文件中产生大量的删除标记，查询过程中标记删除的数据也是需要扫描的。故多次批量更新/删除后，标记删除的数据量过大会严重影响查询的性能。建议在批量更新/删除业务会反复执行的场景下，定期执行 VACUUM FULL 以保持查询性能。

查看数据

- 使用系统表 pg_tables 查询数据库所有表的信息。

```
postgres=# SELECT * FROM pg_tables;
```

- 使用 gsql 的\d+命令查询表的属性。

```
postgres=# \d+ customer_t1;
```

- 查询表中的数据量。

```
postgres=# SELECT count(*) FROM customer_t1;
```

- 查询表中所有数据。

```
postgres=# SELECT * FROM customer_t1;
```

- 只查询表中字段 c_customer_sk 的数据。

```
postgres=# SELECT c_customer_sk FROM customer_t1;
```

- 过滤表中字段 c_customer_sk 的重复数据。

```
postgres=# SELECT DISTINCT( c_customer_sk ) FROM customer_t1;
```

- 查询表中字段 c_customer_sk 为 3869 的所有数据。

```
postgres=# SELECT * FROM customer_t1 WHERE c_customer_sk = 3869;
```

- 将表数据按照字段 c_customer_sk 进行排序。

```
postgres=# SELECT * FROM customer_t1 ORDER BY c_customer_sk;
```

删除表中数据

在使用表的过程中，可能会需要删除已过期的数据，删除数据必须从表中整行的删除。

SQL 不能直接访问独立的行，只能通过声明被删除行匹配的条件进行。如果表中有一个主键，用户可以指定准确行。用户可以删除匹配条件的一组行，或一次删除表中的所有行。

- 使用 DELETE 命令删除行。如删除表 customer_t1 中所有 c_customer_sk 为 3869 的记录：

```
postgres=# DELETE FROM customer_t1 WHERE c_customer_sk = 3869;
```

如果执行如下命令之一，会删除表中所有的行。

```
postgres=# DELETE    FROM customer_t1;
```

或

```
postgres=# TRUNCATE TABLE customer_t1;
```

说明

➢ 全表删除的场景下，建议使用 TRUNCATE，不建议使用 DELETE。

- 删除创建的表：

```
postgres=# DROP TABLE customer_t1;
```

8.3 创建和管理分区表

背景信息

GBase 8c 数据库支持的分区表为范围分区表、列表分区表、哈希分区表。

- 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。这种分区方式是最为常用的，并且分区键经常采用日期，例如将销售数据按照月份进行分区。
- 列表分区表：将数据中包含的键值分别存储在不同的分区中，依次将数据映射到每一个分区，分区中包含的键值由创建分区表时指定。
- 哈希分区表：将数据根据内部哈希算法依次映射到每一个分区中，包含的分区个数由创建分区表时指定。

分区表和普通表相比具有以下优点：

- 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。
- 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。

- 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。
- 均衡 I/O：可以把不同的分区映射到不同的磁盘以平衡 I/O，改善整个系统性能。

普通表若要转成分区表，需要新建分区表，然后把普通表中的数据导入到新建的分区表中。因此在初始设计表时，请根据业务提前规划是否使用分区表。

示例

示例一：使用默认表空间

(1) 创建分区表（假设用户已创建 tpcds schema）

```
postgres=# CREATE TABLE tpcds.customer_address
(
    ca_address_sk         integer                 NOT NULL    ,
    ca_address_id         character(16)           NOT NULL    ,
    ca_street_number      character(10)                       ,
    ca_street_name        character varying(60)               ,
    ca_street_type        character(15)                       ,
    ca_suite_number       character(10)                       ,
    ca_city               character varying(60)               ,
    ca_county             character varying(30)               ,
    ca_state              character(2)                        ,
    ca_zip                character(10)                       ,
    ca_country            character varying(20)               ,
    ca_gmt_offset         numeric(5,2)                        ,
    ca_location_type      character(20)
)
PARTITION BY RANGE (ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(5000),
        PARTITION P2 VALUES LESS THAN(10000),
        PARTITION P3 VALUES LESS THAN(15000),
        PARTITION P4 VALUES LESS THAN(20000),
        PARTITION P5 VALUES LESS THAN(25000),
        PARTITION P6 VALUES LESS THAN(30000),
        PARTITION P7 VALUES LESS THAN(40000),
        PARTITION P8 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

说明

- 创建列存分区表的数量建议不超过 1000 个。

(2) 插入数据

将表 tpcds.customer_address 的数据插入到表 tpcds.web_returns_p2 中。

例如在数据库中创建了一个表 tpcds.customer_address 的备份表 tpcds.web_returns_p2，现在需要将表 tpcds.customer_address 中的数据插入到表 tpcds.web_returns_p2 中，则可以执行如下命令。

```
postgres=# CREATE TABLE tpcds.web_returns_p2
(
    ca_address_sk            integer                       NOT NULL     ,
    ca_address_id            character(16)                 NOT NULL     ,
    ca_street_number         character(10)                              ,
    ca_street_name           character varying(60)                      ,
    ca_street_type           character(15)                              ,
    ca_suite_number          character(10)                               ,
    ca_city                  character varying(60)                      ,
    ca_county                character varying(30)                      ,
    ca_state                 character(2)                               ,
    ca_zip                   character(10)                               ,
    ca_country               character varying(20)                      ,
    ca_gmt_offset            numeric(5,2)                                ,
    ca_location_type         character(20)
)
PARTITION BY RANGE (ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(5000),
        PARTITION P2 VALUES LESS THAN(10000),
        PARTITION P3 VALUES LESS THAN(15000),
        PARTITION P4 VALUES LESS THAN(20000),
        PARTITION P5 VALUES LESS THAN(25000),
        PARTITION P6 VALUES LESS THAN(30000),
        PARTITION P7 VALUES LESS THAN(40000),
        PARTITION P8 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
CREATE TABLE
```

```
postgres=# INSERT INTO tpcds.web_returns_p2 SELECT * FROM tpcds.customer_address;
INSERT 0 0
```

(3) 修改分区表行迁移属性

```
postgres=# ALTER TABLE tpcds.web_returns_p2 DISABLE ROW MOVEMENT;
ALTER TABLE
```

(4) 删除分区

删除分区 P8。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 DROP PARTITION P8;
ALTER TABLE
```

(5) 增加分区

增加分区 P8，范围为 40000<= P8<=MAXVALUE。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 ADD PARTITION P8 VALUES LESS
THAN (MAXVALUE);
ALTER TABLE
```

(6) 重命名分区

重命名分区 P8 为 P_9。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION P8 TO P_9;
ALTER TABLE
```

重命名分区 P_9 为 P8。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION FOR (40000) TO P8;
ALTER TABLE
```

(7) 查询分区

查询分区 P6。

```
postgres=# SELECT * FROM tpcds.web_returns_p2 PARTITION (P6);
postgres=# SELECT * FROM tpcds.web_returns_p2 PARTITION FOR (35888);
```

(8) 删除分区表和表空间

```
postgres=# DROP TABLE tpcds.customer_address;
DROP TABLE
postgres=# DROP TABLE tpcds.web_returns_p2;
DROP TABLE
```

示例二：使用用户自定义表空间

按照以下方式对范围分区表的进行操作。

(1) 创建表空间

```
postgres=# CREATE TABLESPACE example1 RELATIVE LOCATION
'tablespace1/tablespace_1';
postgres=# CREATE TABLESPACE example2 RELATIVE LOCATION
'tablespace2/tablespace_2';
postgres=# CREATE TABLESPACE example3 RELATIVE LOCATION
'tablespace3/tablespace_3';
postgres=# CREATE TABLESPACE example4 RELATIVE LOCATION
'tablespace4/tablespace_4';
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLESPACE
```

(2) 创建分区表

```
postgres=# CREATE TABLE tpcds.customer_address
(
    ca_address_sk         integer                 NOT NULL    ,
    ca_address_id         character(16)           NOT NULL    ,
    ca_street_number      character(10)                       ,
    ca_street_name        character varying(60)               ,
    ca_street_type        character(15)                       ,
    ca_suite_number       character(10)                       ,
    ca_city               character varying(60)               ,
    ca_county             character varying(30)               ,
    ca_state              character(2)                        ,
    ca_zip                character(10)                       ,
    ca_country            character varying(20)               ,
    ca_gmt_offset         numeric(5,2)                        ,
    ca_location_type      character(20)
)
TABLESPACE example1

PARTITION BY RANGE (ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(5000),
        PARTITION P2 VALUES LESS THAN(10000),
        PARTITION P3 VALUES LESS THAN(15000),
        PARTITION P4 VALUES LESS THAN(20000),
        PARTITION P5 VALUES LESS THAN(25000),
```

```
        PARTITION P6 VALUES LESS THAN(30000),
        PARTITION P7 VALUES LESS THAN(40000),
        PARTITION P8 VALUES LESS THAN(MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

说明

- 创建列存分区表的数量建议不超过 1000 个。

(3) 插入数据

将表 tpcds.customer_address 的数据插入到表 tpcds.web_returns_p2 中。

例如在数据库中创建了一个表 tpcds.customer_address 的备份表 tpcds.web_returns_p2，现在需要将表 tpcds.customer_address 中的数据插入到表 tpcds.web_returns_p2 中，则可以执行如下命令。

```
postgres=# CREATE TABLE tpcds.web_returns_p2
(
    ca_address_sk             integer                       NOT NULL     ,
    ca_address_id             character(16)                 NOT NULL     ,
    ca_street_number          character(10)                              ,
    ca_street_name            character varying(60)                      ,
    ca_street_type            character(15)                              ,
    ca_suite_number           character(10)                              ,
    ca_city                   character varying(60)                      ,
    ca_county                 character varying(30)                      ,
    ca_state                  character(2)                               ,
    ca_zip                    character(10)                              ,
    ca_country                character varying(20)                      ,
    ca_gmt_offset             numeric(5,2)                               ,
    ca_location_type          character(20)
)
TABLESPACE example1
PARTITION BY RANGE (ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(5000),
        PARTITION P2 VALUES LESS THAN(10000),
        PARTITION P3 VALUES LESS THAN(15000),
```

```
        PARTITION P4 VALUES LESS THAN(20000),
        PARTITION P5 VALUES LESS THAN(25000),
        PARTITION P6 VALUES LESS THAN(30000),
        PARTITION P7 VALUES LESS THAN(40000),
        PARTITION P8 VALUES LESS THAN(MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
CREATE TABLE
postgres=# INSERT INTO tpcds.web_returns_p2 SELECT * FROM tpcds.customer_address;
INSERT 0 0
```

(4) 修改分区表行迁移属性

```
postgres=# ALTER TABLE tpcds.web_returns_p2 DISABLE ROW MOVEMENT;
ALTER TABLE
```

(5) 删除分区

删除分区 P8。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 DROP PARTITION P8;
ALTER TABLE
```

(6) 增加分区

增加分区 P8，范围为 40000<= P8<=MAXVALUE。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 ADD PARTITION P8 VALUES LESS
THAN (MAXVALUE);
ALTER TABLE
```

(7) 重命名分区

重命名分区 P8 为 P_9。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION P8 TO P_9;
ALTER TABLE
```

重命名分区 P_9 为 P8。

```
postgres=# ALTER TABLE tpcds.web_returns_p2 RENAME PARTITION FOR (40000) TO P8;
ALTER TABLE
```

(8) 修改分区的表空间

修改分区 P6 的表空间为 example3。

```
postgres=#    ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P6 TABLESPACE
example3;
ALTER TABLE
```

修改分区 P4 的表空间为 example4。

```
postgres=#    ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P4 TABLESPACE
example4;
ALTER TABLE
```

(9) 查询分区

查询分区 P6。

```
postgres=# SELECT * FROM tpcds.web_returns_p2 PARTITION (P6);
postgres=# SELECT * FROM tpcds.web_returns_p2 PARTITION FOR (35888);
```

(10) 删除分区表和表空间

```
postgres=# DROP TABLE tpcds.web_returns_p2;
DROP TABLE
postgres=# DROP TABLESPACE example1;
postgres=# DROP TABLESPACE example2;
postgres=# DROP TABLESPACE example3;
postgres=# DROP TABLESPACE example4;
DROP TABLESPACE
```

8.4 创建和管理索引

背景信息

索引可以提高数据的访问速度，但同时也增加了插入、更新和删除操作的处理时间。所以是否要为表增加索引，索引建立在哪些字段上，是创建索引前必须要考虑的问 题。需要分析应用程序的业务处理、数据使用、经常被用作查询的条件或者被要求排序的字段来确定是否建立索引。

- 索引建立在数据库表中的某些列上。因此，在创建索引时，应该仔细考虑在哪些列上创建索引。
- 在经常需要搜索查询的列上创建索引，可以加快搜索的速度。
- 在作为主键的列上创建索引，强制该列的唯一性和组织表中数据的排列结构。
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的。

- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间。
- 在经常使用 WHERE 子句的列上创建索引，加快条件的判断速度。
- 为经常出现在关键字 ORDER BY、GROUP BY、DISTINCT 后面的字段建立索引。

说明

索引创建成功后，系统会自动判断何时引用索引。当系统认为使用索引比顺序扫描更快时，就会使用索引。

索引创建成功后，必须和表保持同步以保证能够准确地找到新数据，这样就增加了数据操作的负荷。因此请定期删除无用的索引。

分区表索引分为 LOCAL 索引与 GLOBAL 索引，一个 LOCAL 索引对应一个具体分区，而 GLOBAL 索引则对应整个分区表。操作创建分区表的参考创建和管理分区表。

创建普通表索引

GBase 8c 支持 4 种创建索引的方式请参见下表。

表 8-3 索引方式

索引方式	描述
唯一索引	可用于约束索引属性值的唯一性，或者属性组合值的唯一性。如果一个表声明了唯一约束或者主键，则 GBase 8c 自动在组成主键或唯一约束的字段上创建唯一索引（可能是多字段索引），以实现这些约束。目前，GBase 8c 只有 B-Tree 可以创建唯一索引。
多字段索引	一个索引可以定义在表中的多个属性上。目前，GBase 8c 中的 B-Tree 支持多字段索引，。
部分索引	建立在一个表的子集上的索引，这种索引方式只包含满足条件表达式的元组。
表达式索引	索引建立在一个函数或者从表中一个或多个属性计算出来的表达式上。表达式索引只有在查询时使用与创建时相同的表达式才会起作用。

- 创建一个普通表

```
postgres=# CREATE TABLE tpcds.customer_address_bak AS TABLE tpcds.customer_address;
INSERT 0 0
```

- 创建普通索引

  如果对于 tpcds.customer_address_bak 表，需要经常进行以下查询。

```
postgres=# SELECT ca_address_sk FROM tpcds.customer_address_bak WHERE
ca_address_sk=14888;
```

通常，数据库系统需要逐行扫描整个 tpcds.customer_address_bak 表以寻找所有匹配的元组。如果表 tpcds.customer_address_bak 的规模很大，但满足 WHERE 条件的只有少数几个（可能是零个或一个），则这种顺序扫描的性能就比较差。如果让数据库系统在 ca_address_sk 属性上维护一个索引，用于快速定位匹配的元组，则数据库系统只需要在搜索树上查找少数的几层就可以找到匹配的元组，这将会大大提高数据查询的性能。同样，在数据库中进行更新和删除操作时，索引也可以提升这些操作的性能。

```
postgres=# CREATE INDEX index_wr_returned_date_sk ON pcds.customer_address_bak
(ca_address_sk);
CREATE INDEX
```

使用以下命令创建索引。

```
postgres=# CREATE INDEX index_wr_returned_date_sk ON
tpcds.customer_address_bak(ca_address_sk);
CREATE INDEX
```

- 创建唯一索引

  在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建唯一索引。

```
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
```

- 创建多字段索引

假如用户需要经常查询表 tpcds.customer_address_bak 中 ca_address_sk 是 5050， 且 ca_street_number 小于 1000 的记录，使用以下命令进行查询。

```
postgres=# SELECT ca_address_sk,ca_address_id FROM tpcds.customer_address_bak
WHERE ca_address_sk = 5050 AND ca_street_number < 1000;
```

使用以下命令在字段 ca_address_sk 和 ca_street_number 上定义一个多字段索引。

```
postgres=# CREATE INDEX more_column_index ON
tpcds.customer_address_bak(ca_address_sk ,ca_street_number );
CREATE INDEX
```

- 创建部分索引

  如果只需要查询 ca_address_sk 为 5050 的记录，可以创建部分索引来提升查询效率。

```
postgres=# CREATE INDEX part_index ON tpcds.customer_address_bak(ca_address_sk)
WHERE ca_address_sk = 5050;
CREATE INDEX
```

- 创建表达式索引

  假如经常需要查询 ca_street_number 小于 1000 的信息，执行如下命令进行查询。

```
postgres=# SELECT * FROM tpcds.customer_address_bak WHERE runc(ca_street_number) <
1000;
```

  可以为上面的查询创建表达式索引：

```
postgres=# CREATE INDEX para_index ON tpcds.customer_address_bak
(trunc(ca_street_number));
CREATE INDEX
```

- 创建分区表索引

  - 创建分区表 LOCAL 索引 tpcds_web_returns_p2_index1，不指定索引分区的名称。

```
postgres=# CREATE INDEX tpcds_web_returns_p2_index1 ON tpcds.web_returns_p2
(ca_address_id) LOCAL;
```

  当结果显示为如下信息，则表示创建成功。

```
CREATE INDEX
```

  - 创建分区表 LOCAL 索引 tpcds_web_returns_p2_index2，并指定索引分区的名称。

```
postgres=# CREATE INDEX tpcds_web_returns_p2_index2 ON tpcds.web_returns_p2
(ca_address_sk) LOCAL
(
PARTITION web_returns_p2_P1_index,
PARTITION web_returns_p2_P2_index TABLESPACE example3, PARTITION
web_returns_p2_P3_index TABLESPACE example4, PARTITION web_returns_p2_P4_index,
PARTITION web_returns_p2_P5_index, PARTITION web_returns_p2_P6_index, PARTITION
web_returns_p2_P7_index, PARTITION web_returns_p2_P8_index
) TABLESPACE example2;
```

  当结果显示为如下信息，则表示创建成功。

```
CREATE INDEX
```

  - 创建分区表 GLOBAL 索引 tpcds_web_returns_p2_global_index。

```
CREATE INDEX tpcds_web_returns_p2_global_index ON tpcds.web_returns_p2
(ca_street_number) GLOBAL;
```

- 修改索引分区的表空间
  - 修改索引分区 web_returns_p2_P2_index 的表空间为 example1。

```
postgres=# ALTER INDEX tpcds.tpcds_web_returns_p2_index2 MOVE PARTITION
web_returns_p2_P2_index TABLESPACE example1;
```

当结果显示为如下信息，则表示修改成功。

```
ALTER INDEX
```

  - 修改索引分区 web_returns_p2_P3_index 的表空间为 example2。

```
postgres=# ALTER INDEX tpcds.tpcds_web_returns_p2_index2 MOVE PARTITION
web_returns_p2_P3_index TABLESPACE example2;
```

当结果显示为如下信息，则表示修改成功。

```
ALTER INDEX
```

- 重命名索引分区

执行如下命令对索引分区进行重命名操作。如将 web_returns_p2_P8_index 重命名为 web_returns_p2_P8_index_new。

```
postgres=# ALTER INDEX tpcds.tpcds_web_returns_p2_index2 RENAME PARTITION
web_returns_p2_P8_index TO web_returns_p2_P8_index_new;
```

当结果显示为如下信息，则表示重命名成功。

```
ALTER INDEX
```

- 查询索引
  - 执行如下命令查询系统和用户定义的所有索引。

```
postgres=# SELECT RELNAME FROM PG_CLASS WHERE RELKIND='i' or RELKIND='I';
```

  - 执行如下命令查询指定索引的信息。

```
postgres=#    \di+ tpcds.tpcds_web_returns_p2_index2
```

- 删除索引

```
postgres=# DROP INDEX tpcds.tpcds_web_returns_p2_index1;
postgres=# DROP INDEX tpcds.tpcds_web_returns_p2_index2;
```

当结果显示为如下信息，则表示删除成功。

```
DROP INDEX
```

- 删除 tpcds.customer_address_bak 表

```
postgres=# DROP TABLE tpcds.customer_address_bak;
DROP TABLE
```

说明

- 索引创建成功后，系统会自动判断何时引用索引。当系统认为使用索引比顺序扫描更快时，就会使用索引。
- 索引创建成功后，必须和表保持同步以保证能够准确地找到新数据，这样就增加了数据操作的负荷。因此请定期删除无用的索引。

8.5 创建和管理视图

背景信息

当用户对数据库中的一张或者多张表的某些字段的组合感兴趣，而又不想每次键入这些查询时，用户就可以定义一个视图，以便解决这个问题。

视图与基本表不同，不是物理上实际存在的，是一个虚表。数据库中仅存放视图的定义，而不存放视图对应的数据，这些数据仍存放在原来的基本表中。若基本表中的数据发生变化，从视图中查询出的数据也随之改变。从这个意义上讲，视图就像一个窗口，透过它可以看到数据库中用户感兴趣的数据及变化。视图每次被引用的时候都会运行一次。

管理视图

- 创建视图

```
postgres=# CREATE OR REPLACE VIEW MyView AS SELECT * FROM tpcds.web_returns
WHERE
trunc(wr_refunded_cash) > 10000;
CREATE VIEW
```

说明

- CREATE VIEW 中的 OR REPLACE 可有可无，当存在 OR REPLACE 时，表示若以前存在该视图就进行替换。

- 查询视图内容

```
postgres=# SELECT * FROM MyView;
```

- 查看某视图的具体信息

```
postgres=# \d+ MyView
View "PG_CATALOG.MyView"
Column |       Type| Modifiers | Storage | Description
---------+----------------------+-----------+----------+------------- USERNAME    | CHARACTER
VARYING(64) |    | extended | View definition:
SELECT PG_AUTHID.ROLNAME::CHARACTER VARYING(64) AS USERNAME FROM
PG_AUTHID;
```

- 删除视图

```
postgres=# DROP VIEW MyView;
DROP VIEW
```

8.6 创建和管理序列

背景信息

序列 Sequence 是用来产生唯一整数的数据库对象。序列的值是按照一定规则自增的整数。因为自增所以不重复，因此说 Sequence 具有唯一标识性。这也是 Sequence 常被用作主键的原因。

通过序列使某字段成为唯一标识符的方法有两种：

- 一种是声明字段的类型为序列整型，由数据库在后台自动创建一个对应的 Sequence。
- 另一种是使用 CREATE SEQUENCE 自定义一个新的 Sequence，然后将 nextval('sequence_name')函数读取的序列值，指定为某一字段的默认值，这样该字段就可以作为唯一标识符。

操作步骤

- 声明字段类型为序列整型来定义标识符字段。例如：

```
postgres=# CREATE TABLE T1 (id serial, name text);
```

当结果显示为如下信息，则表示创建成功。

```
CREATE TABLE
```

- 创建序列，并通过 nextval('sequence_name')函数指定为某一字段的默认值。

① 创建序列

```
postgres=# CREATE SEQUENCE seq1 cache 100;
```

当结果显示为如下信息，则表示创建成功。

```
CREATE SEQUENCE
```

② 指定为某一字段的默认值，使该字段具有唯一标识属性。

```
postgres=# CREATE TABLE T2 (id int not null default nextval('seq1'), name text);
```

当结果显示为如下信息，则表示默认值指定成功。

```
CREATE TABLE
```

③ 指定序列与列的归属关系。

将序列和一个表的指定字段进行关联。这样，在删除那个字段或其所在表的时候会自动删除已关联的序列。

```
postgres=# ALTER SEQUENCE seq1 OWNED BY T2.id;
```

当结果显示为如下信息，则表示指定成功。

```
ALTER SEQUENCE
```

说明

- 除了为序列指定了 cache，方法二所实现的功能基本与方法一类似。但是一旦定义 cache，序列将会产生空洞(序列值为不连贯的数值，如：1.4.5)，并且不能保序。另外为某序列指定从属列后，该列删除，对应的 sequence 也会被删除。 虽然数据库并不限制序列只能为一列产生默认值，但最好不要多列共用同一个序列。
- 当前版本只支持在定义表的时候指定自增列，或者指定某列的默认值为 nextval('seqname')， 不支持在已有表中增加自增列或者增加默认值为 nextval('seqname')的列。

9 访问外部数据库

9.1 oracle_fdw

oracle_fdw（foreign data wrapper for oracle）用于 Oracle 的外部数据包装器，是一款开源插件。

配置客户端

使用 oracle_fdw 前需要完成以下客户端配置。

选择合适的运行环境和版本，下载 Basic Package、SDK package、SQLPlus Package 对应版本的安装包，三者版本号必须一致。

下载链接：

https://www.oracle.com/database/technologies/instant-client/downloads.html

传至数据库主节点，并以 gbase 用户身份解压文件。例如，在/home/gbase.tools/oracle_fdw 路径下解压：

```
# su gbase
$ unzip instanclient-basic-linux.x86-21.5.0.0.0dbru.zip
$ unzip instanclient-sdk-linux.x86-21.5.0.0.0dbru.zip
$ unzip instanclient-sqlplus-linux.x86-21.5.0.0.0dbru.zip
```

安装后配置环境变量，即可使用 oracle_fdw。根据实际情况修改 ORACLE_HOME 具体目录。例如：

```
$ export ORACLE_HOME=/home/gbase/tools/oracle_fdw/instantclient_21_5
$ export LD_LIBRARY_PATH=$ORACLE_HOME:$LD_LIBRARY_PATH
$ export TNS_ADMIN=$ORACLE_HOME/network/admin
```

执行以下命令，使环境变量生效。

```
$ source ~/.bashrc
```

通过客户端连接 Oracle Server，进行配置。

```
$ sqlplus
```

进入 Oracle 客户端目录的 network/admin 目录，添加 tnsnames.ora 文件。

```
$ touch tnsnames.ora
$ vim tnsnames.ora
```

新增文件具体内容如下（host、port、sid、instance_name 请以实际情况为准）：

```
orcl =
    (DESCRIPTION =
        (ADDRESS_LIST =
            (ADDRESS = (PROTOCOL = TCP) (HOST = ip) (PORT = port))
        )
        (CONNECT_DATA =
            (SID = sid)
            (SERVER = DEDICATED)
            (INSTANCE_NAME = instance_name)
        )
    )
```

在 Oracle Server 端添加当前机器的访问权限，进入 product/network/admin 目录，修改 tnsname.ora 配置文件，添加内容如下：

```
    (ADDRESS = (PROTOCOL = TCP) (HOST = ip) (PORT = port))
    (CONNECT_DATA = (SERVER = DEDICATED)（SERVER_NAME = sid）
```

切换 root 用户，在每个节点上执行命令：

```
$ su root
# yum install -y numactl
# yum install -y ndctl
```

其中 Oracle 客户端基础包内含 libclntsh.so.21.1、libnnz21.so、libclntshcore.so.21.1、libociei.so 四个依赖包，需要将其复制到/lib64 目录下。并重启数据库生效。

```
# cp libclntsh.so.21.1 libnnz21.so libclntshcore.so.21.1 libociei.so /lib64
# su gbase
$ gs_om -t restart
```

使用 oracle_fdw

使用 oracle_fdw 需要连接 Oracle，Oracle server 请自行安装。

加载 oracle_fdw 插件

```
CREATE EXTENSION oracle_fdw;
```

创建使用用户并授权

```
CREATE USER user_name IDENTIFIED BY 'password';
GRANT USAGE ON FOREIGN DATA WRAPPER oracle_fdw TO user_name;
```

创建服务器对象

```
CREATE SERVER server_name foreign data wrapper oracle_fdw options(dbserver
'host_ip:port/servername');
```

创建用户映射

```
CREATE USER MAPPING for user_name server server_name options(user 'user',password 'password');
```

创建外部表

```
CREATE FOREIGN TABLE table_name(col_name col_type) server server_name options(schema 'schema_name',table 'table', prefetch 'value');
```

其中，外表的表结构需要与 Oracle 数据库中的表结构保持一致。注意 Oracle server 侧的表的第一个字段必须具有唯一性约束（如 PRIMARY KEY、UNIQUE 等）。

对外表做正常的操作，如 INSERT、UPDATE、DELETE、SELECT、EXPLAIN、ANALYZE、COPY 等。

删除外表

```
DROP FOREIGN TABLE table_name;
```

删除用户映射

```
DROP USER MAPPING;
```

删除服务器对象

```
DROP SERVER server_name CASCADE;
```

删除扩展

```
DROP EXTENSION oracle_fdw;
```

常见问题

- 在 GBase 8c 上建立外表时，不会在 Oracle 数据库中同步建表，需要自行在 Oracle 数据库中建表。
- 执行 CREATE USER MAPPING 时使用的 Oracle 用户需要有远程连接 Oracle 数据库及对表相关操作的权限。使用外表前，可以在 GBase 8c server 所在的机器上，使用 Oracle 的客户端，使用对应的用户名密码确认能否成功连接 Oracle 并进行操作。
- 执行 CREATE Extension oracle_fdw;时，出现 libclntsh.so: cannot open shared object file: No such file or directory。原因是 Oracle 的开发库 libclntsh.so 不在系统的相关路径中，可以先找到 libclntsh.so 的具体路径，然后将该 so 文件所在的文件夹加到/etc/ld.so.conf 中。比如 libclntsh.so 的路径为/usr/lib/oracle/11.2/client64/lib/libclntsh.so.11.1，那么就将该文件的路径/usr/lib/oracle/11.2/client64/lib/加到/etc/ld.so.conf 文件末尾。然后执行 ldconfig 使修改生效即可。注意此操作需要 root 权限。

注意事项

- 两个 Oracle 外表间的 SELECT JOIN 不支持下推到 Oracle server 执行，会被分成两条 SQL 语句传递到 Oracle 执行，然后在 GBase 8c 处汇总处理结果。
- 不支持 IMPORT FOREIGN SCHEMA 语法。
- 不支持对外表进行 CREATE TRIGGER 操作。

9.2 mysql_fdw

mysql_fdw（foreign data wrapper for mysql）是 GBase 8c 用于 MySQL 的外部数据包装器。GBase 8c 基于开源的 mysql_fdw Release 2.5.3 版本进行开发适配。

配置 mysql_fdw

mysql_fdw 同样已集成在 GBase 8c 的安装包中。支持 MariaDB 环境，需要安装客户端，并配置环境变量即可。

选择合适的运行环境和版本，下载对应安装包，并通过执行 rpm -ivh 命令安装对应文件。MariaDB-client 是 MySQL/MariaDB 的客户端工具，也可以根据需要安装，用于连接 MySQL/MariaDB 进行测试。

安装后配置环境变量，即可使用 mysql_fdw。例如：

```
export LD_LIBRARY_PATH=/usr/lib/mysql/lib:LD_LIBRARY_PATH
```

执行 source ~/.bashrc 使环境变量生效。

创建用于访问 mysql_fdw 的用户

```
gs_guc generate -U user_name -S password -D dir -o usermapping
```

使用 mysql_fdw

使用 mysql_fdw 需要连接 MySQL/MariaDB。MySQL 数据库和实例请自行安装准备。

加载 mysql_fdw 扩展

```
CREATE EXTENSION mysql_fdw;
```

授予用户所有权限

```
GRANT ALL PRIVILEGES TO user_name;
```

授予用户访问 mysql_fdw 权限

```
GRANT USAGE ON FOREIGN DATA WRAPPER MYSQL_FDW TO user_name;
```

创建服务器对象

```
CREATE SERVER server_name FOREIGN DATA WRAPPER MYSQL_FDW OPTIONS(host 'host_ip', port 'port');
```

创建用户映射

```
CREATE USER MAPPING FOR user_name server server_name options (username 'user',password 'password');
```

创建外表

```
CREATE FOREIGN TABLE table_name(col_name col_type) server server_name options(dbname 'dbname',table_name 'table_name');
```

其中，外表的表结构需要与 MySQL/MariaDB 侧的表结构保持一致。注意 MySQL/MariaDB 侧的表的第一个字段必须具有唯一性约束（如 PRIMARY KEY、UNIQUE 等）。

对外表做正常的操作，如 INSERT、UPDATE、DELETE、SELECT、EXPLAIN、ANALYZE、COPY 等。

删除外表

```
DROP FOREIGN TABLE table_name;
```

删除用户映射

```
DROP USER MAPPING;
```

删除服务器对象

```
DROP SERVER server_name;
```

删除扩展

```
DROP EXTENSION mysql_fdw;
```

常见问题

- 在 GBase 8c 上建立外表时，不会同步在 MariaDB/MySQL Server 上建表，需要自己利用 MariaDB/MySQL Server 的客户端连接 MariaDB/MySQL Server 建表。
- 创建 USER MAPPING 时使用的 MariaDB/MySQL Server 用户需要有远程连接 MariaDB/MySQL Server 及对表相关操作的权限。使用外表前，可以在 GBase 8c server 所在的机器上，使用 MariaDB/MySQL Server 的客户端，使用对应的用户名密码确认能否成功连接 MariaDB/MySQL Server 并进行操作。
- 对外表执行 DML 操作时，出现 Can't initialize character set SQL_ASCII (path: compiled_in)

错误。由于 MariaDB 不支持 SQL_ASCII 编码格式，目前只能通过修改 GBase 8c database 的编码格式解决该问题。修改 database 编码格式的方式为 update pg_database set encoding = pg_char_to_encoding('UTF-8') where datname = 'postgres'; datname 根据实际情况填写。注意修改完编码格式后，需要重新开启一个 gsql 会话，才能使 mysql_fdw 使用更新后的参数。也可以通过在执行 gs_initdb 时，使用** -locale=LOCALE**，指定默认的编码格式为非 SQL_ASCII 编码。

注意事项

- 两个 mysql 外表间的 SELECT JOIN 不支持下推到 MariaDB/MySQL Server 执行，会被分成两条 SQL 语句传递到 MariaDB/MySQL Server 执行，然后在 GBase 8c 处汇总处理结果。
- 不支持 IMPORT FOREIGN SCHEMA 语法。
- 不支持对外表进行 CREATE TRIGGER 操作。

9.3 postgres_fdw

postgres_fdw 是一款开源插件，其代码随 PostgreSQL 源码一同发布。GBase 8c 基于开源的 postgres_fdw 源码进行开发适配。

postgres_fdw 插件默认参与编译，使用安装包安装好 GBase 8c 后，可直接使用 postgres_fdw，无须其他操作。

使用 postgres_fdw

加载 postgres_fdw 扩展

```
CREATE EXTENSION postgres_fdw;
```

授予用户 postgres_fdw 的使用权限

```
GRANT USAGE ON FOREIGN DATA WRAPPER postgres_fdw TO user_name;
```

创建服务器对象

```
CREATE SERVER server_name FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'host_ip', port 'port', dbname 'dbname');
```

授予用户外部服务器的使用权限

```
GRANT USAGE ON FOREIGN SERVER server_name TO user_name;
```

创建用户映射

```
CREATE USER MAPPING FOR user_name SERVER server_name OPTIONS (user 'user',
password 'password');
```

创建外表

```
CREATE FOREIGN TABLE table_name (col_name col_type) SERVER server_name
OPTIONS (schema_name 'schema', table_name 'table');
```

其中，外表的表结构需要与远端 GBase 8c 侧的表结构保持一致。

对外表做正常的操作，如 INSERT、UPDATE、DELETE、SELECT、EXPLAIN、ANALYZE、COPY 等。

删除外表

```
DROP FOREIGN TABLE table_name;
```

删除用户映射

```
DROP USER MAPPING;
```

删除服务器对象

```
DROP SERVER server_name;
```

删除扩展

```
DROP EXTENSION postgres_fdw;
```

- 查看 REMOTE SQL PLAN：开启 GUC 参数 show_fdw_remote_plan，可以在 explain 时追加打印 remote sql 的计划，对 remote sql 的 explain 会继承 explain 语句的 verbose、costs 选项。

postgres_fdw 下推主要成分

支持的下推成分：

- 稳定表达式：如常量、表的非系统列、内置的稳定级别的函数与操作符、等稳定元素组成的表达式。
- 扫描：支持单表查询，直接查询非系统列，带有简单且稳定的 where 表达式的查询成分的下推。
- 连接：当两个外表均在一个外表服务器，且其访问权限以及连接条件、过滤条件均满足一定的要求时，可以下推到远端执行。
- 分组与聚集：当聚集函数、group by、having 三个部分，满足表达式检查以及其它一些条件时，可以下推到远端执行。

- 排序：当排序键满足一定的条件时，可下推到远端执行。
- LIMIT：当 LIMIT 表达式满足下推条件时，可以下推到远端执行。
- ROWMARK：rowmark 一般都可下推。

不支持的下推成分：

- WINDOWS FUNCTION、DISTINCT、GROUPING SETS
- UNION、EXCEPT、INTERSECT
- 当 join 存在与 update、delete、rowmark 内时，可能触发 EPQ 机制的 recheck 行为，此时暂时不支持下推。
- 其他不满足上述下推要求的成分，如系统列等。

常见问题

- 在 GBase 8c 上建立外表时，不会同步在远端的 GBase 8c 上建表，需要自己利用客户端连接远端 GBase 8c 建表。
- 外表并不会区分远端表的具体表类型，不会检查表结构等是否对应，甚至连是否存在都不会检查。需要用户自己维护与保证这些属性关系。
- 执行 CREATE USER MAPPING 时使用的 GBase 8c 用户需要有远程连接 GBase 8c 及对表相关操作的权限。使用外表前，可以在本地机器上，使用 gsql 的客户端，使用对应的用户名密码确认能否成功连接远端 GBase 8c 并进行操作。

注意事项

- SQL 各类算子的执行具有一定的顺序，当某个算子不能下推后，上层所有后续的算子也都无法下推。
- 不支持 IMPORT FOREIGN SCHEMA 语法。
- 不支持对外表进行 CREATE TRIGGER 操作。
- 外表不支持以分区表的形式创建，不支持映射到某一个具体的分区。

9.4 file_fdw

file_fdw 模块提供了外部数据封装器 file_fdw，可以用来在服务器的文件系统中访问数据文件。数据文件必须是 COPY FROM 可读的格式；具体可参照 COPY 语句的介绍。访问这样的数据文件当前只是可读的。当前不支持对该数据文件的写入操作。

当前 GBase 8c 会默认编译 file_fdw，在 initdb 的时候会在 pg_catalog schema 中创建该插件。

使用 file_fdw 创建的外部表可以有下列选项：

- filename

  指定要读取的文件，必需的参数，且必须是一个绝对路径名。

- format

  远端 server 的文件格式，支持 text/csv/binary 三种格式，和 COPY 语句的 FORMAT 选项相同。

- header

  指定的文件是否有标题行，与 COPY 语句的 HEADER 选项相同。

- delimiter

  指定文件的分隔符，与 COPY 的 DELIMITER 选项相同。

- quote

  指定文件的引用字符，与 COPY 的 QUOTE 选项相同。

- escape

  指定文件的转义字符，与 COPY 的 ESCAPE 选项相同。

- null

  指定文件的 null 字符串，与 COPY 的 NULL 选项相同。

- encoding

  指定文件的编码，与 COPY 的 ENCODING 选项相同。

- force_not_null

  这是一个布尔选项。如果为真，则声明字段的值不应该匹配空字符串（也就是， 文件级别 null 选项）。与 COPY 的 FORCE_NOT_NULL 选项里的字段相同。

file_fdw 不支持 COPY 的 OIDS 和 FORCE_QUOTE 选项。

注意这些选项只能为外部表或它的字段声明，不是在 file_fdw 外部数据封装器的选项里，也不是在使用该封装器的服务器或用户映射的选项里。

修改表级别的选项需要系统管理员权限，因为安全原因：只有系统管理员用户能够决定

读哪个文件。

对于一个使用 file_fdw 的外部表，EXPLAIN 显示要读取的文件名。除非指定了 COSTS OFF，否则也显示文件大小（字节计）。

使用 file_fdw

使用 file_fdw 需要指定要读取的文件，请先准备好该文件，并让数据库有读取权限。

- 创建服务器对象：CREATE SERVER
- 创建用户映射：CREATE USER MAPPING
- 创建外表：CREATE FOREIGN TABLE 外表的表结构需要与指定的文件的数据保持一致。
- 对外表做查询操作，写操作不被允许。
- 删除外表：DROP FOREIGN TABLE
- 删除用户映射：DROP USER MAPPING
- 删除服务器对象：DROP SERVER

注意事项

不支持 DROP Extension file_fdw 操作。

9.5 dblink

dblink 可用于 GBase 8c 和外部数据库之间的会话连接，目前支持 GBase 8c 和 Oracle。同 libpq 支持的连接参数一致。

注意

- 不支持 GBase 8c 数据库访问 PostgreSQL 数据库。
- 仅在 GBase 8c 兼容模式（dbcompatibility）为 A 场景下支持，即 O 兼容模式。

语法

- 加载 dblink 扩展

