ALIBABA CLOUD

阿里云

PolarDB-X (开源版)
从入门到实战

文档版本：20220707

阿里云

法律声明

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

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

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

目录

1.PolarDB-X数据库概述 ---- 06
1.1. PolarDB-X发展历史 ---- 06
1.2. PolarDB-X产品架构 ---- 08
1.3. PolarDB-X适用场景 ---- 10
1.4. 如何联系我们 ---- 10
2.一键安装部署PolarDB-X ---- 12
2.1. 使用Docker镜像安装部署PolarDB-X ---- 12
2.2. （可选）使用PXD工具一键安装PolarDB-X ---- 17
2.3. （可选）使用Kubernetes安装PolarDB-X ---- 20
2.4. （可选）使用源码编译安装PolarDB-X ---- 23
3.使用PolarDB-X开发应用 ---- 25
3.1. 安装PolarDB-X和JDK ---- 25
3.1.1. 第1步：安装PolarDB-X ---- 25
3.1.2. 第2步：登录PolarDB-X ---- 25
3.1.3. 第3步：安装JDK ---- 26
3.2. 体验Spring Boot+PolarDB-X应用开发 ---- 26
3.3. 体验WordPress+PolarDB-X部署博客站点 ---- 33
3.4. PolarDB-X应用开发最佳实践 ---- 38
3.4.1. 如何选择应用端连接池 ---- 39
3.4.2. 透明分布式最佳实践 ---- 42
4.数据导入与导出 ---- 54
4.1. PolarDB-X CDC ---- 54
4.2. Global Binlog ---- 54
4.3. PolarDB-X Replica ---- 61
4.4. 相关资料 ---- 62
5.PolarDB-X集群运维 ---- 64

5.1. 扩容和缩容 64

5.1.1. 第1步：安装环境 66

5.1.2. 第2步：使用PolarDB-X Operator安装PolarDB-X 67

5.1.3. 第3步：体验PolarDB-X集群扩容 70

5.1.4. 第4步：体验PolarDB-X集群缩容 72

5.2. 升配和降配 74

5.3. 备份恢复 77

5.4. 数据库监控 79

5.5. SQL限流和SQL Advisor 80

5.5.1. 第1步：连接PolarDB-X集群 84

5.5.2. 第2步：启动业务 85

5.5.3. 第3步：体验SQL限流和SQL Advisor 90

6.分布式事务与数据分区 93

6.1. 分布式事务 93

6.2. 数据分区 97

7.X-Paxos三副本与高可用 102

7.1. DN高可用方案 102

7.2. X-Paxos协议 103

7.3. DN高可用体系 105

7.4. DN部署和优化 109

7.4.1. 第1步：安装环境 110

7.4.2. 第2步：使用PolarDB-X Operator安装PolarDB-X 111

7.4.3. 第3步：连接PolarDB-X集群 115

7.4.4. 第4步：启动业务 116

7.4.5. 第5步：体验PolarDB-X高可用能力 120

8.常见问题解答 127

1.PolarDB-X数据库概述

PolarDB-X是一款面向超高并发、海量存储、复杂查询场景设计的云原生分布式数据库系统。其采用Shared-Nothing与存储计算分离架构，支持水平扩展、分布式事务、混合负载等能力，具备企业级、云原生、高可用、高度兼容MySQL系统及生态等特点。

PolarDB-X最初为解决阿里巴巴天猫“双十一”核心交易系统数据库扩展性瓶颈而生，之后伴随阿里云一路成长，是一款经过多种核心业务场景验证的、成熟稳定的数据库系统。PolarDB-X的核心特性如下：

- **水平扩展**

  PolarDB-X采用Shared-Nothing架构进行设计，支持多种Hash和Range数据拆分算法，通过隐式主键拆分和数据分片动态调度，实现系统的透明水平扩展。

- **分布式事务**

  PolarDB-X采用MVCC+TSO方案及2PC协议实现分布式事务。事务满足ACID特性，支持RC/RR隔离级别，并通过一阶段提交、只读事务、异步提交等优化实现事务的高性能。

- **混合负载HTAP**

  PolarDB-X通过原生MPP能力实现对分析型查询的支持，通过CPU quota约束、内存池化、存储资源分离等实现了OLTP与OLAP流量的强隔离，通过将数据库中的冷数据归档到OSS或其他低成本存储来降低后台系统的数据存储成本。

- **企业级**

  PolarDB-X为企业场景设计了诸多内核能力，例如SQL限流、SQL Advisor、TDE、三权分立、Flashback Query等。

- **云原生**

  PolarDB-X在阿里云上有多年的云原生实践，支持通过K8S Operator管理集群资源，支持公有云、混合云、专有云等多种形态进行部署，并支持国产化操作系统和芯片。

- **高可用**

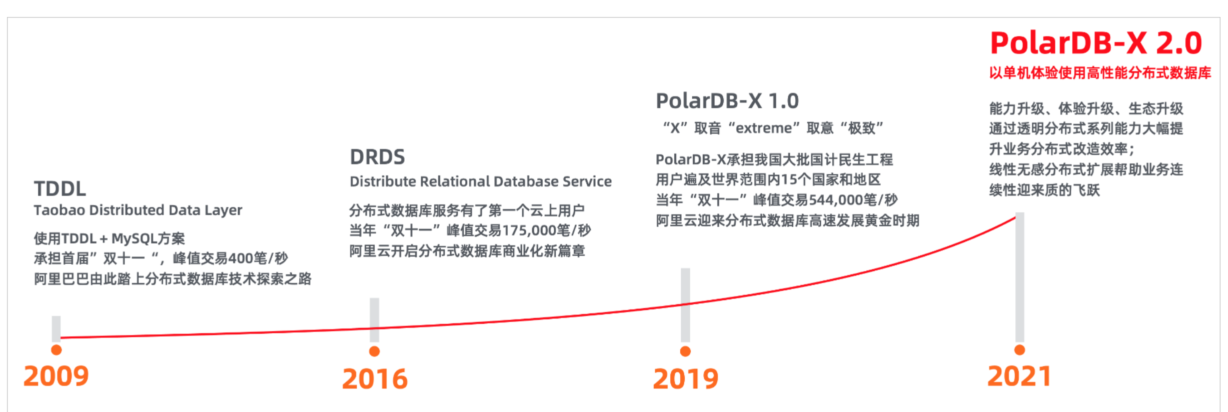
  通过多数派Paxos协议实现数据强一致，支持两地三中心、三地五副本等多种容灾方式，同时通过Table Group、Geo-locality等提高系统可用性。

- **兼容MySQL系统及生态**

  PolarDB-X的目标是完全兼容MySQL，目前兼容的内容包括MySQL协议、MySQL大部分语法、Collation、事务隔离级别、Binlog、Replication等。