```
CREATE EXTENSION dblink;
```

- 打开一个到远程数据库的持久连接

```
SELECT dblink_connect(text connstr);
```

- 关闭一个到远程数据库的持久连接

```
SELECT dblink_disconnect();
```

- 在远程数据库执行查询

```
SELECT * FROM dblink(text connstr, text sql);
```

- 在远程数据库执行命令

```
SELECT dblink_exec(text connstr, text sql);
```

- 返回所有打开的命名 dblink 连接的名称

```
SELECT dblink_get_connections();
```

- 发送一个异步查询到远程数据库

```
SELECT dblink_send_query(text connname, text sql);
```

- 检查连接是否正在忙于一个异步查询

```
SELECT dblink_is_busy(text connname);
```

- 删除扩展

```
DROP EXTENSION dblink;
```

使用流程

(1) 当被访问数据库为 Oracle 时，准备：

创建用户，授予连接权限：

```
create user c##u_link identified by "password";
grant create session, connect, resource to c##u_link;
```

连接，创建数据：

```
conn u_link/password
create table t1 (id int primary key, name varchar(20));
insert into t1 values (1, 'zhangsan');
```

(2) 创建 dblink 远程连接需要创建用户密钥文件：

```
gs_ssh -c "gs_guc generate -S 'password' -D $GAUSSHOME/bin -o usermapping"
# 其中-S 表示自定义密钥，可将 password 替换为自定义密钥
# 创建之后建议就不要做改动了，如果已经创建了 dblink 之后，再使用改语句重新创建密钥文件，会导致已创建的 dblink 失效无法使用
# 如果是单个节点的话，也可以直接执行：gs_guc generate -S 'password' -D $GAUSSHOME/bin -o usermapping
```

(3) 用户创建

```
create user test with sysadmin createrole auditadmin poladmin password 'gbase;123';
create database tdb with template = template0 owner = test encoding = 'UTF-8' LC_COLLATE = 'en_US.UTF-8' LC_CTYPE = 'en_US.UTF-8' dbcompatibility = 'A';
-- 连接 Oracle 时，如果本地数据库的编码格式 Oracle 不支持的话，会报 warning，建议使用 utf-8 编码格式，否则可能出现数据无法识别的问题（目前有问题的编码格式有：SQL_ASCII、LATIN10、WIN874、MULE_INTERNAL）
\c tdb
\c - test
```

(4) 创建插件：

```
create extension oracle_fdw;
create extension postgres_fdw;
```

(5) 使用 dblink 的用户需要使用 postgres_fdw / oracle_fdw 的权限，可使用如下语句：

```
grant usage on foreign data wrapper postgres_fdw to user_name;
grant usage on foreign data wrapper oracle_fdw to user_name;
```

(6) 创建 dblink

```
create [public] database link link_name connect to user_name identified by "password" using 'oracle_fdw://host:port/service_name';
-- 其中 public 可以忽略，oracle_fdw 是在连接 Oracle 时必须要带的，host、port 代表 ip 和端口，service_name 表示 Oracle 远程数据库的实例名。
-- 创建 dblink 连接后需要等待约 10 秒，需要拉取远程表结构信息
```

(7) 使用

```
select * from t1@link_name;
insert into t1 values (1, 'zhangsan');
-- 连接 Oracle 目前只支持查询创建连接的用户的 schema 下面的表，暂不支持查询别的 schema 下的表
-- 目前在连接 Oracle 数据库时，如果要使用的表没有主键，执行 update、delete 操作会失败
-- 连接 GBase 8c 如果不指定 schema 名，会默认使用用户同名的 schema，要查询别的 schema，需要指定，如下：
select * from schema1.t2@link_name;
```

如果远程数据库的表结构发生变化或者新建表，由于 dblink 基于外部表功能实现，需要有要查询的表的结构才可以正常查询，否则会出现问题，目前提供两种给解决方案:

a. 使用 sync_dblink_schema('dblink_name')函数同步一下本地存储的外部表结构，使用示例：
“select * from sync_dblink_schema('link_name');”

b. 设置参数 whale.auto_sync_dblink 为 true，可以在远程数据库发生表结构变化时自动同步表结构信息，由于设置这个参数后，每次操作表都会查询表的结构信息，会有性能损耗，不建议打开改参数，连接 Oracle 时暂不支持打开该参数

当同步表结构信息时，如果表结构发生变化，会重建本地的外部表，会报一些提示信息，忽略即可，除非是 error 级别信息

10 管理数据库安全

10.1 客户端接入认证

10.1.1配置客户端接入认证

背景信息

如果主机需要远程连接数据库，必须在数据库系统的配置文件中增加此主机的信息，并且进行客户端接入认证。配置文件（默认名称为 pg_hba.conf）存放在数据库的数据目录里。hba（host-based authentication）表示是基于主机的认证。

- GBase 8c 支持如下三种认证方式，这三种方式都需要配置 pg_hba.conf 文件。
  - 基于主机的认证：服务器端根据客户端的 IP 地址、用户名及要访问的数据库来查看配置文件从而判断用户是否通过认证。
  - 口令认证：包括远程连接的加密口令认证和本地连接的非加密口令认证。
  - SSL 加密：使用 OpenSSL（开源安全通信库）提供服务器端和客户端安全连接的环境。
- pg_hba.conf 文件的格式是一行写一条信息，表示一个认证规则，空白和注释（以#开头）被忽略。
- 每个认证规则是由若干空格和/，空格和制表符分隔的字段组成。如果字段用引号 包围，则它可以包含空白。一条记录不能跨行存在。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 配置客户端认证方式，允许客户端以"jack”用户连接到本机，此处远程连接禁止使用"gbase”用户（即数据库初始化用户）。

例如，下面示例中配置允许 IP 地址为 10.10.0.30 的客户端访问本机。

```
gs_guc set -N all -I all -h "host all jack 10.10.0.30/32 sha256"
```

说明

- 使用"jack”用户前，需先本地连接数据库，并在数据库中使用如下语句建立"jack”用户：

```
CREATE USER jack PASSWORD 'Test@123';
```

- -N all 表示 GBase 8c 的所有主机。
- -I all 表示主机的所有实例。
- -h 表示指定需要在"pg_hba.conf”增加的语句。
- all 表示允许客户端连接到任意的数据库。
- jack 表示连接数据库的用户。
- 10.10.0.30/32 表示只允许 IP 地址为 10.10.0.30 的主机连接。此处的 IP 地址不能为 GBase 8c 内的 IP，在使用过程中，请根据用户的网络进行配置修改。32 表示子网掩码为 1 的位数，即 255.255.255.255。
- sha256 表示连接时 jack 用户的密码使用 sha256 算法加密。

这条命令在数据库主节点实例对应的"pg_hba.conf”文件中添加了一条规则，用于对连接数据库主节点的客户端进行鉴定。

pg_hba.conf 文件中的每条记录可以是下面四种格式之一，四种格式的参数说明请参见配置文件参考。

```
local       DATABASE USER METHOD [OPTIONS]
host        DATABASE USER ADDRESS METHOD [OPTIONS]
hostssl     DATABASE USER ADDRESS METHOD [OPTIONS]
hostnossl DATABASE USER ADDRESS METHOD [OPTIONS]
```

认证时系统为每个连接请求顺序检查 pg_hba.conf 文件里的记录，所以这些记录的顺序是非常关键的。

对于认证规则的配置建议如下：靠前的记录有比较严格的连接参数和比较弱的认证方法。靠后的记录有比较宽松的连接参数和比较强的认证方法。

说明

- 在配置 pg_hba.conf 文件时，请依据通讯需求按照格式内容从上至下配置记录，优先级高的需求需要配置在前面。GBase 8c 和扩容配置的 IP 优先级最高，用户手动配置的 IP 请放在这二者之后，如果已经进行的客户配置和扩容节点的 IP 在同一网段，请在扩容前删除，扩容成功后再进行配置。
- 一个用户要想成功连接到特定的数据库，不仅需要通过 pg_hba.conf 中的规则检查，还必须要有该数据库上的 CONNECT 权限。如果希望控制某些用户只能连接到指定数据库，赋予/撤销 CONNECT 权限通常比在 pg_hba.conf 中设置规则更为简单。

对应 GBase 8c 外部客户端连接，trust 为不安全的认证方式，请将认证方式设置为 sha256。

异常处理

用户认证失败有很多原因，通过服务器返回给客户端的提示信息，可以看到用户认证失败的原因。常见的错误提示请参见下表。

表 10-1 错误提示

问题现象	解决方法
用户名或密码错误： FATAL: invalid username/password,login denied	这条信息说明用户名或者密码错误，请检查输入是否有误。
连接的数据库不存在： FATAL: database "TESTDB" does not exist	这条信息说明尝试连接的数据库不存在，请检查连接的数据库名输入是否有误。
未找到客户端匹配记录： FATAL: no pg_hba.conf entry for host "10.10.0.60", user "ANDYM", database "TESTDB"	这条信息说明已经连接了服务器，但服务器拒绝了连接请求，因为没有在它的 pg_hba.conf 配置文件里找到匹配的记录。请联系数据库管理员在 pg_hba.conf 配置文件加入用户的信息。
未找到客户端匹配记录： failed to connect 10.10.0.1:12000.	这条信息说明无法连接到指定 IP 和端口的服务器，请联系数据库管理员检查 pg_hba.conf 配置文件里是否有配置对应 IP 白名单。
连接时的用户名不可以包含@字符 @ can't be allowed in username	这条报错说明客户端在连接数据库时使用了包含@的用户名，这是不允许的。

示例

```
TYPE  DATABASE        USER            ADDRESS                 METHOD
"local" is for Unix domain socket connections only
#表示只允许以安装时-U 参数指定的用户从服务器本机进行连接。
local   all             all                                     trust
```

```
IPv4 local connections:
#表示允许 jack 用户从 10.10.0.50 主机上连接到任意数据库，使用 sha256 算法对密码进行加密。
host      all              jack             10.10.0.50/32          sha256
#表示允许任何用户从 10.10.0.0/24 网段的主机上连接到任意数据库，使用 sha256 算法对密码进行加密，并且经过 SSL 加密传输。
hostssl   all              all              10.10.0.0/24           sha256
#表示禁止任何用户从 10.10.0.1/32 网段的主机上连接到任意数据库。
host      all              all              10.10.0.1/32           reject
```

10.1.2配置文件参考

表 10-2 参数说明

参数名称	描述	取值范围
local	表示这条记录只接受通过 Unix 域套接字进行的连接。没有这种类型的记录，就不允许 Unix 域套接字的连接。 只有在从服务器本机使用 gsql 连接且在不指定 -U 参数的情况下，才是通过 Unix 域套接字连接。	-
host	表示这条记录既接受一个普通的 TCP/IP 套接字连接，也接受一个经过 SSL 加密的 TCP/IP 套接字连接。	-
hostssl	表示这条记录只接受一个经过 SSL 加密的 TCP/IP 套接字连接。	用 SSL 进行安全的连接，需要配置申请数字证书并配置相关参数，详细信息请参见用 SSL 进行安全的 TCP/IP 连接。

hostnossl	表示这条记录只接受一个普通的 TCP/IP 套接字连接。	-
DATABAS E	声明记录所匹配且允许访问的数据库。	● all：表示该记录匹配所有数据库。 ● sameuser：表示如果请求访问的数据库和请求的用户同名，则匹配。 ● samerole：表示请求的用户必须是与数据库同名角色中的成员。 ● samegroup：与 samerole 作用完全一致，表示请求的用户必须是与数据库同名角色中的成员。 一个包含数据库名的文件或者文件中的数据库列表：文件可以通过在文件名前面加前缀@来声明。文件中的数据库列表以逗号或者换行符分隔。特定的数据库名称或者用逗号分隔的数据库列表。 说明：值 replication 表示如果请求一个复制链接，则匹配，但复制链接不表示任何特定的数据库。如需使用名为 replication 的数据库，需在 database 列使用记录"replication”作为数据库名。
USER	声明记录所匹配且允许访问的数据库用户。	● all：表明该记录匹配所有用户。 ● +用户角色：表示匹配任何直接或者间接属于这个角色的成员。 说明：+表示前缀符号。 一个包含用户名的文件或者文件中的用户列表：文件可以通过在文件名前面加前缀@来声明。文件中的用户列表以逗号或者换行符分隔。 特定的数据库用户名或者用逗号分隔的用户列表。

ADDRESS	指定与记录匹配且允许访问的 IP 地址范围。	支持 IPv4 和 IPv6，可以使用如下两种形式来表示： ● IP 地址/掩码长度。例如， 10.10.0.0/24 ● IP 地址子网掩码。例如， 10.10.0.0 255.255.255.0 说明：以 IPv4 格式给出的 IP 地址会匹配那些拥有对应地址的 IPv6 连接，比如 127.0.0.1 将匹配 IPv6 地址 ::ffff:127.0.0.1。
METHOD	声明连接时使用的认证方法。	本产品支持如下几种认证方式，详细解释请参见表 10-3： ● trust ● reject ● md5（不推荐使用，默认不支持，可通过 password_encryption_type 参数配置） ● sha256 ● sm3 ● cert ● peer（仅用于 local 模式） 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。

表 10-3 认证方式

认证方式	说明
trust	采用这种认证模式时，本产品只完全信任从服务器本机使用 gsql 且不指定 -U 参数的连接，此时不需要口令。 trust 认证对于单用户工作站的本地连接是非常合适和方便的，通常不适用于多用户环境。如果想使用这种认证方法，可利用文件系统权限限制对服务器的 Unix 域套接字文件的访问。要使用这种限制有两个方法：

	设置参数 unix_socket_permissions 和 unix_socket_group。 设置参数 unix_socket_directory，将 Unix 域套接字文件放在一个经过恰当限制的目录里。 须知：设置文件系统权限只能 Unix 域套接字连接，它不会限制本地 TCP/IP 连接。为保证本地 TCP/IP 安全，GBase 8c 不允许远程连接使用 trust 认证方法。
reject	无条件地拒绝连接。常用于过滤某些主机。
md5	要求客户端提供一个 md5 加密的口令进行认证。 须知： 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。 GBase 8c 保留 md5 认证和密码存储，是为了方便第三方工具的使用。
sha256	要求客户端提供一个 sha256 算法加密的口令进行认证，该口令在传送过程中结合 salt（服务器发送给客户端的随机数）的单向 sha256 加密，增强了安全性。
sm3	要求客户端提供一个 sm3 算法加密口令进行认证，该口令在传送过程中结合 salt (服务器发送给客户端的随机数) 的单项 sm3 的加密，增加了安全性。
cert	客户端证书认证模式，此模式需进行 SSL 连接配置且需要客户端提供有效的 SSL 证书，不需要提供用户密码。 须知：该认证方式只支持 hostssl 类型的规则。
peer	获取客户端所在操作系统用户名，并检查与数据库初始用户名是否一致。此方式只支持数据库初始用户通过 local 模式本地连接，并支持通过配置 pg_ident.conf 建立操作系统用户与数据库初始用户映射关系。 假设操作系统用户名为 gbase，数据库初始用户为 dbAdmin，在 pg_hba.conf 中配置 local 模式为 peer 认证： `local all all peer map=mymap` 其中 map=mymap 指定使用的用户名映射，并在 pg_ident.conf 中添加映射名

	称为 mymap 的用户名映射如下：
	`# MAPNAME SYSTEM-USERNAME PG-USERNAME`
	`mymap gbase dbAdmin`
	说明
	通过 gs_guc reload 方式修改 pg_hba.conf 配置可以立即生效，无需重启数据库。直接编辑修改 pg_ident.conf 配置后下次连接时自动生效，无需重启数据库。

10.1.3用 SSL 进行安全的 TCP/IP 连接

背景信息

GBase 8c 支持 SSL 标准协议（TLS 1.2），SSL 协议是安全性更高的协议标准，它们加入了数字签名和数字证书来实现客户端和服务器的双向身份验证，保证了通信双方更加安全的数据传输。

前提条件

从 CA 认证中心申请到正式的服务器、客户端的证书和密钥。（假设服务器的私钥为 server.key，证书为 server.crt，客户端的私钥为 client.key，证书为 client.crt，CA 根证 书名称为 cacert.pem。）

注意事项

- 当用户远程连接到数据库主节点时，需要使用 sha256 的认证方式。
- 当内部服务器之间连接时，需要使用 trust 的认证方式，支持 IP 白名单认证。

操作步骤

GBase 8c 在数据库部署完成后，默认已开启 SSL 认证模式。服务器端证书，私钥以及根证书已经默认配置完成。用户需要配置客户端的相关参数。

配置 SSL 认证相关的数字证书参数，具体要求请参见表 10-4。

(1) 配置客户端参数。

已从 CA 认证中心申请到客户端默认证书，私钥，根证书以及私钥密码加密文件。假设证书、私钥和根证书都放在"/home/gbase”目录。

双向认证需配置如下参数：

```
export PGSSLCERT="/home/gbase/client.crt"
export PGSSLKEY="/home/gbase/client.key"
export PGSSLMODE="verify-ca"
export PGSSLROOTCERT="/home/gbase/cacert.pem"
```

单向认证需要配置如下参数：

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

(2) 修改客户端密钥的权限。

客户端根证书，密钥，证书以及密钥密码加密文件的权限，需保证权限为 600。如果权限不满足要求，则客户端无法以 SSL 连接到 GBase 8c。

```
chmod 600 client.key
chmod 600 client.crt
chmod 600 client.key.cipher
chmod 600 client.key.rand
chmod 600 cacert.pem
```

须知

- 从安全性考虑，建议使用双向认证方式。

配置客户端环境变量，必须包含文件的绝对路径。

表 10-4 认证方式

认证方式	含义	配置客户端环境变量	维护建议
双向认证（推荐）	客户端验证服务器证书的有效性，同时服务器端也要验证客户端证书的有效性，只有认证成功，连接才能建立。	设置如下环境变量： PGSSLCERT PGSSLKEY PGSSLROOTCERT PGSSLMODE	该方式应用于安全性要求较高的场景。使用此方式时，建议设置客户端的 PGSSLMODE 变量为 verify-ca。确保了网络数据的安全性。
单向认证	客户端只验证服务器证书的有效性，而服务器端不验证客户端证书的有效性。服	设置如下环境变量： PGSSLROOTCERT	为防止基于 TCP 链接的欺骗，建议使用 SSL 证书认证功能。除配置

	务器加载证书信息并发送给客户端，客户端使用根证书来验证服务器端证书的有效性。	PGSSLMODE	客户端根证书外，建议客户端使用PGSSLMODE 变量为verify-ca 方式连接。

相关参考

在服务器端的 postgresql.conf 文件中配置相关参数，详细信息请参见表 10-5。

表 10-5 服务器参数

参数	描述	取值范围
ssl	表示是否启动 SSL 功能。	● on：开启 SSL 功能。 ● off：关闭 SSL 功能。 默认值：on
require_ssl	设置服务器端是否强制要求 SSL 连接。该参数只有当参数 ssl 为 on 时才有效。	● on：服务器端强制要求 SSL 连接。 ● off：服务器端对是否通过 SSL 连接不作强制要求。 默认值：off
ssl_cert_file	指定服务器证书文件，包含服务器端的公钥。服务器证书用以表明服务器身份的合法性，公钥将发送给对端用来对数据进行加密。	请以实际的证书名为准，其相对路径是相对于数据目录的。 默认值：server.crt
ssl_key_file	指定服务器私钥文件，用以对公钥加密的数据进行解密。	请以实际的服务器私钥名称为准，其相对路径是相对于数据目录的。 默认值：server.key
ssl_ca_file	CA 服务器的根证书。此参数可选择配置，需要验证客户端证书的合法性时才需要配置。	请以实际的 CA 服务器根证书名称为准。 默认值：cacert.pem

ssl_crl_file	证书吊销列表，如果客户端证书在该列表中，则当前客户端证书被视为无效证书。	请以实际的证书吊销列表名称为准。 默认值：空，表示没有吊销列表。
ssl_ciphers	SSL 通讯使用的加密算法。	本产品支持的加密算法的详细信息请参见表 10-7。 默认值：ALL，表示允许对端使用产品支持的所有加密算法，但不包含 ADH、LOW、EXP、MD5 算法。
ssl_cert_notify_time	SSL 服务器证书到期前提醒的天数。	请按照需求配置证书过期前提醒天数。 默认值：90

在客户端配置 SSL 认证相关的环境变量，详细信息请参见表 10-6。

说明

- 客户端环境变量的路径以"/home/gbase”为例，在实际操作中请使用实际路径进行替换。

表 10-6 客户端参数

环境变量	描述	取值范围
PGSSLC ERT	指定客户端证书文件，包含客户端的公钥。客户端证书用以表明客户端身份的合法性，公钥将发送给对端用来对数据进行加密。	必须包含文件的绝对路径，如： export PGSSLCERT='/home/gbase/client.crt' 默认值：空
PGSSLK EY	指定客户端私钥文件，用以对公钥加密的数据进行解密。	必须包含文件的绝对路径，如： export PGSSLKEY='/home/gbase/client.key' 默认值：空
PGSSLM ODE	设置是否和服务器进行 SSL 连	取值及含义：

	接协商，以及指定 SSL 连接的优先级。	disable：只尝试非 SSL 连接。 allow：首先尝试非 SSL 连 接，如果连接失败，再尝试 SSL 连接。 prefer：首先尝试 SSL 连接， 如果连接失败，将尝试非 SSL 连接。 require：只尝试 SSL 连接。如果存在 CA 文件，则按设置成 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。 默认值：prefer
PGSSLROOTCERT	指定为客户端颁发证书的根证书文 件，根证书用于验证服务器证书的有效性。	必须包含文件的绝对路径，如： `export PGSSLROOTCERT='/home/gbase/certca.pem'` 默认值：空
PGSSLCRL	指定证书吊销列表文件，用于验证服务器证书是否在废弃证书列表中，如果在，则服务器证书将会被视为无效证书。	必须包含文件的绝对路径，如： `export PGSSLCRL='/home/gbase/sslcrl-file.crl'` 默认值：空
PGSSLCERT	指定客户端证书文件，包含客户端的公钥。客户端证书用以表明客户端身份的合法性，公钥将发送给对端用来对数据进行加密。	必须包含文件的绝对路径，如： `export PGSSLCERT='/home/gbase/client.crt'` 默认值：空

服务器端参数 ssl、require_ssl 与客户端参数 sslmode 配置组合结果如下：

ssl（服务器）	sslmode（客户端）	require_ssl（服务器）	结果
on	disable	on	由于服务器端要求使用 SSL，但客户端针对该连接禁用了 SSL，因此无法建立连接。
	disable	off	连接未加密。
	allow	on	连接经过加密。
	allow	off	连接未加密。
	prefer	on	连接经过加密。
	prefer	off	连接经过加密。
	require	on	连接经过加密。
	require	off	连接经过加密。
	verify-ca	on	连接经过加密，且验证了服务器证书。
	verify-ca	off	连接经过加密，且验证了服务器证书。
	verify-full	on	连接经过加密，且验证了服务器证书和主机名。
	verify-full	off	连接经过加密，且验证了服务器证书和主机名。
off	disable	on	连接未加密。
	disable	off	连接未加密。
	allow	on	连接未加密。
	allow	off	连接未加密。
	prefer	on	连接未加密。

	prefer	off	连接未加密。
	require	on	由于客户端要求使用 SSL，但服务器端禁用了 SSL，因此无法建立连接。
	require	off	由于客户端要求使用 SSL，但服务器端禁用了 SSL，因此无法建立连接。
	verify-ca	on	由于客户端要求使用 SSL，但服务器端禁用了 SSL，因此无法建立连接。
	verify-ca	off	由于客户端要求使用 SSL，但服务器端禁用了 SSL，因此无法建立连接。
	verify-full	on	由于客户端要求使用 SSL，但服务器端禁用了 SSL，因此无法建立连接。
	verify-full	off	由于客户端要求使用 SSL，但服务器端禁用了 SSL，因此无法建立连接。

SSL 传输支持一系列不同强度的加密和认证算法。用户可以通过修改 postgresql.conf 中的 ssl_ciphers 参数指定数据库服务器使用的加密算法。目前 GBase 8c SSL 支持的加密算法如表 10-7 所示。

表 10-7 加密算法套件

OpenSSL 套件名	IANA 套件名	安全程度
ECDHE-RSA-AES128-GCM-SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	HIGH
ECDHE-RSA-AES256-GCM-SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	HIGH
ECDHE-ECDSA-AES128-	TLS_ECDHE_ECDSA_WITH_AES_	HIGH

GCM-SHA256	128_GCM_SHA256	
ECDHE-ECDSA-AES256-GCM-SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	HIGH
DHE-RSA-AES128-GCM-SHA256	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	HIGH
DHE-RSA-AES256-GCM-SHA384	TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	HIGH

说明

- 目前只支持上表中的 6 种加密算法套件。
- 配置参数 ssl_ciphers 的默认值为 ALL，表示支持上表中的所有加密算法。如果对加密算法没有特殊要求，建议用户使用该默认值。为保持前向兼容保留了 DHE 算法套件，非兼容场景不推荐使用。
- 如需指定以上加密算法套件，可以设置 ssl_ciphers 为上表中 OpenSSL 套件名称，加密算法套件之间需要使用分号分割，如在 postgresql.conf 设置：

```
ssl_ciphers='ECDHE-RSA-AES128-GCM-SHA256;ECDHE-ECDSA-AES128-GCM-SHA256'
```

- SSL 连接认证不仅增加了登录（创建 SSL 环境）及退出过程（清理 SSL 环境）的时间消耗，同时需要消耗额外的时间用于加解密所需传输的内容，因此对性能有一定影响。特别的，对于频繁的登录登出，短时查询等场景有较大的影响。
- 在证书有效期小于 7 天的时候，连接登录会在日志中产生告警提醒。

10.1.4用 SSH 隧道进行安全的 TCP/IP 连接

背景信息

为了保证服务器和客户端之间的安全通讯，可以在服务器和客户端之间构建安全的 SSH 隧道。SSH 是目前较可靠，专为远程登录会话和其他网络服务提供安全性的协议。

从 SSH 客户端来看，SSH 提供了两种级别的安全验证：

- 基于口令的安全验证：使用帐号和口令登录到远程主机。所有传输的数据都会被加密，但是不能保证正在连接的服务器就是需要连接的服务器。可能会有其他服务器冒充真正的服务器，也就是受到"中间人”方式的攻击。

- 基于密钥的安全验证：用户必须为自己创建一对密钥，并把公用密钥放在需要访问的服务器上。这种级别的认证不仅加密所有传送的数据，而且避免"中间人”攻击方式。但是整个登录的过程可能需要 10 秒。

前提条件

SSH 服务和数据库运行在同一台服务器上。

操作以 OpenSSH 为例介绍配置 SSH 隧道，对于如何配置基于密钥的安全验证不作赘述，OpenSSH 提供了多种配置适应网络的各种限制，更多详细信息请参考 OpenSSH 的相关文档。

从本地主机建立到服务器的 SSH 隧道。

```
ssh -L 63333:localhost:15400 username@hostIP
```

说明

- -L 参数的第一串数字（63333）是通道本端的端口号，可以自由选择。
- 第二串数字（15400）是通道远端的端口号，也就是服务器使用的端口号。
- localhost 是本机 IP 地址，username 是要连接的服务器上的用户名，hostIP 是要连接的主机 IP 地址。

10.1.5查看数据库连接数

背景信息

当用户连接数达到上限后，无法建立新的连接。因此，当数据库管理员发现某用户无法连接到数据库时，需要查看是否连接数达到了上限。控制数据库连接的主要以下几种选项。

- 全局的最大连接数：由运行参数 max_connections 指定。
- 某用户的连接数：在创建用户时由 CREATE ROLE 命令的 CONNECTION LIMIT connlimit 子句直接设定，也可以在设定以后用 ALTER ROLE 的 CONNECTION LIMIT connlimit 子句修改。
- 某数据库的连接数：在创建数据库时，由 CREATE DATABASE 的 CONNECTION LIMIT connlimit 参数指定。

操作步骤

(1) 在数据库主节点登录数据库。

```
$ gsql -d postgres -p 15400
```

(2) 查看全局会话连接数限制：

```
postgres=# SHOW max_connections;
```

返回最大会话连接数。

(3) 查看已使用的会话连接数，具体命令请参见下表：

须知

- 除了创建的时候用双引号引起的数据库和用户名称外，以下命令中用到的数据库名称和用户名称，其中包含的英文字母必须使用小写。

表 10-8 查看会话连接数

描述	示例命令
查看指定用户的会话连接数上限。 若显示-1，则表示没有对用户设置连接数的限制。	**SELECT ROLNAME,ROLCONNLIMIT FROM PG_ROLES WHERE ROLNAME='gbase';** rolname | rolconnlimit ----------+-------------- gbase | -1 (1 row)
查看指定用户已使用的会话连接数。	**SELECT COUNT(*) FROM DV_SESSIONS WHERE USERNAME='gbase';** count ------- 1 (1 row)
查看指定数据库的会话连接数上限。 若显示-1，则表示没有对数据库设置连接数的限制。	**SELECT DATNAME,DATCONNLIMIT FROM PG_DATABASE WHERE DATNAME='postgres';** datname | datconnlimit ----------+-------------- postgres | -1 (1 row)
查看指定数据库已使用的会话连接数。	**SELECT COUNT(*) FROM PG_STAT_ACTIVITY WHERE DATNAME='postgres';** count -------

	1 (1 row)
查看所有用户已使用会话连接数。	**SELECT COUNT(*) FROM DV_SESSIONS;** count ------- 10 (1 row)

10.1.6SSL 证书管理

GBase 8c 默认配置了通过 openssl 生成的安全证书、私钥。并且提供证书替换的接口，方便用户进行证书的替换。

10.1.6.1 证书生成

背景信息

在测试环境下，用户可以用通过以下方式进行数字证书测试。在客户的运行环境中， 请使用从 CA 认证中心申请的数字证书。

前提条件

Linux 环境安装了 openssl 组件。

自认证证书生成过程

(1) 搭建 CA 环境。

假设用户为 gbase 已存在，搭建 CA 的路径为 test。

```
$ mkdir test
$ cd test
```

copy 配置文件 openssl.cnf 到 test 下。

```
$ cp /etc/pki/tls/openssl.cnf ./
```

进入 test 文件夹下，开始搭建 CA 环境。

```
$ mkdir ./demoCA ./demoCA/newcerts ./demoCA/private chmod 700 ./demoCA/private
$ touch ./demoCA/index.txt
```

修改 openssl.cnf 配置文件中的参数。

```
$ vim openssl.cnf
dir = ./demoCA
```

```
default_md= sha256
```

至此 CA 环境搭建完成。

(2) 生成根私钥

RSA 证书：

```
openssl genrsa -aes256 -out demoCA/private/cakey.pem 2048
```

ECDSA 证书：

```
openssl ecparam -name prime256v1 -genkey -out demoCA/private/cakey.pem
```

交互过程中，设置根私钥的保护密码，并再次输入私钥密码。

(3) 生成根证书请求文件。

生成 CA 根证书申请文件 careq.pem

```
openssl req -config openssl.cnf -new -key demoCA/private/cakey.pem -out demoCA/careq.pem
```

交互过程设置根私钥密码和名称。请牢记，生成服务器证书和客户端证书时填写的信息需要与键入的保持一致。

(4) 生成自签发根证书。

生成根证书时，需要修改 openssl.cnf 文件，设置 basicConstraints=CA:TRUE。

```
vi openssl.cnf
basicConstraints=CA:TRUE
```

生成 CA 自签发根证书。

```
openssl ca -config openssl.cnf -out demoCA/cacert.pem -keyfile demoCA/private/cakey.pem
-selfsign -infiles demoCA/careq.pem
```

交互过程中设置根密钥，CA 根证书自签发完成，根证书 demoCA/cacert.pem。

(5) 生成服务端证书私钥，RSA 和 ECDSA 加密方式可以根据需要选择其中一种。

生成 RSA 服务端证书私钥文件 server.key。

```
openssl genrsa -aes256 -out server.key 2048
```

交互过程设置服务器私钥的保护密码。

生成 ECDSA 服务端证书私钥文件 server.key。

```
openssl ecparam -name prime256v1 -genkey -out server.key
```

对 ECDSA 证书私钥进行加密保护，根据提示输入加密密码：

```
openssl ec -in server.key -aes256 -out server.key
```

根据提示输入服务端私钥的密码，加密后会生成 server.key.cipher,server.key.rand 两个私钥密码保护文件。

```
gs_guc encrypt -M server -D ./
```

(6) 生成服务端证书请求文件。

```
openssl req -config openssl.cnf -new -key server.key -out server.req
```

以下填写的信息与创建 CA 时的信息一致。

```
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:shanxi Locality Name (eg, city) []:xian
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Abc Organizational Unit Name
(eg, section) []:hello
```

Cgbaseon Name 可以随意命名。

```
Cgbaseon Name (eg, YOUR name) []:world Email Address []:
```

以下信息可以选择性填写：

```
Please enter the following 'extra' attributes to be sent with your certificate request
A challenge password []:
An optional company name []:
```

(7) 生成服务端证书。

生成服务端/客户端证书时，修改 openssl.cnf 文件，设置：

```
vi openssl.cnf
basicConstraints=CA:FALSE
```

修改 demoCA/index.txt.attr 中属性为 no。

```
vi demoCA/index.txt.attr
```

对生成的服务端证书请求文件进行签发，签发后将生成正式的服务端证书 server.crt

```
openssl ca -config openssl.cnf -in server.req -out server.crt -days 3650 -md sha256
```

交互过程中，输入’y’确认对证书进行签发。

(8) 客户端证书和私钥的生成。

生成客户端证书和客户端证书私钥的方法和要求与服务器相同。生成客户端证书私钥，RSA 和 ECDSA 加密方式可以根据需要选择其中一种。

RSA 证书私钥：

```
openssl genrsa -aes256 -out client.key 2048
```

ECDSA 证书私钥：

```
openssl ecparam -name prime256v1 -genkey -out client.key
```

对于 ECDSA 证书私钥，需要执行如下命令进行加密保护，根据提示输入加密密码：

```
openssl ec -in server.key -aes256 -out server.key
```

根据提示输入客户端私钥的密码，加密后会生成 client.key.cipher,client.key.rand 两个私钥密码保护文件。

```
gs_guc encrypt -M client -D ./
```

生成客户端证书请求文件。

```
openssl req -config openssl.cnf -new -key client.key -out client.req
```

对生成的客户端证书请求文件进行签发，签发后将生成正式的客户端证书 client.crt。

```
openssl ca -config openssl.cnf -in client.req -out client.crt -days 3650 -md sha256
```

根据需要将客户端密钥转化为 DER 格式，方法如下：

```
openssl pkcs8 -topk8 -outform DER -in client.key -out client.key.pk8 -nocrypt
```

(9) 吊销证书列表。

创建 crlnumber 文件。

```
echo '00'>./demoCA/crlnumber
```

吊销服务器证书。

```
openssl ca -config openssl.cnf -revoke server.crt
```

生成证书吊销列表 sslcrl-file.crl。

```
openssl ca -config openssl.cnf -gencrl -out sslcrl-file.crl
```

10.1.6.2 证书替换

操作场景

GBase 8c 默认配置了 SSL 连接所需要的安全的证书、私钥，用户如果需要替换为自己的证书、私钥则可按照此方法进行替换。

前提条件

用户需要从 CA 认证中心申请到正式的服务器、客户端的证书和密钥。

注意事项

GBase 8c 目前只支持 X509v3 的 PEM 格式证书。

(1) 准备证书、私钥。

服务端各个配置文件名称约定：

- 证书名称约定：server.crt。
- 私钥名称约定：server.key。
- 私钥密码加密文件约定：server.key.cipher、server.key.rand。

客户端各个配置文件名称约定：

- 证书名称约定：client.crt。
- 私钥名称约定：client.key。
- 私钥密码加密文件约定：client.key.cipher、client.key.rand。
- 根证书名称约定：cacert.pem。
- 吊销证书列表文件名称约定：sslcrl-file.crl。

(2) 制作压缩包。

压缩包名称约定：db-cert-replacement.zip。

压缩包格式约定：ZIP。

压缩包文件列表约定：server.crt、server.key、server.key.cipher、server.key.rand、client.crt、client.key、client.key.cipher、client.key.rand、cacert.pem。如果需要配置吊销证书列表，则列表中包含 sslcrl-file.crl。

(3) 调用接口，执行替换。

将制作好的压缩包 db-cert-replacement.zip 上传到 GBase 8c 用户下的任意路径。

例如：/home/xxxx/db-cert-replacement.zip。

调用如下命令进行替换。

```
gs_om    -t    cert --cert-file=/home/xxxx/db-cert-replacement.zip
```

(4) 重启 GBase 8c。