1.1. PolarDB-X发展历史

2009年，阿里巴巴在进行年度收支核算时，发现业务对数据存取资源的需求呈指数级扩张，并与业务指标增长之间产生鸿沟。此时的阿里巴巴清晰的意识到，如果不通过技术创新平衡成本和增长需求，必将对未来发展产生影响。于是"去IOE"的说法被首次提出。"去IOE"的目的绝不仅仅是摆脱几个传统IT厂商的产品，而是在互联网+、云计算快速兴起后，企业用户迫切需要能够保证其业务发展和创新的更加开放、灵活、高效的IT基础架构。概括地说，"去IOE"推动了用横向扩展取代纵向扩展，用开源软件代替商业软件的进程，可以看作是云计算的奠基之举。其中对传统商业数据库的替换又是"去IOE"的核心。



TDDL阶段

关键字：阿里巴巴大规模应用；分库分表技术开创者

去掉传统商业数据库后，是否有更适合的产品和解决方案来替代呢？对于数据库来说，答案是明确的：开源+分布式，开源解决成本问题，分布式解决性能和容量问题。

同年11月11日，TDDL（Taobao Distributed Data Layer）首次发布，开创了分布式数据库中间件+开源数据库应用在高并发交易系统的先河。

当时的TDDL虽然是一个客户端jar，但创造性地提出了三层（Matrix、Group、Atom）拆分拓扑结构，满足应用按需制定拆分策略的同时，解决了弹性扩容、本地高可用等企业应用难题。

2011~2015，TDDL成为阿里巴巴数据库系统的统一接入标准，开始面向阿里巴巴所有业务提供分布式数据库服务。目前集团内运行实例约30万套，业务覆盖支付、资金、即时通信、媒体等十余大类。

丰富的业务模型造就了TDDL优秀的MySQL语法兼容性，庞大的业务规模使TDDL打磨出优异的内核稳定性，历年双十一的加持孵化了TDDL业界顶尖的高性能高吞吐。

与此同时，阿里巴巴分布式数据库的商业化进程悄然启动。

DRDS阶段

关键字：云端商业化；高性能SQL引擎

DRDS（Distributed Relational Database Service）于2016年初迎来了第一个公有云付费客户。自此，DRDS一直在不断努力提升单位资源的处理能力，以求最大限度帮助客户降本增效。

DRDS研发团队于2017年发布的新一代高性能分布式SQL引擎，通过PlanCache、FastSQL、定制化的底层驱动使Batch写、含拆分键Select、读写分离等操作具有300%的性能提升；跨库聚合、分布式Join、分布式事务等操作具有200%性能提升。

DRDS提供更低使用成本，包括对不同维度的表的Join操作的支持、内存中二次排序的支持和对内存结果做函数计算的支持等。

DRDS还针对分布式数据库使用场景提供一系列的企业级特性，包括全局Sequence服务、读写分离、数据库账号体系和DRDS后台运维指令集。

凭借优异性能和相对优良的体验，DRDS迅速在公有云积累了一批忠实用户。

DRDS的商业化成功，标志着阿里巴巴分布式数据库技术完成了从内部孵化到市场化运营的阶段性转变，以及从分布式数据库中间件到分布式数据库系统实质性跨越。

PolarDB-X 1.0阶段

关键字：架构与品牌升级；国计民生项目

2018~2019年，DRDS凭借优异稳定性、超高性能以及丰富的企业特性，承接众多政企行业的国计民生项目，积极投入我国信息系统基础设施数字化转型建设，品牌声誉得到大幅提升，逐步成长为代表阿里巴巴的名片级产品。

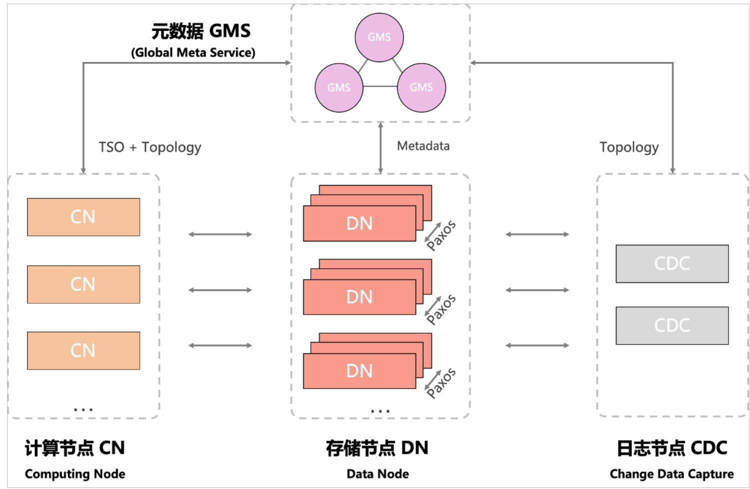
与此同时，DRDS进行品牌升级，命名为PolarDB-X，"PolarDB"是阿里云自研关系型数据库产品家族名称，"X"取音"Extreme"，取意"极致"。

PolarDB-X 1.0支持以PolarDB MySQL作为存储节点，大幅提高集群IO能力以及柔性分布式事务，且面向政企客户需求增强了安全特性，例如，一致性备份恢复、SQL闪回、SQL审计等。

PolarDB-X 2.0阶段

关键字：透明分布式、开源

PolarDB-X 2.0是阿里巴巴分布式数据库有史以来最大幅度的版本更新。产品基于透明分布式理念提供了默认主键拆分策略、基于TSO和MVCC的高性能强一致分布式事务、基于一致性Hash分区策略的分布式线性扩展能力、全局一致性Binlog和全局一致性备份能力。数据节点(DN)采用阿里巴巴自研的基于X-Paxos的三副本强一致MySQL分支，确保在容灾过程中RPO=0。



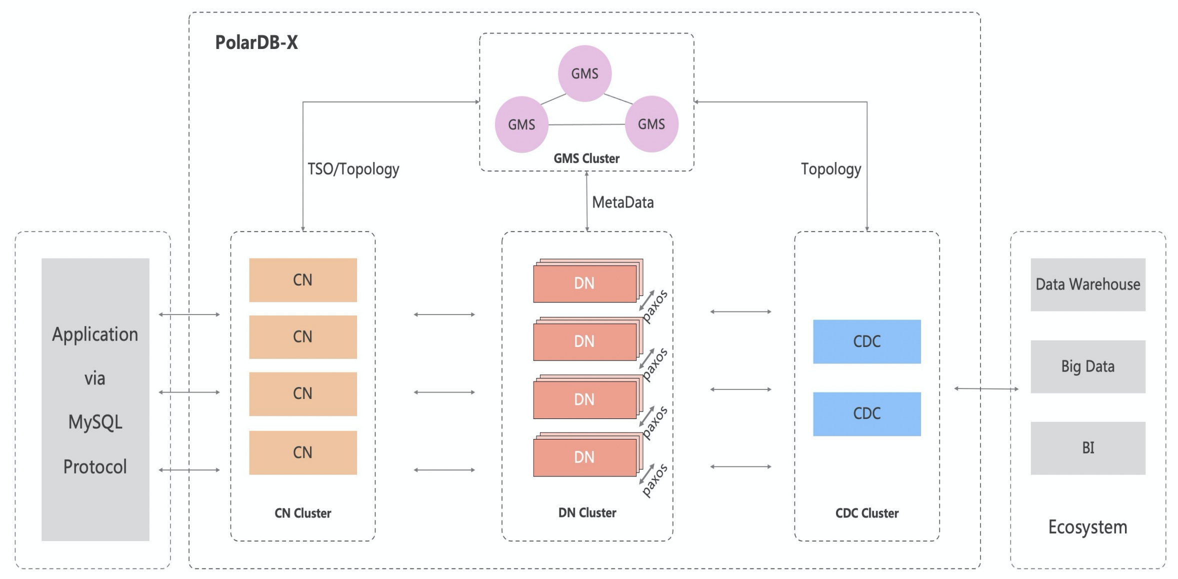
目前PolarDB-X 2.0已经采用全新架构在阿里云世界范围内的13个国家和地区提供服务。并提供公有云、专有云、软件版和DBStack多种部署形态满足不同行业客户的需求。