```
gs_om -t stop
gs_om -t start
```

说明

证书具有 rollback 功能，可以把上一次执行证书替换之前的证书，进行回退。可以使用 gs_om -t cert --rollback 进行远程调用该接口；使用 gs_om -t cert --rollback -L 进行本地调用该接口。以上一次成功执行证书替换后，被替换的证书版本为基础进行回退。

10.2 管理用户及权限

10.2.1默认权限机制

数据库对象创建后，进行对象创建的用户就是该对象的所有者。数据库安装后的默认情况下，未开启三权分立，数据库系统管理员具有与对象所有者相同的权限。也就是说对象创建后，默认只有对象所有者或者系统管理员可以查询、修改和销毁对象，以及通过《GBase 8c V5_5.0.0_SQL 参考手册》 GRANT 将对象的权限授予其他用户。

为使其他用户能够使用对象，必须向用户或包含该用户的角色授予必要的权限。

GBase 8c 支持以下的权限：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、CREATE、CONNECT、EXECUTE、USAGE、ALTER、DROP、CgbaseENT、INDEX 和 VACUUM。不同的权限与不同的对象类型关联。有关各权限的 详细信息，请参见《GBase 8c V5_5.0.0_SQL 参考手册》 GRANT。

要撤消已经授予的权限，可以使用《GBase 8c V5_5.0.0_SQL 参考手册》 REVOKE。对象所有者的权限(例如 ALTER、DROP、CgbaseENT、INDEX、VACUUM、GRANT 和 REVOKE)是隐式拥有的，即只要拥有对象就可以执行对象所有者的这些隐式权限。对象所有者可以撤消自己的普通权限，例如，使表对自己以及其他人只读，系统管理员用户除外。

系统表和系统视图要么只对系统管理员可见，要么对所有用户可见。标识了需要系统管理员权限的系统表和视图只有系统管理员可以查询。 有关信息，请参考 20 系统表和系统视图。

数据库提供对象隔离的特性，对象隔离特性开启时，用户只能查看有权限访问的对象(表、视图、字段、函数)，系统管理员不受影响。有关信息，请参考《GBase 8c V5_5.0.0_SQL 参考手册》ALTER DATABASE。

10.2.2管理员

初始用户

数据库安装过程中自动生成的帐户称为初始用户。初始用户拥有系统的最高权限，能够

执行所有的操作。如果安装时不指定初始用户名称则该帐户与进行数据库安装的操作系统用户同名。如果在安装时不指定初始用户的密码，安装完成后密码为空，在执行其他操作前需要通过 gsql 客户端修改初始用户的密码。如果初始用户密码为空，则除修改密码外无法执行其他 SQL 操作以及升级、扩容、节点替换等操作。

初始用户会绕过所有权限检查。建议仅将此初始用户作为 DBA 管理用途，而非业务应用。

系统管理员

系统管理员是指具有 SYSADMIN 属性的帐户，默认安装情况下具有与对象所有者相同的权限，但不包括 dbe_perf 模式的对象权限。

要创建新的系统管理员，请以初始用户或者系统管理员用户身份连接数据库，并使用带 SYSADMIN 选项的《GBase 8c V5_5.0.0_SQL 参考手册》 CREATE USER 语句或《GBase 8c V5_5.0.0_SQL 参考手册》 ALTER USER 语句进行设置。

```
CREATE USER username WITH SYSADMIN password "password";
```

或者

```
ALTER USER username SYSADMIN;
```

ALTER USER 时，要求用户已存在。

监控管理员

监控管理员是指具有 MONADMIN 属性的帐户，具有查看 dbe_perf 模式下视图和函数的权限，亦可以对 dbe_perf 模式的对象权限进行授予或收回。

要创建新的监控管理员，请以系统管理员身份连接数据库，并使用带 MONADMIN 选 项的《GBase 8c V5_5.0.0_SQL 参考手册》CREATE USER 语句或《GBase 8c V5_5.0.0_SQL 参考手册》ALTER USER 语句进行设置。

```
CREATE USER username WITH MONADMIN password "password";
```

或者

```
ALTER USER username MONADMIN;
```

ALTER USER 时，要求用户已存在。

运维管理员

运维管理员是指具有 OPRADMIN 属性的帐户，具有使用 Roach 工具执行备份恢复的权限。

要创建新的运维管理员，请以初始用户身份连接数据库，并使用带 OPRADMIN 选项的 CREATE USER 语句或《GBase 8c V5_5.0.0_SQL 参考手册》ALTER USER 语句进行设置。

```
CREATE USER username WITH OPRADMIN password "password";
```

或者

```
ALTER USER username OPRADMIN;
```

ALTER USER 时，要求用户已存在。

安全策略管理员

安全策略管理员是指具有 POLADMIN 属性的帐户，具有创建资源标签，脱敏策略和统一审计策略的权限。

要创建新的安全策略管理员，请以系统管理员用户身份连接数据库，并使用带 POLADMIN 选项的《GBase 8c V5_5.0.0_SQL 参考手册》 CREATE USER 语句或《GBase 8c V5_5.0.0_SQL 参考手册》 ALTER USER 语句进行设置。

```
CREATE USER username WITH POLADMIN password "password";
```

或者

```
ALTER USER username POLADMIN;
```

ALTER USER 时，要求用户已存在。

10.2.3三权分立

默认权限机制和管理员两节的描述基于的是数据库创建之初的默认情况。从前面的介绍可以看出，默认情况下拥有 SYSADMIN 属性的系统管理员，具备系统最高权限。

在实际业务管理中，为了避免系统管理员拥有过度集中的权利带来高风险，可以设置三权分立。将系统管理员的部分权限分立给安全管理员和审计管理员，形成系统管理员、安全管理员和审计管理员三权分立。

三权分立后，系统管理员将不再具有 CREATEROLE 属性（安全管理员）和 AUDITADMIN 属性（审计管理员）能力。即不再拥有创建角色和用户的权限，并不再拥有查看和维护数据库审计日志的权限。关于 CREATEROLE 属性和 AUDITADMIN 属性 的更多信息请参考《GBase 8c V5_5.0.0_SQL 参考手册》CREATE ROLE。

三权分立后，系统管理员只会对自己作为所有者的对象有权限。

初始用户的权限不受三权分立设置影响。因此建议仅将此初始用户作为 DBA 管理用途，

而非业务应用。

三权分立的设置办法为：将参数 enableSeparationOfDuty 设置为 on。

三权分立前的权限详情及三权分立后的权限变化，请分别参见表 10-9 和表 10-10。

表 10-9 默认的用户权限

对象名称	初始用户(id 为 10)	系统管理员	安全管理员	审计管理员	普通用户
表空间	具有除私有用户表对象访问权限外，所有的权限。	对表空间有创建、修改、删除、访问、分配操作的权限。	不具有对表空间进行创建、修改、删除、分配的权限，访问需要被赋权。		
表		对所有表有所有的权限。	仅对自己的表有所有的权限，对其他用户的表无权限。		
索引		可以在所有的表上建立索引。	仅可以在自己的表上建立索引。		
模式		对除 dbe_perf 以外的所有模式有所有的权限。	仅对自己的模式有所有的权限，对其他用户的模式无权限。		
函数		对除 dbe_perf 模式下的函数以外的所有的函数有所有的权限。	仅对自己的函数有所有的权限，对其他用户放在 public 这个公共模式下的函数有调用的权限，对其他用户放在其他模式下的函数无权限。		
自定义视图		对除 dbe_perf 模式下的视图以外的所有的视图有所有的权限。	仅对自己的视图有所有的权限，对其他用户的视图无权限。		
系统表和系统视图		可以查看所有系统表和视图。	只可以查看部分系统表和视图。详细请参见《GBase 8c V5_5.0.0_SQL 参考手册》中系统表和系统视图。		

表 10-10 三权分立较非三权分立权限变化说明

对象名称	初始用户（id 为 10）	系统管理员	安全管理员	审计管理员	普通用户
表空间	无变化。 依然具有除私有用户表对象访问权限外，所有的权限。	无变化。	无变化。		
表		对所有表有所有的权限。	仅对自己的表有所有的权限，对其他用户的表无权限。		
索引		权限缩小。 只可以在自己的表及其他用户放在 public 模式下的表上建立索引。	无变化。		
模式		权限缩小。 只对自己的模式有所有的权限，对其他用户的模式无权限。	无变化。		
函数		权限缩小。 只对自己的函数及其他用户放在 public 模式下的函数有所有的权限，对其他用户放在属于各自模式下的函数无权限。	无变化。		
自定义视图		权限缩小。 只对自己的视图及其他用户放在 public 模式下的视图有所有的权限，对其他用户放在属于各自模式下的视图无权限。	无变化。		

系统表和系统视图		无变化。	无变化。

10.2.4用户

使用 CREATE USER 和 ALTER USER 可以创建和管理数据库用户。GBase 8c 包含一个或多个已命名数据库用户和角色在 GBase 8c 范围内是共享的，但是其数据并不共享。 即用户可以连接任何数据库，但当连接成功后，任何用户都只能访问连接请求里声明的那个数据库。

非三权分立下，GBase 8c 用户帐户只能由系统管理员或拥有 CREATEROLE 属性的安全管理员创建和删除。三权分立时，用户帐户只能由初始用户和安全管理员创建。

在用户登录 GBase 8c 时，会对其进行身份验证。用户可以拥有数据库和数据库对象（例如表），并且可以向用户和角色授予对这些对象的权限以控制谁可以访问哪个对象。除系统管理员外，具有 CREATEDB 属性的用户可以创建数据库并授予对这些数据库的权限。

- 要创建用户，请使用 SQL 语句《GBase 8c V5_5.0.0_SQL 参考手册》 CREATE USER。

  例如：创建用户 joe，并设置用户拥有 CREATEDB 属性。

```
postgres= CREATE USER joe WITH CREATEDB PASSWORD "Gbase,12";
```

- 要创建系统管理员，请使用带有 SYSADMIN 选项的《GBase 8c V5_5.0.0_SQL 参考手册》 CREATE USER 语句。
- 要删除现有用户，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 DROP USER 语句。
- 要更改用户帐户（例如，重命名用户或更改密码），请使用《GBase 8c V5_5.0.0_SQL 参考手册》ALTER USER 语句。
- 要查看用户列表，请查询视图 PG_USER：

```
SELECT * FROM pg_user;
```

- 要查看用户属性，请查询系统表 PG_AUTHID：

```
SELECT * FROM pg_authid;
```

私有用户

对于有多个业务部门，各部门间使用不同的数据库用户进行业务操作，同时有一个同级的数据库维护部门使用数据库管理员进行维护操作的场景下，业务部门可能希望在未经授权的情况下，管理员用户只能对各部门的数据进行控制操作（DROP、ALTER、TRUNCATE），

但是不能进行访问操作（INSERT、DELETE、UPDATE、SELECT、COPY）。即针对管理员用户，表对象的控制权和访问权要能够分离，提高普通用户数据安全性。

三权分立情况下，管理员对其他用户放在属于各自模式下的表无权限。但是，这种无权限包含了无控制权限，因此不能满足上面的诉求。为此，GBase 8c 提供了私有用户方案。即在非三权分立模式下，创建具有 INDEPENDENT 属性的私有用户。

```
postgres=# CREATE USER user_independent WITH INDEPENDENT IDENTIFIED BY
"1234@abc";
```

针对该用户的对象，系统管理员和拥有 CREATEROLE 属性的安全管理员在未经其授权前，只能进行控制操作（DROP、ALTER、TRUNCATE），无权进行 INSERT、DELETE、SELECT、UPDATE、COPY、GRANT、REVOKE、ALTER OWNER 操作。

须知

- PG_STATISTIC 系统表和 PG_STATISTIC_EXT 系统表存储了统计对象的一些敏感信息，如高频值 MCV。系统管理员仍然可以通过访问这两张系统表，得到私有用户所属表的统计信息里的这些信息。

永久用户

GBase 8c 提供永久用户方案，即创建具有 PERSISTENCE 属性的永久用户。

```
postgres=# CREATE USER user_persistence WITH PERSISTENCE IDENTIFIED BY
"1234@abc";
```

只允许初始用户创建、修改和删除具有 PERSISTENCE 属性的永久用户。

10.2.5角色

角色是一组用户的集合。通过 GRANT 把角色授予用户后，用户即具有了角色的所有权限。推荐使用角色进行高效权限分配。例如，可以为设计、开发和维护人员创建不同的角色，将角色 GRANT 给用户后，再向每个角色中的用户授予其工作所需数据的差异权限。在角色级别授予或撤消权限时，这些更改将作用到角色下的所有成员。

GBase 8c 提供了一个隐式定义的拥有所有角色的组 PUBLIC，所有创建的用户和角色默认拥有 PUBLIC 所拥有的权限。关于 PUBLIC 默认拥有的权限请参考《GBase 8c V5_5.0.0_SQL 参考手册》GRANT。要撤销或重新授予用户和角色对 PUBLIC 的权限， 可通过在 GRANT 和 REVOKE 指定关键字 PUBLIC 实现。

要查看所有角色，请查询系统表 PG_ROLES：

```
SELECT * FROM    PG_ROLES;
```

创建、修改和删除角色

非三权分立时，只有系统管理员和具有 CREATEROLE 属性的用户才能创建、修改或删除角色。三权分立下，只有初始用户和具有 CREATEROLE 属性的用户才能创建、修改或删除角色。

- 要创建角色，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 CREATE ROLE。
- 要在现有角色中添加或删除用户，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 ALTER ROLE。
- 要删除角色，请使用《GBase 8c V5_5.0.0_SQL 参考手册》DROP ROLE。DROP ROLE 只会删除角色，并不会删除角色中的成员用户帐户。

内置角色

GBase 8c 提供了一组默认角色，以 gs_role_开头命名。它们提供对特定的、通常需要高权限的操作的访问，可以将这些角色 GRANT 给数据库内的其他用户或角色，让这些用户能够使用特定的功能。在授予这些角色时应当非常小心，以确保它们被用在需要的地方。表 10-11 描述了内置角色允许的权限范围：

表 10-11 内置角色权限描述

角色	权限描述
gs_role_copy_files	具有执行 copy … to/from filename 的权限，但需要先打开 GUC 参数 enable_copy_server_files。
gs_role_signal _backend	具有调用函数 pg_cancel_backend、pg_terminate_backend 和 pg_terminate_session 来取消或终止其他会话的权限，但不能操作属于初始用户和 PERSISTENCE 用户的会话。
gs_role_tables pace	具有创建表空间（tablespace）的权限。
gs_role_replication	具有调用逻辑复制相关函数的权限，例如 kill_snapshot、pg_create_logical_replication_slot 、pg_create_physical_replication_slot、pg_drop_replication_slot、pg_replication_slot_advance 、pg_create_physical_replication_slot_extern 、

	pg_logical_slot_get_changes、pg_logical_slot_peek_changes、pg_logical_slot_get_binary_changes、pg_logical_slot_peek_binary_changes。
gs_role_account_lock	具有加解锁用户的权限，但不能加解锁初始用户和 PERSISTENCE 用户。
gs_role_pldebugger	具有执行 dbe_pldebugger 下调试函数的权限。
gs_role_directory_create	具有执行创建 directory 对象的权限，但需要先打开 GUC 参数 enable_access_server_directory。
gs_role_directory_drop	具有执行删除 directory 对象的权限，但需要先打开 GUC 参数 enable_access_server_directory。

关于内置角色的管理有如下约束：

- 以 gs_role_开头的角色名作为数据库的内置角色保留名，禁止新建以"gs_role_" 开头的用户/角色，也禁止将已有的用户/角色重命名为以"gs_role_"开头；
- 禁止对内置角色的 ALTER 和 DROP 操作；
- 内置角色默认没有 LOGIN 权限，不设预置密码；
- gsql 元命令\du 和\dg 不显示内置角色的相关信息，但若显示指定了 pattern 为特定内置角色则会显示。
- 三权分立关闭时，初始用户、具有 SYSADMIN 权限的用户和具有内置角色 ADMIN OPTION 权限的用户有权对内置角色执行 GRANT/REVOKE 管理。三权分立打开时，初始用户和具有内置角色 ADMIN OPTION 权限的用户有权对内置角色执行 GRANT/REVOKE 管理。例如：

```
GRANT gs_role_signal_backend TO username;
REVOKE gs_role_signal_backend FROM username;
```

10.2.6Schema

Schema 又称作模式。通过管理 Schema，允许多个用户使用同一数据库而不相互干 扰，可以将数据库对象组织成易于管理的逻辑组，同时便于将第三方应用添加到相应的 Schema 下而不引起冲突。

每个数据库包含一个或多个 Schema。数据库中的每个 Schema 包含表和其他类型的对象。数据库创建初始，默认具有一个名为 public 的 Schema，且所有用户都拥有此 Schema 的 usage 权限，只有系统管理员和初始化用户可以在 public Schema 下创建函数、存储过程和同义词对象，其他用户即使赋予 create 权限后也不可以创建上述三种对 象。可以通过 Schema 分组数据库对象。Schema 类似于操作系统目录，但 Schema 不能嵌套。默认只有初始化用户可以在 pg_catalog 模式下创建对象。

相同的数据库对象名称可以应用在同一数据库的不同 Schema 中，而没有冲突。例如，a_schema 和 b_schema 都可以包含名为 mytable 的表。具有所需权限的用户可以访问数据库的多个 Schema 中的对象。

通过 CREATE USER 创建用户的同时，系统会在执行该命令的数据库中，为该用户创建一个同名的 SCHEMA。

数据库对象是创建在数据库搜索路径中的第一个 Schema 内的。有关默认情况下的第一个 Schema 情况及如何变更 Schema 顺序等更多信息，请参见搜索路径。

创建、修改和删除 Schema

要创建 Schema，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 CREATE SCHEMA。默认初始用户和系统管理员可以创建 Schema，其他用户需要具备数据库的 CREATE 权限才可以在该数据库中创建 Schema，赋权方式请参考《GBase 8c V5_5.0.0_SQL 参考手册》GRANT 中将数据库的访问权限赋予指定的用户或角色中的语法。

- 要更改 Schema 名称或者所有者，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 ALTER SCHEMA。Schema 所有者可以更改 Schema。
- 要删除 Schema 及其对象，请使用《GBase 8c V5_5.0.0_SQL 参考手册》DROP SCHEMA。Schema 所有者可以删除 Schema。
- 要在 Schema 内创建表，请以 schema_name.table_name 格式创建表。不指定 schema_name 时，对象默认创建到搜索路径中的第一个 Schema 内。
- 要查看 Schema 所有者，请对系统表 PG_NAMESPACE 和 PG_USER 执行如下关联查询。语句中的 schema_name 请替换为实际要查找的 Schema 名称。

```
SELECT s.nspname,u.usename AS nspowner FROM pg_namespace s, pg_user u WHERE nspname='schema_name' AND s.nspowner = u.usesysid;
```

- 要查看所有 Schema 的列表，请查询 PG_NAMESPACE 系统表。

```
SELECT * FROM pg_namespace;
```

- 要查看属于某 Schema 下的表列表，请查询系统视图 PG_TABLES。例如，以下查询会返回 Schema PG_CATALOG 中的表列表。

```
SELECT * FROM pg_namespace;
```

搜索路径

搜索路径定义在 search_path 参数中，参数取值形式为采用逗号分隔的 Schema 名称列表。如果创建对象时未指定目标 Schema，则该对象会被添加到搜索路径中列出的第一个 Schema 中。当不同 Schema 中存在同名的对象时，查询对象未指定 Schema 的情况 下，将从搜索路径中包含该对象的第一个 Schema 中返回对象。

- 要查看当前搜索路径，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 SHOW。

```
SHOW SEARCH_PATH;
```

search_path 参数的默认值为："$user",public。$user 表示与当前会话用户名同名的 Schema 名，如果这样的模式不存在，$user 将被忽略。所以默认情况下，用户连接数据库后，如果数据库下存在同名 Schema，则对象会添加到同名 Schema 下，否则对象被添加到 Public Schema 下。

- 要更改当前会话的默认 Schema，请使用 SET 命令。

例如，将搜索路径设置为 myschema、public，首先搜索 myschema：

```
postgres=# SET SEARCH_PATH TO myschema, public;
SET
```

10.2.7用户权限设置

- 给用户直接授予某对象的权限，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 GRANT。

将 Schema 中的表或者视图对象授权给其他用户或角色时，需要将表或视图所属 Schema 的 USAGE 权限同时授予该用户或角色。否则用户或角色将只能看到这些对象的名称，并不能实际进行对象访问。

例如，下面示例将 Schema tpcds 的权限赋给用户 joe 后，将表 tpcds.web_returns 的 select 权限赋给用户 joe：

```
postgres=# GRANT USAGE ON SCHEMA tpcds TO joe;
postgres=# GRANT SELECT ON TABLE tpcds.web_returns to joe;
```

- 给用户指定角色，使用户继承角色所拥有的对象权限
  - 创建角色。

新建一个角色 lily，同时给角色指定系统权限 CREATEDB：

```
postgres=# CREATE ROLE lily WITH CREATEDB PASSWORD "Gbase.12";
```

- 给角色赋予对象权限，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 GRANT。

例如，将模式 tpcds 的权限赋给角色 lily 后，将表 tpcds.web_returns 的 select 权限赋给角色 lily。

```
postgres=# GRANT USAGE ON SCHEMA tpcds TO lily;
postgres=# GRANT SELECT ON TABLE tpcds.web_returns to lily;
```

- 将角色的权限赋予用户。

```
postgres=# GRANT lily to joe;
```

说明

- 当将角色的权限赋予用户时，角色的属性并不会传递到用户。

- 回收用户权限，请使用《GBase 8c V5_5.0.0_SQL 参考手册》 REVOKE。

10.2.8行级访问控制

行级访问控制特性将数据库访问控制精确到数据表行级别，使数据库达到行级访问控制的能力。不同用户执行相同的 SQL 查询操作，读取到的结果是不同的。

用户可以在数据表创建行访问控制(Row Level Security)策略，该策略是指针对特定数据库用户、特定 SQL 操作生效的表达式。当数据库用户对数据表访问时，若 SQL 满足数据表特定的 Row Level Security 策略，在查询优化阶段将满足条件的表达式，按照属性(PERMISSIVE | RESTRICTIVE)类型，通过 AND 或 OR 方式拼接，应用到执行计划上。

行级访问控制的目的是控制表中行级数据可见性，通过在数据表上预定义 Filter，在查询优化阶段将满足条件的表达式应用到执行计划上，影响最终的执行结果。当前受影响的 SQL 语句包括 SELECT，UPDATE，DELETE，支持 with check、enable、security relevant columns 子句。

示例：某表中汇总了不同用户的数据，但是不同用户只能查看自身相关的数据信息，不能查看其他用户的数据信息。

```
--创建用户 alice, bob, peter
postgres=# CREATE USER alice PASSWORD 'Gbase,123';
postgres=# CREATE USER bob PASSWORD 'Gbase,123';
postgres=# CREATE USER peter PASSWORD 'Gbase,123';
--创建表 all_data，包含不同用户数据信息
```

```
postgres=# CREATE TABLE all_data(id int, role varchar(100), data varchar(100));
--向数据表插入数据
postgres=# INSERT INTO all_data VALUES(1, 'alice', 'alice data');
postgres=# INSERT INTO all_data VALUES(2, 'bob', 'bob data');
postgres=# INSERT INTO all_data VALUES(3, 'peter', 'peter data');
--将表 all_data 的读取权限赋予 alice，bob 和 peter 用户
postgres=# GRANT SELECT ON all_data TO alice, bob, peter;
--打开行访问控制策略开关
postgres=# ALTER TABLE all_data ENABLE ROW LEVEL SECURITY;
--创建行访问控制策略，当前用户只能查看用户自身的数据
postgres=# CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role
= CURRENT_USER);
--查看表详细信息
postgres=# \d+ all_data
                                        Table "public.all_data"
 Column |            Type             | Modifiers | Storage  | Stats target | Descrip
tion
--------+-----------------------------+-----------+----------+--------------+--------
-----
 id     | integer                     |           | plain    |              |
 role   | character varying(100)      |           | extended |              |
 data   | character varying(100)      |           | extended |              |
Row Level Security Policies:
    POLICY "all_data_rls" FOR ALL
      TO public
      USING (((role)::name = "current_user"()))
      SECURITY RELEVANT COLUMNS (("*"))
      ENABLE (t)
Has OIDs: no
Options: orientation=row, compression=no, enable_rowsecurity=true

--切换至用户 alice，执行 SQL"SELECT * FROM public.all_data"
postgres=> SELECT * FROM public.all_data;
id | role | data
----+-------+------------
1 | alice | alice data
(1 row)

postgres=> EXPLAIN(COSTS OFF) SELECT * FROM public.all_data;
                                QUERY PLAN
-----------------------------------------------------------------
```

```
 Seq Scan on all_data
    Filter: ((role)::name = 'alice'::name)
 Notice: This query is influenced by row level security feature
(3 rows)

--切换至用户 peter，执行 SQL"SELECT * FROM public.all_data"
postgres=>SELECT * FROM public.all_data;
id | role | data
----+-------+------------
3 | peter | peter data
(1 row)
postgres=>EXPLAIN(COSTS OFF) SELECT * FROM public.all_data; QUERY PLAN
-------------------------------------------------------------
Streaming (type: GATHER) Node/s: All datanodes
-> Seq Scan on all_data
Filter: ((role)::name = 'peter'::name)
Notice: This query is influenced by row level security feature
(5 rows)
```

须知

- PG_STATISTIC 和 PG_STATISTIC_EXT 系统表存储了统计对象的一些敏感信息，如高频值 MCV。若创建行级访问控制后，将这两张系统表的查询权限授予普通用户，则普通用户仍然可以通过访问这两张系统表，得到统计对象里的这些信息。

10.2.9设置安全策略

10.2.9.1 设置帐户安全策略

背景信息

GBase 8c 为帐户提供了自动锁定和解锁帐户、手动锁定和解锁异常帐户和删除不再使用的帐户等一系列的安全措施，保证数据安全。

自动锁定和解锁账户

- 为了保证帐户安全，如果用户输入密码次数超过一定次数（failed_login_attempts），系统将自动锁定该帐户，默认值为 10。次数设置越小越安全，但是在使用过程中会带来不便。
- 当帐户被锁定时间超过设定值（password_lock_time），则当前帐户自动解锁， 默认值为 1 天。时间设置越长越安全，但是在使用过程中会带来不便。

说明

- 参数 password_lock_time 的整数部分表示天数，小数部分可以换算成时、分、秒。
- 当 failed_login_attempts 设置为 0 时，表示不限制密码错误次数。当 password_lock_time 设置为 0 时，表示即使超过密码错误次数限制导致帐户锁定，也会在短时间内自动解锁。因此，只有两个配置参数都为正数时，才可以进行常规的密码失败检查、帐户锁定和解锁操作。
- 这两个参数的默认值都符合安全标准，用户可以根据需要重新设置参数，提高安全等级。建议用户使用默认值。

配置 failed_login_attempts 参数。

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(3) 查看已配置的参数。

```
postgres=# SHOW failed_login_attempts;
failed_login_attempts
----------------------
10
(1 row)
```

如果显示结果不为 10，执行"\q”命令退出数据库。

(4) 执行如下命令设置成默认值 10。例如，存储目录为/opt/database/install/data。

```
gs_guc reload -Z datanode -D /opt/database/install/data/dn -c "failed_login_attempts=10"
```

配置 password_lock_time 参数。

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(3) 查看已配置的参数。

```
postgres=# SHOW password_lock_time;
password_lock_time
----------------------
1
```

```
(1 row)
```

如果显示结果不为 1，执行"\q”命令退出数据库。

(4) 执行如下命令设置成默认值 1。

```
gs_guc reload -Z datanode -N all -I all -c    "password_lock_time=1"
```

手动锁定和解锁账户

若管理员发现某帐户被盗、非法访问等异常情况，可手动锁定该帐户。当管理员认为帐户恢复正常后，可手动解锁该帐户。

命令格式如下：

- 手动锁定

```
ALTER USER username ACCOUNT LOCK;
```

- 手动解锁

```
ALTER USER username ACCOUNT UNLOCK;
```

删除不再使用的账户

当确认帐户不再使用，管理员可以删除帐户。该操作不可恢复。

当删除的用户正处于活动状态时，此会话状态不会立马断开，用户在会话状态断开后才会被完全删除。

命令格式如下：

```
DROP USER username CASCADE;
```

10.2.9.2 设置账号有效期

注意事项

创建新用户时，需要限制用户的操作期限（有效开始时间和有效结束时间）。不在有效操作期内的用户需要重新设定帐号的有效操作期。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(3) 创建用户并制定用户的有效开始时间和有效结束时间。

```
postgres=# CREATE USER joe WITH PASSWORD 'gbase;123' VALID BEGIN '2022-6-6 08:00:00' VALID UNTIL '2022-6-6 18:00:00';
```

显示如下信息表示创建用户成功。

```
CREATE ROLE
```

(4) 用户已不在有效使用期内，需要重新设定帐号的有效期，这包括有效开始时间和有效结束时间。

```
postgres=# ALTER USER joe WITH VALID BEGIN '2022-6-6 08:00:00' VALID UNTIL '2022-6-6 18:00:00';
```

显示如下信息表示重新设定成功。

```
ALTER ROLE
```

须知

- CREATE ROLE 语法中不指定"VALID BEGIN”和"VALID UNTIL”时，表示不对用户的开始操作时间和结束操作时间进行限定。
- ALTER ROLE 语法中不指定"VALID BEGIN”和"VALID UNTIL”时，表示不对用户的开始操作时间和结束操作时间进行修改，沿用之前设置。

10.2.9.3 设置密码安全策略

用户密码存储在系统表 pg_authid 中，为防止用户密码泄露，GBase 8c 对用户密码进行加密存储，所采用的加密算法由配置参数 password_encryption_type 决定。

- 当参数 password_encryption_type 设置为 0 时，表示采用 md5 方式对密码加密。MD5 加密算法安全性低，存在安全风险，不建议使用。
- 当参数 password_encryption_type 设置为 1 时，表示采用 sha256 和 md5 方式对密码加密。MD5 加密算法安全性低，存在安全风险，不建议使用。
- 当参数 password_encryption_type 设置为 2 时，表示采用 sha256 方式对密码加密， 为默认配置。
- 当参数 password_encryption_type 设置为 3 时，表示采用 sm3 方式对密码加密。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(3) 查看已配置的加密算法。

```
postgres=# SHOW password_encryption_type;
password_encryption_type
-------------------------
2
(1 row)
```

如果显示结果为 0 或 1，执行"\q”命令退出数据库。

(4) 执行如下命令将其设置为安全的加密算法。

```
gs_guc reload -Z datanode -N all -I all -c "password_encryption_type=2"
```

须知

- 为防止用户密码泄露，在执行 CREATE USER/ROLE 命令创建数据库用户时，不能指定 UNENCRYPTED 属性，即新创建的用户的密码只能是加密存储的。

(5) 配置密码安全参数。

密码复杂度

初始化数据库、创建用户、修改用户时需要指定密码。密码必须要符合复杂度（password_policy）的要求，否则会提示用户重新输入密码。

- 参数 password_policy 设置为 1 时表示采用密码复杂度校验，默认值。
- 参数 password_policy 设置为 0 时表示不采用密码复杂度校验，但需满足密码不能为空并且只包含有效字符，有效字符范围为大写字母（A-Z）、小写字母（a-z）、数字（0-9）及特殊字符详见表 10-12。设置为 0 会存在安全风险，不建议设置为 0。如需要设置，则需将所有数据库节点中的 password_policy 都设置为 0 才能生效。

配置 password_policy 参数。

(1) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(2) 查看已配置的参数。

```
postgres=# SHOW password_policy;
password_policy
--------------------
 1
(1 row)
```

如果显示结果不为 1，执行"\q”命令退出数据库。

(3) 执行如下命令设置成默认值 1。

```
gs_guc reload -Z datanode -N all -I all -c "password_policy=1"
```

帐户密码的复杂度要求如下：

- 包含大写字母（A-Z）的最少个数（password_min_uppercase）
- 包含小写字母（a-z）的最少个数（password_min_lowercase）
- 包含数字（0-9）的最少个数（password_min_digital）
- 包含特殊字符的最少个数(password_min_special)(特殊字符的列表请参见表 10-12)
- 密码的最小长度（password_min_length）
- 密码的最大长度（password_max_length）
- 至少包含上述四类字符中的三类。
- 不能和用户名、用户名倒写相同，本要求为非大小写敏感。
- 不能和当前密码、当前密码的倒写相同。

不能是弱口令。

- 弱口令指的是强度较低，容易被破解的密码，对于不同的用户或群体，弱口令的定义可能会有所区别，用户需自己添加定制化的弱口令。
- 弱口令字典中的口令存放在 gs_global_config 系统表中，当创建用户、修改用户需要设置密码时，系统将会把用户设置口令和弱口令字典中存放的口令进行对比，如果符合，则会提示用户该口令为弱口令，设置密码失败。
- 弱口令字典默认为空，用户通过以下语法可以对弱口令字典进行增加和删除，格式如下：

```
CREATE WEAK PASSWORD DICTIONARY WITH VALUES ('password1'),
('password2');
DROP WEAK PASSWORD DICTIONARY;
```

密码重用

用户修改密码时，只有超过不可重用天数（password_reuse_time）或不可重用次数（password_reuse_max）的密码才可以使用。参数配置说明如表 10-13 所示。

说明

- 不可重用天数默认值为 60 天，不可重用次数默认值是 0。这两个参数值越大越安全，但是在使用过程中会带来不便，其默认值符合安全标准，用户可以根据需要重新设置参数，提高安全等级。

配置 password_reuse_time 参数。

(1) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(2) 查看已配置的参数。

```
postgres=# SHOW password_reuse_time;
password_reuse_time
--------------------
 60
(1 row)
```

如果显示结果不为 60，执行"\q”命令退出数据库。

(3) 执行如下命令设置成默认值 60。

```
gs_guc reload    -N all -I all -c "password_reuse_time=60"
```

说明

- 不建议设置为 0。如需要设置，则需将所有节点中的 password_reuse_time 都设置为 0，才能生效。

配置 password_reuse_max 参数。

(1) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(2) 查看已配置的参数。

```
postgres=# SHOW password_reuse_max;
password_reuse_max
-------------------
 0
(1 row)
```

如果显示结果不为 0，执行"\q”命令退出数据库。

(3) 执行如下命令设置成默认值 0。

```
gs_guc reload    -N all -I all -c "password_reuse_max = 0"
```

密码有效期限

数据库用户的密码都有密码有效期（password_effect_time），当达到密码到期提醒天数（password_notify_time）时，系统会在用户登录数据库时提示用户修改密码。

说明

- 考虑到数据库使用特殊性及业务连续性，密码过期后用户还可以登录数据库，但是每次登录都会提示修改密码，直至修改为止。

配置 password_effect_time 参数。

(1) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(2) 查看已配置的参数。

```
postgres=# SHOW password_effect_time;
password_effect_time
---------------------
90
(1 row)
```

如果显示结果不为 90，执行"\q”命令退出数据库。

(3) 执行如下命令设置成默认值 90（不建议设置为 0）。

```
gs_guc reload    -N all -I all -c "password_effect_time = 90"
```

配置 password_notify_time 参数。

(1) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(2) 查看已配置的参数。

```
postgres=# SHOW password_notify_time;
password_notify_time
---------------------
 7
(1 row)
```

(3) 如果显示结果不为 7，执行如下命令设置成默认值 7（不建议设置为 0）。

```
gs_guc reload    -N all -I all -c "password_notify_time = 7"
```

密码修改

- 在安装数据库时，会新建一个和初始化用户重名的操作系统用户，为了保证帐户安全，请定期修改操作系统用户的密码。

以修改用户 user1 密码为例，命令格式如下：

```
passwd user1
```

根据提示信息完成修改密码操作。

- 建议系统管理员和普通用户都要定期修改自己的帐户密码，避免帐户密码被非法窃取。

以修改用户 user1 密码为例，以系统管理员用户连接数据库并执行如下命令：

```
postgres=# ALTER USER user1 IDENTIFIED BY "1234@abc" REPLACE "5678@def";
ALTER ROLE
```

说明

- 1234@abc、5678@def 分别代表用户 user1 的新密码和原始密码，这些密码要符合规则，否则会执行失败。
- 管理员可以修改自己的或者其他帐户的密码。通过修改其他帐户的密码，解决用户密码遗失所造成无法登录的问题。

以修改用户 joe 帐户密码为例，命令格式如下：

```
postgres=# ALTER USER joe IDENTIFIED BY "abc@1234";
ALTER ROLE
```

说明

- 系统管理员之间不允许互相修改对方密码。
- 系统管理员可以修改普通用户密码且不需要用户原密码。
- 系统管理员修改自己密码但需要管理员原密码。

密码验证

设置当前会话的用户和角色时，需要验证密码。如果输入密码与用户的存储密码不一致，则会报错。

```
postgres=# SET ROLE joe PASSWORD "abc@1234";
ERROR: Invalid username/password,set role denied.
```

以设置用户 joe 为例，命令格式如下：

```
postgres=# SET ROLE joe PASSWORD "abc@1234";
```

ERROR: Invalid username/password,set role denied.

表 10-12 特殊字符

编号	字符	编号	字符	编号	字符	编号	字符
1	~	9	*	17	|	25	<
2	!	10	(	18	[	26	.
3	@	11	)	19	{	27	>
4	#	12	-	20	}	28	/
5	$	13	_	21	]	29	?
6	%	14	=	22	;	-	-
7	^	15	+	23	:	-	-

表 10-13 不可重用天数和不可重用次数参数说明

参数	取值范围	配置说明
不可重用天数 password_reuse_time	正数或 0，其中整数部分表示天数，小数部分可以换算成时，分，秒。 默认值为 60。	如果参数变小，则后续修改密码按新的参数进行检查。 如果参数变大（比如由 a 变大为 b），因为 b 天之前的历史密码可能已经删除，所以 b 天之前的密码仍有可能被重用。则后续修改密码按新的参数进行检查。 说明：时间以绝对时间为准，历史密码记录的都是当时的时间，不识别时间的修改。
不可重用次数 password_reuse_max	正整数或 0。 默认值为 0，表示不检查重用次数。	如果参数变小，则后续修改密码按新的参数进行检查。 如果参数变大（比如由 a 变大为 b），因为 b 次之前的历史密码可能已经删除，所以 b 次之

		前的密码仍有可能被重用。则后续修改密码按新的参数进行检查。

(4) 设置用户密码失效。

具有 CREATEROLE 权限的用户在创建用户时可以强制用户密码失效，新用户首次登陆数据库后需要修改密码才允许执行其他查询操作，命令格式如下：

```
postgres=# CREATE USER joe PASSWORD "abc@1234" EXPIRED;
CREATE ROLE
```

具有 CREATEROLE 权限的用户可以强制用户密码失效或者强制修改密码且失效，命令格式如下：

```
postgres=# ALTER USER joe PASSWORD EXPIRED;
ALTER ROLE
postgres=# ALTER USER joe PASSWORD "abc@2345" EXPIRED;
ALTER ROLE
```

说明

- 密码失效的用户登录数据库后，当执行简单查询或者扩展查询时，会提示用户修改密码。修改密码后可以正常执行语句。
- 只有初始用户、系统管理员（sysadmin）或拥有创建用户（CREATEROLE）权限的用户才可以设置用户密码失效，其中系统管理员也可以设置自己或其他系统管理员密码失效。不允许设置初始用户密码失效。

10.3 设置数据库审计

10.3.1审计概述

背景信息

数据库安全对数据库系统来说至关重要。GBase 8c 将用户对数据库的所有操作写入审计日志。数据库安全管理员可以利用这些日志信息，重现导致数据库现状的一系列事件，找出非法操作的用户、时间和内容等。

关于审计功能，用户需要了解以下几点内容：

- 审计总开关 audit_enabled 支持动态加载。在数据库运行期间修改该配置项的值会立即生效，无需重启数据库。默认值为 on，表示开启审计功能。

- 除了审计总开关，各个审计项也有对应的开关。只有开关开启，对应的审计功能才能生效。
- 各审计项的开关支持动态加载。在数据库运行期间修改审计开关的值，不需要重启数据库便可生效。

目前，GBase 8c 支持以下审计项如表 10-14 所示。

表 10-14 配置审计项

配置项	描述
用户登录、注销审计	参数：audit_login_logout 默认值为 7，表示开启用户登录、退出的审计功能。设置为 0 表示关闭用户登录、退出的审计功能。不推荐设置除 0 和 7 之外的值。
数据库启动、停 止、恢复和切换审计	参数：audit_database_process 默认值为 1，表示开启数据库启动、停止、恢复和切换的审计功能。
用户锁定和解锁审计	参数：audit_user_locked 默认值为 1，表示开启审计用户锁定和解锁功能。
用户访问越权审计	参数：audit_user_violation 默认值为 0，表示关闭用户越权操作审计功能。
授权和回收权限审计	参数：audit_grant_revoke 默认值为 1，表示开启审计用户权限授予和回收功能。
数据库对象的 CREATE，ALTER，DROP 操作审计	参数：audit_system_object 默认值为 12295，表示只对 DATABASE、SCHEMA、USER、DATA SOURCE 这四类数据库对象的 CREATE、ALTER、DROP 操作进行审计。
具体表的 INSERT、UPDATE 和 DELETE	参数：audit_dml_state 默认值为 0，表示关闭具体表的 DML 操作（SELECT 除外）审计功能。

操作审计	
SELECT 操作审计	参数：audit_dml_state_select 默认值为 0，表示关闭 SELECT 操作审计功能。
COPY 审计	参数：audit_copy_exec 默认值为 1，表示开启 copy 操作审计功能。
存储过程和自定义函数的执行审计	参数：audit_function_exec 默认值为 0，表示不记录存储过程和自定义函数的执行审计日志。
SET 审计	参数：audit_set_parameter 默认值为 1，表示记录 set 操作审计日志
事务 ID 记录	参数：audit_xid_info 默认值为 0，表示关闭审计日志记录事务 ID 功能。

安全相关参数及说明请参见表 10-15。

表 10-15 安全相关参数及说明

参数名	说明
ssl	指定是否启用 SSL 连接。
require_ssl	指定服务器端是否强制要求 SSL 连接。
ssl_ciphers	指定 SSL 支持的加密算法列表。
ssl_cert_file	指定包含 SSL 服务器证书的文件的名称。
ssl_key_file	指定包含 SSL 私钥的文件名称。
ssl_ca_file	指定包含 CA 信息的文件的名称。
ssl_crl_file	指定包含 CRL 信息的文件的名称。

password_policy	指定是否进行密码复杂度检查。
password_reuse_time	指定是否对新密码进行可重用天数检查。
password_reuse_max	指定是否对新密码进行可重用次数检查。
password_lock_time	指定帐户被锁定后自动解锁的时间。
failed_login_attempts	如果输入密码错误的次数达到此参数值时，当前帐户被锁定。
password_encryption_type	指定采用何种加密方式对用户密码进行加密存储。
password_min_uppercase	密码中至少需要包含大写字母的个数。
password_min_lowercase	密码中至少需要包含小写字母的个数。
password_min_digital	密码中至少需要包含数字的个数。
password_min_special	密码中至少需要包含特殊字符的个数。
password_min_length	密码的最小长度。 说明：在设置此参数时，请将其设置成不大于password_max_length，否则进行涉及密码的操作会一直出现密码长度错误的提示
password_max_length	密码的最大长度。 说明：在设置此参数时，请将其设置成不小于password_min_length，否则进行涉及密码的操作会一直出现密码长度错误的提示。
password_effect_time	密码的有效期限。
password_notify_time	密码到期提醒的天数。
audit_enabled	控制审计进程的开启和关闭。
audit_directory	审计文件的存储目录。

audit_data_format	审计日志文件的格式，当前仅支持二进制格式（binary）。
audit_rotation_interval	指定创建一个新审计日志文件的时间间隔。当现在的时间减去上次创建一个审计日志的时间超过了此参数值时，服务器将生成一个新的审计日志文件。
audit_rotation_size	指定审计日志文件的最大容量。当审计日志消息的总量超过此参数值时，服务器将生成一个新的审计日志文件。
audit_resource_policy	控制审计日志的保存策略，以空间还是时间限制为优先策略，on 表示以空间为优先策略。
audit_file_remain_time	表示需记录审计日志的最短时间要求，该参数在 audit_resource_policy 为 off 时生效。
audit_space_limit	审计文件占用磁盘空间的最大值。
audit_file_remain_threshold	审计目录下审计文件的最大数量。
audit_login_logout	指定是否审计数据库用户的登录（包括登录成功和登录失败）、注销。
audit_database_process	指定是否审计数据库启动、停止、切换和恢复的操作。
audit_user_locked	指定是否审计数据库用户的锁定和解锁。
audit_user_violation	指定是否审计数据库用户的越权访问操作。
audit_grant_revoke	指定是否审计数据库用户权限授予和回收的操作。
audit_system_object	指定是否审计数据库对象的 CREATE、DROP、ALTER 操作。
audit_dml_state	指定是否审计具体表的 INSERT、UPDATE、DELETE 操作。
audit_dml_state_select	指定是否审计 SELECT 操作。
audit_copy_exec	指定是否审计 COPY 操作。

audit_function_exec	指定在执行存储过程、匿名块或自定义函数（不包括系统自带函数）时是否记录审计信息。
audit_set_parameter	指定是否审计 SET 操作。
enableSeparationOfDuty	指定是否开启三权分立。
session_timeout	建立连接会话后，如果超过此参数的设置时间，则会自动断开连接。
auth_iteration_count	认证加密信息生成过程中使用的迭代次数。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(3) 检查审计总开关状态。

用 show 命令显示审计总开关 audit_enabled 的值。

```
SHOW audit_enabled;
```

如果显示为 off，执行"\q”命令退出数据库。

执行如下命令开启审计功能，参数设置立即生效。

```
gs_guc set -N all -I all    -c "audit_enabled=on"
```

(4) 配置具体的审计项。

说明

- 只有开启审计功能，用户的操作才会被记录到审计文件中。
- 各审计项的默认参数都符合安全标准，用户可以根据需要开启其他审计功能，但会对性能有一定影响。

以开启对数据库所有对象的增删改操作的审计开关为例，其他配置项的修改方法与此相同，修改配置项的方法如下所示：

```
gs_guc reload    -N all -I all -c "audit_system_object=12295"
```

其中，audit_system_object 代表审计项开关，12295 为该审计开关的值。

10.3.2查看审计结果

前提条件

- 审计功能总开关已开启。
- 需要审计的审计项开关已开启。
- 数据库正常运行，并且对数据库执行了一系列增、删、改、查操作，保证在查询时段内有审计结果产生。
- 数据库各个节点审计日志单独记录。

背景信息

- 只有拥有 AUDITADMIN 属性的用户才可以查看审计记录。有关数据库用户及创建用户的办法请参见用户。
- 审计查询命令是数据库提供的 sql 函数 pg_query_audit，其原型为：

```
pg_query_audit(timestamptz      startime,timestamptz    endtime,audit_log)
```

参数 startime 和 endtime 分别表示审计记录的开始时间和结束时间，audit_log 表示所查看的审计日志信息所在的物理文件路径，当不指定 audit_log 时，默认查看连接当前实例的审计日志信息。

说明

- startime 和 endtime 的差值代表要查询的时间段，其有效值为从 startime 日期中的 00:00:00 开始到 endtime 日期中的 23:59:59 之间的任何值。请正确指定这两个参数，否则将查不到需要的审计信息。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(3) 查询审计记录。

```
postgres=# select * from pg_query_audit('2022-06-06 16:00:00','2022-06-06 17:00:00');
```

查询结果如下：

```
            time              |        type         | result | userid | username | database
| client_conninfo |     object_name     |
```

```
                                        detail_info
                                            | node_name |                  thread_id                  |
local_port | remote_port
----------------------+---------------+--------+--------+----------+----------
+-----------------+-----------------+-----------------------------------------
-----------------------------------------------------------------------------
--------------------------------+-----------+--------------------------------+-
-----------+-------------
 2022-06-06 16:00:00+08 | login_success | ok        | 10        | gbase       | postgres
| gs_ctl@XX.X.X.X | postgres              | login db(postgres) success,the current use
r is:gbase, SSL=off
                                            | hg1           | 140125749114624@707817600787469 |
5601            | 40412
......
```

该条记录表明，用户 gbase 在 time 字段标识的时间点登录数据库 postgres。其中 client_conninfo 字段在 log_hostname 启动且 IP 连接时，字符@后显示反向 DNS 查找得到的主机名。

说明

- 对于登录操作的记录，审计日志 detail_info 结尾会记录 SSL 信息，SSL=on 表示客户端通过 SSL 连接，SSL=off 表示客户端没有通过 SSL 连接。

10.3.3维护审计日志

前提条件

用户必须拥有审计权限。

背景信息

- 与审计日志相关的配置参数与其含义请参见表 10-16。

表 10-16 审计日志相关配置参数

配置项	含义	默认值
audit_directory	审计文件的存储目录。	/var/log/gbase/gbase/pg_audit/dn_6001
audit_resource_policy	审计日志的保存策略。	on（表示使用空间配置策略）
audit_space_limit	审计文件占用的磁盘	1GB

	空间总量。	
audit_file_remain_time	审计日志文件的最小保存时间。	90
audit_file_remain_thresh old	审计目录下审计文件的最大数量。	1048576

说明

- 如果使用 gs_om 工具部署数据库，则审计日志路径为"/var/log/gbase/gbase/pg_audit/dn_6001”。
- 审计日志删除命令为数据库提供的 sql 函数 pg_delete_audit，其原型为：

```
pg_delete_audit(timestamp startime,timestamp endtime)
```

其中参数 startime 和 endtime 分别表示审计记录的开始时间和结束时间。

- 目前常用的记录审计内容的方式有两种：记录到数据库的表中、记录到 OS 文件中。这两种方式的优缺点比较如表 10-17 所示。

表 10-17 审计日志保存方式比较

方式	优点	缺点
记录到表中	不需要用户维护审计日志。	由于表是数据库的对象，如果一个数据库用户具有一定的权限，就能够访问到审计表。如果该用户非法操作审计表，审计记录的准确性难以得到保证。
记录到 OS 文件中	比较安全，即使一个帐户可以访问数据库，但不一定有访问 OS 这个文件的权限。	需要用户维护审计日志。

从数据库安全角度出发，GBase 8c 采用记录到 OS 文件的方式来保存审计结果， 保证了审计结果的可靠性。

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

(3) 选择日志维护方式进行维护。

- 设置自动删除审计日志

审计文件占用的磁盘空间或者审计文件的个数超过指定的最大值时，系统将删除最早的审计文件，并记录审计文件删除信息到审计日志中。

说明

- 审计文件占用的磁盘空间大小默认值为 1024MB，用户可以根据磁盘空间大小重新设置参数。

① 配置审计文件占用磁盘空间的大小（audit_space_limit）。

查看已配置的参数。

```
postgres=# SHOW audit_space_limit;
audit_space_limit
------------------
1GB
(1 row)
```

如果显示结果不为 1GB（1024MB），执行"\q”命令退出数据库。

建议执行如下命令设置成默认值 1024MB。

```
gs_guc reload    -N all -I all -c "audit_space_limit=1024MB"
```

② 配置审计文件个数的最大值（audit_file_remain_threshold）。

查看已配置的参数。

```
postgres=# SHOW audit_file_remain_threshold;
audit_file_remain_threshold
---------------------------
1048576
(1 row)
```

如果显示结果不为 1048576，执行"\q”命令退出数据库。

建议执行如下命令设置成默认值 1048576。

```
gs_guc reload    -N all -I all -c "audit_file_remain_threshold=1048576"
```

- 手动备份审计文件

当审计文件占用的磁盘空间或者审计文件的个数超过配置文件指定的值时，系统将会自动删除较早的审计文件，因此建议用户周期性地对比较重要的审计日志进行保存。

使用 show 命令获得审计文件所在目录（audit_directory）。

```
postgres=# SHOW audit_directory;
```

将审计目录整个拷贝出来进行保存。

- 手动删除审计日志

当不再需要某时段的审计记录时，可以使用审计接口命令 pg_delete_audit 进行手动删除。

以删除 2012/9/20 到 2012/9/21 之间的审计记录为例：

```
postgres=# SELECT pg_delete_audit('2012-09-20 00:00:00','2012-09-21 23:59:59');
```

10.3.4设置文件权限安全策略

背景信息

数据库在安装过程中，会自动对其文件权限（包括运行过程中生成的文件，如日志文件等）进行设置。其权限规则如下：

- 数据库程序目录的权限为 0750。
- 数据库数据文件目录的权限为 0700。
- 数据库部署时通过创建 xml 配置文件中的 tmpMppdbPath 参数指定目录（若未指定，则默认创建/tmp/$USER_mppdb 目录）来存放".s.PGSQL.*”文件，该目录 和文件权限设置为 0700。
- 数据库的数据文件、审计日志和其他数据库程序生成的数据文件的权限为 0600， 运行日志的权限默认不高于 0640。
- 普通操作系统用户不允许修改和删除数据库文件和日志文件。

数据库程序目录及文件权限

数据库安装后，部分程序目录及文件权限如表 10-18 所示。

表 10-18 文件及目录权限

文件/目录	父目录	权限

bin	-	0700
lib	-	0700
share	-	0700
data(数据库节点/数据库主节点)	-	0700
base	实例数据目录	0700
global	实例数据目录	0700
pg_audit	实例数据目录（可配置）	0700
pg_log	实例数据目录（可配置）	0700
pg_xlog	实例数据目录	0700
postgresql.conf	实例数据目录	0600
pg_hba.conf	实例数据目录	0600
postmaster.opts	实例数据目录	0600
pg_ident.conf	实例数据目录	0600
gs_initdb	bin	0700
gs_dump	bin	0700
gs_ctl	bin	0700
gs_guc	bin	0700
gsql	bin	0700
archive_status	pg_xlog	0700
libpq.so.5.5	lib	0600

建议

数据库在安装过程中，会自动对其文件权限（包括运行过程中生成的文件，如日志文件等）进行设置，适合大多数情况下的权限要求。如果用户产品对相关权限有特殊要求，建议用户安装后定期检查相关权限设置，确保完全符合产品要求。

10.4 设置密态等值查询

10.4.1密态等值查询概述

随着企业数据上云，数据的安全隐私保护面临越来越严重的挑战。密态数据库将解决数据整个生命周期中的隐私保护问题，涵盖网络传输、数据存储以及数据运行态；更进一步，密态数据库可以实现云化场景下的数据隐私权限分离，即实现数据拥有者和实际数据管理者的数据读取能力分离。密态等值查询将优先解决密文数据的等值类查询问题。密态等值查询目前支持客户端工具 gsql 和 JDBC。接下来分别介绍如何使用客户端工具执行密态等值查询的相关操作。

10.4.2使用 gsql 操作密态数据库

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 执行以下命令打开密态开关，连接密态数据库。

```
gsql -p PORT postgres -r -C
```

(3) 创建客户端主密钥 CMK 和列加密密钥 CEK。具体涉及到的新增创建 CMK 的语法参考《GBase 8c V5_5.0.0_SQL 参考手册》CREATE CLIENT MASTER KEY 章节，创建的 CEK 的语法参考《GBase 8c V5_5.0.0_SQL 参考手册》 CREATE COLUMN ENCRYPTION KEY

```
--创建客户端加密主密钥(CMK)
postgres=# CREATE CLIENT MASTER KEY ImgCMK1 WITH (KEY_STORE = localkms,
KEY_PATH ="key_path_value1", ALGORITHM = RSA_2048);
postgres=# CREATE CLIENT MASTER KEY ImgCMK WITH (KEY_STORE = localkms,
KEY_PATH ="key_path_value2", ALGORITHM = RSA_2048);
postgres=# CREATE COLUMN ENCRYPTION KEY ImgCEK1 WITH VALUES
(CLIENT_MASTER_KEY = ImgCMK1, ALGORITHM =
AEAD_AES_256_CBC_HMAC_SHA256);
CREATE COLUMN ENCRYPTION KEY
```

查询存储密钥信息的系统表结果如下。

```
postgres=# SELECT * FROM gs_client_global_keys;
global_key_name | key_namespace | key_owner |    key_acl |     create_date
----------------+---------------+-----------+---------+----------------------------
imgcmk1        | 2200          |     10    | 2022-06-06 11:04:00.656617
imgcmk         | 2200          |     10    | 2022-06-06 11:04:05.389746
(2 rows)
postgres=# SELECT column_key_name,column_key_distributed_id ,global_key_id,key_owner
FROM gs_column_keys;
column_key_name | column_key_distributed_id | global_key_id | key_owner
----------------+---------------------------+---------------+-----------
imgcmk1        |                           |     10    |   2022-06-06 11:04:00.656617
imgcmk         |                           |     10    |   2022-06-06 11:04:05.389746
(2 rows)
postgres=# SELECT column_key_name,column_key_distributed_id ,global_key_id,key_owner
FROM gs_column_keys;
column_key_name | column_key_distributed_id | global_key_id | key_owner
----------------+---------------------------+---------------+-----------
imgcek1     |    760411027    |            16392        |      10
imgcek      |    3618369306 |            16398        |      10
(2 rows)
```

(4) 创建加密表。

```
postgres=# CREATE TABLE creditcard_info (id_number       int, name   text encrypted with
(column_encryption_key = ImgCEK, encryption_type = DETERMINISTIC), credit_card
varchar(19) encrypted with (column_encryption_key = ImgCEK1, encryption_type =
DETERMINISTIC));
NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'id_number' as the distribution
column by default.
HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

查询表的详细信息如下，Modifiers 值为 encrypted 则表示该列是加密列。

```
postgres=# \d creditcard_info
Table "public.creditcard_info" Column       |      Type| Modifiers
------------+------------------+------------
id_number | integer
name     | text
credit_card | character varying | encrypted
```

(5) 向加密表插入数据并进行等值查询。

```
postgres=# INSERT INTO creditcard_info VALUES (1,'joe','6217986500001288393');
```

```
INSERT 0 1
postgres=# INSERT INTO creditcard_info VALUES (2, 'joy','6219985678349154323 3');
INSERT 0 1
postgres=# select * from creditcard_info where name = 'joe';
id_number | name |       credit_card
-----------+------+--------------------
1 | joe | 6217986500001288393
(1 row)
```

注意：使用非密态客户端查看该加密表数据时是密文

```
postgres=# select id_number,name from creditcard_info;
id_number |   name
-----------+--------------------
1 |
\x011aefabd754ded0a536a96664790622487c4d366d313aecd5839e410a46d29cba96a60e48310
00000ee7905 6a114c9a6c041bb552b78052e912a8b730609142074c63791abebd0d38
2 |
\x011aefabd76853108eb406c0f90e7c773b71648fa6e2b8028cf634b49aec65b4fcfb376f3531000
000f7471c868 6682de215d09aa87113f6fb03884be2031ef4dd967afc6f7901646b
(2 rows)
```

(6) （可选）对加密表进行 alter 和 update 操作。

```
postgres=# ALTER TABLE creditcard_info ADD COLUMN age int ENCRYPTED WITH
(COLUMN_ENCRYPTION_KEY = ImgCEK, ENCRYPTION_TYPE = DETERMINISTIC);
ALTER TABLE
postgres=# \d creditcard_info
Table "public.creditcard_info" Column      | Type| Modifiers
-------------+-------------------+------------
id_number | integer
name      | text
credit_card | character varying | encrypted
age       | integer       | encrypted
postgres=# ALTER TABLE creditcard_info DROP COLUMN age;
ALTER TABLE
postgres=# update creditcard_info set credit_card = '154320000011111111' where name = 'joy';
UPDATE 1
postgres=# select * from creditcard_info where name = 'joy';
id_number | name | credit_card
-----------+------+-------------------
2 | joy | 154320000011111111
(1 row)
```

10.4.3使用 JDBC 操作密态数据库

连接密态数据库

连接密态数据库需要使用驱动包 gsjdbc4.jar，具体 JDBC 连接参数参考基于 JDBC 开发章节介绍。JDBC 支持密态数据库相关操作，需要设置 enable_ce=1，示例如下。

```
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://10.10.0.13:15400/postgres?enable_ce=1"; 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;
};
```

- 【建议】使用 JDBC 操作密态数据库时，一个数据库连接对象对应一个线程，否则，不同线程变更可能导致冲突。
- 【建议】使用 JDBC 操作密态数据库时，不同 connection 对密态配置数据有变更，由客户端调用 isvalid 方法保证 connection 能够持有变更后的密态配置数据，此时需要保证参数 refreshClientEncryption 为 1(默认值为 1)，在单客户端操作密态数据场景下，

refreshClientEncryption 参数可以设置为 0。

使用 isValid 方法刷新缓存示例

```
// 创建客户端主密钥
Connection conn1 = DriverManager.getConnection("url","user","password");
// conn1 通过调用 isValid 刷新缓存 try {
if (!conn1.isValid(60)) {
System.out.println("isValid Failed for connection 1");
}
} catch (SQLException e) { e.printStackTrace();
return null;
}
```

执行密态等值查询相关的创建密钥语句

```
// 创建客户端主密钥
Connection conn = DriverManager.getConnection("url","user","password"); Statement stmt =
conn.createStatement();
int rc = stmt.executeUpdate("CREATE CLIENT MASTER KEY ImgCMK1 WITH
( KEY_STORE = localkms, KEY_PATH = \"key_path_value\" , ALGORITHM = RSA_2048);
```

说明

- 创建密钥之前需要使用 gs_ktool 工具提前生成密钥，才能创建 CMK 成功。

```
// 创建列加密密钥
int rc2 = stmt.executeUpdate("CREATE COLUMN ENCRYPTION KEY ImgCEK1 WITH
VALUES (CLIENT_MASTER_KEY = ImgCMK1, ALGORITHM =
AEAD_AES_256_CBC_HMAC_SHA256);");
```

执行密态等值查询相关的创建加密表的语句

```
int rc3 = stmt.executeUpdate("CREATE TABLE creditcard_info (id_number int, name
varchar(50) encrypted with (column_encryption_key = ImgCEK1, encryption_type =
DETERMINISTIC),credit_card varchar(19) encrypted with (column_encryption_key =
ImgCEK1, encryption_type = DETERMINISTIC));");
// 插入数据
int rc4 = stmt.executeUpdate("INSERT INTO creditcard_info VALUES
(1,'joe','6217986500001288393');");
// 查询加密表
ResultSet rs = null;
rs = stmt.executeQuery("select * from creditcard_info where name = 'joe';");
// 关闭语句对象
stmt.close();
```

执行加密表的预编译 SQL 语句

```
// 调用 Connection 的 prepareStatement 方法创建预编译语句对象。
PreparedStatement pstmt = con.prepareStatement("INSERT INTO creditcard_info VALUES
(?, ?, ?);");
// 调用 PreparedStatement 的 setShort 设置参数。
pstmt.setInt(1, 2); pstmt.setString(2, "joy");
pstmt.setString(3, "6219985678349154323 3");
// 调用 PreparedStatement 的 executeUpdate 方法执行预编译 SQL 语句。
int rowcount = pstmt.executeUpdate();
// 调用 PreparedStatement 的 close 方法关闭预编译语句对象。
pstmt.close();
```

执行加密表的批处理操作

```
// 调用 Connection 的 prepareStatement 方法创建预编译语句对象。
Connection conn = DriverManager.getConnection("url","user","password");
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO batch_table (id, name,
address) VALUES (?,?,?)");
// 针对每条数据都要调用 setShort 设置参数，以及调用 addBatch 确认该条设置完毕。
int loopCount = 20;
for (int i = 1; i < loopCount + 1; ++i) { statemnet.setInt(1, i);
statemnet.setString(2, "Name " + i); statemnet.setString(3, "Address " + i);
// Add row to the batch. statemnet.addBatch();
}
// 调用 PreparedStatement 的 executeBatch 方法执行批处理。
int[] rowcount = pstmt.executeBatch();
// 调用 PreparedStatement 的 close 方法关闭预编译语句对象。
pstmt.close();
```

10.4.4密态支持函数/存储过程

密态支持函数/存储过程当前版本只支持 sql 和 PL/pgsql 两种语言。 由于密态支持存储过程中创建和执行函数/存储过程对用户是无感知的，因此语法和非密态无区别。

密态等值查询支持函数存储过程新增系统表 gs_encrypted_proc，用于存储参数返回的原始数据类型。

创建并执行涉及加密列的函数/存储过程

(1) 创建密钥，详细步骤请参考使用 gsql 操作密态数据库和使用 JDBC 操作密态数据库。

(2) 创建加密表。

```
postgres=# CREATE TABLE creditcard_info ( id_number int, name text, credit_card
varchar(19) encrypted with (column_encryption_key = ImgCEK1, encryption_type =
DETERMINISTIC)) with (orientation=row);
CREATE TABLE
```

(3) 插入数据。

```
postgres=# insert into creditcard_info values(1, 'Avi', '1234567890123456');
INSERT 0 1
postgres=# insert into creditcard_info values(2, 'Eli', '2345678901234567');
INSERT 0 1
```

(4) 创建函数支持密态等值查询。

```
postgres=# CREATE FUNCTION f_encrypt_in_sql(val1 text, val2 varchar(19)) RETURNS
text AS 'SELECT name from creditcard_info where name=$1 or credit_card=$2 LIMIT 1'
LANGUAGE SQL;
CREATE FUNCTION

postgres=# CREATE FUNCTION f_encrypt_in_plpgsql (val1 text, val2 varchar(19)) ETURNS
text AS $$ DECLARE c text;
postgres$# BEGIN
postgres$# SELECT into c name from creditcard_info where name=$1 or credit_card =$2
LIMIT 1;
postgres$#     RETURN c;
postgres$# END; $$
postgres-# LANGUAGE plpgsql;
CREATE FUNCTION
```

(5) 执行函数。

```
postgres=# SELECT f_encrypt_in_sql('Avi','1234567890123456');
f_encrypt_in_sql
-----------------
Avi
(1 row)

postgres=# SELECT f_encrypt_in_plpgsql('Avi', val2=>'1234567890123456');
f_encrypt_in_plpgsql
---------------------
Avi
(1 row)
```

- 函数/存储过程中的"执行动态查询语句"中的查询是在执行过程中编译，因此函数/存储过程中的表名、列名不能在创建阶段未知，输入参数不能用于表名、列名或以任何方式

连接。

- 在 RETURNS、IN 和 OUT 的参数中，不支持混合使用加密和非加密类型参数。虽然参数类型都是原始数据类型，但实际类型不同。
- 当前版本函数/存储过程的 LANGUAGE 只支持 SQL 和 PL/pgSQL，不支持 C 和 JAVA 等其他过程语言。
- 不支持在函数/存储过程中执行其他查询加密列的函数/存储过程。
- 当前版本不支持 default、DECLARE 中为变量赋予默认值，且不支持对 DECLARE 中的返回值进行解密，用户可以用执行函数时用输入参数，输出参数来代替使用。
- 不支持 gs_dump 对涉及加密列的 function 进行备份。
- 不支持在函数/存储过程中创建密钥。
- 该版本密态函数/存储过程不支持触发器
- 密态等值查询函数/存储过程不支持对 plpgsql 语言对语法进行转义，对于语法主体带有引号的语法 CREATE FUNCTION AS ‘语法主体'，可以用 CREATE FUNCTION AS $$语法主体$$代替。
- 不支持在密态等值查询函数/存储过程中执行修改加密列定义的操作，包括对创建加密表，添　加加密列，由于执行函数是在服务端，客户端没法判断是否是需要刷新缓存，得断开连接后或触发刷新客户端加密列缓存才可以对该列做加密操作。

10.5　设置账本数据库

10.5.1账本数据库概述

背景信息

账本数据库融合了区块链思想，将用户操作记录至两种历史表中：用户历史表和全局区块表。当用户创建防篡改用户表时，系统将自动为该表添加一个 hash 列来保存每行数据的 hash 摘要信息，同时在 blockchain 模式下会创建一张用户历史表来记录对应用户表中每条数据的变更行为；而用户对防篡改用户表的一次修改行为将记录至全局区块表中。由于历史表具有只可追加不可修改的特点，因此历史表记录串联起来便形成了用户对防篡改用户表的修改历史。

用户历史表命名和结构如下：

表 10-19 用户历史表 blockchain.<schemaname>_<tablename>_hist 所包含的字段

字段名	类型	描述
rec_num	bigint	行级修改操作在历史表中的执行序号。
hash_ins	hash16	INSERT 或 UPDATE 操作插入的数据行的 hash 值。
hash_del	hash16	DELETE 或 UPDATE 操作删除的数据行的 hash 值。
pre_hash	hash32	当前用户历史表的数据整体摘要。

表 10-20 hash_ins 与 hash_del 场景对应关系

-	hash_ins	hash_del
INSERT	(√) 插入行的 hash 值	空
DELETE	空	(√) 删除行的 hash 值。
UPDATE	(√) 新插入数据的 hash 值	(√) 删除前该行的 hash 值。

操作步骤

(1) 创建防篡改模式。

例如，创建防篡改模式 ledgernsp。

```
postgres=# CREATE SCHEMA ledgernsp WITH BLOCKCHAIN;
```

(2) 在防篡改模式下创建防篡改用户表。

例如，创建防篡改用户表 ledgernsp.usertable。

```
postgres=# CREATE TABLE ledgernsp.usertable(id int, name text);
```

查看防篡改用户表结构及其对应的用户历史表结构。

```
postgres=# \d+ ledgernsp.usertable;
postgres=# \d+ blockchain.ledgernsp_usertable_hist;
```

执行结果如下：

```
postgres=# \d+ ledgernsp.usertable;
                                Table "ledgernsp.usertable"
 Column |    Type     | Modifiers | Storage    | Stats target | Description
--------+-------------+-----------+------------+--------------+-------------
 id     | integer |                 | plain      |              |
 name   | text    |                 | extended |                |
 hash   | hash16  |                 | plain      |              |
Has OIDs: no
Options: orientation=row, compression=no
History table name: ledgernsp_usertable_hist

postgres=# \d+ blockchain.ledgernsp_usertable_hist;
                    Table "blockchain.ledgernsp_usertable_hist"
  Column   |   Type    | Modifiers | Storage | Stats target | Description
-----------+-----------+-----------+---------+--------------+-------------
 rec_num   | bigint |              | plain   |              |
 hash_ins | hash16 |              | plain   |              |
 hash_del | hash16 |              | plain   |              |
 pre_hash | hash32 |              | plain   |              |
Indexes:
    "gs_hist_16388_index" PRIMARY KEY, btree (rec_num int4_ops) TABLESPACE
pg_default
Has OIDs: no
Options: internal_mask=263
```

说明

- 防篡改表不支持非行存表、临时表、外表、unlog 表、非行存表均无防篡改属性。
- 防篡改表在创建时会自动增加一个名为 hash 的系统列，所以防篡改表单表最大列数为 1599。

注意

- dbe_perf 和 snapshot 两个模式不能 ALTER 为 blockchain 属性，如：ALTER SCHEMA dbe_perf WITH BLOCKCHAIN;。
- 系统模式不能 ALTER 为 blockchain 属性，如：ALTER SCHEMA pg_catalog WITH BLOCKCHAIN;。
- 包含表的 SCHEMA 不能通过 ALTER SCHEMA 语句修改属性为 blockchain。

(3) 修改防篡改用户表数据。

例如，对防篡改用户表执行 INSERT/UPDATE/DELETE。

```
postgres=# INSERT INTO ledgernsp.usertable VALUES(1, 'alex'), (2, 'bob'), (3, 'peter');
INSERT 0 3

postgres=# SELECT *, hash FROM ledgernsp.usertable ORDER BY id;
 id | name  |              hash
----+-------+------------------
  1 | alex  | 1f2e543c580cb8c5
  2 | bob   | 8fcd74a8a6a4b484
  3 | peter | f51b4b1b12d0354b
(3 rows)

postgres=# UPDATE ledgernsp.usertable SET name = 'bob2' WHERE id = 2;
UPDATE 1

postgres=# SELECT *, hash FROM ledgernsp.usertable ORDER BY id;
 id | name  |              hash
----+-------+------------------
  1 | alex  | 1f2e543c580cb8c5
  2 | bob2  | 437761affbb7c605
  3 | peter | f51b4b1b12d0354b
(3 rows)

postgres=# DELETE FROM ledgernsp.usertable WHERE id = 3;
DELETE 1

postgres=# SELECT *, hash FROM ledgernsp.usertable ORDER BY id;
 id | name |              hash
----+------+------------------
  1 | alex | 1f2e543c580cb8c5
  2 | bob2 | 437761affbb7c605
(2 rows)
```

10.5.2查看账本历史操作记录

前提条件

- 系统中需要有审计管理员或者具有审计管理员权限的角色。
- 数据库正常运行，并且对防篡改数据库执行了一系列增、删、改等操作，保证在查询时

段内有账本操作记录结果产生。

背景信息

- 只有拥有 AUDITADMIN 属性的用户才可以查看账本历史操作记录。有关数据库用户及创建用户的办法请参见用户。
- 查询全局区块表命令是直接查询 gs_global_chain 表，操作为：

```
SELECT * FROM gs_global_chain;
```

该表有 11 个字段，每个字段的含义见 GS_GLOBAL_CHAIN。

- 查询用户历史表的命令是直接查询 BLOCKCHAIN 模式下的用户历史表，操作为： 例如用户表所在的模式为 ledgernsp，表名为 usertable，则对应的用户历史表名为 blockchain.ledgernsp_usertable_hist;

```
SELECT * FROM blockchain.ledgernsp_usertable_hist;
```

用户历史表有 4 个字段，每个字段的含义见表 10-19。

说明

➢ 用户历史表的表名一般为 blockchain.<schemaname>_<tablename>_hist 形式。当防篡改用户表模式名或者表名过长导致前述方式生成的表名超出表名长度限制，则会采用 blockchain.<schema_oid>_<table_oid>_hist 的方式命名。

操作步骤

(1) 连接主节点，登录数据库查询全局区块表记录。

```
postgres=# SELECT * FROM gs_global_chain;
```

例如，查询结果如下：

```
blocknum | dbname | username |     starttime | relid | relnsp     | relname |      relhash | globalhash
txcgbaseand
----------+----------+----------+-------------------------------+-------+-----------+-----------+-------------
-----+----------------------------------+------------------
-----------------------------------------------------------
0 | postgres | gbase        | 2021-04-14 07:00:46.32757+08 | 16393 | ledgernsp | usertable |
a41714001181a294 | 6b5624e039e8aee36bff3e8295c75b40 | insert into ledge
rnsp.usertable values(1, 'alex'), (2, 'bob'), (3, 'peter');
1 | postgres | gbase        | 2021-04-14 07:01:19.767799+08 | 16393 | ledgernsp | usertable |
b3a9ed0755131181 | 328b48c4370faed930937869783c23e0 | update ledgernsp.
usertable set name = 'bob2' where id = 2;
```

```
2 | postgres | gbase       | 2021-04-14 07:01:29.896148+08 | 16393 | ledgernsp | usertable |
0ae4b4e4ed2fcab5 | aa8f0a236357cac4e5bc1648a739f2ef | delete from ledge
rnsp.usertable where id = 3;
```

该结果表明，用户 gbase 连续执行了三条 DML 命令，包括 INSERT、UPDATE 和 DELETE 操作。

(2) 查询历史表记录。

```
postgres=# SELECT * FROM blockchain.ledgernsp_usertable_hist;
```

查询结果如下：

```
rec_num |     hash_ins |      hash_del |     pre_hash
---------+------------------+-----------------+---------------------------------
0   | 1f2e543c580cb8c5 |                       |e1b664970d925d09caa295abd38d9b35
1   | 8fcd74a8a6a4b484 |                       |dad3ed8939a141bf3682043891776b67
2   | f51b4b1b12d0354b |                       | 53eb887fc7c4302402343c8914e43c69
3   | 437761affbb7c605 | 8fcd74a8a6a4b484 | c2868c5b49550801d0dbbbaa77a83a10
4   |                       | f51b4b1b12d0354b | 9c512619f6ffef38c098477933499fe3
(5 rows)
```

查询结果显示，用户 gbase 对 ledgernsp.usertable 表插入了 3 条数据，更新了 1 条数据，随后删除了 1 行数据，最后剩余 2 行数据，hash 值分别为 1f2e543c580cb8c5 和 437761affbb7c605。

(3) 查询用户表数据及 hash 校验列。

```
postgres=# SELECT *, hash FROM ledgernsp.usertable;
```

查询结果如下：

```
id | name |     hash
----+------+------------------
1 | alex | 1f2e543c580cb8c5
2 | bob2 | 437761affbb7c605
(2 rows)
```

查询结果显示，用户表中剩余 2 条数据，与步骤 2 中的记录一致。

10.5.3校验账本数据一致性

前提条件

数据库正常运行，并且对防篡改数据库执行了一系列增、删、改等操作，保证在查询时段内有账本操作记录结果产生。

背景信息

- 账本数据库校验功能目前提供两种校验接口，分别为： ledger_hist_check(text,...和 ledger_gchain_check(text...。普通用户调用校验接口，仅能校验自己有权限访问的表。
- 校验防篡改用户表和用户历史表的接口为 pg_catalog.ledger_hist_check，操作 为：

```
SELECT pg_catalog.ledger_hist_check(schema_name text,table_name text);
```

如果校验通过，函数返回 t，反之则返回 f。

校验防篡改用户表、用户历史表和全局区块表三者是否一致的接口为 pg_catalog.ledger_gchain_check，操作为：

```
SELECT pg_catalog.ledger_gchain_check(schema_name text, table_name text);
```

如果校验通过，函数返回 t，反之则返回 f。

操作步骤

(1) 校验防篡改用户表 ledgernsp.usertable 与其对应的历史表是否一致。

```
postgres=# SELECT pg_catalog.ledger_hist_check('ledgernsp', 'usertable');
```

查询结果如下：

```
ledger_hist_check
------------------
t
(1 row)
```

该结果表明：防篡改用户表和用户历史表中记录的结果能够一一对应，保持一致。

(2) 查询防篡改用户表 ledgernsp.usertable 与其对应的历史表以及全局区块表中关于该表的记录是否一致。

```
postgres=# SELECT pg_catalog.ledger_gchain_check('ledgernsp', 'usertable');
```

查询结果如下：

```
ledger_gchain_check
--------------------
t
(1 row)
```

查询结果显示，上述三表中关于 ledgernsp.usertable 的记录保持一致，未发生篡改行为。

10.5.4归档账本数据库

前提条件

- 系统中需要有审计管理员或者具有审计管理员权限的角色。
- 数据库正常运行，并且对防篡改数据库执行了一系列增、删、改等操作，保证在查询时段内有账本操作记录结果产生。
- 数据库已经正确配置审计文件的存储路径 audit_directory。

背景信息

- 账本数据库归档功能目前提供两种校验接口，分别为：ledger_hist_archive(text, text)和ledger_gchain_archive(void)，详见《GBase 8c V5_5.0.0_SQL 参考手册》中"账本数据库的函数”章节。账本数据库接口仅审计管理员可以调用。
- 归档用户历史表的接口为 pg_catalog.ledger_hist_archive，表示归档当前 DN 节点的用户历史表数据。操作为：

```
SELECT pg_catalog.ledger_hist_archive(schema_name text,table_name text);
```