PolarDB-X 2.0已全面拥抱开源，毫无保留地和全世界的MySQL社区共同成长，将高性能分布式数据库质量进一步提升，使用门槛尽可能降低。

未来，国产化、HTAP混合负载、快速容量变更、多区域容灾多活等均是PolarDB-X 2.0的探索方向。

1.2. PolarDB-X产品架构

PolarDB-X 采用 Shared-nothing 与存储计算分离架构进行设计，系统由4个核心组件组成。



- **计算节点（CN, Compute Node）**

  计算节点是系统的入口，采用无状态设计（目前最多可扩展至1024个节点），包括 SQL 解析器、优化器、执行器等模块。负责数据分布式路由、计算及动态调度，负责分布式事务 2PC 协调、全局二级索引维护等，同时提供 SQL 限流、三权分立等企业级特性。

- **存储节点（DN, Data Node）**

  存储节点负责数据的持久化，基于多数派 Paxos 协议提供数据高可靠、强一致保障，同时通过 MVCC 维护分布式事务可见性。与CN相同，DN目前最多支持扩展至1024个节点。

- **元数据服务（GMS, Global Meta Service）**

  元数据服务负责维护全局强一致的 Table/Schema, Statistics 等系统 Meta 信息，维护账号、权限等安全信息，同时提供全局授时服务（即 TSO）。

- **日志节点（CDC, Change Data Capture）**

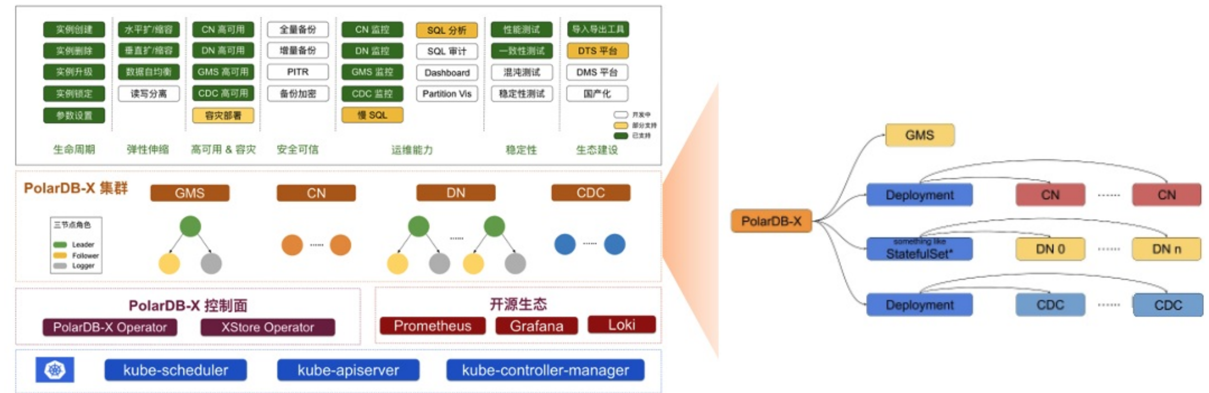
  日志节点提供完全兼容 MySQL Binlog 格式和协议的增量订阅能力，提供兼容 MySQL Replication 协议的主从复制能力。

PolarDB-X 提供通过 K8S Operator 方式管理以上4个组件，同时计算节点与存储节点之间可通过私有协议进行 RPC 通信，这些组件对应的仓库如下在：

组件名称	GitHub地址
计算节点（CN, Compute Node）	galaxysql
元数据服务（GMS, Global Meta Service）	galaxyengine
存储节点（DN, Data Node）	galaxyengine
日志节点（CDC, Change Data Capture）	galaxycdc
私有协议	galaxyglue
K8S Operator	galaxykube

PolarDB-X积极拥抱Kubernetes生态，在Kubernetes上进行扩展，提供了PolarDB-X Operator来支持PolarDB-X集群的相关工作。例如，我们基于Kubernetes的scheduler、apiserver和controller manager，构建了PolarDB-X自己的控制面，在此基础上，让我们能够对PolarDB-X集群进行部署运维以及相关的一些生命周期产品能力管理；采用Deployment来管理无状态的CN和CDC两个组件；采用类似StatefulSet的方式来管理采用了X-Paxos的DN节点。

除了部署、运维等基本的生命周期能力外，PolarDB-X还提供了弹性伸缩、高可用、监控审计、备份恢复等能力。



1.3. PolarDB-X适用场景

PolarDB-X在以下业务场景中，能够提供

金融级高可靠场景

基于X-Paxos实现的数据三副本强一致，让业务轻松具备跨多可用区的高可用与容灾能力。

海量数据归集场景

PolarDB-X的平滑扩展能力，让客户轻松实现横向和纵向的容量变更且保持访问性能不变。

超高并发访问场景

深度优化的PolarDB-X计算下推能力显著提升在线事务处理性能，令访问尖峰丝般顺滑。

HTAP混合负载场景

无需进行ETL，可对在线数据做实时报表分析。

1.4. 如何联系我们

欢迎广大PolarDB-X数据库的客户、用户和爱好者联系我们，反馈您在使用PolarDB-X过程中的任何问题和建议：

- 公共云版官网：https://www.aliyun.com/product/drds
- 开源版官网：https://www.polardbx.com
- 开源版项目：https://github.com/ApsaraDB/galaxysql
- 钉钉群：
  - 群号：32432897

- 二维码：

2.一键安装部署PolarDB-X

本节将介绍基于一台配置了CentOS 8.5操作系统的ECS实例（云服务器），通过Docker镜像部署PolarDB-X的安装部署方法，并体验PolarDB-X数据库的分布式特性。此外，本节还将介绍使用PXD工具一键安装PolarDB-X、使用Kubernetes安装PolarDB-X和使用源码编译安装PolarDB-X多种不同的安装编译方法。