如果归档成功，函数返回 t，反之则返回 f。

归档全局区块表的接口为 pg_catalog.ledger_gchain_archive，表示归档当前主节点的全局历史表数据。操作为：

```
SELECT pg_catalog.ledger_gchain_archive();
```

如果归档成功，函数返回 t，反之则返回 f。

操作步骤

(1) 使用 EXECUTE DIRECT 对某个 DN 节点进行归档操作。例如：

```
postgres=# EXECUTE DIRECT ON (dn1) 'SELECT
pg_catalog.ledger_hist_archive("ledgernsp", "usertable");';
```

执行结果如下：

```
ledger_hist_archive
--------------------
t
(1 row)
```

用户历史表将归档为一条数据：

```
postgres=# EXECUTE DIRECT ON (datanode1) 'SELECT * FROM
blockchain.ledgernsp_usertable_hist;';
rec_num |      hash_ins |      hash_del |      pre_hash
---------+-----------------+-----------------+--------------------------------
3 | e78e75b00d396899 | 8fcd74a8a6a4b484 | fd61cb772033da297d10c4e658e898d7
(1 row)
```

该结果表明当前节点用户历史表导出成功。

(2) 连接登录主节点，执行全局区块表导出操作。

```
postgres=# SELECT pg_catalog.ledger_gchain_archive();
```

执行结果如下：

```
ledger_gchain_archive
----------------------
t
(1 row)
```

全局历史表将以用户表为单位归档为 N(用户表数量)条数据：

```
postgres=# SELECT * FROM gs_global_chain;
blocknum |      dbname     | username |   starttime | relid |    relnsp      |    relname |      relhash |
    globalhash      | txcgbaseand
----------+----------+----------+-----------------------------+-------+-----------+-----------+-------------
-----+----------------------------------+-----------
1 | postgres | libc    | 2021-05-10 19:59:38.619472+08 | 16388 | ledgernsp | usertable |
57c101076694b415 | be82f98ee68b2bc4e375f69209345406 | Archived.
(1 row)
```

该结果表明，当前节点全局区块表导出成功。

10.5.5修复账本数据库

前提条件

- 系统中需要有审计管理员或者具有审计管理员权限的角色。
- 数据库正常运行，并且对防篡改数据库执行了一系列增、删、改等操作，保证在查询时段内有账本操作记录结果产生。

背景信息

- 当在异常情况或表被损坏时需要使用账本数据库的函数章节中的ledger_gchain_repair(text, text)或 ledger_hist_repair(text, text)接口对全局区块表或用户历

史表进行修复，修复后调用全局区块表或用户历史表校验接口结果为 true。

- 修复用户历史表的接口为 pg_catalog.ledger_hist_repair，操作为：

```
SELECT pg_catalog.ledger_hist_repair(schema_name text,table_name text);
```

如果修复成功，函数返回修复过程中用户历史表 hash 的增量。

- 归档全局区块表的接口为 pg_catalog.ledger_gchain_repair，操作为：

```
SELECT pg_catalog.ledger_gchain_repair(schema_name text,table_name text);
```

如果修复成功，函数返回修复过程中全局区块表 hash 的增量。

操作步骤

(1) 执行历史表修复操作。

```
postgres=# SELECT pg_catalog.ledger_hist_repair('ledgernsp', 'usertable');
```

查询结果如下：

```
ledger_hist_repair
--------------------
84e8bfc3b974e9cf
 (1 row)
```

该结果表明当前节点用户历史表修复成功，修复造成的用户历史表 hash 增量为 84e8bfc3b974e9cf。

(2) 执行全局区块表修复操作。

```
postgres=# SELECT pg_catalog.ledger_gchain_repair('ledgernsp', 'usertable');
```

查询结果如下：

```
ledger_gchain_repair
----------------------
a41714001181a294
(1 row)
```

该结果表明，全局区块表修复成功，且插入一条修复数据，其 hash 值为 a41714001181a294。

11 参数配置

GBase 8c 数据库提供了许多运行参数，配置这些 GUC 参数可以影响数据库系统的行为。为了使数据库与业务的配合度更高，用户需要根据业务场景和数据量的大小进行 GUC 参数调整，详见《GBase 8c V5_5.0.0_数据库参考手册》。

11.1 查看参数

操作步骤

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 使用如下命令连接数据库。

```
gsql -d postgres -p 15400
```

postgres 为需要连接的数据库名称，15400 为数据库主节点的端口号。

(3) 查看数据库运行参数当前取值。

- 使用 SHOW 命令。

  使用如下命令查看单个参数：

```
SHOW server_version;
```

  server_version 显示数据库版本信息的参数。

  使用如下命令查看所有参数：

```
SHOW ALL;
```

- 使用 pg_settings 视图。

  使用如下命令查看单个参数：

```
SELECT * FROM pg_settings WHERE NAME='server_version';
```

  使用如下命令查看所有参数：

```
SELECT * FROM pg_settings;
```

示例

查看服务器的版本号。

```
postgres=# SHOW server_version;
 server_version
----------------
```

```
 9.2.4
(1 row)
```

11.2 设置参数

GBase 8c 提供了多种修改 GUC 参数的方法，用户可以方便的针对数据库、用户、会话进行设置。

- 参数名称不区分大小写。
- 参数取值有整型、浮点型、字符串、布尔型和枚举型五类。

  布尔值可以是（on，off）、（true，false）、（yes，no）或者（1，0），且不区分大小写。

  枚举类型的取值是在系统表 pg_settings 的 enumvals 字段取值定义的。
- 对于有单位的参数，在设置时请指定单位，否则将使用默认的单位。

  参数的默认单位在系统表 pg_settings 的 unit 字段定义的。

  内存单位有：KB（千字节）、MB（兆字节）和 GB（吉字节）。

  时间单位：ms（毫秒）、s（秒）、min（分钟）、h（小时）和 d（天）。

  GBase 8c 提供了六类 GUC 参数，具体分类和设置方式请参考下表：

表 11-1 GUC 参数说明

参数类型	说明	设置方式
INTERNAL	固定参数，在创建数据库的时候确定，用户无法修改，只能通过 show 语法或者 pg_settings 视图进行查看。	无
POSTMASTER	数据库服务端参数，在数据库启动时确定，可以通过配置文件指定。	支持表 11-2 中的方式一、方式四。
SIGHUP	数据库全局参数，可在数据库启动时设置或者在数据库启动后，发送指令重新加载。	支持表 11-2 中的方式一、方式二、方式四。
BACKEND	会话连接参数。在创建会话连接时指定，连接建	支持表 11-2 中的方

	立后无法修改。连接断掉后参数失效。内部使用参数，不推荐用户设置。	式一、方式二、方式四。 说明： 设置该参数后，下一次建立会话连接时生效。
SUSET	数据库管理员参数。可在数据库启动时、数据库启动后或者数据库管理员通过 SQL 进行设置。	支持表 11-2 中的方式一、方式二或由数据库管理员通过方式三设置。
USERSET	普通用户参数。可被任何用户在任何时刻设置。	支持表 11-2 中的方式一、方式二或方式三设置。

GBase 8c 提供了四种方式来修改 GUC 参数，具体操作参考表 11-2。

表 11-2 GUC 参数设置方式

序号	设置方法

方式一	使用如下命令修改参数。
	`gs_guc set -Z datanode -D datadir -c "paraname=value"`
	说明：
	如果参数是一个字符串变量，则使用-c parameter="'value'"或者使用-c "parameter = 'value'"。
	使用以下命令在数据库节点上同时设置某个参数。
	`gs_guc set -Z datanode -N all -I all -c "paraname=value"`
	重启数据库使参数生效。
	说明：
	重启 GBase 8c 数据库的操作，会导致用户执行操作中断，请在操作之前规划好合适的执行窗口。

	gs_om -t restart
方式二	gs_guc reload -D datadir -c "paraname=value" 说明： 使用以下命令在数据库节点上同时设置某个参数。 gs_guc reload -Z datanode -N all -I all -c "paraname=value"
方式三	● 修改指定数据库、用户、会话级别的参数。 设置数据库级别的参数 ALTER DATABASE dbname SET paraname TO value; 在下次会话中生效。 ● 设置用户级别的参数 ALTER USER username SET paraname TO value; 在下次会话中生效。 ● 设置会话级别的参数 SET paraname TO value; 修改本次会话中的取值。退出会话后，设置将失效。 说明： SET 设置的会话级参数优先级最高，其次是 ALTER 设置的，其中 ALTER DATABASE 设置的参数值优先级高于 ALTER USER 设置，这三种设置方式设置的优先级都高于 gs_guc 设置方式。
方式四	使用 ALTER SYSTEM SET 修改数据库参数。 ● 设置 POSTMASERT 级别的参数 ALTER SYSTEM SET paraname TO value; 重启后生效。 ● 设置 SIGHUP 级别的参数 ALTER SYSTEM SET paraname TO value; 立刻生效(实际等待线程重新加载参数略有延迟)。 ● 设置 BACKEND 级别的参数

	ALTER SYSTEM SET paraname TO value; 在下次会话中生效。

注意

- 使用方式一和方式二设置参数时，若所设参数不属于当前环境，数据库会提示参数不在支持范围内的相关信息。

12 内存优化表 MOT 管理

12.1 MOT 介绍

本章介绍了 GBase 8c 内存优化表（Memory-Optimized Table，MOT）的特性及价值、关键技术、应用场景、性能基准和竞争优势。

12.1.1MOT 简介

GBase 8c 提供的 MOT 存储引擎是一种事务性行存储，针对多核和大内存服务器进行了优化。MOT 为事务性工作负载提供更高的性能。MOT 完全支持 ACID 特性，并包括严格的持久性和高可用性支持。企业可以在关键任务、性能敏感的在线事务处理（OLTP）中使用 MOT，以实现高性能、高吞吐、可预测低延迟以及多核服务器的高利用率。MOT 尤其适合在多路和多核处理器的现代服务器上运行。

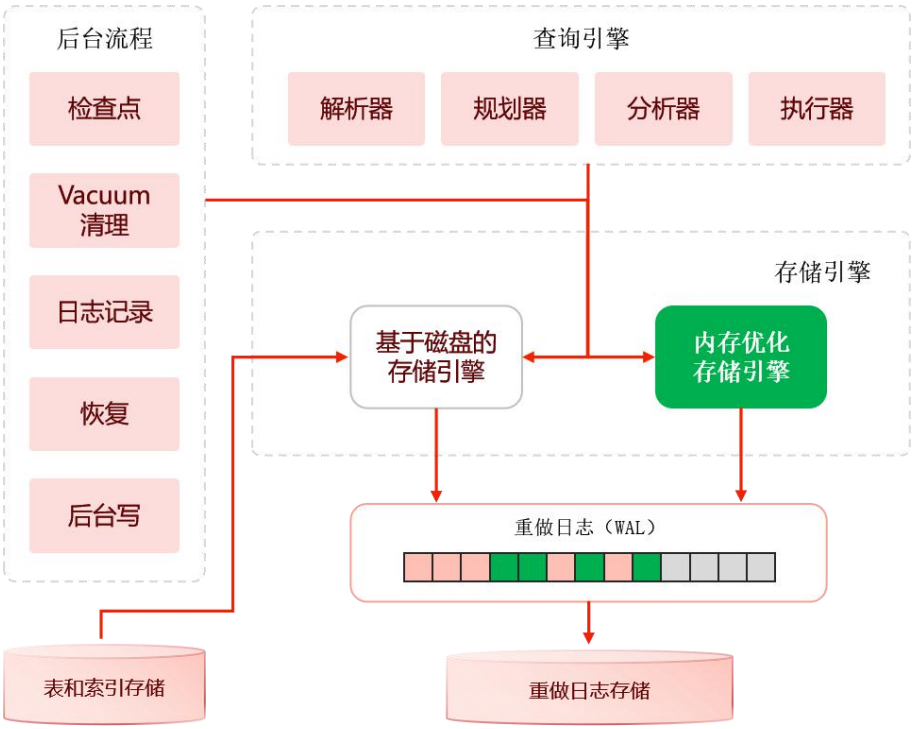


图 12-1 GBase 8c 内存优化存储引擎

如上图所示，GBase 8c 数据库内存优化存储引擎组件（绿色部分）负责管理 MOT 和事务。

MOT 与基于磁盘的普通表并排创建。MOT 的有效设计实现了几乎完全的 SQL 覆盖，并且支持完整的数据库功能集，如存储过程和自定义函数（限制参见 MOT SQL 覆盖和限制）。

通过完全存储在内存中的数据和索引、非统一内存访问感知（NUMA-aware）设计、消除锁和锁存争用的算法，MOT 可提供更快的数据访问和更高效的事务执行。

MOT 有效的几乎无锁的设计和高度调优的实现，使其在多核服务器上实现了卓越的近线性吞吐量扩展，这可能是业界最好的。

MOT 完全支持 ACID 特性：

- 原子性（Atomicity）：原子事务是一系列不可分割的数据库操作。在事务完成（分别提交或中止）之后，这些操作要么全部发生，要么全部不发生。
- 一致性（Consistency）：事务结束后，数据库处于一致状态，保留数据完整性。
- 隔离性（Isolation）：事务之间不能相互干扰。MOT 支持不同的重复读和读提交隔离级别。在下一个版本中，MOT 还将支持可序列化隔离。更多信息，请参见 MOT 隔离级别。
- 持久性（Durability）：即使发生崩溃和失败，成功完成（提交）的事务效果持久保存。MOT 完全集成了 GBase 8c 的基于 WAL 的日志记录。同时支持同步和异步日志记录选项。MOT 还支持同步+面向 NUMA 优化的组提交。详见 MOT 持久性概念。

12.1.2MOT 特性及价值

MOT 在高性能（查询和事务延迟）、高可扩展性（吞吐量和并发量）以及高资源利用率（某些程度上节约成本）方面拥有显著优势。

- 低延迟（Low Latency）：提供快速的查询和事务响应时间。
- 高吞吐量（High Throughput）：支持峰值和持续高用户并发。
- 高资源利用率（High Resource Utilization）：充分利用硬件。

此外，高负载和高争用的场景是所有领先的行业数据库都会遇到的公认问题，而 MOT 能够在这种情况下极高地利用服务器资源。使用 MOT 后，4 路服务器的资源利用率达到 99%，远远领先其他行业数据库。

这种能力在现代的多核服务器上尤为明显和重要。

12.1.3MOT 关键技术

MOT 的关键技术如下：

内存优化数据结构：以实现高并发吞吐量和可预测的低延迟为目标，所有数据和索引都

在内存中，不使用中间页缓冲区，并使用持续时间最短的锁。数据结构和所有算法都是专门为内存设计而优化的。

免锁事务管理：MOT 在保证严格一致性和数据完整性的前提下，采用乐观的策略实现高并发和高吞吐。在事务过程中，MOT 不会对正在更新的数据行的任何版本加锁，从而大大降低了一些大内存系统中的争用。事务中的乐观并发控制（Optimistic Concurrency Control，OCC）语句是在没有锁的情况下实现的，所有的数据修改都是在内存中专门用于私有事务的部分（也称为私有事务内存）中进行的。这就意味着在事务过程中，相关数据在私有事务内存中更新，从而实现了无锁读写；而且只有在提交阶段才会短时间加锁。更多详细信息，请参见 MOT 并发控制机制。

免锁索引：由于内存表的数据和索引完全存储在内存中，因此拥有一个高效的索引数据结构和算法非常重要。MOT 索引机制基于最先进的 Masstree，这是一种用于多核系统的快速和可扩展的键值（Key Value，KV）存储索引，以 B+树的 Trie 实现。通过这种方式，高并发工作负载在多核服务器上可以获得卓越的性能。同时 MOT 应用了各种先进的技术以优化性能，如优化锁方法、高速缓存感知和内存预取。

NUMA-aware 的内存管理：MOT 内存访问的设计支持非统一内存访问（NUMA）感知。NUMA-aware 算法增强了内存中数据布局的性能，使线程访问物理上连接到线程运行的核心的内存。这是由内存控制器处理的，不需要通过使用互连（如英特尔 QPI）进行额外的跳转。MOT 的智能内存控制模块，为各种内存对象预先分配了内存池，提高了性能，减少了锁，保证了稳定性。事务的内存对象的分配始终是 NUMA 本地的。本地处理的对象会返回到池中。同时在事务中尽量减少系统内存分配（OS malloc）的使用，避免不必要的锁。

高效持久性：日志和检查点是实现磁盘持久化的关键能力，也是 ACID 的关键要求之一（D 代表持久性）。目前所有的磁盘（包括 SSD 和 NVMe）都明显慢于内存，因此持久化是基于内存数据库引擎的瓶颈。作为一个基于内存的存储引擎，MOT 的持久化设计必须实现各种各样的算法优化，以确保持久化的同时还能达到设计时的速度和吞吐量目标。这些优化包括：

并行日志，所有磁盘表都支持。

每个事务的日志缓冲和无锁事务准备。

增量更新记录，即只记录变化。

除了同步和异步之外，创新的 NUMA 感知组提交日志记录。

最先进的数据库检查点（CALC）使内存和计算开销降到最低。

高 SQL 覆盖率和功能集：MOT 通过扩展的 PostgreSQL 外部数据封装（FDW）以及索引，几乎支持完整的 SQL 范围，包括存储过程、用户定义函数和系统函数调用。有关不支持的功能的列表，请参阅 MOT SQL 覆盖和限制。

MOT 和数据库的无缝集成：MOT 是一个高性能的面向内存优化的存储引擎，已集成在 GBase 8c 数据库包中。MOT 的主内存引擎和基于磁盘的存储引擎并存，以支持多种应用场景，同时在内部重用数据库辅助服务，如 WAL 重做日志、复制、检查点和恢复高可用性等。用户可以从基于磁盘的表和 MOT 的统一部署、配置和访问中受益。根据特定需求，灵活且低成本地选择使用哪种存储引擎。例如，把会导致瓶颈的高度性能敏感数据放入内存中。

12.1.4MOT 应用场景

MOT 可以根据负载的特点，显著加快应用程序的整体性能。MOT 通过提高数据访问和事务执行的效率，并通过消除并发执行事务之间的锁和锁存争用，最大程度地减少重定向，从而提高了事务处理的性能。

MOT 的极速不仅因为它在内存中，还因为它围绕并发内存使用管理进行了优化。数据存储、访问和处理算法从头开始设计，以利用内存和高并发计算的最新先进技术。

GBase 8c 允许应用程序随意组合 MOT 和基于标准磁盘的表。对于启用已证明是瓶颈的最活跃、高争用和对性能敏感的应用程序表，以及需要可预测的低延迟访问和高吞吐量的表来说，MOT 特别有用。

MOT 可用于各种应用，例如：

高吞吐事务处理：这是使用 MOT 的主要场景，因为它支持海量事务，同时要求单个事务的延迟较低。这类应用的例子有实时决策系统、支付系统、金融工具交易、体育博彩、移动游戏、广告投放等。

性能瓶颈加速：存在高争用现象的表可以通过使用 MOT 受益，即使该表是磁盘表。由于延迟更低、竞争和锁更少以及服务器吞吐量能力增加，此类表（除了相关表和在查询和事务中一起引用的表之外）的转换使得性能显著提升。

消除中间层缓存：云计算和移动应用往往会有周期性或峰值的高工作负载。此外，许多应用都有 80%以上负载是读负载，并伴有频繁的重复查询。为了满足峰值负载单独要求，以及降低响应延迟提供最佳的用户体验，应用程序通常会部署中间缓存层。这样的附加层增加了开发的复杂性和时间，也增加了运营成本。MOT 提供了一个很好的替代方案，通过一致的高性能数据存储来简化应用架构，缩短开发周期，降低 CAPEX 和 OPEX 成本。

大规模流数据提取：MOT 可以满足云端（针对移动、M2M 和物联网）、事务处理

（Transactional Processing，TP）、分析处理（Analytical Processing，AP）和机器学习（Machine Learning，ML）的大规模流数据的提取要求。MOT 尤其擅长持续快速地同时提取来自许多不同来源的大量数据。这些数据可以在以后进行处理、转换，并在速度较慢的基于磁盘的表中进行移动。另外，MOT 还可以查询到一致的、最新的数据，从而得出实时结果。在有许多实时数据流的物联网和云计算应用中，通常会有专门的数据摄取和处理。例如，一个 Apache Kafka 集群可以用来提取 10 万个事件/秒的数据，延迟为 10ms。一个周期性的批处理任务会将收集到的数据收集起来，并将转换格式，放入关系型数据库中进行进一步分析。MOT 可以通过将数据流直接存储在 MOT 关系表中，为分析和决策做好准备，从而支持这样的场景（同时消除单独的数据处理层）。这样可以更快地收集和处理数据，MOT 避免了代价高昂的分层和缓慢的批处理，提高了一致性，增加了分析数据的实时性，同时降低了总拥有成本（Total Cost of Ownership，TCO）。

降低 TCO：提高资源利用率和消除中间层可以节省 30%到 90%的 TCO。

12.2 MOT 使用

本章介绍如何部署、使用和管理 MOT。使用 MOT 的方法非常简单，相关命令语法与 GBase 8c 基于磁盘的表相同。只有 MOT 中的创建和删除表语句有所不同。本章主要介绍 MOT 入门、如何将基于磁盘的表转换为 MOT 以及 MOT 的局限性和覆盖面。还包括 MOT 管理选项，以及如何进行 TPC-C 基准测试。

12.2.1MOT 使用概述

MOT 作为 GBase 8c 的一部分自动部署。有关如何计算和规划所需的内存和存储资源以维持工作负载的说明，请参阅 MOT 准备。参考 MOT 部署了解 MOT 中所有的配置，以及服务器优化的非必须选项。

使用 MOT 的方法非常简单。MOT 命令的语法与基于磁盘的表的语法相同，并支持大多数标准，如 PostgreSQL SQL、DDL 和 DML 命令和功能，如存储过程。只有 MOT 中的创建和删除表语句有所不同。您可以参考 MOT 使用了解这两个简单命令的说明，如何将基于磁盘的表转换为 MOT 和 PREPARE 语句获得更高的性能，以及了解外部工具支持和 MOT 引擎的限制。

MOT 管理介绍了如何进行数据库维护，以及监控和分析日志和错误报告。最后，MOT 样例 TPC-C 基准介绍了如何执行标准 TPC-C 基准测试。

12.2.2MOT 准备

下文介绍了使用 MOT 的前提条件以及内存和存储规划。

前提条件

- 硬件支持

MOT 支持最新硬件和现有硬件平台，支持 x86 架构和华为鲲鹏 Arm 架构。

MOT 能够支持 GBase 8c 数据库支持的硬件。

- CPU

MOT 在多核服务器（扩容）上提供卓越的性能。在这些环境中，MOT 的性能明显优于友商，并提供近线性扩展和极高的资源利用率。

用户也可以开始在低端、中端和高端服务器上实现 MOT 的性能优势，无论 CPU 槽位是 1 或 2 个，还是 4 个，甚至是 8 个也没问题。在 16 路甚至 32 路的高端服务器上，性能和资源利用率也非常高。

- 内存

MOT 支持标准 RAM/DRAM 用于其数据和事务管理。所有 MOT 数据和索引都驻留在内存中，因此内存容量必须能够支撑数据容量，并且还有进一步增长的空间。

- 存储 IO

MOT 是一个持久的数据库，使用永久性存储设备（磁盘/SSD/NVMe 驱动器）进行事务日志操作和存储定期检查点。

推荐采用低延迟的存储设备，如配置 RAID-1 的 SSD、NVMe 或者任何企业级存储系统。当使用适当的硬件时，数据库事务处理和竞争将成为瓶颈，而非 IO。

- 操作系统支持

MOT 支持裸机和虚拟化环境

- 操作系统优化

MOT 不需要任何特殊修改或安装新软件。但是，一些优化可以提高性能。

MOT 内存和存储规划

本节描述了为满足特定应用程序需求，在评估、估计和规划内存和存储容量数量时，需要注意的事项和准则，以及影响所需内存数量的各种数据，例如计划表的数据和索引大小、

维持事务管理的内存以及数据增长的速度。

MOT 内存规划

MOT 是一种内存数据库存储引擎（IMDB），其中所有表和索引完全驻留在内存中。

服务器上必须有足够的物理内存以维持内存表的状态，并满足工作负载和数据的增长。所有这些都是在传统的基于磁盘的引擎、表和会话所需的内存之外的要求。因此，提前规划好足够的内存来容纳这些内容是非常有必要的。

说明

内存存储是易失的，需要电力来维护所存储的信息。磁盘存储是持久的，写入磁盘是非易失性存储。MOT 使用两种存储，既把所有数据保存在内存中，也把事务性更改同步（通过 WAL 日志记录）到磁盘上以保持严格一致性（使用同步日志记录模式）。

开始可以使用任何数量的内存并执行基本任务和评估测试。但当准备好生产时，应解决以下问题：

内存配置

GBase 8c 数据库和标准 Postgres 类似，其内存上限是由 max_process_memory 设置的，该上限在 postgresql.conf 文件中定义。MOT 及其所有组件和线程，都驻留在进程中。因此，分配给 MOT 的内存也是在整个数据库进程的 max_process_memory 定义的上限内分配。

MOT 为自己保留的内存是 max_process_memory 的一部分。可以通过百分比或通过小于 max_process_memory 的绝对值定义。这个部分在 mot.conf 配置文件中由_mot__memory 配置项定义。

max_process_memory 中可以除了被 MOT 使用的部分之外，必须为 Postgres（GBase 8c）封装留下至少 2GB 的可用空间。为了确保这一点，MOT 在数据库启动过程中会进行如下校验：

```
(max_mot_global_memory + max_mot_local_memory) + 2GB < max_process_memory
```

如果违反此限制，则调整 MOT 内存内部限制，最大可能地满足上述限制范围。该调整在启动时进行，并据此计算 MOT 最大内存值。

说明

MOT 最大内存值是配置或调整值（max_mot_global_memory + max_mot_local_memory）的逻辑计算值。

此时，会向服务器日志发出警告，如下所示：

以下是报告问题的警告消息示例：

```
[WARNING] <Configuration> MOT engine maximum memory definitions (global: 9830 MB, local: 1843 MB, session large store: 0 MB, total: 11673 MB) breach GaussDB maximum process memory restriction (12288 MB) and/or total system memory (64243 MB). MOT values shall be adjusted accordingly to preserve required gap (2048 MB).
```

以下警告消息示例提示 MOT 正在自动调整内存限制：

```
[WARNING] <Configuration> Adjusting MOT memory limits: global = 8623 MB, local = 1617 MB, session large store = 0 MB, total = 10240 MB
```

新内存限制仅在此处显示。

此外，当总内存使用量接近所选内存限制时，MOT 不再允许插入额外数据。不再允许额外数据插入的阈值即是 MOT 最大内存百分比（如上所述，这是一个计算值）。MOT 最大内存百分比默认值为 90，即 90%。尝试添加超过此阈值的额外数据时，会向用户返回错误，并且也会注册到数据库日志文件中。

最小值和最大值

为了确保内存安全，MOT 根据最小的全局和本地设置预先分配内存。数据库管理员应指定 MOT 和会话维持工作负载所需的最小内存量。这样可以确保即使另一个消耗内存的应用程序与数据库在同一台服务器上运行，并且与数据库竞争内存资源，也能够将这个最小的内存分配给 MOT。最大值用于限制内存增长。

全局和本地

MOT 使用的内存由两部分组成：

全局内存：全局内存是一个长期内存池，包含 MOT 的数据和索引。它平均分布在 NUMA 节点，由所有 CPU 核共享。

本地内存：本地内存是用于短期对象的内存池。它的主要使用者是处理事务的会话。这些会话将数据更改存储在专门用于相关特定事务的内存部分（称为事务专用内存）。在提交阶段，数据更改将被移动到全局内存中。内存对象分配以 NUMA-local 方式执行，以实现尽可能低的延迟。

被释放的对象被放回相关的内存池中。在事务期间尽量少使用操作系统内存分配（malloc）函数，避免不必要的锁和锁存。

这两个内存的分配由专用的 min/max_mot_global_memory 和 min/max_mot_local_memory 设置控制。如果 MOT 全局内存使用量太接近最大值，则 MOT

会保护自身，不接受新数据。超出此限制的内存分配尝试将被拒绝，并向用户报告错误。

最低内存要求

在开始执行对 MOT 性能的最小评估前，请确保：

除了磁盘表缓冲区和额外的内存，max_process_memory（在 postgresql.conf 中定义）还有足够的容量用于 MOT 和会话（由 mix/max_mot_global_memory 和 mix/max_mot_local_memory 配置）。对于简单的测试，可以使用 mot.conf 的默认设置。

生产过程中实际内存需求

在典型的 OLTP 工作负载中，平均读写比例为 80:20，每个表的 MOT 内存使用率比基于磁盘的表高 60%（包括数据和索引）。这是因为使用了更优化的数据结构和算法，使得访问速度更快，并具有 CPU 缓存感知和内存预取功能。

特定应用程序的实际内存需求取决于数据量、预期工作负载，特别是数据增长。

最大全局内存规划：数据和索引大小

要规划最大全局内存，需满足：

确定特定磁盘表（包括其数据和所有索引）的大小。如下统计查询可以确定 customer 表的数据大小和 customer_pkey 索引大小：

数据大小：选择 pg_relation_size（'customer'）；

索引：选择 pg_relation_size（'customer_pkey'）；

额外增加 60%的内存，相对于基于磁盘的数据和索引的当前大小，这是 MOT 中的常见要求。

额外增加数据预期增长百分比。例如：

5%月增长率 = 80%年增长率（1.05^12）。因此，为了维持年增长，需分配比表当前使用的还多 80%的内存。

至此，max_mot_global_memory 值的估计和规划就完成了。实际设置可以用绝对值或 Postgres max_process_memory 的百分比来定义。具体的值通常在部署期间进行微调。

最大本地内存规划：并发会话支持

本地内存需求主要是并发会话数量的函数。平均会话的典型 OLTP 工作负载最大占用 8MB。此值乘以会话的数量，再加一点额外的值。

可以通过这种方式进行内存计算，然后进行微调：

```
SESSION_COUNT * SESSION_SIZE (8 MB) + SOME_EXTRA (100MB should be enough)
```

默认指定 Postgres 最大进程内存（默认为 12GB）的 15%。相当于 1.8GB 可满足 230 个会话，即 max_mot_local 内存需求。实际设置可以用绝对值或 Postgres max_process_memory 的百分比来定义。具体的值通常在部署期间进行微调。

异常大事务

某些事务非常大，因为它们将更改应用于大量行。这可能导致单个会话的本地内存增加到允许的最大限制，即 1GB。例如：

```
delete from SOME_VERY_LARGE_TABLE;
```

在配置 max_mot_local_memory 设置和应用程序开发时，请考虑此场景。

存储 IO

MOT 是一个内存优化的持久化数据库存储引擎。需要磁盘驱动器来存储 WAL 重做日志和定期检查点。

推荐采用低延迟的存储设备，如配置 RAID-1 的 SSD、NVMe 或者任何企业级存储系统。当使用适当的硬件时，数据库事务处理和竞争将成为瓶颈，而非 IO。

由于持久性存储比 RAM 内存慢得多，因此 IO 操作（日志和检查点）可能成为内存中数据库和内存优化数据库的瓶颈。但是，MOT 具有针对现代硬件（如 SSD、NVMe）进行优化的高效持久性设计和实现。此外，MOT 最小化和优化了写入点（例如，使用并行日志记录、每个事务的单日志记录和 NUMA-aware 事务组写入），并且最小化了写入磁盘的数据（例如，只把更改记录的增量或更新列记录到日志，并且只记录提交阶段的事务）。

容量需求

所需容量取决于检查点和记录的要求，如下所述：

检查点

检查点将所有数据的快照保存到磁盘。

需要给检查点分配两倍数据大小的容量。不需要为检查点索引分配空间。

检查点 = 2 x MOT 数据大小（仅表示行，索引非持久）。

检查点之所以需要两倍大小，是因为快照会保存数据的全部大小到磁盘上，此外还应该为正在进行的检查点分配同样数量的空间。当检查点进程结束时，以前的检查点文件将被删除。

说明

MOT 增量检查点特性，这将大大降低存储容量需求。

日志记录

MOT 日志记录与基于磁盘的表的其它记录写入同一个数据库事务日志。

日志的大小取决于事务吞吐量、数据更改的大小和检查点之间的时间（每次检查点，重做日志被截断并重新开始扩展）。

与基于磁盘的表相比，MOT 使用较少的日志带宽和较低的 IO 争用。这由多种机制实现。

例如，MOT 不会在事务完成之前记录每个操作。它只在提交阶段记录，并且只记录更新的增量记录（不像基于磁盘的表那样的完整记录）。

为了确保日志 IO 设备不会成为瓶颈，日志文件必须放在具有低延迟的驱动器上。

12.2.3MOT 部署

以下各小节介绍了各种必需和可选的设置，以达到最佳部署效果。

12.2.3.1 MOT 服务器优化：X86

通常情况下，数据库由以下组件绑定：

- CPU：更快的 CPU 可以加速任何 CPU 绑定的数据库。
- 磁盘：高速 SSD/NVME 可加速任何 I/O 绑定数据库。
- 网络：更快的网络可以加速任何 SQL*Net 绑定数据库。

除以上内容外，以下通用服务器设置默认使用，可能会明显影响数据库的性能。

MOT 性能调优是确保快速的应用程序功能和数据检索的关键 MOT 支持最新的硬件，因此调整每个系统以达到最大吞吐量是极为重要的。

BIOS

Hyper Threading 设置为 ON。

建议打开超线程（HT=ON）。建议在 MOT 上运行 OLTP 工作负载时打开超线程。当使用超线程时，某些 OLTP 工作负载显示高达 40%的性能增益。

操作系统环境设置

- NUMA

禁用 NUMA 平衡，如下所示。MOT 以极其高效的 NUMA-aware 方式进行内存管理，

远远超过操作系统使用的默认方法。

```
echo 0 > /proc/sys/kernel/numa_balancing
```

- 服务

  禁用如下服务：

```
service irqbalance stop                 # MANADATORY
service sysmonitor stop                  # OPTIONAL，performance
service rsyslog stop            # OPTIONAL，performance
```

- 调优服务

  以下为必填项。

  服务器必须运行 throughput-performance 配置文件。

```
[...]$ tuned-adm profile throughput-performance
```

  throughput-performance 配置文件是广泛适用的调优，它为各种常见服务器工作负载提供卓越的性能。

  其他不太适合 GBase 8c 和 MOT 服务器的配置可能会影响 MOT 的整体性能，包括：平衡配置、桌面配置、延迟性能配置、网络延迟配置、网络吞吐量配置和节能配置。

- 系统命令

  推荐使用下列操作系统设置以获得最佳性能。

  在/etc/sysctl.conf 文件中添加如下配置，然后执行 sysctl -p 命令：

```
net.ipv4.ip_local_port_range = 9000 65535
kernel.sysrq = 1
kernel.panic_on_oops = 1
kernel.panic = 5
kernel.hung_task_timeout_secs = 3600
kernel.hung_task_panic = 1
vm.oom_dump_tasks = 1
kernel.softlockup_panic = 1
fs.file-max = 640000
kernel.msgmnb = 7000000
kernel.sched_min_granularity_ns = 10000000
kernel.sched_wakeup_granularity_ns = 15000000
kernel.numa_balancing=0
vm.max_map_count = 1048576
net.ipv4.tcp_max_tw_buckets = 10000
```

```
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_keepalive_time = 30
net.ipv4.tcp_keepalive_probes = 9
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_retries2 = 80
kernel.sem = 250 6400000 1000 215432
net.core.wmem_max = 21299200
net.core.rmem_max = 21299200
net.core.wmem_default = 21299200
net.core.rmem_default = 21299200
#net.sctp.sctp_mem = 94500000 915000000 927000000
#net.sctp.sctp_rmem = 8192 250000 16777216
#net.sctp.sctp_wmem = 8192 250000 16777216
net.ipv4.tcp_rmem = 8192 250000 16777216
net.ipv4.tcp_wmem = 8192 250000 16777216
net.core.somaxconn = 65535
vm.min_free_kbytes = 26351629
net.core.netdev_max_backlog = 65535
net.ipv4.tcp_max_syn_backlog = 65535
#net.sctp.addip_enable = 0
net.ipv4.tcp_syncookies = 1
vm.overcommit_memory = 0
net.ipv4.tcp_retries1 = 5
net.ipv4.tcp_syn_retries = 5
```

按如下方式修改/etc/security/limits.conf 对应部分：

```
<user> soft nofile 100000
<user> hard nofile 100000
```

软限制和硬限制设置可指定一个进程同时打开的文件数量。软限制可由各自运行这些限制的进程进行更改，直至达到硬限制值。

磁盘/SSD

下面以数据库同步提交模式为例，介绍如何保证磁盘读写性能适合数据库同步提交模式。

按如下方式运行磁盘/SSD 性能测试：

```
[...]$ sync; dd if=/dev/zero of=testfile bs=1M count=1024; sync
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied，1.36034 s，789 MB/s
```

当磁盘带宽明显低于 789MB/s 时，可能会造成 GBase 8c 性能瓶颈，尤其是造成 MOT 性能瓶颈。

网络

需要使用 10Gbps 以上网络。

运行 iperf 命令进行验证：

```
Server side: iperf -s
Client side: iperf -c <IP>
rc.local：网卡调优
```

以下可选设置对性能有显著影响：

将 https://gist.github.com/SaveTheRbtz/8875474 下的 set_irq_privacy.sh 文件拷贝到/var/scripts/目录下。

进入/etc/rc.d/rc.local，执行 chmod 命令，确保在 boot 时执行以下脚本：

```
'chmod +x /etc/rc.d/rc.local'
var/scripts/set_irq_affinity.sh -x all <DEVNAME>
ethtool -K <DEVNAME> gro off
ethtool -C <DEVNAME> adaptive-rx on adaptive-tx on
Replace <DEVNAME> with the network card，i.e. ens5f1
```

12.2.3.2 MOT 服务器优化：Arm

以下是基于 Arm/鲲鹏架构的华为 TaiShan 2280 v2 服务器（2 路 128 核）和 TaiShan 2480 v2 服务器（4 路 256 核）上运行 MOT 时的建议配置。

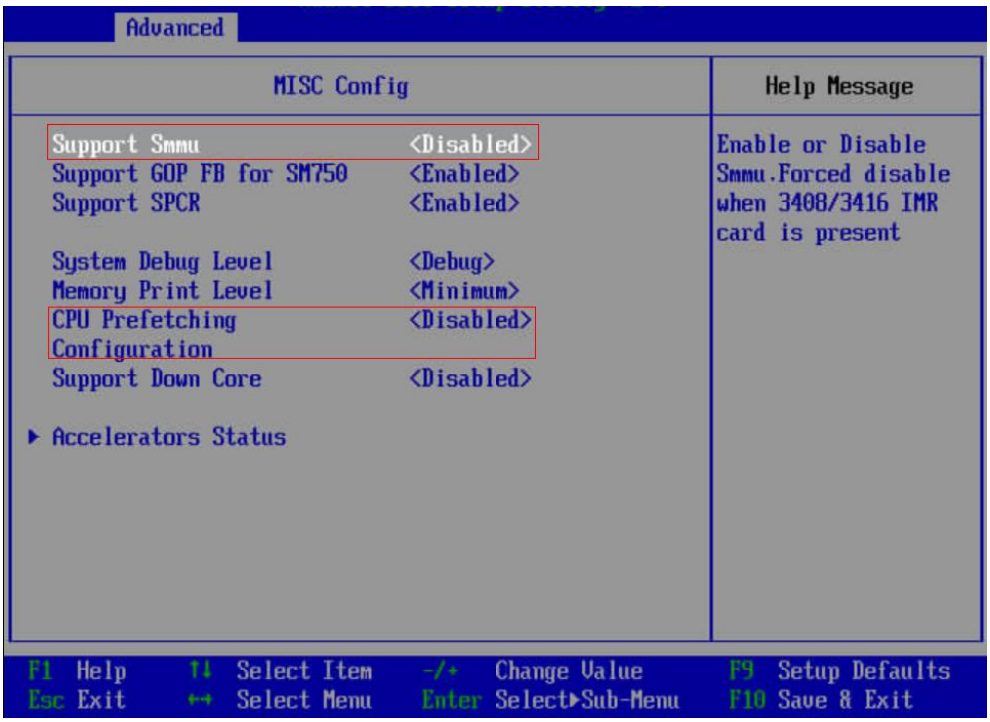
除非另有说明，以下设置适用于客户端和服务器的机器。

BIOS

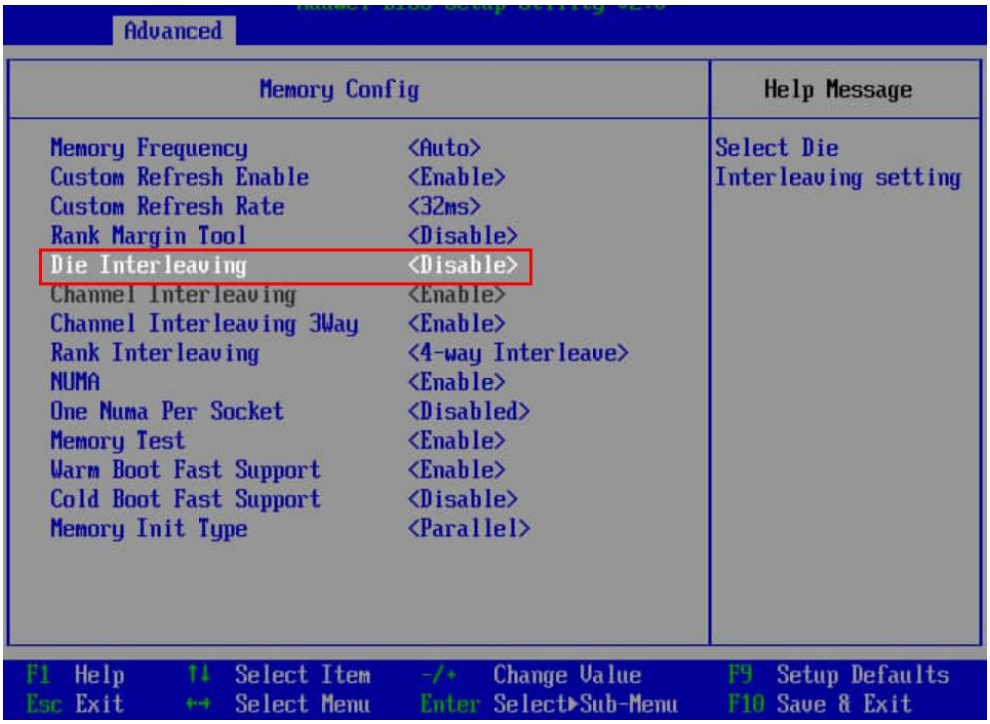
修改 BIOS 相关设置：

(1) 选择 BIOS > Advanced > MISC Config。设置 Support Smmu 为 Disabled。

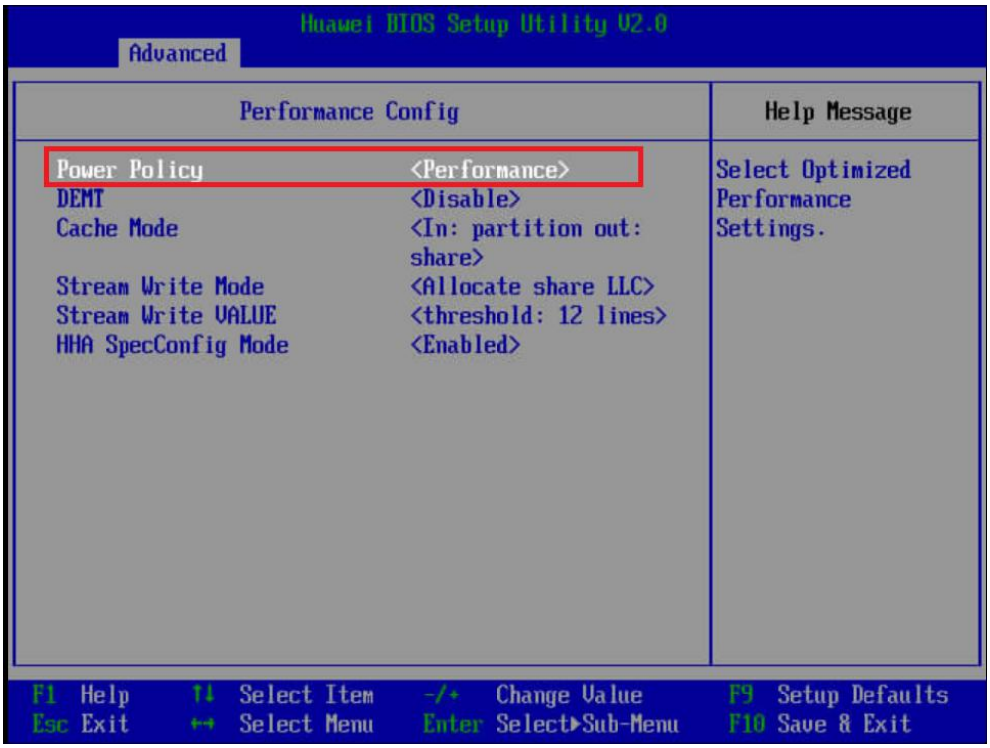
(2) 选择 BIOS > Advanced > MISC Config。设置 CPU Prefetching Configuration 为 Disabled。



(3) 选择 BIOS > Advanced > Memory Config。设置 Die Interleaving 为 Disabled 。



(4) 选择 BIOS > Advanced > Performance Config。设置 Power Policy 为 Performance。



操作系统：内核和启动

(1) 以下操作系统内核和启动参数通常由 sysadmin 配置。

配置内核参数，如下所示。

```
net.ipv4.ip_local_port_range = 9000 65535
kernel.sysrq = 1
kernel.panic_on_oops = 1
kernel.panic = 5
kernel.hung_task_timeout_secs = 3600
kernel.hung_task_panic = 1
vm.oom_dump_tasks = 1
kernel.softlockup_panic = 1
fs.file-max = 640000
kernel.msgmnb = 7000000
kernel.sched_min_granularity_ns = 10000000
kernel.sched_wakeup_granularity_ns = 15000000
kernel.numa_balancing=0
vm.max_map_count = 1048576
net.ipv4.tcp_max_tw_buckets = 10000
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_keepalive_time = 30
```

```
net.ipv4.tcp_keepalive_probes = 9
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_retries2 = 80
kernel.sem = 32000 1024000000          500         32000
kernel.shmall = 52805669
kernel.shmmax = 18446744073692774399
sys.fs.file-max = 6536438
net.core.wmem_max = 21299200
net.core.rmem_max = 21299200
net.core.wmem_default = 21299200
net.core.rmem_default = 21299200
net.ipv4.tcp_rmem = 8192 250000 16777216
net.ipv4.tcp_wmem = 8192 250000 16777216
net.core.somaxconn = 65535
vm.min_free_kbytes = 5270325
net.core.netdev_max_backlog = 65535
net.ipv4.tcp_max_syn_backlog = 65535
net.ipv4.tcp_syncookies = 1
vm.overcommit_memory = 0
net.ipv4.tcp_retries1 = 5
net.ipv4.tcp_syn_retries = 5
##NEW
kernel.sched_autogroup_enabled=0
kernel.sched_min_granularity_ns=2000000
kernel.sched_latency_ns=10000000
kernel.sched_wakeup_granularity_ns=5000000
kernel.sched_migration_cost_ns=500000
vm.dirty_background_bytes=33554432
kernel.shmmax=21474836480
net.ipv4.tcp_timestamps = 0
net.ipv6.conf.all.disable_ipv6=1
net.ipv6.conf.default.disable_ipv6=1
net.ipv4.tcp_keepalive_time=600
net.ipv4.tcp_keepalive_probes=3
kernel.core_uses_pid=1
```

(2) 调优服务

以下为必填项。

服务器必须运行 throughput-performance 配置文件 :

```
[...]$ tuned-adm profile throughput-performance
```

throughput-performance 配置文件是广泛适用的调优，它为各种常见服务器工作负载提供卓越的性能。

其他不太适合 GBase 8c 和 MOT 服务器的配置可能会影响 MOT 的整体性能，包括：平衡配置、桌面配置、延迟性能配置、网络延迟配置、网络吞吐量配置和节能配置。

(3) 启动调优

在内核启动参数中添加 igbaseu.passthrough=1。

在 pass-through 模式下运行时，适配器需要 DMA 转换到内存，从而提高性能。

12.2.3.3 MOT 配置

预置 MOT 用于创建工作 MOT。为了获得最佳效果，建议根据应用程序的特定要求和偏好自定义 MOT 配置（在 mot.conf 文件中定义）。

该文件在服务器启动时只读。如果在系统运行中编辑此文件，则必须重新加载服务器才能使修改内容生效。

mot.conf 文件与 postgresql.conf 配置文件在同一文件夹下。

在主备部署模式下，主备节点的 mot.conf 文件需要完全相同，否则，系统行为不明确。

阅读总体原则，根据需要查看和配置 mot.conf 文件。

说明

以上描述了 mot.conf 文件中的各个设置。除上述内容外，要了解特定 MOT 功能（如恢复），可参考本用户手册的相关章节。例如，MOT 恢复说明了 mot.conf 文件的恢复，包含影响 MOT 恢复的设置。此外，有关恢复的完整说明，请参阅"MOT 管理”章节的 MOT 恢复。下文各相关章节中还提供了参考链接。

以下介绍了 mot.conf 文件中的各个部分，其包含的设置以及默认值。

总体原则

以下是编辑 mot.conf 文件的总体原则。

- 每个设置项都带有默认值，如下所示：

```
# name = value
```

- 可以接受空格或留空。
- 在各行添加#号可进行注释。

- 每个设置项的默认值将作为注释显示在整个文件中。
- 如果参数没有注释并且置入了新值，则定义新设置。
- 对 mot.conf 文件的更改仅在数据库服务器启动或重装时生效。

内存单元的表示如下：

- KB：千字节
- MB：兆字节
- GB：吉字节
- TB：太字节

如果未指定内存单元，则假定为字节。

某些内存单位为 postgresql.conf 中的 max_process_memory 的百分比值。例如，20%。

时间单位表示如下：

- us：微秒
- ms：毫秒
- s：秒
- min：分钟
- h：小时
- d：天

如果未指定时间单位，则假定为微秒。

重做日志（MOT）

- enable_group_commit = false

是否使用组提交。

该选项仅在 GBase 8c 配置为使用同步提交时相关，即仅当 postgresql.conf 中的 synchronization_commit 设置为除 off 以外的任何值时相关。

- group_commit_size = 16
- group_commit_timeout = 10 ms

只有当 MOT 引擎配置为同步组提交日志记录时，此选项才相关。即 postgresql.conf 中

的 synchronization_commit 配置为 true，mot.conf 配置文件中的 enable_group_commit 配置为 true。

当一组事务记录在 WAL 重做日志中时，需确定以下设置项取值：

group_commit_size：一组已提交的事务数。例如，16 表示当同一组中的 16 个事务已由它们的客户端应用程序提交时，则针对 16 个事务中的每个事务，在磁盘的 WAL 重做日志中写入一个条目。

group_commit_timeout：超时时间，单位为毫秒。例如，10 表示在 10 毫秒之后，为同一组由客户端应用程序在最近 10 毫秒内提交的每个事务，在磁盘的 WAL 重做日志中写入一个条目。

提交组在到达配置的事务数后或者在超时后关闭。组关闭后，组中的所有事务等待一个组落盘完成执行，然后通知客户端每个事务都已经结束。

检查点（MOT）

- checkpoint_dir =

指定检查点数据存放目录。默认位置在每个数据节点的 data 文件夹中。

- checkpoint_segsize = 16 MB

指定检查点时使用的段大小。分段执行检查点。当一个段已满时，它将被序列化到磁盘，并为后续的检查点数据打开一个新的段。

- checkpoint_workers = 3

指定在检查点期间要使用的工作线程数。

检查点由多个 MOT 引擎工作线程并行执行。工作线程的数量可能会大大影响整个检查点操作的整体性能，以及其它正在运行的事务的操作。为了实现较短的检查点持续时间，应使用更多线程，直至达到最佳数量（根据硬件和工作负载的不同而不同）。但请注意，如果这个数目太大，可能会对其他正在运行的事务的执行时间产生负面影响。尽可能低这个数字，以最小化对其他运行事务的运行时的影响。当此数目过高时，检查点持续时间会较长。

说明

➢ 有关配置的更多信息，请参阅 MOT 持久性中的 MOT 检查点。

恢复（MOT）

- checkpoint_recovery_workers = 3

指定在检查点数据恢复期间要使用的工作线程数。每个 MOT 引擎工作线程在自己的核上运行，通过将不同的表读入内存，可以并行处理不同的表。缺省值为 3，可将此参数设置为可处理的核数。恢复后，将停止并杀死这些线程。

说明

- 有关配置的详细信息，请参阅 MOT 恢复。

统计（MOT）

- enable_stats = false

  设置周期性统计打印信息。

- print_stats_period = 10 minute

  设置汇总统计报表打印的时间范围。

- print_full_stats_period = 1 hours

  设置全量统计报表打印的时间范围。

  以下设置为周期性统计报表中的各个部分。如果没有配置，则抑制统计报表。

- enable_log_recovery_stats = false

  日志恢复统计信息包含各种重做日志的恢复指标。

- enable_db_session_stats = false

  数据库会话统计信息包含事务事件，如提交、回滚等。

- enable_network_stats = false

  网络统计信息包括连接/断连事件。

- enable_log_stats = false

  日志统计信息包含重做日志详情。

- enable_memory_stats = false

  内存统计信息包含内存层详情。

- enable_process_stats = false

  进程统计信息包含当前进程的内存和 CPU 消耗总量。

- enable_system_stats = false

系统统计信息包含整个系统的内存和 CPU 消耗总量。

- enable_jit_stats = false

JIT 统计信息。

错误日志（MOT）

- log_level = INFO

设置 MOT 引擎下发的消息在数据库服务器的错误日志中记录的日志级别。有效值为 PANIC、ERROR、WARN、INFO、TRACE、DEBUG、DIAG1、DIAG2。

- Log/COMPONENT/LOGGER=LOG_LEVEL

使用以下语法设置特定的日志记录器。

例如，要为系统组件中的 ThreadIdPool 日志记录器配置 TRACE 日志级别，请使用以下语法：

- Log.System.ThreadIdPool.log_level=TRACE

要为某个组件下的所有记录器配置日志级别，请使用以下语法：

- Log.COMPONENT.log_level=LOG_LEVEL

例如：

```
Log.System.log_level=DEBUG
```

内存（MOT）

- enable_numa = true

指定是否使用可识别 NUMA 的内存。禁用时，所有亲和性配置也将被禁用。MOT 引擎假定所有可用的 NUMA 节点都有内存。如果计算机具有某些特殊配置，其中某些 NUMA 节点没有内存，则 MOT 引擎初始化将因此失败，因此数据库服务器启动将失败。在此类计算机中，建议将此配置值设置为 false，以防止启动失败并让 MOT 引擎在不使用可识别 NUMA 的内存分配的情况下正常运行。

- affinity_mode = fill-physical-first

设置用户会话和内部 MOT 任务的线程亲和模式。

使用线程池时，用户会话将忽略此值，因为它们的亲和性由线程池控制。但内部 MOT 任务仍然使用。

有效值为 fill-socket-first、equal-per-socket、fill-physical-first、none。

- Fill-socket-first 将线程连接到同一个槽位的核上，直到槽位已满，然后移动到下一个槽位。
- Equal-per-socket 使线程均匀分布在所有槽位中。
- Fill-physical-first 将线程连接到同一个槽位中的物理核，直到用尽所有物理核，然后移动到下一个槽位。当所有物理核用尽时，该过程再次从超线程核开始。
- None 禁用任何亲和配置，并让系统调度程序确定每个线程调度在哪个核上运行。

- lazy_load_chunk_directory = true

  设置块目录模式，用于内存块查找。

Lazy 模式将块目录设置为按需加载部分目录，从而减少初始内存占用（大约从 1GB 减少到 1MB）。然而，这可能会导致轻微的性能损失和极端情况下的内存损坏。相反，使用 non-lazy 块目录会额外分配 1GB 的初始内存，产生略高的性能，并确保在内存损坏期间避免块目录错误。

- reserve_memory_mode = virtual

  设置内存预留模式（取值为 physical 或 virtual）。

每当从内核分配内存时，都会参考此配置值来确定所分配的内存是常驻（physical）还是非常驻（virtual）。这主要与预分配有关，但也可能影响运行时分配。对于 physical 保留模式，通过强制内存区域所跨越的所有页出现页错误，使整个分配的内存区域常驻。配置 virtual 内存预留可加速内存分配（特别是在预分配期间），但可能在初始访问期间出现页错误（因此导致轻微的性能影响），并在物理内存不可用时出现更多服务器错误。相反，物理内存分配速度较慢，但后续访问速度更快且有保障。

- store_memory_policy = compact

  设置内存存储策略（取值为 compact 或 expanding）。

当定义了 compact 策略时，未使用的内存会释放回内核，直到达到内存下限（请参见下面的 min_mot_memory）。在 expanding 策略中，未使用的内存存储在 MOT 引擎中，以便后续再使用。compact 存储策略可以减少 MOT 引擎的内存占用，但偶尔会导致性能轻微下降。此外，在内存损坏时，它还可能导致内存不可用。相反，expanding 模式会占用更多的内存，但是会更快地分配内存，并且能够更好地保证在解分配后能够重新分配内存。

- chunk_alloc_policy = auto

设置全局内存的块分配策略。

MOT 内存以 2MB 的块为单位组织。源 NUMA 节点和每个块的内存布局会影响表数据在 NUMA 节点间的分布，因此对数据访问时间有很大影响。在特定 NUMA 节点上分配块时，会参考分配策略。

可用值包括 auto、local、page-interleaved、chunk-interleaved、native。

- Auto 策略根据当前硬件情况选择块分配策略。
- Local 策略在各自的 NUMA 节点上分配每个数据块。
- Page-interleaved 策略从所有 NUMA 节点分配由交插内存 4 千字节页组成的数据块。
- Chunk-interleaved 策略以轮循调度方式从所有 NUMA 节点分配数据块。
- Native 策略通过调用原生系统内存分配器来分配块。

- chunk_prealloc_worker_count = 8

  设置每个 NUMA 节点参与内存预分配的工作线程数。

- max_mot_global_memory = 80%

  设置 MOT 引擎全局内存的最大限制。

  指定百分比值与 postgresql.conf 中 max_process_memory 定义的总量有关。

  MOT 引擎内存分为全局（长期）内存，主要用于存储用户数据，以及本地（短期）内存，主要用于用户会话，以满足本地需求。

  任何试图分配超出此限制的内存的尝试将被拒绝，并向用户报告错误。请确保 max_mot_global_memory 与 max_mot_local_memory 之和不超过 postgresql.conf 中配置的 max_process_memory。

- min_mot_global_memory = 0 MB

  设置 MOT 引擎全局内存的最小限制。

  指定百分比值与 postgresql.conf 中 max_process_memory 定义的总量有关。

  此值用于启动期间的内存预分配，以及确保 MOT 引擎在正常运行期间有最小的内存可用量。当使用 compact 存储策略时（参阅上文 store_memory_policy），该值指定了下限，超过下限的内存不会释放回内核，而是保留在 MOT 引擎中以便后续重用。

- max_mot_local_memory = 15%

设置 MOT 引擎本地内存的最大限制。

指定百分比值与 postgresql.conf 中 max_process_memory 定义的总量有关。

MOT 引擎内存分为全局（长期）内存，主要用于存储用户数据，以及本地（短期）内存，主要用于用户会话，以满足本地需求。

任何试图分配超出此限制的内存的尝试将被拒绝，并向用户报告错误。请确保 max_mot_global_memory 与 max_mot_local_memory 之和不超过 postgresql.conf 中配置的 max_process_memory。

- min_mot_local_memory = 0 MB

  设置 MOT 引擎本地内存的最小限制。

  指定百分比值与 postgresql.conf 中 max_process_memory 定义的总量有关。

  此值用于在启动期间预分配内存，以及确保 MOT 引擎在正常运行期间有最小的可用内存。当使用 compact 存储策略时（参阅上文 store_memory_policy），该值指定了下限，超过下限的内存不会释放回内核，而是保留在 MOT 引擎中以便后续重用。

- max_mot_session_memory = 0 MB

  设置 MOT 引擎中单个会话的最大内存限制。

  指定百分比值与 postgresql.conf 中 max_process_memory 定义的总量有关。

  通常，MOT 引擎中的会话可以根据需要分配尽可能多的本地内存，只要没有超出本地内存限制即可。为了避免单个会话占用过多的内存，从而拒绝其他会话的内存，通过该配置项限制小会话的本地内存分配（最大 1022KB）。

  请确保该配置项不影响大会话的本地内存分配。

  0 表示不会限制每个小会话的本地分配，除非是由 max_mot_local_memory 配置的本地内存分配限制引起的。

- min_mot_session_memory = 0 MB

  设置 MOT 引擎中单个会话的最小内存预留。

  指定百分比值与 postgresql.conf 中 max_process_memory 定义的总量有关。

  此值用于在会话创建期间预分配内存，以及确保会话有最小的可用内存量来执行其正常操作。

- session_large_buffer_store_size = 0 MB

设置会话的大缓冲区存储。

当用户会话执行需要大量内存的查询时（例如，使用许多行），大缓冲区存储用于增加此类内存可用的确定级别，并更快地为这个内存请求提供服务。对于超过 1022KB 的会话，任何内存分配都是大内存分配。如果未使用或耗尽了大缓冲区存储，则这些分配将被视为直接从内核提供的巨大分配。

- session_large_buffer_store_max_object_size = 0 MB

  设置会话的大分配缓冲区存储中的最大对象大小。

  大缓冲区存储内部被划分为不同大小的对象。此值用于对源自大缓冲区存储的对象设置上限，以及确定缓冲区存储内部划分为不同大小的对象。

  此大小不能超过 session_large_buffer_store_size 的 1/8。如果超过，则将其调整到最大可能。

- session_max_huge_object_size = 1 GB

  设置会话单个大内存分配的最大尺寸。

  巨大分配直接从内核中提供，因此不能保证成功。

  此值也适用于全局（非会话相关）内存分配。

  垃圾收集（MOT）

- enable_gc = true

  是否使用垃圾收集器（Garbage Collector，GC）。

- reclaim_threshold = 512 KB

  设置垃圾收集器的内存阈值。

  每个会话管理自己的待回收对象列表，并在事务提交时执行自己的垃圾回收。此值决定了等待回收的对象的总内存阈值，超过该阈值，会话将触发垃圾回收。

  一般来说，这里是在权衡未回收对象与垃圾收集频率。设置低值会使未回收的内存保持在较少的水平，但会导致频繁的垃圾回收，从而影响性能。设置高值可以减少触发垃圾回收的频率，但会导致未回收的内存过多。此设置取决于整体工作负载。

- reclaim_batch_size = 15432

  设置垃圾回收的批次大小。

  垃圾收集器从对象中批量回收内存，以便限制在一次垃圾收集传递中回收的对象数量。

此目的是最小化单个垃圾收集传递的操作时间。

- high_reclaim_threshold = 8 MB

  设置垃圾回收的高内存阈值。

  由于垃圾收集是批量工作的，因此会话可能有许多可以回收的对象，但这些对象不能回收。在这种情况下，为了防止垃圾收集列表变得过于膨胀，尽管已经达到批处理大小限制，此值继续单独回收对象，直到待回收对象小于该阈值，或者没有更多符合回收条件的对象。

  默认 MOT.conf 文件

  最小设置和配置指定将 postgresql.conf 文件指向 MOT.conf 文件的位置：

```
postgresql.conf
mot_config_file = '/tmp/gauss/ MOT.conf'
```

  确保 max_process_memory 设置的值足够包含 MOT 的全局（数据和索引）和本地（会话）内存。

  MOT.conf 的默认内容满足开始使用的需求。设置内容后续可以优化。

12.2.4MOT 使用

使用 MOT 非常简单，以下几个小节将会进行描述。

GBase 8c 允许应用程序使用 MOT 和基于标准磁盘的表。MOT 适用于最活跃、高竞争和对吞吐量敏感的应用程序表，也可用于所有应用程序的表。

以下命令介绍如何创建 MOT，以及如何将现有的基于磁盘的表转换为 MOT，以加速应用程序的数据库相关性能。MOT 尤其有利于已证明是瓶颈的表。

12.2.4.1 授予用户权限

以授予数据库用户对 MOT 存储引擎的访问权限为例。每个数据库用户仅执行一次，通常在初始配置阶段完成。

说明

- MOT 通过外部数据封装器（Foreign Data Wrapper，FDW）机制与 GBase 8c 数据库集成，所以需要授权用户权限。

要使特定用户能够创建和访问 MOT（DDL、DML、SELECT），以下语句只执行一次：

```
GRANT USAGE ON FOREIGN SERVER mot_server TO <user>;
```

所有关键字不区分大小写。

12.2.4.2 创建/删除 MOT

创建 MOT 非常简单。只有 MOT 中的创建和删除表语句与 GBase 8c 中基于磁盘的表的语句不同。SELECT、DML 和 DDL 的所有其他命令的语法对于 MOT 表和 GBase 8c 基于磁盘的表是一样的。

- 创建 MOT：

```
create FOREIGN table test(x int) [server mot_server];
```

- 以上语句中：
  - 始终使用 FOREIGN 关键字引用 MOT。
  - 在创建 MOT 表时，[server mot_server]部分是可选的，因为 MOT 是一个集成的引擎，而不是一个独立的服务器。
  - 上文以创建一个名为 test 的内存表（表中有一个名为 x 的整数列）为例。在下一节（创建索引）中将提供一个更现实的例子。
  - 如果 postgresql.conf 中开启了增量检查点，则无法创建 MOT。因此请在创建 MOT 前将 enable_incremental_checkpoint 设置为 off。
- 删除名为 test 的 MOT：

```
drop FOREIGN table test;
```

有关 MOT 的功能限制（如数据类型），请参见 MOT SQL 覆盖和限制。

12.2.4.3 为 MOT 创建索引

支持标准的 PostgreSQL 创建和删除索引语句。

例如：

```
create index    text_index1 on test(x) ;
```

创建一个用于 TPC-C 的 ORDER 表，并创建索引：

```
create FOREIGN table bmsql_oorder (
  o_w_id          integer          not null,
  o_d_id          integer          not null,
  o_id            integer          not null,
  o_c_id          integer not null,
```

```
    o_carrier_id integer,
    o_ol_cnt        integer,
    o_all_local    integer,
    o_entry_d       timestamp,
    primary key (o_w_id, o_d_id, o_id)
);
create index    bmsql_oorder_index1 on bmsql_oorder(o_w_id, o_d_id, o_c_id, o_id) ;
```

说明

- 在 MOT 名字之前不需要指定 FOREIGN 关键字，仅用于创建和删除表的命令。
- 有关 MOT 索引限制，请参见 MOT SQL 覆盖和限制的索引部分内容。

12.2.4.4 将磁盘表转换为 MOT

磁盘表直接转换为 MOT 尚不能实现，这意味着尚不存在将基于磁盘的表转换为 MOT 的 ALTER TABLE 语句。

下面介绍如何手动将基于磁盘的表转换为 MOT，如何使用 gs_dump 工具导出数据，以及如何使用 gs_restore 工具导入数据。

前置条件检查

- 检查待转换为 MOT 的磁盘表的模式是否包含所有需要的列。
- 检查架构是否包含任何不支持的列数据类型，具体参见 MOT SQL 覆盖和限制中的不支持的数据类型。
- 如果不支持特定列，则建议首先创建一个更新了模式的备磁盘表。此模式与原始表相同，只是所有不支持的类型都已转换为支持的类型。
- 使用以下脚本导出该备磁盘表，然后导入到 MOT 中。

转换

要将基于磁盘的表转换为 MOT，请执行以下

(1) 暂停应用程序活动。

(2) 使用 gs_dump 工具将表数据转储到磁盘的物理文件中。请确保使用 data only。

(3) 重命名原始基于磁盘的表。

(4) 创建同名同模式的 MOT。请确保使用创建 FOREIGN 关键字指定该表为 MOT。

(5) 使用 gs_restore 将磁盘文件的数据加载/恢复到数据库表中。

(6) 浏览或手动验证所有原始数据是否正确导入到新的 MOT 中。下面将举例说明。

(7) 恢复应用程序活动。

须知

- 由于表名称保持不变，应用程序查询和相关数据库存储过程将能够无缝访问新的 MOT，而无需更改代码。请注意，MOT 目前不支持跨引擎多表查询（如使用 Join、Union 和子查询）和跨引擎多表事务。因此，如果在多表查询、存储过程或事务中访问原始表，则必须将所有相关的磁盘表转换为 MOT，或者更改应用程序或数据库中的相关代码。

转换示例

假设要将数据库 benchmarksql 中一个基于磁盘的表 customer 迁移到 MOT 中。将 customer 表迁移到 MOT，操作下：

(1) 检查源表列类型。验证 MOT 支持所有类型，详情请参阅 MOT SQL 覆盖和限制中的不支持的数据类型。

```
benchmarksql-# \d+ customer
                                Table "public.customer"
 Column |    Type     | Modifiers | Storage | Stats target | Description
--------+-------------+-----------+---------+--------------+-------------
 x      | integer |           | plain   |              |
 y      | integer |           | plain   |              |
Has OIDs: no
Options: orientation=row, compression=no
```

(2) 请检查源表数据。

```
benchmarksql=# select * from customer;
 x | y
---+---
 1 | 2
 3 | 4
(2 rows)
```

(3) 只能使用 gs_dump 转储表数据。

```
$ gs_dump -Fc benchmarksql -a --table customer -f customer.dump -p 16000
gs_dump[port='15500'][benchmarksql][2020-06-04 16:45:38]: dump database benchmarksql successfully
gs_dump[port='15500'][benchmarksql][2020-06-04 16:45:38]: total time: 332  ms
```

(4) 重命名源表。

```
benchmarksql=# alter table customer rename to customer_bk;
ALTER TABLE
```

(5) 创建与源表完全相同的 MOT。

```
benchmarksql=# create foreign table customer (x int, y int);
CREATE FOREIGN TABLE
benchmarksql=# select * from customer;
 x | y
---+---
(0 rows)
```

(6) 将源转储数据导入到新 MOT 中。

```
$ gs_restore -C -d benchmarksql customer.dump -p 16000
restore operation successful
total time: 24    ms
Check that the data was imported successfully.
benchmarksql=# select * from customer;
 x | y
---+---
 1 | 2
 3 | 4
(2 rows)

benchmarksql=# \d
                                    List of relations
 Schema |       Name        |       Type        | Owner |                  Storage
--------+-------------------+-------------------+-------+-----------------------------------
 public | customer          | foreign table | aharon |
 public | customer_bk | table                 | aharon | {orientation=row,compression=no}
(2 rows)
```

12.2.4.5 重试中止事务

在乐观并发控制（OCC）中，在 COMMIT 阶段前的事务期间（使用任何隔离级别）不会对记录进行锁定。这是一个能显著提高性能的强大优势。它的缺点是，如果另一个会话尝试更新相同的记录，则更新可能会失败。所以必须中止整个事务。这些所谓的更新冲突是由 MOT 在提交时通过版本检查机制检测到的。

说明

- 使用悲观并发控制的引擎，如标准 Postgres 和 GBase 8c 基于磁盘的表，当使用 SERIALIZABLE 或 REPEATABLE-READ 隔离级别时，也会发生类似的异常中止。

这种更新冲突在常见的 OLTP 场景中非常少见，在使用 MOT 时尤其少见。但是，由于仍有可能发生这种情况，开发人员应该考虑使用事务重试代码来解决此问题。

下面以多个会话同时尝试更新同一个表为例，说明如何重试表命令。有关更多详细信息，请参阅"OCC 与 2PL 的区别举例”部分。下面以 TPC-C 支付事务为例。

```
int commitAborts = 0;
while (commitAborts < RETRY_LIMIT) {
    try {
        stmt =db.stmtPaymentUpdateDistrict;
        stmt.setDouble(1, 100);
        stmt.setInt(2, 1);
        stmt.setInt(3, 1);
        stmt.executeUpdate();
        db.commit();
        break;
    }
    catch (SQLException se) {
        if(se != null && se.getMessage().contains("could not serialize access due to
concurrent update")) {
            log.error("cgbasemit abort = " + se.getMessage());
            commitAborts++;
            continue;
        }else {
            db.rollback();
        }
        break;
    }
}
```

12.2.4.6 MOT 外部支持工具

为了支持 MOT，修改了以下外部 GBase 8c 工具。请确保使用的工具是最新版本。下面将介绍与 MOT 相关的用法。

- gs_ctl（全量和增量）

此工具用于从主服务器创建备服务器，以及当服务器的时间线偏离后，将服务器与其副本进行同步。

在操作结束时，工具将获取最新的 MOT 检查点，同时考虑 checkpoint_dir 配置值。

检查点从源服务器的 checkpoint_dir 读取到目标服务器的 checkpoint_dir。

目前 MOT 不支持增量检查点。因此，gs_ctl 增量构建对于 MOT 来说不是以增量方式工作，而是以全量方式工作。Postgres 磁盘表仍然可以增量构建。

- gs_basebackup

  gs_basebackup 用于准备运行中服务器的基础备份，不影响其他数据库客户端。

  MOT 检查点也会在操作结束时获取。但是，检查点的位置是从源服务器中的 checkpoint_dir 获取的，并传输到源数据目录中，以便正确备份。

- gs_dump

  gs_dump 用于将数据库模式和数据导出到文件中。支持 MOT。

- gs_restore

  gs_restore 用于从文件中导入数据库模式和数据。支持 MOT。

12.2.4.7 MOT SQL 覆盖和限制

MOT 设计几乎能够覆盖 SQL 和未来特性集。例如，大多数支持标准的 Postgres SQL，也支持常见的数据库特性，如存储过程、自定义函数等。

下面介绍各种 SQL 覆盖和限制。

不支持的特性

MOT 不支持以下特性：

- 跨引擎操作：不支持跨引擎（磁盘+MOT）的查询、视图或事务。计划于 2021 年实现该特性。
- MVCC、隔离：不支持没有快照/可序列化隔离。计划于 2021 年实现该特性。
- 本地内存限制为 1GB。一个事务只能更改小于 1GB 的数据。
- 容量（数据+索引）受限于可用内存。未来将提供 Anti-caching 和数据分层功能。
- 不支持全文检索索引。
- 不支持逻辑复制特性。

此外，下面详细列出了 MOT、MOT 索引、查询和 DML 语法的各种通用限制。

MOT 限制

MOT 功能限制：

- 按范围分区
- AES 加密
- 流操作
- 自定义类型
- 子事务
- DML 触发器
- DDL 触发器
- "C”或"POSIX”以外的排序规则

不支持的 DDL 操作

- 修改表结构
- 创建 including 表
- 创建 as select 表
- 按范围分区
- 创建无日志记录子句（no-logging clause）的表
- 创建可延迟约束主键（DEFERRABLE）
- 重建索引
- 表空间
- 使用子命令创建架构

不支持的数据类型

- UUID
- User-Defined Type (UDF)
- Array data type
- NVARCHAR2(n)
- NVARCHAR(n)

- Clob
- Name
- Blob
- Raw
- Path
- Circle
- Reltime
- Bit varying(10)
- Tsvector
- Tsquery
- JSON
- HSTORE
- Box
- Text
- Line
- Point
- LSEG
- POLYGON
- INET
- CIDR
- MACADDR
- Smalldatetime
- BYTEA
- Bit
- Varbit
- OID

- Money
- 无限制的 varchar/character varying
- HSTORE
- XML

不支持的索引 DDL 和索引

- 在小数和数值类型上创建索引
- 在可空列上创建索引
- 单表创建索引总数>9
- 在键大小>256 的表上创建索引

键大小包括以字节为单位的列大小+列附加大小，这是维护索引所需的开销。下表列出了不同列类型的列附加大小。

此外，如果索引不是唯一的，额外需要 8 字节。

下面是伪代码计算键大小：

```
keySize =0;

for each (column in index){
        keySize += (columnSize + columnAddSize);
}
if (index is non_unique) {
        keySize += 8;
}
```

列类型	列大小	列附加大小
varchar	N	4
tinyint	1	1
smallint	2	1
int	4	1
longint	8	1

float	4	2
double	8	3

上表中未指定的类型，列附加大小为零（例如时间戳）。

不支持的 DML

- Merge into
- Select into
- Lock table
- Copy from table

12.2.5MOT 管理

12.2.5.1 MOT 持久性

持久性是指长期的数据保护（也称为磁盘持久化）。持久性意味着存储的数据不会遭受任何形式的退化或损坏，因此数据不会丢失或损坏。持久性可确保在有计划停机（例如维护）或计划外崩溃（例如电源故障）后数据和 MOT 引擎恢复到一致状态。

内存存储是易失的、需要电力来维护所存储的信息。另一方面，磁盘存储是非易失性的，这意味着它不需要电源来维护存储的信息，因此不用担心停电。MOT 同时使用这两种类型的存储。内存中存储了所有数据，同时将事务性更改持久化到磁盘，并保持频繁的定期 MOT 检查点，以确保在关机时恢复数据。

用户必须保证有足够的磁盘空间用于日志记录和检查点操作。检查点使用单独的驱动器，通过减少磁盘 I/O 负载来提高性能。

参考 MOT 关键技术了解如何在 MOT 引擎中实现持久化。

若要设置持久性：

为保证严格一致性，请在 postgresql.conf 配置文件中将参数 sync_commit 配置为 on。

MOT 的 WAL 重做日志和检查点开启持久性，下面将详细介绍。

MOT 日志记录：WAL 重做日志

为保证持久性，MOT 全面集成 GBase 8c 的 WAL 机制，通过 GBase 8c 的 XLOG 接口持久化 WAL 记录。这意味着，每次 MOT 记录的添加、更新和删除都记录在 WAL 中。确保