> **说明**
>
> 本节实验操作部分主要通过阿里云官方网站的云起实验室进行，详情可登录阿里云官方网站，访问如何一键安装部署PolarDB-X了解。

2.1. 使用Docker镜像安装部署PolarDB-X

本节将介绍基于一台配置了CentOS 8.5操作系统的ECS实例（云服务器），通过Docker镜像部署PolarDB-X的安装部署方法，并体验PolarDB-X数据库的分布式特性。

背景信息

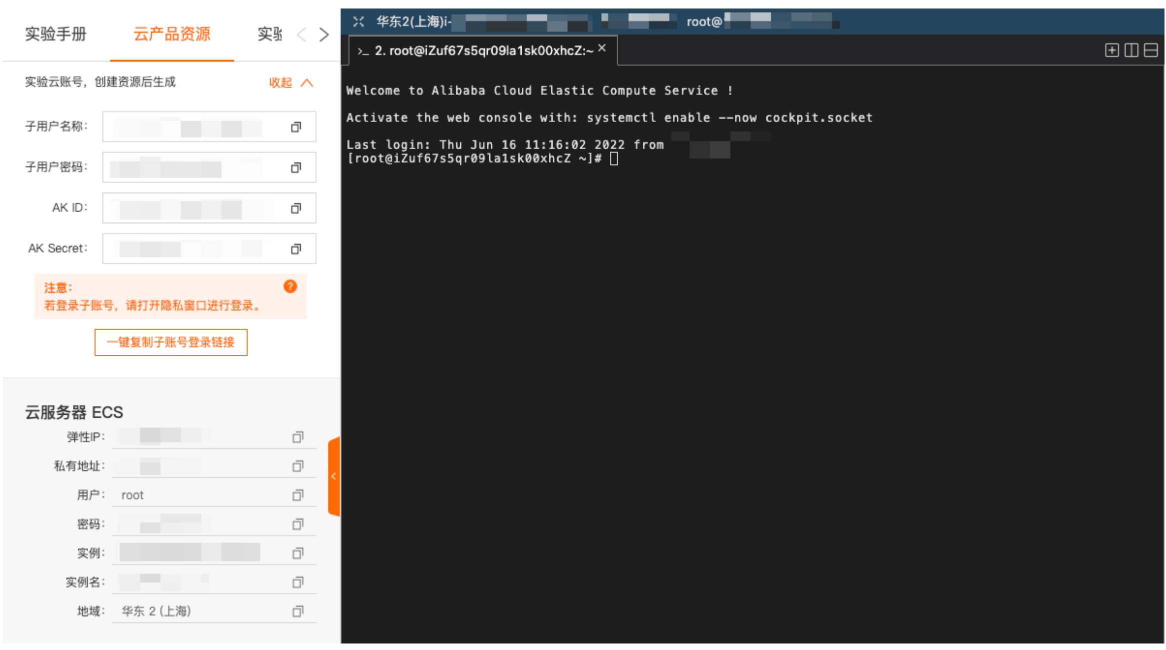
PolarDB-X支持通过MySQL Client命令行、第三方客户端以及符合MySQL交互协议的第三方程序代码进行连接。

操作步骤

1. 创建ECS实例资源。
    i. 在实验室页面，单击**创建资源**。

    > **说明**
    >
    > 资源创建过程需要1~3分钟。

    ii. （可选）在实验室页面左侧导航栏中，单击**云产品资源列表**，可查看本次实验资源相关信息（例如IP地址、用户信息等）。



2. 安装并启动依赖Docker。

   i. 执行如下命令，安装Docker。

```
curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun
```

   ii. 执行如下命令，启动Docker。

```
systemctl start docker
```

3. 使用Docker镜像安装PolarDB-X。

   i. 执行如下命令，拉取PolarDB-X容器镜像。

```
docker pull polardbx/polardb-x
```

   ii. 执行如下命令，运行PolarDB-X容器。

```
docker run -d --name some-polardb-x -p 8527:8527 polardbx/polardb-x
```

4. 通过MySQL Client命令行连接到PolarDB-X数据库，以登录PolarDB-X数据库。

   i. 执行如下命令，安装MySQL。

```
yum install mysql -y
```

   ii. （可选）执行如下命令，查看MySQL版本号。

```
mysql -V
```

   返回结果如下，表示您已成功安装MySQL。

```
[root@                    ~]# mysql -V
mysql  Ver 8.0.26 for Linux on x86_64 (Source distribution)
```

iii. 执行如下命令，登录PolarDB-X数据库。

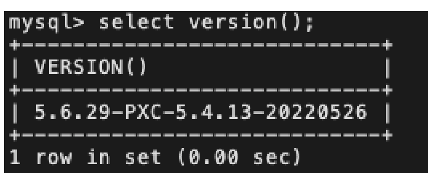
```
mysql -h127.0.0.1 -P8527 -upolardbx_root -p123456
```

> **说明**
>
> - 本实验场景中的PolarDB-X数据库用户名和密码已预设，请您使用上方的命令登录即可。
> - 如遇到报错
>
> ```
> mysql: [Warning] Using a password on the command line interface can be insecure. ERROR 2013 (HY000): Lost connection to MySQL server at 'reading initial communication packet', system error: 0
> ```
>
> ，重新执行登录命令即可。

iv. （可选）执行如下命令，查看连接到的数据库的版本。

```
select version();
```

可以看到当前连接到的数据库的版本（5.6.29）与mysql -V查询到的MySQL版本（8.0.26）并不一致，说明此处连接到的是PolarDB-X的数据库实例，而非连接到MySQL数据库。



v. （可选）执行如下SQL语句，检查GMS。

```
select * from information_schema.schemata;
```

```
mysql> select * from information_schema.schemata;
+--------------+--------------------+----------------------------+------------------------+----------+--------------------+
| CATALOG_NAME | SCHEMA_NAME        | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME | SQL_PATH | DEFAULT_ENCRYPTION |
+--------------+--------------------+----------------------------+------------------------+----------+--------------------+
| def          | information_schema | utf8                       | UTF8_GENERAL_CI        | NULL     | NO                 |
| def          | __cdc__            | utf8                       | UTF8_GENERAL_CI        | NULL     | NO                 |
+--------------+--------------------+----------------------------+------------------------+----------+--------------------+
2 rows in set (0.07 sec)
```

5. （可选）体验PolarDB-X的分布式特性。

i. 执行如下SQL语句，创建数据库。

```
create database polarx_example mode='auto';
```

> **说明**
>
> 数据库模式（参数mode）目前有两种，一种是partition，一种是auto。设置mode='auto'，表示“透明”，亦即PolarDB-X会透明、自动地处理好与分布式相关的一些工作。

ii. 执行如下SQL语句，使用polarx_example数据库。

```
use polarx_example;
```

iii. 执行如下SQL语句，创建数据表。

```
create table example (
  `id` bigint(11) auto_increment NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `score` bigint(11) DEFAULT NULL,
  primary key (`id`)
) engine=InnoDB default charset=utf8
partition by hash(id)
partitions 8;
```

说明

如上命令显式指明了用于拆分的列以及拆分的分片数（根据id列使用Hash拆分为8个分片）。

为证明在auto模式下，我们可以不必关心具体分片的方法（分片方式对用户透明），可以将建表语句中的partition部分去掉，再创建一张表（假定名为example2），来进行对比。

```
create table example2 (
  `id` bigint(11) auto_increment NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `score` bigint(11) DEFAULT NULL,
  primary key (`id`)
) engine=InnoDB default charset=utf8;
```

iv. 执行如下SQL语句，向example数据表中插入数据。

```
insert into example values(null,'lily',375),(null,'lisa',400),(null,'ljh',500);
```

v. 执行如下SQL语句，查询example表所有数据。

```
select * from example;
```

```
mysql> select * from example;
+--------+------+-------+
| id     | name | score |
+--------+------+-------+
| 100001 | lily |   375 |
| 100002 | lisa |   400 |
| 100003 | ljh  |   500 |
+--------+------+-------+
3 rows in set (0.04 sec)
```

vi. 执行如下SQL语句，查看example和example2两个数据表的分区。

```
show topology from example;
```

```
show topology from example2;
```

说明

在单机部署数据库时，数据表表一定是本地存储的。但在分布式部署数据库时，数据表可能会分成若干个分片。使用

```
show topology
```

命令可以展示一张数据表里的分片情况。

返回结果如下，您可以看到example数据表分布在8个分区。

```
mysql> show topology from example;
+----+------------------------------+--------------------+----------------+
| ID | GROUP_NAME                   | TABLE_NAME         | PARTITION_NAME |
+----+------------------------------+--------------------+----------------+
|  0 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00000 | p1             |
|  1 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00001 | p2             |
|  2 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00002 | p3             |
|  3 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00003 | p4             |
|  4 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00004 | p5             |
|  5 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00005 | p6             |
|  6 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00006 | p7             |
|  7 | POLARX_EXAMPLE_P00000_GROUP | example_fjUe_00007 | p8             |
+----+------------------------------+--------------------+----------------+
8 rows in set (0.00 sec)
```

example2数据表同样分布在8个分区。

```
mysql> show topology from example2;
+----+------------------------------+---------------------+----------------+
| ID | GROUP_NAME                   | TABLE_NAME          | PARTITION_NAME |
+----+------------------------------+---------------------+----------------+
|  0 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00000 | p1             |
|  1 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00001 | p2             |
|  2 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00002 | p3             |
|  3 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00003 | p4             |
|  4 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00004 | p5             |
|  5 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00005 | p6             |
|  6 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00006 | p7             |
|  7 | POLARX_EXAMPLE_P00000_GROUP | example2_c8tt_00007 | p8             |
+----+------------------------------+---------------------+----------------+
8 rows in set (0.00 sec)
```

可以看到，在不显式指定分片方式的情况下，PolarDB-X会自动处理与分布式相关的工作，降低学习和使用的门槛。

6. （可选）执行如下SQL语句，检查CDC。

```
show master status;  # 查看当前节点状态（Binlog记录到的位置）
show binlog events  in 'binlog.000001' from 4; # 查看Binglog中的事件
```

```
mysql> show master status;
+----------------+----------+--------------+------------------+-------------------+
| FILE           | POSITION | BINLOG_DO_DB | BINLOG_IGNORE_DB | EXECUTED_GTID_SET |
+----------------+----------+--------------+------------------+-------------------+
| binlog.000001  |    72777 |              |                  |                   |
+----------------+----------+--------------+------------------+-------------------+
1 row in set (0.35 sec)

mysql> show binlog events  in 'binlog.000001' from 4;
| LOG_NAME       | POS    | EVENT_TYPE   | SERVER_ID | END_LOG_POS | INFO

| binlog.000001 |      4 | Format_desc | 193317851 |         123 | Server ver: 5.6.29-PXC-5.4.13-2022
0526, Binlog ver: 4

| binlog.000001 |    123 | Rows_query  | 193317851 |         206 | CTS::69433886248345272961474453840
```

7. （可选）执行如下SQL语句，检查DN和CN。

```
show storage;
show mpp;
```

```
mysql> show storage;
+------------------+----------------+------------+-----------+----------+-------------+--------+-----------+-------+--------+
| STORAGE_INST_ID  | LEADER_NODE    | IS_HEALTHY | INST_KIND | DB_COUNT | GROUP_COUNT | STATUS | DELETABLE | DELAY | ACTIVE |
+------------------+----------------+------------+-----------+----------+-------------+--------+-----------+-------+--------+
| polardbx_dn_0    | 127.0.0.1:4886 | true       | MASTER    | 2        | 3           | 0      | false     | null  | null   |
| polardbx_meta    | 127.0.0.1:4886 | true       | META_DB   | 2        | 2           | 0      | false     | null  | null   |
+------------------+----------------+------------+-----------+----------+-------------+--------+-----------+-------+--------+
2 rows in set (0.01 sec)
```

```
mysql> show mpp;
+------------------+-----------------+------+--------+
| ID               | NODE            | ROLE | LEADER |
+------------------+-----------------+------+--------+
| polardbx-polardbx | 172.17.0.2:9090 | W    | Y      |
+------------------+-----------------+------+--------+
1 row in set (0.00 sec)
```

8. 输入exit退出数据库。

```
mysql> exit
Bye
```

9. 执行如下命令，停止PolarDB-X容器。

```
docker stop some-polardb-x
```

2.2. （可选）使用PXD工具一键安装PolarDB-X

PXD是PolarDB-X的部署工具，除了支持在本地一键快速拉起测试环境外，也支持在Linux集群中通过指定的拓扑的方式部署PolarDB-X分布式数据库。本节将介绍如何使用PXD工具一键安装PolarDB-X。

前提条件

- 确保已安装并启动Docker。
- 确保已安装MySQL Client。

操作步骤

> **说明**
> - 本节的实验场景使用virtual environment安装PXD工具。
> - 有关使用PXD工具一键安装PolarDB-X的详情，请参见通过PXD部署集群

1. 安装PXD。
   i. 执行如下命令，创建并激活虚拟场景。

   ```
   python3 -m venv venv
   source venv/bin/activate
   ```

   ii. 执行如下命令，升级pip。

   ```
   pip install --upgrade pip
   ```

   iii. 执行如下命令，安装PXD。

   ```
   pip install pxd
   ```

2. 部署PolarDB-X。
   - 执行如下命令，创建一个PolarDB-X数据库，其中GMS、CN、DN和CDC节点个数为1。

   ```
   pxd tryout
   ```

   - 执行如下命令，创建一个PolarDB-X数据库，指定CN、DN和CDC节点个数为1以及版本为latest。

   ```
   pxd tryout -cn_replica 1 -cn_version latest -dn_replica 1 -dn_version latest -cdc_rep
   lica 1 -cdc_version latest
   ```