了可以从这个非易失性日志重新生成和恢复最新的数据状态。例如，如果向表中添加了 3 行，删除了 2 行，更新了 1 行，那么日志中将记录 6 个条目。

MOT 日志记录和 GBase 8c 磁盘表的其他记录写入同一个 WAL 中。

MOT 只记录事务提交阶段的操作。

MOT 只记录更新的增量记录，以便最小化写入磁盘的数据量。

在恢复期间，从最后一个已知或特定检查点加载数据；然后使用 WAL 重做日志完成从该点开始发生的数据更改。

WAL 重做日志将保留所有表行修改，直到执行检查点（如上所述）。然后可以截断日志，以减少恢复时间和节省磁盘空间。

⚠ 注意

为确保日志 IO 设备不会成为瓶颈，日志文件必须放在具有低延迟的驱动器上。

MOT 日志类型

支持两个同步事务日志选项和一个异步事务日志选项。MOT 还支持同步的组提交日志记录与 NUMA-aware 优化，如下所述。

根据您的配置，实现以下类型的日志记录：

- 同步重做日志记录

同步重做日志记录选项是最简单、最严格的重做日志记录器。当客户端应用程序提交事务时，事务重做条目记录在 WAL 重做日志中，如下所示：

(1) 当事务正在进行时，它存储在 MOT 内存中。

(2) 事务完成后，客户端应用程序发送 Commit 命令，该事务被锁定，然后写入磁盘上的 WAL 重做日志。当事务日志条目写入日志时，客户端应用程序仍在等待响应。

(3) 一旦事务的整个缓冲区被写入日志，就更改内存中的数据，然后提交事务。事务提交后，通知客户端应用程序事务完成。

总结

同步重做日志记录选项是最安全、最严格的，因为它确保了客户端应用程序和每个事务提交时的 WAL 重做日志条目的完全同步，从而确保了总的持久性和一致性，并且绝对不会丢失数据。此日志记录选项可防止客户端应用程序在事务尚未持久化到磁盘时将事务标记为成功的情况。

同步重做日志记录选项的缺点是，它是三个选项中最慢的日志机制。因为客户端应用程序必须等待所有数据都写入磁盘，并且磁盘写入过于频繁导致数据库变慢。

- 组同步重做日志记录

组同步重做日志记录选项与同步重做日志记录选项非常相似，它确保完全持久性，绝对不会丢失数据，并保证客户端应用程序和 WAL 重做日志条目的完全同步。不同的是，组同步重做日志记录选项将事务重做条目组同时写入磁盘上的 WAL 重做日志，而不是在提交时写入每个事务。使用组同步重做日志记录可以减少磁盘 I/O 数量，从而提高性能，特别是在运行繁重的工作负载时。

MOT 引擎通过根据运行事务的核的 NUMA 槽位自动对事务进行分组，使用 NUMA 感知优化来执行同步的组提交记录。

有关 NUMA-aware 内存访问的更多信息，请参阅 NUMA-aware 分配和亲和性。

当一个事务提交时，一组条目记录在 WAL 重做日志中，如下所示：

(1) 当事务正在进行时，它存储在内存中。MOT 引擎根据运行事务的核的 NUMA 槽位对桶中的事务进行分组。在同一槽位上运行的所有事务都被分组在一起，并且多个组将根据事务运行的核并行填充。

这样，将事务写入 WAL 更为有效，因为来自同一个槽位的所有缓冲区都一起写入磁盘。

注意：每个线程在属于单槽位的单核/CPU 上运行，每个线程只写在各自运行的核的槽位上。

(2) 在事务完成并且客户端应用程序发送 Commit 命令之后，事务重做日志条目将与同组的其他事务一起序列化。

(3) 当特定一组事务满足配置条件后，如重做日志（MOT）小节中描述的已提交的事务数或超时时间，该组中的事务将被写入磁盘的 WAL 中。当这些日志条目被写入日志时，发出提交请求的客户端应用程序正在等待响应。

(4) 一旦 NUMA-aware 组中的所有事务缓冲区都写入日志，该组中的所有事务都将对内存存储执行必要的更改，并且通知客户端这些事务已完成。

总结

组同步重做日志记录选项是一个极其安全和严格的日志记录选项，因为它确保了客户端应用程序和 WAL 重做日志条目的完全同步，从而确保了总的持久性和一致性，而且绝对不会丢失数据。此日志记录选项可防止客户端应用程序在事务尚未持久化到磁盘时将事务标记

为成功的情况。

该选项的磁盘写入次数比同步重做日志记录选项少，这可能意味着它更快。缺点是事务被锁定的时间更长。在同一 NUMA 内存中的所有事务都写入磁盘的 WAL 重做日志之前，它们一直处于锁定状态。

是否使用此选项取决于事务工作负载的类型。例如，此选项有利于事务较多的系统。而对于事务少的系统而言，磁盘写入量也很少，因此不建议使用。

- 异步重做日志记录

异步重做日志记录选项是最快的日志记录方法，但是它不能确保数据不会丢失，某些仍位于缓冲区中且尚未写入磁盘的数据在电源故障或数据库崩溃时可能会丢失。当客户端应用程序提交事务时，事务重做条目将记录在内部缓冲区中，并按预先配置的时间间隔写入磁盘。客户端应用程序不会等待数据写入磁盘。它将继续进行下一个事务。这就是异步重做日志记录最快的原因。

当客户端应用程序提交事务时，事务重做条目记录在 WAL 重做日志中，如下所示：

(1) 当事务正在进行时，它存储在 MOT 内存中。

(2) 在事务完成并且客户端应用程序发送 Commit 命令后，事务重做条目将被写入内部缓冲区，但尚未写入磁盘。然后更改 MOT 数据内存，并通知客户端应用程序事务已提交。

(3) 后台运行的重做日志线程按预先配置的时间间隔收集所有缓存的重做日志条目，并将它们写入磁盘。

总结

异步重做日志记录选项是最快的日志记录选项，因为它不需要客户端应用程序等待数据写入磁盘。此外，它将许多事务重做条目分组并把它们写入一起，从而减少降低 MOT 引擎速度的磁盘 I/O 数量。

异步重做日志记录选项的缺点是它不能确保在崩溃或失败时数据不会丢失。已提交但尚未写入磁盘的数据在提交时是不持久的，因此在出现故障时无法恢复。异步重做日志记录选项对于愿意牺牲数据恢复（一致性）而不是性能的应用程序来说最为适用。

配置日志

标准 GBase 8c 磁盘引擎支持两个同步事务日志选项和一个异步事务日志选项。

配置日志记录

(1) 在 postgresql.conf 配置文件中的 sync_commit (On = Synchronous)参数中指定是否执行同

步或异步事务日志记录。

(2) 在重做日志章节中的 mot.conf 配置文件里，将 enable_redo_log 参数设置为 True。

如果已选择事务日志记录的同步模式（如上文所述，synchronous_commit = on），则在 mot.conf 配置文件中的 enable_group_commit 参数中指定 Group Synchronous Redo Logging 选项或 Synchronous Redo Logging 选项。如果选择 Group Synchronous Redo Logging，必须在 mot.conf 文件中定义以下阈值，决定何时将一组事务记录在 WAL 中。

- group_commit_size：一组已提交的事务数。例如，16 表示当同一组中的 16 个事务已由它们的客户端应用程序提交时，则针对 16 个事务中的每个事务，在磁盘的 WAL 重做日志中写入一个条目。
- group_commit_timeout：超时时间，单位为毫秒。例如，10 表示在 10 毫秒之后，为同一组由客户端应用程序在最近 10 毫秒内提交的每个事务，在磁盘的 WAL 重做日志中写入一个条目。

说明

➢ 有关配置的详细信息，请参阅 MOT 持久性中的重做日志。

MOT 检查点

检查点是一个时间点。在这个时间点，表行的所有数据都保存在持久存储上的文件中，以便创建完整持久的数据库镜像。这是一个数据在某个时间点的快照。

检查点减少了为确保持久性而必须重放的 WAL 重做日志条目的数量，以此缩短数据库的恢复时间。检查点还减少了保存所有日志条目所需的存储空间。

如果没有检查点，那么为了恢复数据库，所有 WAL 重做条目必须从开始时间进行重放，可能需要几天或几周的时间，这取决于数据库中的记录数量。检查点记录数据库的当前状态，并允许丢弃旧的重做条目。

检查点在恢复方案（特别是冷启动）中是必不可少的。首先，从最后一个已知或特定检查点加载数据；然后使用 WAL 完成此后发生的数据更改。

例如，如果同一表行被修改 100 次，则日志中将记录 100 个条目。当使用检查点后，即使表行被修改了 100 次，检查点也可以一次性记录。在记录检查点之后，可以基于该检查点执行恢复，并且只需要播放自该检查点之后发生的 WAL 重做日志条目。

12.2.5.2 MOT 恢复

MOT 恢复的主要目标是在有计划停机（例如维护）或计划外崩溃（例如电源故障后）后，将数据和 MOT 引擎恢复到一致状态。

MOT 恢复是随着 GBase 8c 数据库其余部分的恢复而自动执行的，并且完全集成到 GBase 8c 恢复过程（也称为冷启动）。

MOT 恢复包括两个阶段：

- 检查点恢复：必须通过将数据加载到内存行并创建索引，从磁盘上的最新检查点文件恢复数据。
- WAL 重做日志恢复：从检查点恢复中使用检查点后，必须通过重放之后添加到日志中的记录，从 WAL 重做日志中恢复最近的数据（在检查点中未捕获）。

  GBase 8c 管理和触发 WAL 重做日志恢复。

  配置恢复。
- 虽然 WAL 恢复以串行方式执行，但可以将检查点恢复配置为以多线程方式运行（即由多个工作线程并行运行）。
- 在 mot.conf 文件中配置 checkpoint_recovery_workers 参数。

12.2.5.3 MOT 复制和高可用

由于 MOT 集成到 GBase 8c 中，并且使用或支持其复制和高可用，因此，MOT 原厂功能即支持同步复制和异步复制。

gs_ctl 工具用于可用性控制和数据库操作。这包括 gs_ctl 切换、gs_ctl 故障切换、gs_ctl 构建等。

有关配置复制和高可用性的更多信息，请参见《GBase 8c V5_5.0.0_工具与命令参考手册》。

12.2.5.4 MOT 内存管理

规划和微调，请参见 MOT 准备和 MOT 配置。

12.2.5.5 MOT VACUUM 清理

使用 VACUUM 进行垃圾收集，并有选择地分析数据库，如下所示。

- 【Postgres】

在 Postgres 中，VACUUM 用于回收死元组占用的存储空间。在正常的 Postgres 操作中，删除的元组或因更新而作废的元组不会从表中物理删除。只能由 VACUUM 清理。因此，需要定期执行 VACUUM，特别是在频繁更新的表上。

- 【MOT 扩展】

MOT 不需要周期性的 VACUUM 操作，因为新元组会重用失效元组和空元组。只有当 MOT 的大小急剧减少，并且不计划恢复到原来大小时，才需要 VACUUM 操作。

例如，应用程序定期（如每周一次）大量删除表数据的同时插入新数据，这需要几天时间，并且不一定是相同数量的行。在这种情况下，可以使用 VACUUM。

对 MOT 的 VACUUM 操作总是被转换为带有排他表锁的 VACUUM FULL。

- 支持的语法和限制

按规范激活 VACUUM 操作。

```
VACUUM [FULL | ANALYZE] [ table ];
```

只支持 FULL 和 ANALYZE VACUUM 两种类型。VACUUM 操作只能对整个 MOT 进行。

不支持以下 Postgres VACUUM 选项：

- FREEZE
- VERBOSE
- Column specification
- LAZY 模式（部分表扫描）

此外，不支持以下功能：

- AUTOVACUUM

12.2.5.6 MOT 统计

统计信息主要用于性能分析或调试。在生产环境中，通常不打开它们（默认是关闭的）。统计信息主要由数据库开发人员使用，数据库用户较少使用。

对性能有一定影响，特别是对服务器。对用户的影响可以忽略不计。

统计信息保存在数据库服务器日志中。该日志位于 data 文件夹中，命名为 postgresql-DATE-TIME.log。

有关详细的配置选项，请参阅 MOT 配置中的统计（MOT）。

12.2.5.7 MOT 监控

监控的所有语法支持基于 Postgres 的 FDW 表，包括下面的表或索引大小。此外，还存在用于监控 MOT 内存消耗的特殊函数，包括 MOT 全局内存、MOT 本地内存和单个客户端会话。

- 表和索引大小

  可以通过查询 pg_relation_size 来监控表和索引的大小。

  例如：

  数据大小

```
select pg_relation_size('customer');
```

  索引

```
select pg_relation_size('customer_pkey');
```

- MOT 全局内存详情

  检查 MOT 全局内存大小，主要是数据和索引。

```
select * from mot_global_memory_detail();
```

  结果如下。

```
numa_node    | reserved_size             | used_size
---------------+----------------+-------------
-1                  | 194716368896          | 25908215808
0                   | 446693376             | 446693376
1                   | 452984832             | 452984832
2                   | 452984832             | 452984832
3                   | 452984832             | 452984832
4                   | 452984832             | 452984832
5                   | 364904448             | 364904448
6                   | 301989888             | 301989888
7                   | 301989888             | 301989888
```

  其中，-1 为总内存，0 -7 为 NUMA 内存节点。

- MOT 本地内存详情

  检查 MOT 本地内存大小，包括会话内存。

```
select * from mot_local_memory_detail();
```

结果如下。

```
numa_node    | reserved_size          | used_size
---------------+----------------+-------------
-1                  | 144703488           | 144703488
0                   | 25165824            | 25165824
1                   | 25165824            | 25165824
2                   | 18874368            | 18874368
3                   | 18874368            | 18874368
4                   | 18874368            | 18874368
5                   | 12582912            | 12582912
6                   | 12582912            | 12582912
7                   | 12582912            | 12582912
```

其中，-1 为总内存，0 -7 为 NUMA 内存节点。

- 会话内存

  会话管理的内存从 MOT 本地内存中获取。

  所有活动会话（连接）的内存使用量可以通过以下查询。

```
select * from mot_session_memory_detail();
```

结果如下。

```
sessid                              | total_size | free_size | used_size
-------------------------------- - - - - - -+-----------+----------+----------
1591175063.139755603855104 | 6291456       | 1800704     | 4490752
```

其中，total_size：分配给会话的内存，free_size：未使用的内存，used_size：使用中的内存。

DBA 可以通过以下查询确定当前会话使用的本地内存状态。

```
select * from mot_session_memory_detail()
 where sessid = pg_current_sessionid();
```

结果如下。

```
postgres=# select * from mot_session_memory_detail() where sessid = pg_current_sessionid();
          sessid           | total_size | free_size | used_size
---------------------------+------------+-----------+-----------
 1591955545.139837641324288 |    6291456 |   1861992 |   4429464
(1 row)
```

12.2.5.8 MOT 错误消息

错误可能由多种场景引起。所有错误都记录在数据库服务器日志文件中。此外，与用户相关的错误作为对查询、事务或存储过程执行或数据库管理操作的响应的一部分返回给用户。

- 服务器日志中报告的错误包括函数、实体、上下文、错误消息、错误描述和严重性。
- 向用户报告的错误被翻译成标准 PostgreSQL 错误码，可能由 MOT 特定的消息和描述组成。

错误提示、错误描述和错误码见下文。该错误码实际上是内部代码，不记录也不返回给用户。

写入日志文件的错误

所有错误都记录在数据库服务器日志文件中。以下列出了写入数据库服务器日志文件但未返回给用户的错误。该日志位于 data 文件夹中，命名为 postgresql-DATE-TIME.log。

表 12-1 只写入日志文件的错误

日志消息	内部错误代码
Error code denoting success	MOT_NO_ERROR 0
Out of memory	MOT_ERROR_OOM 1
Invalid configuration	MOT_ERROR_INVALID_CFG 2
Invalid argument passed to function	MOT_ERROR_INVALID_ARG 3
System call failed	MOT_ERROR_SYSTEM_FAILURE 4
Resource limit reached	MOT_ERROR_RESOURCE_LIMIT 5
Internal logic error	MOT_ERROR_INTERNAL 6
Resource unavailable	MOT_ERROR_RESOURCE_UNAVAILABLE 7
Unique violation	MOT_ERROR_UNIQUE_VIOLATION 8
Invalid memory allocation size	MOT_ERROR_INVALID_MEMORY_SIZE 9

Index out of range	MOT_ERROR_INDEX_OUT_OF_RANGE 10
Error code unknown	MOT_ERROR_INVALID_STATE 11

返回给用户的错误

下面列出了写入数据库服务器日志文件并返回给用户的错误。

MOT 使用返回码（Return Code，RC）返回 Postgres 标准错误代码至封装。某些 RC 会导致向正在与数据库交互的用户生成错误消息。

MOT 从内部返回 Postgres 代码（见下文）到数据库包，数据库封装根据标准的 Postgres 行为对其做出反应。

说明

提示信息中的%s、%u、%lu 指代相应的错误信息（如查询、表名或其他信息）。

- %s：字符串
- %u：数字
- %lu：数字

表 12-2 返回给用户并记录到日志文件的错误

返回给用户的短/长描述	Postgres 代码	内部错误码
Success. Denotes success	ERRCODESUCCESSFUL COMPLETION	RC_OK = 0
Failure Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_ERROR = 1
Unknown error has occurred. Denotes aborted operation.	ERRCODE_FDW_ERROR	RC_ABORT

返回给用户的短/长描述	Postgres 代码	内部错误码
Column definition of %s is not supported. Column type %s is not supported yet.	ERRCODE_INVALID_COLUMN_DEFINITION	RC_UNSUPPORTED_COL_TYPE
Column definition of %s is not supported. Column type Array of %s is not supported yet.	ERRCODE_INVALID_COLUMN_DEFINITION	RC_UNSUPPORTED_COL_TYPE_ARR
Column size %d exceeds max tuple size %u. Column definition of %s is not supported.	ERRCODE_FEATURE_NOT_SUPPORTED	RC_EXCEEDS_MAX_ROW_SIZE
Column name %s exceeds max name size %u. Column definition of %s is not supported.	ERRCODE_INVALID_COLUMN_DEFINITION	RC_COL_NAME_EXCEEDS_MAX_SIZE
Column size %d exceeds max size %u. Column definition of %s is not supported.	ERRCODE_INVALID_COLUMN_DEFINITION	RC_COL_SIZE_INVLALID
Cannot create table. Cannot add column %s; as the number of declared columns exceeds the maximum declared columns.	ERRCODE_FEATURENOTSUPPORTED	RC_TABLE_EXCEEDSMAX DECLARED_COLS

返回给用户的短/长描述	Postgres 代码	内部错误码
Cannot create index. Total column size is greater than maximum index size %u.	ERRCODE_FDW_KEYSIZE EXCEEDS_MAX_ALLOWED	RC_INDEX_EXCEEDS_MAX_SIZE
Cannot create index. Total number of indexes for table %s is greater than the maximum number of indexes allowed %u.	ERRCODE_FDW_TOOMANY INDEXES	RC_TABLE_EXCEEDS_MAX_INDEXES
Cannot execute statement. Maximum number of DDLs per transaction reached the maximum %u.	ERRCODE_FDW_TOOMANY DDL_CHANGESIN TRANSACTIONNOT ALLOWED	RC_TXN_EXCEEDS_MAX_DDLS
Unique constraint violation Duplicate key value violates unique constraint \"%s\. Key %s already exists.	ERRCODEUNIQUE VIOLATION	RC_UNIQUE_VIOLATION
Table \"%s\" does not exist.	ERRCODE_UNDEFINED_TABLE	RC_TABLE_NOT_FOUND
Index \"%s\" does not exist.	ERRCODE_UNDEFINED_TABLE	RC_INDEX_NOT_FOUND
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_LOCAL_ROW_FOUND

返回给用户的短/长描述	Postgres 代码	内部错误码
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_LOCAL_ROW_NOT_FOUND
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_LOCAL_ROW_DELETED
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_INSERT_ON_EXIST
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_INDEX_RETRY_INSERT
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_INDEX_DELETE
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_LOCAL_ROW_NOT_VISIBLE
Memory is temporarily unavailable.	ERRCODE_OUT_OF_LOGICAL_MEMORY	RC_MEMORY_ALLOCATION_ERROR
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_ILLEGAL_ROW_STATE
Null constraint violated. NULL value cannot be inserted into non-null column %s at table %s.	ERRCODE_FDW_ERROR	RC_NULL_VIOLATION
Critical error. Critical error: %s.	ERRCODE_FDW_ERROR	RC_PANIC
A checkpoint is in progress – cannot truncate table.	ERRCODE_FDW_OPERATION_NOT_SUPPORTED	RC_NA
Unknown error has occurred.	ERRCODE_FDW_ERROR	RC_MAX_VALUE

返回给用户的短/长描述	Postgres 代码	内部错误码
<recovery message>	-	ERRCODE_CONFIG_FILE_ERROR
<recovery message>	-	ERRCODE_INVALIDTABLE DEFINITION
Memory engine – Failed to perform commit prepared.	-	ERRCODE_INVALIDTRANSACTION STATE
Invalid option <option name>	-	ERRCODE_FDW_INVALID OPTION NAME
Invalid memory allocation request size.	-	ERRCODE_INVALIDPARAMETER VALUE
Memory is temporarily unavailable.	-	ERRCODE_OUT_OFLOGICAL MEMORY
Could not serialize access due to concurrent update.	-	ERRCODE_T_RSERIALIZATION FAILURE

返回给用户的短/长描述	Postgres 代码	内部错误码
Alter table operation is not supported for memory table. Cannot create MOT tables while incremental checkpoint is enabled. Re-index is not supported for memory tables.	-	ERRCODE_FDW_OPERATIONNOT SUPPORTED
Allocation of table metadata failed.	-	ERRCODE_OUT_OF_MEMORY
Database with OID %u does not exist.	-	ERRCODE_UNDEFINED_DATABASE
Value exceeds maximum precision: %d.	-	ERRCODE_NUMERIC_VALUEOUT OF_RANGE
You have reached a maximum logical capacity %lu of allowed %lu.	-	ERRCODE_OUT_OFLOGICAL MEMORY

12.3 MOT 的概念

本章介绍 GBase 8c MOT 的设计和工作原理，阐明其高级特性、功能及使用方法，旨在让读者了解 MOT 操作上的技术细节、重要特性细节和创新点。本章内容有助于决策 MOT 是否适合于特定的应用需求，以及进行最有效的使用和管理。

12.3.1MOT 纵向扩容架构

纵向扩容即为同一台机器添加额外的核以增加算力。纵向扩容是传统上为单对控制器和多核的机器增加算力的常见形式。纵向扩容架构受限于控制器的可扩展性。

技术要求

MOT 旨在实现以下目标：

- 线性扩容：MOT 提供事务性存储引擎，利用单个 NUMA 架构服务器的所有核，以提供近线性的扩容性能。这意味着 MOT 的目标是在机器的核数和性能提升倍数之间实现直接的、近线性的关系。
- 无最大核数限制：MOT 对最大核数不做任何限制。这意味着 MOT 可从单核扩展到高达 1000 秒的多核，并且新增的核退化速度最小，即便是在跨 NUMA 槽位边界的情况下。
- 极高的事务性吞吐量：MOT 提供了一个事务性存储引擎，与市场上任何其他 OLTP 供应商相比，它能够实现极高的事务性吞吐量。
- 极低的事务性时延：与市场上任何其他 OLTP 供应商相比，MOT 提供事务性存储引擎，可以达到极低的事务时延。
- 无缝集成和利用产品：MOT 事务引擎与 GBase 8c 产品标准无缝集成。通过这种方式，MOT 最大限度地重用了位于其事务性存储引擎顶部的功能。

设计原则

为了实现上述要求（特别是在多核的环境中），我们存储引擎的体系结构实施了以下技术和策略：

- 数据和索引只存在于内存中。
- 数据和索引不用物理分区来布局（因为对于某些类型的应用程序，这些分区的性能可能会降低）。
- 事务并发控制基于乐观并发控制（OCC），没有集中的争用点。有关 OCC 的详细信息，请参见 MOT 并发控制机制。
- 使用平行重做日志（最后单位为核）来有效避免中央锁定点。
- 使用免锁索引。有关免锁索引的详细信息，请参见 MOT 索引。
- 使用 NUMA 感知内存分配，避免跨槽位访问，特别是会话生命周期对象。有关 NUMA 感知的更多信息，请参见 NUMA-aware 分配和亲和性。
- 使用带有预缓存对象池的自定义 MOT 内存管理分配器，避免昂贵的运行时间分配和额外的争用点。这种专用的 MOT 内存分配器按需预先访问操作系统中较大的内存块，然后按需将内存分配给 MOT，使内存分配更加高效。

使用外部数据封装（FDW）进行集成

MOT 遵循并利用了 GBase 8c 的标准扩展机制——外部数据封装（FDW），如下图所示。

在 PostgreSQL 外部数据封装特性的支持下，作为其他数据源的代理的 MOT 数据库可以创建外表，如 MySQL、Oracle、PostgreSQL 等。当对外表执行查询时，FDW 将查询外部数据源并返回结果，就像查询内表一样。

GBase 8c 依赖 PostgreSQL 外部数据封装和索引支持，因此 SQL 完全覆盖，包括存储过程、用户定义函数、系统函数调用。

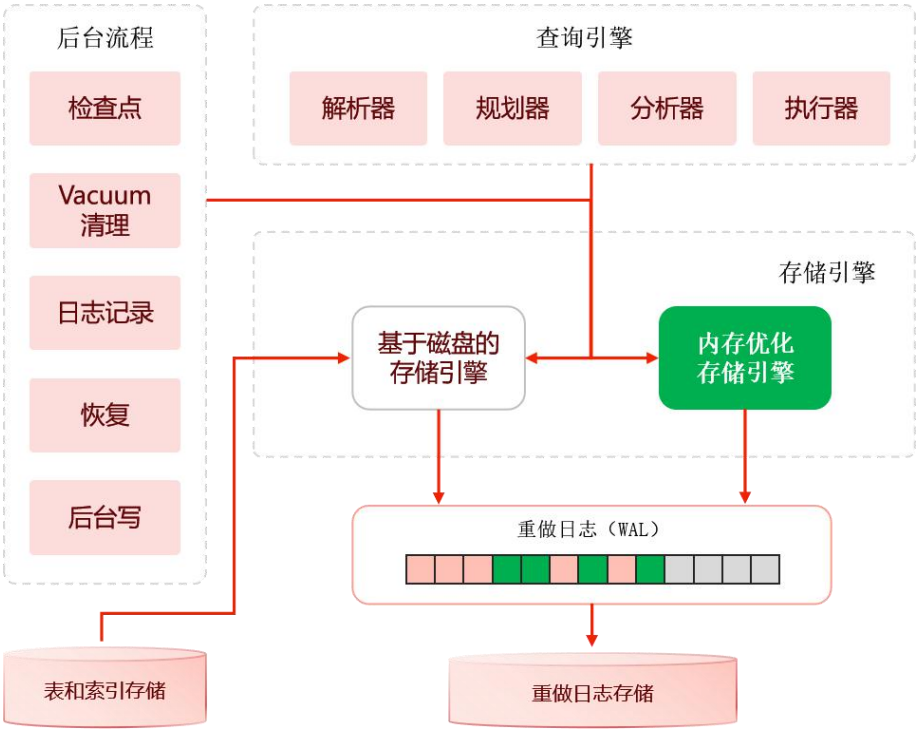


图 12-2 MOT 架构

上图中绿色表示 MOT 引擎，蓝色表示现有的 GBase 8c 组件。由此可见，FDW 在 MOT 引擎和 GBase 8c 组件之间进行中介。

与 MOT 相关的 FDW 定制

通过 FDW 集成 MOT 可以重用最上层的 GBase 8c 功能，从而显著缩短 MOT 的上市时间，同时不影响 SQL 的覆盖范围。

但是，GBase 8c 中原有的 FDW 机制并不是为存储引擎扩展而设计的，因此缺少以下基本功能：

- 查询规划阶段待计算的外表的索引感知
- 完整的 DDL 接口

- 完整的事务生命周期接口
- 检查点接口
- 重做日志接口
- 恢复接口
- 真空接口

为了支持所有缺失的功能，SQL 层和 FDW 接口层已扩展，从而为插入 MOT 事务存储引擎提供必要的基础设施。

12.3.2MOT 并发控制机制

MOT 完全符合原子性、一致性、隔离性、持久性（ACID）特性，如 MOT 简介所述。下面介绍 MOT 的并发控制机制。

MOT 本地内存和全局内存

SILO 管理本地内存和全局内存，如下图所示。

全局内存是所有核共享的长期内存，主要用于存储所有的表数据和索引。

本地内存是短期内存，主要由会话使用，用于处理事务及将数据更改存储到事务内存中，直到提交阶段。

当事务需要更改时，SILO 将该事务的所有数据从全局内存复制到本地内存。使用 OCC 方法，全局内存中放置的是最小的锁，因此争用时间极短。事务更改完成后，该数据从本地内存回推到全局内存中。

本地内存与 SILO 增强并发控制的基本交互式事务流如下所示：

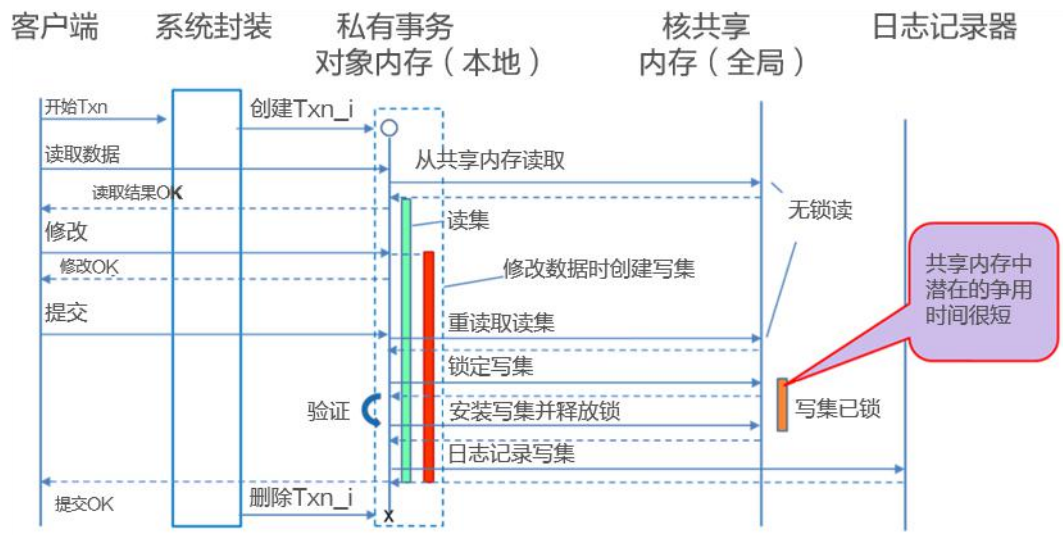


图 12-3 私有（本地）内存（每个事务）和全局内存（所有核的所有事务）

MOT SILO 增强特性

SILO 凭借其基本算法流程，优于我们在研究实验中测试的许多其他符合 ACID 的 OCC 算法。然而，为了使 SILO 成为产品级机制，我们必须用许多在最初设计中缺失的基本功能来增强它，例如：

- 新增对交互式事务的支持，其中事务的 SQL 运行在客户端实现，而不是作为服务器端的单个行。
- 新增乐观插入。
- 新增对非唯一索引的支持。
- 新增对事务中写后读校验（RAW）的支持，使用户能够在提交之前查看更改。
- 新增对无锁协同垃圾回收的支持。
- 新增对无锁检查点的支持。
- 新增对快速恢复的支持。
- 新增对分布式部署两阶段提交的支持。

在不破坏原始 SILO 的可扩展特性的前提下添加这些增强是非常具有挑战性的。

MOT 隔离级别

即使 MOT 完全兼容 ACID，GBase 8c 并非支持所有的隔离级别。下表介绍了各隔离级别，以及 MOT 支持和不支持的内容。

表 12-3 隔离级别

隔离级别	说明
READ UNCOMMITTED	MOT 不支持
READ COMMITTED	MOT 支持 READ COMMITTED（读已提交）隔离级别保证任何正在读取的数据在上一次读取时都已提交。它只是限制读者看到任何中间数据、未提交数据或脏读。数据被读取后可以自由更改，因此，读已提交隔离级别并不保证事务再次读取时能找到相同的数据。
SNAPSHOT	MOT 不支持 SNAPSHOT（快照）隔离级别提供与 SERIALIZABLE（可序列化）相同的保证，同时支持并发事务修改数据。相反，它迫使每个读者看到自己的世界版本（自己的快照）。不阻止并发更新使得编程非常容易，且可扩展性很强。然而，在许多实现中，这种隔离级别需要更高的服务器资源。
REPEATABLE READ	MOT 支持 REPEATABLE READ（可重复读）是一个更高的隔离级别，除了 READ COMMITTED 隔离级别的保证之外，它还保证任何读取的数据都不能更改。如果一个事务再次读取相同的数据，它将找出该数据，不做更改，并且保证它可读取。 乐观模型使得并发事务能更新该事务读取的行。在提交时，该事务将验证 REPEATABLE READ 隔离级别是否被违反。若违反，则回滚该事务，必须重试。
SERIALIZABLE	MOT 不支持 SERIALIZABLE（可序列化）隔离提供了更强的保证。除了 REPEATABLE READ 隔离级别保证的所有内容外，它还保证后续读取不会看到新数据。

	它之所以被命名为 SERIALIZABLE，是因为隔离非常严格，几乎有点像事务串行运行，而不是并行运行。

下表显示了不同隔离级别启用的并发副作用。

表 12-4 隔离级别启用的并发副作用

隔离级别	说明	不可重复读	幻读
READ UNCOMMITTED	是	是	是
READ COMMITTED	否	是	是
REPEATABLE READ	否	否	是
SNAPSHOT	否	否	否
SERIALIZABLE	否	否	否

MOT 乐观并发控制

并发控制模块（简称 CC 模块）提供了主内存引擎的所有事务性需求。CC 模块的主要目标是为主内存引擎提供各种隔离级别的支持。

乐观 OCC 与悲观 2PL

悲观 2PL（2 阶段锁定）和乐观并发控制（OCC）的功能差异在于对事务完整性分别采用悲观和乐观方法。

基于磁盘的表使用悲观方法，这是最常用的数据库方法。MOT 引擎使用的是乐观方法。

悲观方法和乐观方法的主要功能区别在于，如果冲突发生，

- 悲观的方法会导致客户端等待；
- 而乐观方法会导致其中一个事务失败，使得客户端必须重试失败的事务。

乐观并发控制方法（MOT 使用）

乐观并发控制（OCC）方法在冲突发生时检测冲突，并在提交时执行验证检查。

乐观方法开销较小，而且通常效率更高，原因之一是事务冲突在大多数应用程序中并不

常见。

当强制执行 REPEATABLE READ 隔离级别时，乐观方法与悲观方法之间的函数差异更大，而当强制执行 SERIALIZABLE 隔离级别时，函数差异最大。

悲观方法（MOT 未使用）

悲观并发控制（2PL，或称 2 阶段锁定）方法使用锁阻止在潜在冲突的发生。执行语句时应用锁，提交事务时释放锁。基于磁盘的行存储使用这种方法，并且添加了多版本并发控制（Multi-version Concurrency Control，MVCC）。

在 2PL 算法中，当一个事务正在写入行时，其他事务不能访问该行；当一个行正在读取时，其他事务不能覆盖该行。在访问时锁定每个行，以进行读写；在提交时释放锁。这些算法需要一个处理和避免死锁的方案。死锁可以通过计算等待图中的周期来检测。死锁可以通过使用 TSO 保持时序或使用某种回退方案来避免。

遇时锁定（ETL）

另一种方法是遇时锁定（ETL），它以乐观的方式处理读取，但写入操作锁定它们访问的数据。因此，来自不同 ETL 事务的写入操作相互感知，并可以决定中止。实验证明，ETL 通过两种方式提高 OCC 的性能：

- 首先，ETL 会在早期检测冲突，并通常能增加事务吞吐量。这是因为事务不会执行无用的操作。（通常）在提交时发现的冲突无法在不中止至少一个事务的情况下解决。
- 其次，ETL 写后读校验（RAW）运行高效，无需昂贵或复杂的机制。

OCC 与 2PL 的区别举例

下面是会话同时更新同一个表时，两种用户体验的区别：悲观（针对基于磁盘的表）和乐观（针对 MOT 表）。

本例中，使用如下表测试命令：

```
table "TEST”  – create table test (x int, y int, z int, primary key(x));
```

本示例描述同一测试的两个方面：用户体验（本示例中的操作）和重试要求。

悲观方法示例——用于基于磁盘的表

下面是一个悲观方法例子（非 MOT）。任何隔离级别都可能适用。

以下两个会话执行尝试更新单个表的事务。

WAIT LOCK 操作发生，客户端体验是：会话 2 卡住，直到会话 1 完成 COMMIT，会

话 2 才能进行。

但是，使用这种方法时，两个会话都成功，并且不会发生异常中止（除非应用了 SERIALIZABLE 或 REPEATABLE-READ 隔离级别），这会导致整个事务需要重试。

表 12-5 悲观方法代码示例

时间	会话 1	会话 2
t0	Begin	Begin
t1	update test set y=200 where x=1;	
t2	y=200	Update test set y=300 where x=1; – Wait on lock
t4	Commit	
		Unlock
		Commit (in READ-COMMITTED this will succeed, in SERIALIZABLE it will fail)
		y = 300

乐观方法示例——用于 MOT

下面是一个乐观方法的例子。

它描述了创建一个 MOT 表，然后有两个并发会话同时更新同一个 MOT 表的情况。

```
create foreign table test (x int, y int, z int, primary key(x));
```

- OCC 的优点是，在 COMMIT 之前没有锁。
- OCC 的缺点是，如果另一个会话更新了相同的记录，则更新可能会失败。如果更新失败（在所有支持的隔离级别中），则必须重试整个会话#2 事务。
- 更新冲突由内核在提交时通过版本检查机制检测。
- 会话 2 将不会等待其更新操作，并且由于在提交阶段检测到冲突而中止。

表 12-6 乐观方法代码示例——用于 MOT

时间	会话 1	会话 2
t0	Begin	Begin
t1	update test set y=200 where x=1;	
t2	y=200	Update test set y=300 where x=1;
t4	Commit	y = 300
		Commit
		ABORT
		y = 200

12.3.3扩展 FDW 与其他特性

为了在 GBase 8c 数据库中集成 MOT 存储引擎，利用并扩展了现有的 FDW 机制，可以将这些外表和数据源呈现为统一、本地可访问的关系来访问外部管理的数据库。MOT 存储引擎是嵌入在 GBase 8c 数据库内部的，表由 GBase 8c 管理。GBase 8c 规划器和执行器控制表的访问。