   返回结果如下，表示您已成功部署PolarDB-X数据库，您可以看到输出的连接信息，通过MySQL Client即可连接。

   ```
   Status: Downloaded newer image for polardbx/xstore-tools:latest
   Processing  [#########---------------------------]   25%    create gms node
   Processing  [############------------------------]   33%    create gms db and tables
   Processing  [###############---------------------]   41%    create PolarDB-X root account
   Processing  [##################------------------]   50%    create dn
   Processing  [#####################---------------]   58%    register dn to gms
   Processing  [########################------------]   66%    create cn
   Processing  [###########################---------]   75%    wait cn ready
   Processing  [##############################------]   83%    create cdc containers
   Processing  [#################################---]   91%    wait PolarDB-X ready
   Processing  [####################################]  100%


   PolarDB-X cluster create successfully, you can try it out now.
   Connect PolarDB-X using the following command:

     mysql -h127.0.0.1 -P9069 -upola[illegible] root -phPv[illegible]wv
   ```

注意

PolarDB-X管理员账号的密码随机生成，仅出现这一次，请注意保存。

3. 执行如下命令，登录PolarDB-X数据库。

```
mysql -h127.0.0.1 -P8527 -upolardbx_root -p123456 # 请将账号密码替换为上一步中生成的管理员账号和密码
```

4. 体验PolarDB-X。

   i. 执行如下SQL语句，检查GMS。

```
select * from information_schema.schemata;
```

```
mysql> select * from information_schema.schemata;
+--------------+--------------------+----------------------------+------------------------+----------+----------------------+
| CATALOG_NAME | SCHEMA_NAME        | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME | SQL_PATH | DEFAULT_ENCRYPTION |
+--------------+--------------------+----------------------------+------------------------+----------+----------------------+
| def          | information_schema | utf8                       | UTF8_GENERAL_CI        | NULL     | NO                 |
| def          | __cdc__            | utf8                       | UTF8_GENERAL_CI        | NULL     | NO                 |
+--------------+--------------------+----------------------------+------------------------+----------+----------------------+
2 rows in set (0.07 sec)
```

   ii. 执行如下SQL语句，创建数据库。

```
create database polarx_example mode='auto';
```

   iii. 执行如下SQL语句，使用polarx_example数据库。

```
use polarx_example;
```

   iv. 执行如下SQL语句，创建数据表。

```
create table example (
  `id` bigint(11) auto_increment NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `score` bigint(11) DEFAULT NULL,
  primary key (`id`)
) engine=InnoDB default charset=utf8
partition by hash(id)
partitions 8;
```

   v. 执行如下SQL语句，向example数据表中插入数据。

```
insert into example values(null,'lily',375),(null,'lisa',400),(null,'ljh',500);
```

   vi. 执行如下SQL语句，查询example表所有数据。

```
select * from example;
```

```
mysql> select * from example;
+--------+------+-------+
| id     | name | score |
+--------+------+-------+
| 100001 | lily |   375 |
| 100002 | lisa |   400 |
| 100003 | ljh  |   500 |
+--------+------+-------+
3 rows in set (0.03 sec)
```

vii. 执行如下SQL语句，查看example数据表的分区。

```
show topology from example;
```

返回结果如下，您可以看到example数据表分布在8个分区。

```
mysql> show topology from example;
+----+-----------------------------+--------------------+----------------+
| ID | GROUP_NAME                  | TABLE_NAME         | PARTITION_NAME |
+----+-----------------------------+--------------------+----------------+
|  0 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00000 | p1             |
|  1 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00001 | p2             |
|  2 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00002 | p3             |
|  3 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00003 | p4             |
|  4 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00004 | p5             |
|  5 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00005 | p6             |
|  6 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00006 | p7             |
|  7 | POLARX_EXAMPLE_P00000_GROUP | example_lb5Z_00007 | p8             |
+----+-----------------------------+--------------------+----------------+
8 rows in set (0.00 sec)
```

viii. 执行如下SQL语句，检查CDC。

```
show master status;
show binlog events  in 'binlog.000001' from 4;
```

ix. 执行如下SQL语句，检查DN和CN。

```
show storage;
show mpp;
```

x. 输入exit退出数据库。

```
mysql> exit
Bye
```

5. 执行如下命令，查看当前环境的PolarDB-X状态。

```
pxd list
```

返回如下结果，您可查看到PolarDB-X状态。

```
NAME                CN      DN      CDC     STATUS
pxc-tryout          1       1       1       running
```

6. 执行如下命令，清理本地环境所有的PolarDB-X。

```
pxd cleanup
```

2.3. （可选）使用Kubernetes安装PolarDB-X

本节介绍如何创建一个简单的Kubernetes集群并部署PolarDB-X Operator，并使用Operator部署一个完整的PolarDB-X集群。

前提条件

确保已安装并启动Docker。

操作步骤

> **说明**
>
> 有关使用Kubernetes安装部署PolarDB-X的详情，请参见通过Kubernetes部署集群。

1. 安装kubectl。
   i. 执行如下命令，下载kubectl文件。
   ```
   curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
   ```
   ii. 执行如下命令，赋予可执行权限。
   ```
   chmod +x ./kubectl
   ```
   iii. 执行如下命令，移动到系统目录。
   ```
   mv ./kubectl /usr/local/bin/kubectl
   ```
2. 执行如下命令，下载并安装minikube。
   ```
   curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
   sudo install minikube-linux-amd64 /usr/local/bin/minikube
   ```
   > **说明**
   >
   > minikube是由社区维护的用于快速创建Kubernetes测试集群的工具，适合测试和学习Kubernetes。使用minikube创建的Kubernetes集群可以运行在容器或是虚拟机中。
3. 安装Helm3。
   i. 执行如下命令，下载Helm3。
   ```
   wget https://labfileapp.oss-cn-hangzhou.aliyuncs.com/helm-v3.9.0-linux-amd64.tar.gz
   ```
   ii. 执行如下命令，解压Helm3。
   ```
   tar -zxvf helm-v3.9.0-linux-amd64.tar.gz
   ```
   iii. 执行如下命令，移动到系统目录。
   ```
   mv linux-amd64/helm /usr/local/bin/helm
   ```
4. 使用minikube创建Kubernetes集群。
   > **说明**
   >
   > 本实验以在CentOS 8.5上使用minikube创建Kubernetes为例。如您使用其他操作系统部署minikube（例如macOS或Windows），部分步骤可能略有不同。

i. 执行如下命令，新建账号galaxykube，并将galaxykube加入Docker组中。

```
useradd -ms /bin/bash galaxykube
usermod -aG docker galaxykube
```

> **说明**
>
> minikube要求使用非root账号进行部署，所以您需要新建一个账号。

ii. 执行如下命令，切换到账号galaxykube。

```
su galaxykube
```

iii. 执行如下命令，进入到home/galaxykube目录。

```
cd
```

iv. 执行如下命令，启动一个minikube。

```
minikube start --cpus 4 --memory 7168 --image-mirror-country cn --registry-mirror=h
ttps://docker.mirrors.sjtug.sjtu.edu.cn
```

> **说明**
>
> 这里我们使用的是阿里云的minikube镜像源以及USTC提供的Docker镜像源来加速镜像的拉取。

返回结果如下，表示minikube已经正常运行，minikube将自动设置kubectl的配置文件。

```
😄 minikube v1.25.2 on Centos 8.5.2111 (amd64)
✨ Automatically selected the docker driver
❗ Your cgroup does not allow setting memory.
    ▪ More information: https://docs.docker.com/engine/install/linux-postinstall/#your-kernel-does-not-support-cgroup-swap-limit-capabilities

🧯 The requested memory allocation of 7168MiB does not leave room for system overhead (total system memory: 7768MiB). You may face stability issues.
💡 Suggestion: Start minikube with less memory allocated: 'minikube start --memory=7168mb'

✅ Using image repository registry.cn-hangzhou.aliyuncs.com/google_containers
👍 Starting control plane node minikube in cluster minikube
🚜 Pulling base image ...
    > registry.cn-hangzhou.aliyun...: 379.06 MiB / 379.06 MiB  100.00% 6.37 MiB
🔥 Creating docker container (CPUs=4, Memory=7168MB) ...
🐳 Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
    > kubectl.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubelet.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubeadm.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubeadm: 43.12 MiB / 43.12 MiB [---------------] 100.00% 4.17 MiB p/s 11s
    > kubectl: 44.43 MiB / 44.43 MiB [---------------] 100.00% 4.23 MiB p/s 11s
    > kubelet: 118.75 MiB / 118.75 MiB [-------------] 100.00% 7.05 MiB p/s 17s
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎 Verifying Kubernetes components...
    ▪ Using image registry.cn-hangzhou.aliyuncs.com/google_containers/storage-provisioner:v5
🌟 Enabled addons: storage-provisioner, default-storageclass
🏄 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
(venv) [galaxykube@iZbp1dgzpaxwc0ykoczcmiZ ~]$
```

v. 执行如下命令，使用kubectl查看集群信息。

```
minikube kubectl -- cluster-info
```

返回如下结果，您可以查看到集群相关信息。

```
[galaxyku[illegible]x9dZ ~]$ kubectl cluster-info
Kubernetes control plane is running at https://19[illegible]8443
CoreDNS is running at https://[illegible]3/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

5. 部署 PolarDB-X Operator。

i. 执行如下命令，创建一个名为polardbx-operator-system的命名空间。

```
kubectl create namespace polardbx-operator-system
```

ii. 执行如下命令，安装PolarDB-X Operator。

```
helm install --namespace polardbx-operator-system polardbx-operator https://github.
com/ApsaraDB/galaxykube/releases/download/v1.2.1/polardbx-operator-1.2.1.tgz
```

```
[galaxykube@iZuf6i7k7oqyxzpl66zt9cZ ~]$ helm install --namespace polardbx-operator-system polardbx-operator https://github.com/ApsaraDB/galaxykube/releases/download/v1
.2.1/polardbx-operator-1.2.1.tgz
NAME: polardbx-operator
LAST DEPLOYED: Sat Jun 18 14:54:47 2022
NAMESPACE: polardbx-operator-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
polardbx-operator is installed. Please check the status of components:

    kubectl get pods --namespace polardbx-operator-system

Now have fun with your first PolarDB-X cluster.

Here's the manifest for quick start:

```yaml
apiVersion: polardbx.aliyun.com/v1
kind: PolarDBXCluster
metadata:
  name: quick-start
  annotations:
    polardbx/topology-mode-guide: quick-start
```
```

iii. 执行如下命令，查看PolarDB-X Operator组件的运行情况。

```
kubectl get pods --namespace polardbx-operator-system
```

请您耐心等待2分钟，将看到返回结果如下，等待所有组件都进入Running状态，表示PolarDB-X Operator已经安装完成。

```
[galaxykub...tZ ~]$ kubectl get pods --namespace polardbx-operator-system
NAME                                        READY   STATUS    RESTARTS   AGE
polardbx-controller-manager-7978bc7bd5-sr5qf   1/1     Running   0          2m11s
polardbx-hpfs-br7wh                         1/1     Running   0          2m11s
polardbx-tools-updater-f2zpw                1/1     Running   0          2m11s
```

6. 部署 PolarDB-X 集群。

i. 执行如下命令，部署一个PolarDB-X集群，它包含1个GMS 节点、1个 CN节点、1个DN节点和1 个CDC节点。

```
echo "apiVersion: polardbx.aliyun.com/v1
kind: PolarDBXCluster
metadata:
  name: quick-start
  annotations:
    polardbx/topology-mode-guide: quick-start" | kubectl apply -f -
```

ii. 执行如下命令，查看创建状态。

```
kubectl get polardbxcluster -w
```

请您耐心等待大约七分钟，将获得返回结果如下。当PHASE显示为Running时，表示PolarDB-X集群已经部署完成。现在您可以开始连接并体验PolarDB-X分布式数据库了。

```
[galaxykube@i...Z ~]$ kubectl get polardbxCluster polardb-x -o wide -w
NAME        PROTOCOL   GMS   CN    DN    CDC   PHASE      DISK      STAGE   REBALANCE   VERSION                              AGE
polardb-x   8.0        0/1   0/1   0/1   0/1   Creating                                                                   2m28s
polardb-x   8.0        1/1   0/1   1/1   0/1   Creating                                                                   3m33s
polardb-x   8.0        1/1   0/1   1/1   1/1   Creating                                                                   5m3s
polardb-x   8.0        1/1   1/1   1/1   1/1   Creating                                                                   6m32s
polardb-x   8.0        1/1   1/1   1/1   1/1   Running    7.2 GiB                       8.0.3-PXC-5.4.13-16534775/8.0.18    6m44s
polardb-x   8.0        1/1   1/1   1/1   1/1   Running    7.2 GiB                       8.0.3-PXC-5.4.13-16534775/8.0.18    7m44s
```

2.4. （可选）使用源码编译安装PolarDB-X

本节将介绍如何使用源码编译安装PolarDB-X。

操作步骤

> 说明
>
> 有关使用源码编译安装PolarDB-X的详情，请参见通过编译源码安装部署。

1. 执行如下命令，安装git。

```
yum -y install git
```

2. 执行如下命令，下载源码。

```
git clone https://github.com/ApsaraDB/PolarDB-X.git
```

3. 执行如下命令，进入PolarDB-X目录。

```
cd PolarDB-X
```

4. 执行如下命令，编译。

```
make
```

> 说明
>
> make将下载所有源码、工具和库，并构建和安装PolarDB-X。源码将下载到*./build*目录下，编译好的二进制文件将安装到*./build/run*目录下。您可以运行
>
> ```
> make clean
> ```
>
> 来移除安装，并尝试再次构建PolarDB-X。您也可以运行
>
> ```
> make cleanAll
> ```
>
> 来删除*./build*下的所有东西。这个过程可能需要半个多小时，取决于您的网络带宽。

5. 执行如下命令，运行PolarDB-X。