MOT 从 GBase 8c 获取日志和检查点服务，并参与恢复过程和其他过程。

下图显示了 MOT 存储引擎如何嵌入到数据库中，及其对数据库功能的双向访问。

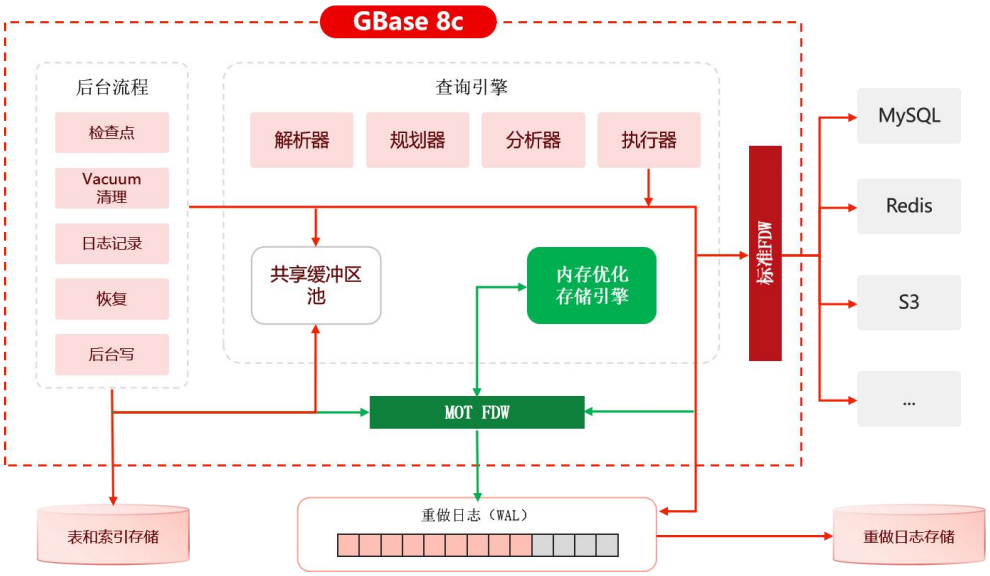


图 12-4 GBase 8c 内置 MOT 存储引擎——外部数据库的 FDW 访问

通过扩展和修改 FdwRoutine 结构来扩展 FDW 的能力，以便引入在 MOT 引入之前不需要的特性和调用。例如，新增了对以下功能的支持：添加索引、删除索引/表、截断、真空和表/索引内存统计。重点放在了 FdwRoutine 结构与日志、复制和检查点机制的集成，以便通过故障为跨表事务提供一致性。在这种情况下，MOT 本身有时会通过 FDW 层发起对 GBase 8c 功能的调用。

创建表和索引

为了支持 MOT 表的创建，重用了标准的 FDW 语法。

例如，创建 FOREIGN 表。

MOT DW 机制将指令传递给 MOT 存储引擎，用于实际建表。同样，我们支持创建索引（create index …）。此功能以前在 FDW 中不可用，因为表由外部管理，不需要此功能。

为了在 MOT FDW 中支持两者，ValidateTableDef 函数实际上创建了指定的表。它还处理该关系的索引创建、DROP TABLE 和 DROP INDEX 以及先前在 FDW 中不支持的 VACUUM 和 ALTER TABLE。

索引规划与执行的使用方法

查询分为两个阶段：规划和执行。在规划阶段（可能在多次执行中才出现一次），选择扫描的最佳索引。该选择基于匹配查询的 WHERE 子句、JOIN 子句和 ORDER BY 条件。在执行期间，查询迭代相关的表行，并执行各种任务，如每次迭代的更新或删除。插入是一种特殊情况——表将行添加到所有索引中，且不需要扫描。

- 规划器：在标准 FDW 中，将查询传递给外部数据源执行。这意味着索引过滤和实际规划（例如索引的选择）不在数据库中本地执行，而是在外部数据源中执行。在内部，FDW 向数据库规划器返回总体计划。MOT 表的处理方式与磁盘表类似。这意味着相关的 MOT 索引得到过滤和匹配，最小化遍历行集的索引被选择并添加到计划中。
- 执行器：查询执行器使用所选的 MOT 索引来迭代表的相关行。每个行都由 GBase 8c 封装检查，根据查询条件调用 update 或 delete 处理相应的行。

持久性、复制性和高可用性

存储引擎负责存储、读取、更新和删除底层内存和存储系统中的数据。存储引擎不处理日志、检查点和恢复，特别是因为某些事务包含多个不同存储引擎的表。因此，为了数据持久化和复制，GBase 8c 封装使用如下高可用性设施：

- 持久性：MOT 引擎通过 WAL 记录使数据持久化，WAL 记录使用 GBase 8c 的 XLOG 接口。这为 GBase 8c 提供了使用相同 API 进行复制的好处。具体请参见 MOT 持久性概念。
- 检查点设定：通过向 GBase 8c Checkpointer 注册回调来启用 MOT 检查点每当执行通用数据库检查点时，MOT 检查点也被调用。MOT 保留了检查点的日志序列号（LSN），以便与 GBase 8c 恢复对齐。MOT Checkpointing 算法是高度优化的异步算法，不会停止并发事务。具体请参见 MOT 检查点概念。
- 恢复：启动时，GBase 8c 首先调用 MOT 回调，通过加载到内存行并创建索引来恢复 MOT 检查点，然后根据检查点的 LSN 重放记录来执行 WAL 恢复。MOT 检查点使用多线程并行恢复，每个线程读取不同的数据段。这使 MOT 检查点在多核硬件上的恢复速度相当快，尽管可能比仅重放 WAL 记录的基于磁盘的表慢一些。具体请参见 MOT 恢复概念。

VACUUM 和 DROP

为了最大化 MOT 功能，我们增加了对 VACUUM、DROP TABLE 和 DROP INDEX 的支持。这三个操作都使用排他表锁执行，这意味着不允许在表上并发事务。系统 VACUUM 调用一个新的 FDW 函数执行 MOT 真空，而 ValidateTableDef()函数中增加了 DROP。

删除内存池

每个索引和表都跟踪它使用的所有内存池。DROP INDEX 命令用于删除元数据。内存池作为单个连续块被删除。MOT VACUUM 只对已使用的内存进行压缩，因为内存回收由基于 epoch 的垃圾收集器（GC）在后台持续进行。为了执行压缩，我们将索引或表切换到新

的内存池，遍历所有实时数据，删除每行并使用新池插入数据，最后删除池，就像执行 DROP 那样。

12.3.4NUMA-aware 分配和亲和性

非统一内存访问（NUMA）是一种计算机内存设计，用于多重处理，其中内存访问时间取决于内存相对于处理器的位置。处理器可以利用 NUMA 的优势，优先访问本地内存（速度更快），而不是访问非本地内存（这意味着它不会访问另一个处理器的本地内存或处理器之间共享的内存）。

MOT 内存访问设计时采用了 NUMA 感知。即 MOT 意识到内存不是统一的，而是通过访问最快和最本地的内存来获得最佳性能。

NUMA 的优点仅限于某些类型的工作负载，特别是数据通常与某些任务或用户强相关的服务器上的工作负载。

在 NUMA 平台上运行的内储存数据库系统面临一些问题，例如访问远程主内存时，时延增加和带宽降低。为了应对这些 NUMA 相关问题，NUMA 感知必须被看作是数据库系统基本架构的主要设计原则。

为了便于快速操作和高效利用 NUMA 节点，MOT 为每个表的行分配一个指定的内存池，同时为索引的节点分配一个指定的内存池。每个内存池由多个 2MB 的块组成。指定 API 从本地 NUMA 节点、来自所有节点的页面或通过轮询分配这些块，每个块在下一个节点上分配。默认情况下，共享数据池以轮询方式分配，以保持访问均衡，同时避免在不同 NUMA 节点之间拆分行。但是，线程专用内存是从一个本地节点分配的，必须验证线程始终运行在同一个 NUMA 节点中。

总结

MOT 有一个智能内存控制模块，它预先为各种类型的内存对象分配了内存池。这种智能内存控制可以提高性能，减少锁并保证稳定性。事务的内存对象分配始终是 NUMA-local，从而保证了 CPU 内存访问的最佳性能，降低时延和争用。被释放的对象返回到内存池中。在事务期间最小化使用操作系统的 malloc 函数可以避免不必要的锁。

12.3.5MOT 索引

MOT 索引基于最先进的 Masstree 的免锁索引，用于多核系统的快速和可扩展的键值（KV）存储，通过 B+树的 Trie 实现。在多核服务器和高并发工作负载上，性能优异。它使用各种先进的技术，如乐观锁方法、缓存感知和内存预取。

Masstree 是 Trie 和 B+树的组合，用以谨慎利用缓存、预取、乐观导航和细粒度锁定。它针对高争用进行了优化，并对其前代产品增加了许多优化，如 OLFIT。然而，Masstree 索引的缺点是它的内存消耗更高。虽然行数据占用相同的内存大小，但每个索引（主索引或辅助索引）的每行内存平均高了 16 字节——基于磁盘的表使用基于锁的 B 树，大小为 29 字节，而 MOT 的 Masstree 大小为 45 字节。

实证研究表明，成熟的免锁 Masstree 实现与我们对 Silo 的强大改进相结合，恰能为我们解决这一方面的问题。

另一个挑战是对具有多个索引的表使用乐观插入。

Masstree 索引是用于数据和索引管理的 MOT 内存布局的核心。我们的团队增强并显著改进了 Masstree，同时提交了一些关键贡献给 Masstree 开源。这些改进包括：

- 每个索引都有专用内存池：高效分配和快速索引下移
- Masstree 全球 GC：快速按需内存回收
- 具有插入键访问的大众树迭代器实现
- ARM 架构支持

MOT 的主要创新是增强了原有的 Masstree 数据结构和算法，它不支持非唯一索引（作为二级索引）。设计细节请参见非唯一索引。

MOT 支持主索引、辅助索引和无键索引（受不支持的索引 DDL 和索引中提到的限制）。

非唯一索引

一个非唯一索引可以包含多个具有相同键的行。非唯一索引仅用于通过维护频繁使用的数据值的排序来提高查询性能。例如，数据库可能使用非唯一索引对来自同一家庭的所有人员进行分组。但是，Masstree 数据结构实现不允许将多个对象映射到同一个键。我们用于创建非唯一索引的解决方案（如下图所示）是为映射行的键添加一个打破对称的后缀。这个添加的后缀是指向行本身的指针，该行具有 8 个字节的常量大小，并且值对该行是唯一的。当插入到非唯一索引时，哨兵的插入总是成功的，这使执行事务分配的行能够被使用。这种方法还使 MOT 能够为非唯一索引提供一个快速、可靠、基于顺序的迭代器。

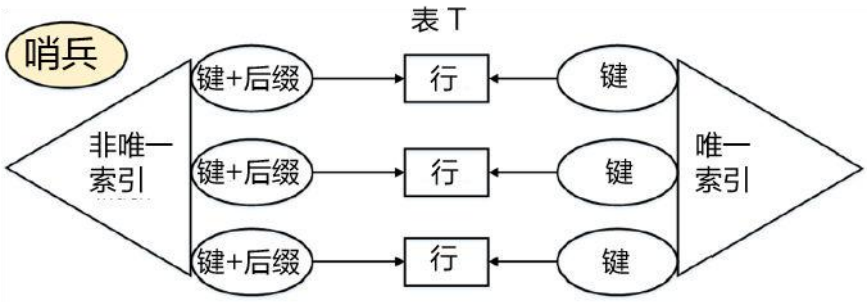


图 12-5 非唯一索引

上图描述了一个 MOT 的 T 表的结构，它有三个行和两个索引。矩形表示数据行，索引指向指向行的哨兵（椭圆形）。哨兵用键插入唯一索引，用键+后缀插入非唯一索引。哨兵可以方便维护操作，无需接触索引数据结构就可替换行。此外，在哨兵中嵌入了各种标志和参考计数，以便于乐观插入。

查找非唯一辅助索引时，会使用所需的键（如姓氏）。全串联键只用于插入和删除操作。插入和删除操作总是将行作为参数获取，从而可以创建整个键，并在执行删除或插入索引的特定行时使用它。

12.3.6MOT 持久性概念

持久性是指长期的数据保护（也称为磁盘持久性）。持久性意味着存储的数据不会遭受任何形式的退化或破坏，因此数据不会丢失或损坏。持久性可确保在有计划停机（例如维护）或计划外崩溃（例如电源故障）后数据和 MOT 引擎恢复到一致状态。

内存存储是易失的，需要电源来维护所存储的信息。另一方面，磁盘存储是非易失性的，这意味着它不需要电源来维护存储的信息，因此它不用担心停电。MOT 使用这两种类型的存储，它拥有内存中的所有数据，同时将事务性更改持久化到磁盘，并保持频繁的定期 MOT 检查点，以确保在关机时恢复数据。

用户必须保证有足够的磁盘空间用于日志记录和检查点操作。检查点使用单独的驱动器，通过减少磁盘 I/O 负载来提高性能。

有关如何在 MOT 引擎中实现持久性的概述，请参见 MOT 关键技术。

MOT 的 WAL 重做日志和检查点启用了持久性，如下所述。

MOT 日志记录：WAL 重做日志概念

概述

预写日志记录（WAL）是确保数据持久性的标准方法。WAL 的主要概念是，数据文件（表和索引所在的位置）的更改只有在记录这些更改之后才会写入，即只有在描述这些更改的日志记录被刷新到永久存储之后才会写入。

MOT 全面集成 GBase 8c 的封装日志记录设施。除持久性外，这种方法的另一个好处是能够将 WAL 用于复制目的。

支持三种日志记录方式：两种标准同步和一种异步方式。标准 GBase 8c 磁盘引擎也支持这三种日志记录方式。此外，在 MOT 中，组提交（Group-Commit）选项还提供了特殊的 NUMA 感知优化。Group-Commit 在维护 ACID 属性的同时提供最高性能。

为保证持久性，MOT 全面集成 GBase 8c 的 WAL 机制，通过 GBase 8c 的 XLOG 接口持久化 WAL 记录。这意味着，每次 MOT 记录的添加、更新和删除都记录在 WAL 中。这确保了可以从这个非易失性日志中重新生成和恢复最新的数据状态。例如，如果向表中添加了 3 个新行，删除了 2 个，更新了 1 个，那么日志中将记录 6 个条目。

- MOT 日志记录和 GBase 8c 磁盘表的其他记录写入同一个 WAL 中。
- MOT 只记录事务提交阶段的操作。
- MOT 只记录更新的增量记录，以便最小化写入磁盘的数据量。
- 在恢复期间，从最后一个已知或特定检查点加载数据；然后使用 WAL 重做日志完成从该点开始发生的数据更改。
- WAL 重做日志将保留所有表行修改，直到执行一个检查点为止（如上所述）。然后可以截断日志，以减少恢复时间和节省磁盘空间。

说明

➢ 为了确保日志 IO 设备不会成为瓶颈，日志文件必须放在具有低时延的驱动器上。

日志类型

支持两个同步事务日志选项和一个异步事务日志选项（标准 GBase 8c 磁盘引擎也支持这些选项）。MOT 还支持同步的组提交日志记录与 NUMA 感知优化，如下所述。

根据您的配置，实现以下类型的日志记录：

- 同步重做日志记录

同步重做日志记录选项是最简单、最严格的重做日志记录器。当客户端应用程序提交事务时，事务重做条目记录在 WAL 重做日志中，如下所示：

(1) 当事务正在进行时，它存储在 MOT 内存中。

(2) 事务完成后，客户端应用程序发送提交命令，该事务被锁定，然后写入磁盘上的 WAL 重做日志。这意味着，当事务日志条目写入日志时，客户端应用程序仍在等待响应。

(3) 一旦事务的整个缓冲区被写入日志，就更改内存中的数据，然后提交事务。事务提交后，客户端应用程序收到事务完成通知。

技术说明

当事务结束时，同步重做日志处理程序（SynchronousRedoLogHandler）序列化事务缓冲区，并写入 XLOG iLogger 实现。

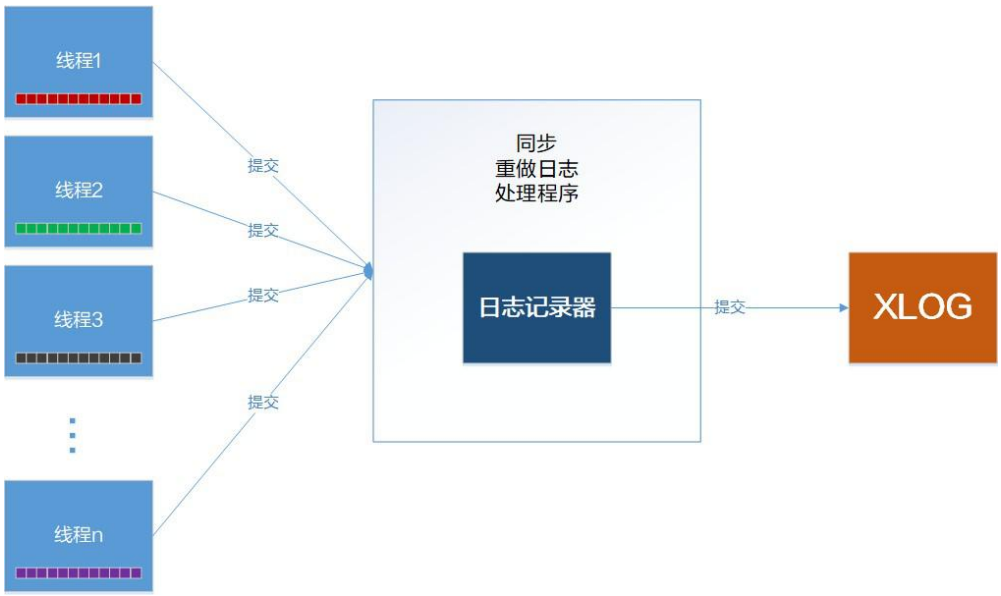


图 12-6 同步日志记录

总结

同步重做日志记录选项是最安全、最严格的，因为它确保了客户端应用程序和每个事务提交时的 WAL 重做日志条目的完全同步，从而确保了总的持久性和一致性，并且绝对不会丢失数据。此日志记录选项可防止客户端应用程序在事务尚未持久化到磁盘时将事务标记为成功的情况。

同步重做日志记录选项的缺点是，它是三个选项中最慢的日志机制。这是因为客户端应用程序必须等到所有数据都写入磁盘，并且磁盘频繁写入（这通常使数据库变慢）。

- 组同步重做日志记录

组同步重做日志记录选项与同步重做日志记录选项非常相似，因为它还确保完全持久性，

绝对不会丢失数据，并保证客户端应用程序和 WAL 重做日志条目的完全同步。不同的是，组同步重做日志记录选项将事务重做条目组同时写入磁盘上的 WAL 重做日志，而不是在提交时写入每个事务。使用组同步重做日志记录可以减少磁盘 I/O 数量，从而提高性能，特别是在运行繁重的工作负载时。

MOT 引擎通过根据运行事务的核的 NUMA 槽位自动对事务进行分组，使用非统一内存访问（NUMA）感知优化来执行同步的组提交记录。

有关 NUMA 感知内存访问的更多信息，请参见 NUMA-aware 分配和亲和性。

当一个事务提交时，一组条目记录在 WAL 重做日志中，如下所示：

(1) 当事务正在进行时，它存储在内存中。MOT 引擎根据运行事务的核的 NUMA 槽位对桶中的事务进行分组。这意味着在同一槽位上运行的所有事务都被分在一组，并且多个组将根据事务运行的核心并行填充。

这样，将事务写入 WAL 更为有效，因为来自同一个槽位的所有缓冲区都一起写入磁盘。

说明

每个线程在属于单个槽位的单核/CPU 上运行，每个线程只写运行于其上的核的槽位。

(2) 在事务完成并且客户端应用程序发送 Commit 命令之后，事务重做日志条目将与属于同一组的其他事务一起序列化。

(3) 当特定一组事务满足配置条件后，如重做日志（MOT）小节中描述的已提交的事务数或超时时间，该组中的事务将被写入磁盘的 WAL 中。这意味着，当这些日志条目被写入日志时，发出提交请求的客户端应用程序正在等待响应。

(4) 一旦 NUMA-aware 组中的所有事务缓冲区都写入日志，该组中的所有事务都将对内存存储执行必要的更改，并且通知客户端这些事务已完成。

技术说明

4 种颜色分别代表 4 个 NUMA 节点。因此，每个 NUMA 节点都有自己的内存日志，允许多个连接的组提交。

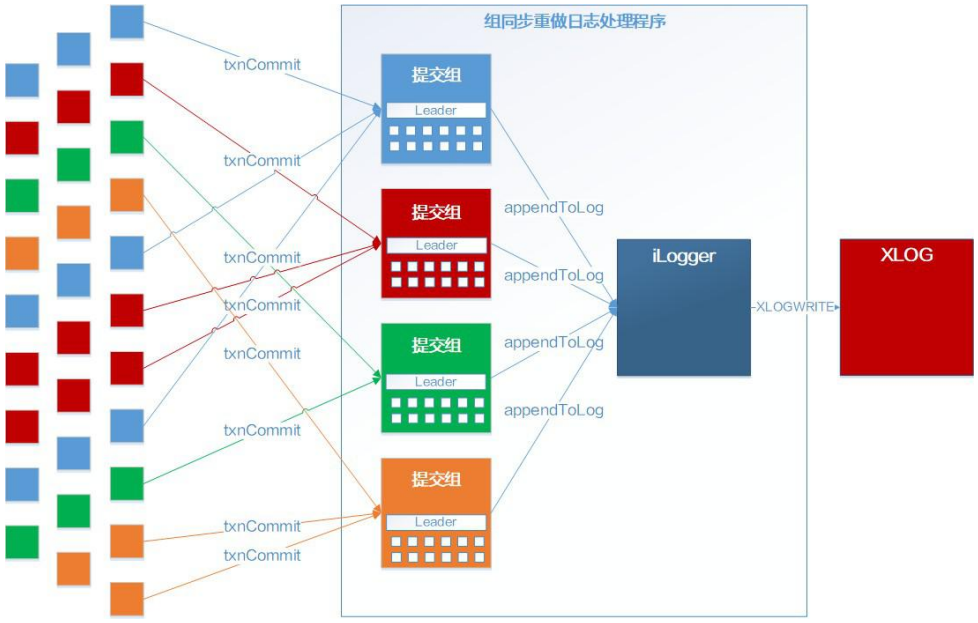


图 12-7 组提交——具有 NUMA 感知

总结

组同步重做日志记录选项是一个极其安全和严格的日志记录选项，因为它保证了客户端应用程序和 WAL 重做日志条目的完全同步，从而确保总的持久性和一致性，并且绝不会丢失数据。此日志记录选项可防止客户端应用程序在事务尚未持久化到磁盘时将事务标记为成功的情况。

一方面，该选项的磁盘写入次数比同步重做日志记录选项少，这可能意味着它更快。缺点是事务被锁定的时间更长，这意味着它们被锁定，直到同一 NUMA 内存中的所有事务都写入磁盘上的 WAL 重做日志为止。

使用此选项的好处取决于事务工作负载的类型。例如，此选项有利于事务多的系统（而对于事务少的系统而言，则较少使用，因为磁盘写入量也很少）。

- 异步重做日志记录

异步重做日志记录选项是最快的日志记录方法，但是，它不能确保数据不会丢失。也就是说，某些仍位于缓冲区且尚未写入磁盘的数据在电源故障或数据库崩溃时可能会丢失。当客户端应用程序提交事务时，事务重做条目将记录在内部缓冲区中，并按预先配置的时间间隔写入磁盘。客户端应用程序不会等待数据写入磁盘，而是继续到下一个事务。因此异步重做日志记录的速度最快。

当客户端应用程序提交事务时，事务重做条目记录在 WAL 重做日志中，如下所示：

(1) 当事务正在进行时，它存储在 MOT 内存中。

(2) 在事务完成并且客户端应用程序发送 Commit 命令后，事务重做条目将被写入内部缓冲区，但尚未写入磁盘。然后更改 MOT 数据内存，并通知客户端应用程序事务已提交。

(3) 后台运行的重做日志线程按预先配置的时间间隔收集所有缓存的重做日志条目，并将它们写入磁盘。

技术说明

在事务提交时，事务缓冲区被移到集中缓冲区（指针分配，而不是数据副本），并为事务分配一个新的事务缓冲区。一旦事务缓冲区移动到集中缓冲区，且事务线程不被阻塞，事务就会被释放。实际写入日志使用 Postgres WALWRITER 线程。当 WALWRITER 计时器到期时，它首先调用异步重做日志处理程序（通过注册的回调）来写缓冲区，然后继续其逻辑，并将数据刷新到 XLOG 中。

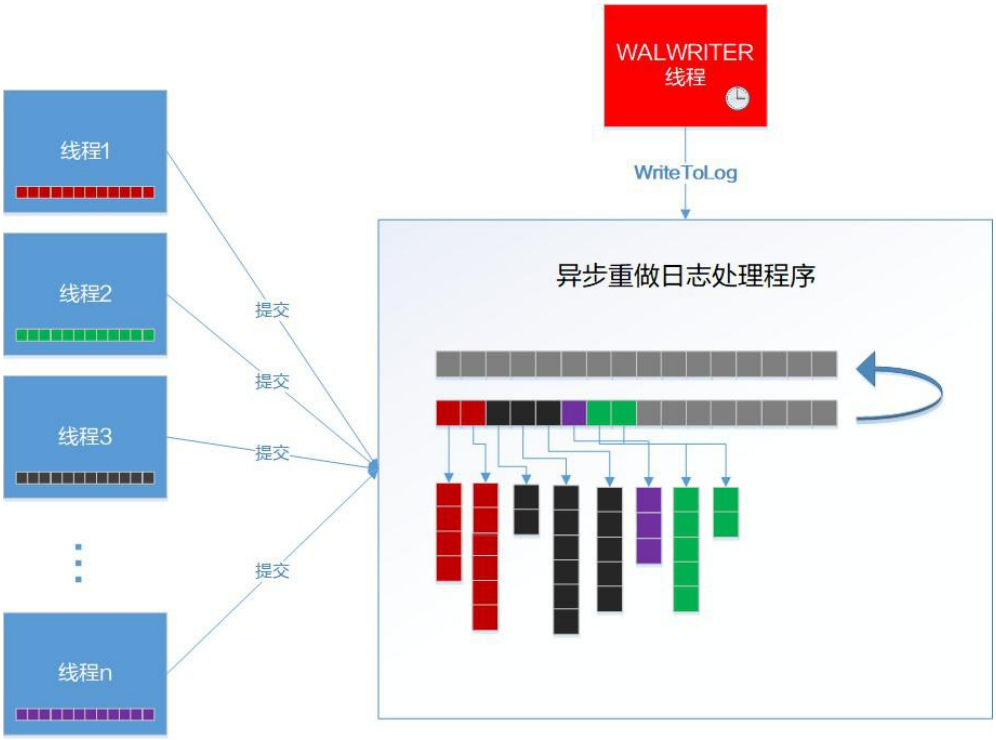


图 12-8 异步日志记录

总结

异步重做日志记录选项是最快的日志记录选项，因为它不需要客户端应用程序等待数据写入磁盘。此外，它将许多事务重做条目分组并把它们写入一起，从而减少降低 MOT 引擎速度的磁盘 I/O 数量。

异步重做日志记录选项的缺点是它不能确保在崩溃或失败时数据不会丢失。已提交但尚

未写入磁盘的数据在提交时是不持久的，因此在出现故障时无法恢复。异步重做日志记录选项对于愿意牺牲数据恢复（一致性）而不是性能的应用程序来说最为相关。

- 日志记录设计细节

下面将详细介绍内储存引擎模块中与持久化相关的各个组件的设计细节。

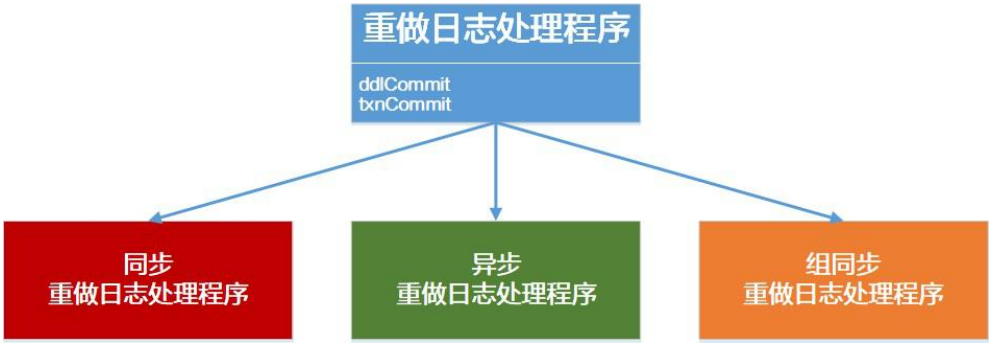


图 12-9 三种日志记录选项

重做日志组件由使用内储存引擎的后端线程和 WAL 编写器使用，以便持久化其数据。检查点通过 Checkpoint 管理器执行，由 Postgres 的 Checkpointer 触发。

- 日志记录设计概述

预写日志记录（WAL）是确保数据持久性的标准方法。WAL 的核心概念是，数据文件（表和索引所在的位置）的更改只有在记录了这些更改之后才会写入，这意味着在描述这些更改的日志记录被刷新到永久存储之后。

在内储存引擎中，我们使用现有的 GBase 8c 日志设施，并没有从头开始开发低级别的日志 API，以减少开发时间并使其可用于复制目的。

- 单事务日志记录

在内储存引擎中，事务日志记录存储在事务缓冲区中，事务缓冲区是事务对象（TXN）的一部分。在调用 addToLog()时记录事务缓冲区 -如果缓冲区超过阈值，则将其刷新并重新使用。当事务提交并通过验证阶段（OCC SILO[对比：磁盘与 MOT]验证）或由于某种原因中止时，相应的消息也会保存在日志中，以便能够在恢复期间确定事务的状态。

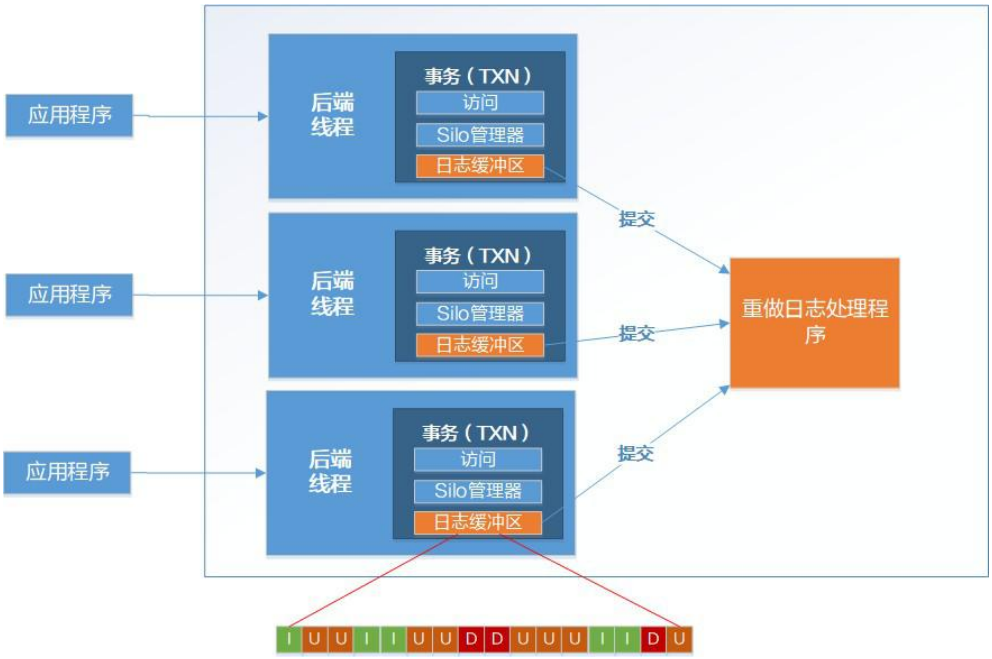


图 12-10 单事务日志记录

并行日志记录由 MOT 和磁盘引擎执行。但是，MOT 引擎通过每个事务的日志缓冲区、无锁准备和单个日志记录增强了这种设计。

- 异常处理

持久化模块通过 Postgres 错误报告基础设施（ereport）处理异常。系统日志中会记录每个错误情况的错误信息。此外，使用 Postgres 内置的错误报告基础设施将错误报告到封装。

该模块上报有如下异常：

表 12-7 异常处理

异常条件	异常码	场景描述	最终结果
WAL 写入失败	ERRCODE_FDW_ERROR	在任何情况下，WAL 写入失败	事务终止
文件 IO 错误：写入、打开等	ERRCODE_IO_ERROR	检查点：在任何文件访问错误时调用	严重：进程存在

内存不足	ERRCODE_INSUFFICIENT_RESOURCES	检查点：本地内存分配失败	严重：进程存在
逻辑、DB 错误	ERRCODEINTERNAL 错误	检查点：算法失败或无法检索表数据或索引	严重：进程存在

MOT 检查点概念

在 GBase 8c 数据库中，检查点是事务序列中一个点的快照，在该点上，可以保证堆和索引数据文件已经同步了检查点之前写入的所有信息。

在执行检查点时，所有脏数据页都会刷新到磁盘，并将一个特殊的检查点记录写入日志文件。

数据直接存储在内存中。MOT 没有像 GBase 8c 那样存储数据，因此不存在脏页的概念。为此，使用 CALC 算法，用于主内存数据库系统中的低开销异步检查点。

CALC 检查点算法：内存和计算开销低

检查点算法具有以下优点：

降低内存使用量-每条记录在任何时候最多存储两个副本。在记录处于活动且稳定版本相同或没有记录任何检查点时，仅存储记录的一个物理副本，可以最大限度地减少内存使用。

- 低开销：CALC 的开销比其他异步检查点算法小。
- 使用虚拟一致性点：CALC 不需要静默数据库以实现物理一致性点。

检查点激活

MOT 检查点被集成到 GBase 8c 的封装的检查点机制中。检查点流程可以通过执行 CHECKPOINT；命令手动触发，也可以根据封装的检查点触发设置（时间/大小）自动触发。

检查点配置在 mot.conf 文件中执行，请参见 MOT 配置的检查点（MOT）部分。

12.3.7MOT 恢复概念

MOT 恢复模块提供了恢复 MOT 表数据所需的所有功能。恢复模块的主要目标是在计划（例如维护）关闭或计划外（例如电源故障）崩溃后，将数据和 MOT 引擎恢复到一致的状态。

GBase 8c 数据库恢复（有时也称为冷启动）包括 MOT 表，并且随着数据库其余部分的恢复而自动执行。MOT 恢复模块无缝、全面地集成到 GBase 8c 恢复过程中。

MOT 恢复有两个主要阶段：检查点恢复和 WAL 恢复（重做日志）。

MOT 检查点恢复在封装的恢复发生之前执行。仅在冷启动事件（PG 进程的启动）中执行此操作。它首先恢复元数据，然后插入当前有效检查点的所有行，这由 checkpoint_recovery_workers 并行完成，每个行都在不同的表中工作。索引在插入过程中创建。

在检查点时，表被分成多个 16MB 的块，以便多个恢复工作进程可以并行地恢复表。这样做是为了加快检查点恢复速度，它被实现为一个多线程过程，其中每个线程负责恢复不同的段。不同段之间没有依赖关系，因此线程之间没有争用，在更新表或插入新行时也不需要使用锁。

WAL 记录作为封装的 WAL 恢复的一部分进行恢复。GBase 8c 封装会迭代 XLOG，根据 xlog 记录类型执行必要的操作。如果是记录类型为 MOT 的条目，封装将它转发给 MOT 恢复管理器进行处理。如果 XLOG 条目太旧（即 XLOG 条目的 LSN 比检查点的 LSN 旧），MOT 恢复将忽略该条目。

在主备部署中，备用服务器始终处于 Recovery 状态，以便自动 WAL 恢复过程。

MOT 恢复参数在 mot.conf 文件中配置，参见 MOT 恢复。

对比：磁盘与 MOT

下表简要对比了存储引擎和 MOT 存储引擎的各种特性。

表 12-8 对比：基于磁盘与 MOT

特性	GBase 8c 磁盘存储	GBase 8c MOT 引擎
Interl x86+鲲鹏 ARM	是	是
SQL 和功能集覆盖率	100%	98%
纵向扩容（多核，NUMA）	低效	高效
吞吐量	高	极高
时延	低	极低

分布式	是	是
隔离级别	RC+SI RR 序列化	RC RR RC+SI（V2 版本）
并发控制策略	悲观	乐观
数据容量（数据+索引）	不受限制	受限于 DRAM
复制、恢复	是	是
复制选项	2（同步，异步）	3（同步、异步、组提交）

其中，

- RR 表示可重复读取
- RC 表示读已提交
- SI 表示快照隔离

12.4 附录

表 12-9 术语表

缩略语	定义描述
2PL	2 阶段锁（2-Phase Locking）
ACID	原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）
AP	分析处理（Analytical Processing）
Arm	高级 RISC 机器（Advanced RISC Machine）、x86 的替代硬件架构
CC	并发控制（Concurrency Control）

CPU	中央处理器（Central Processing Unit）
DB	数据库（Database）
DBA	数据库管理员（Database Administrator）
DBMS	数据库管理系统（DataBase Management System）
DDL	数据定义语言（Data Definition Language）数据库模式管理语言
DML	数据修改语言（Data Modification Language）
ETL	提取、转换、加载或遇时锁定（Extract、Transform、 Load or Encounter Time Locking）
FDW	外部数据封装（Foreign Data Wrapper）
GC	垃圾收集器（Garbage Collector）
HA	高可用性（High Availability）
HTAP	事务分析混合处理（Hybrid Transactional-Analytical Processing）
IoT	物联网（Internet of Things）
IM	内储存（In-Memory）
IMDB	内储存数据库（In-Memory Database）
IR	源代码的中间表示（Intermediate Representation），用于编译和优化
JIT	准时（Just In Time）
JSON	JavaScript 对象表示法（JavaScript Object Notation）
KV	键值（Key Value）
LLVM	低级虚拟机（Low-Level Virtual Machine），指编译代码或 IR 查询
M2M	机对机（Machine-to-Machine）

ML	机器学习（Machine Learning）
MM	主内存（Main Memory）
MO	内存优化（Memory Optimized）
MOT	内存优化表存储引擎（SE），读作/em/ /oh/ /tee/
MVCC	多版本并发控制（Multi-Version Concurrency Control）
NUMA	非一致性内存访问（Non-Uniform Memory Access）
OCC	乐观并发控制（Optimistic Concurrency Control）
OLTP	在线事务处理（On-Line Transaction Processing），多用户在线交易类业务
PG	PostgreSQL
RAW	写后读校验（Reads-After-Writes）
RC	返回码（Return Code）
RTO	目标恢复时间（Recovery Time Objective）
SE	存储引擎（Storage Engine）
SQL	结构化查询语言（Structured Query Language）
TCO	总体拥有成本（Total Cost of Ownership）
TP	事务处理（Transactional Processing）
TPC-C	一种联机事务处理基准
Tpm-C	每分钟事务数-C. TPC-C 基准的性能指标，用于统计新订单事务
TVM	微小虚拟机（Tiny Virtual Machine）
TSO	分时选项（Time Sharing Option）

UDT	自定义类型
WAL	预写日志（Write Ahead Log）
XLOG	事务日志的 PostgreSQL 实现（WAL，如上文所述）

GBASE®

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微信二维码

技术支持热线：400-013-9696