```
./build/run/bin/polardb-x.sh start
```

6. 执行如下命令，停止PolarDB-X。

```
./build/run/bin/polardb-x.sh stop
```

3.使用PolarDB-X开发应用

本节在安装部署好的PolarDB-X基础上，介绍Spring Boot、WordPress与PolarDB-X配合开发应用的方法，以及PolarDB-X连接池和透明分布式的最佳实践经验。

> **说明**
>
> 本节实验操作部分主要通过阿里云官方网站的云起实验室进行，详情可登录阿里云官方网站，访问如何使用PolarDB-X了解。

3.1. 安装PolarDB-X和JDK

3.1.1. 第1步：安装PolarDB-X

操作步骤

1. 安装并启动Docker。
   i. 执行如下命令，安装Docker。
   ```
   curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun
   ```
   ii. 执行如下命令，启动Docker。
   ```
   systemctl start docker
   ```
2. 执行如下命令，安装PolarDB-X。
   ```
   docker run -d --name some-polardb-x -p 8527:8527 polardbx/polardb-x:2.1.0
   ```

3.1.2. 第2步：登录PolarDB-X

PolarDB-X支持通过MySQL Client命令行、第三方客户端以及符合MySQL交互协议的第三方程序代码进行连接。本节主要介绍如何通过MySQL Client命令行连接到PolarDB-X数据库。

操作步骤

1. 执行如下命令，安装MySQL。
   ```
   yum install mysql -y
   ```
2. 执行如下命令，查看MySQL版本号。
   ```
   mysql -V
   ```
   返回结果如下，表示您已成功安装MySQL。
   ```
   [roo        ~]# mysql -V
   mysql  Ver 8.0.26 for Linux on x86_64 (Source distribution)
   ```
3. 执行如下命令，登录PolarDB-X数据库。
   ```
   mysql -h127.0.0.1 -P8527 -upolardbx_root -p123456
   ```

> 说明
>
> - 本实验场景中的PolarDB-X数据库用户名和密码已预设，请您使用上述命令登录即可。
> - 如遇到报错：
>
> ```
> mysql: [Warning] Using a password on the command line interface can be insecure. ERROR 2013 (HY000): Lost connection to MySQL server at 'reading initial communication packet', system error: 0
> ```
>
> ，请您稍等一分钟，重新执行登录命令即可。

返回结果如下，表示您已成功登录PolarDB-X数据库。

```
[root@iZbp          31Z ~]# mysql -h127.0.0.1 -P8527 -upolardbx_root -p123456
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.6.29 Tddl Server (ALIBABA)

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

4. 输入exit退出数据库。

```
mysql> exit
Bye
```

3.1.3. 第3步：安装JDK

操作步骤

1. 执行如下命令，使用yum安装JDK 1.8。

```
yum -y install java-1.8.0-openjdk*
```

2. 执行如下命令，查看是否安装成功。

```
java -version
```

返回结果如下，表示您已成功安装JDK 1.8。

```
[root@iZbp          iicblZ ~]# java -version
openjdk version "1.8.0_312"
OpenJDK Runtime Environment (build 1.8.0_312-b07)
OpenJDK 64-Bit Server VM (build 25.312-b07, mixed mode)
```

3.2. 体验Spring Boot+PolarDB-X应用开发

本节介绍如何下载并编辑Spring Boot样例工程，并连接到PolarDB-X数据库。

前提条件

确保已安装PolarDB-X和JDK，详情请参见安装PolarDB-X和JDK。

操作步骤

> **说明**
>
> 有关Spring Boot的详情，请参见Spring Boot样例教程。

1. 执行如下命令，安装Git。

```
yum -y install git
```

2. 下载Spring Boot样例工程。

   i. 执行如下命令，下载Spring Boot样例工程。

   ```
   git clone https://github.com/spring-guides/gs-accessing-data-mysql.git
   ```

   ii. 执行如下命令，进入initial目录。

   ```
   cd gs-accessing-data-mysql/initial
   git checkout b8408e3a1e05008811d542b706107d45160556ac
   ```

   iii. 执行如下命令，查看样例工程代码。

   ```
   ls
   ```

```
[root@iZbp173hgzx7f5esuhel7yZ initial]# ls
build.gradle  gradle  gradlew  gradlew.bat  mvnw  mvnw.cmd  pom.xml  settings.gradle  src
```

3. 创建数据库。

   i. 执行如下命令，登录PolarDB-X数据库。

   ```
   mysql -h127.0.0.1 -P8527 -upolardbx_root -p123456
   ```

   ii. 执行如下SQL语句，创建数据库db_example。

   ```
   create database db_example;
   ```

   iii. 执行如下SQL语句，创建用户springuser。

   ```
   create user 'springuser'@'%' identified by 'ThePassword';
   ```

   iv. 执行如下SQL语句，给用户springuser授权。

   ```
   grant all on db_example.* to 'springuser'@'%';
   ```

   v. 退出数据库。

   ```
   exit
   ```

4. 配置application.properties文件，将数据库连接到Spring Boot样例工程。

   i. 执行如下命令，打开application.properties配置文件。

```
vim src/main/resources/application.properties
```

ii. 按i键进入编辑模式，找到参数spring.datasource.url，并将参数值中的端口号修改为*8527*。

```
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:8527/db_example
```

iii. 修改完成后的文件内容如下图所示。按下Esc键后，输入:wq后按下Enter键保存并退出。

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:8527/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
~
```

5. 创建Entity Model。

i. 执行如下命令，创建一个User类。

```
vim src/main/java/com/example/accessingdatamysql/User.java
```

ii. 将如下代码复制粘贴到User类中。

```
package com.example.accessingdatamysql;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity // This tells Hibernate to make a table out of this class
public class User {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;
  private String name;
  private String email;
  public Integer getId() {
    return id;
  }
  public void setId(Integer id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
}
```

iii. 修改完成后的文件内容如下图所示。按下Esc键后，输入:wq后按下Enter键保存并退出。

```
package com.example.accessingdatamysql;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity // This tells Hibernate to make a table out of this class
public class User {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;

  private String name;

  private String email;

  public Integer getId() {
    return id;
  }

  public void setId(Integer id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }
}
~
~
~
"src/main/java/com/example/accessingdatamysql/User.java" 41L, 737C
```

6. 创建Repository，保存用户记录。

i. 执行如下命令，创建一个UserRepository类。

```
vim src/main/java/com/example/accessingdatamysql/UserRepository.java
```

ii. 将如下代码复制粘贴到UserRepository类中。

```
package com.example.accessingdatamysql;
import org.springframework.data.repository.CrudRepository;
import com.example.accessingdatamysql.User;
// This will be AUTO IMPLEMENTED by Spring into a Bean called userRepository
// CRUD refers Create, Read, Update, Delete
public interface UserRepository extends CrudRepository<User, Integer> {
}
```

iii. 修改完成后的文件内容如下图所示。按下Esc键后，输入:wq后按下Enter键保存并退出。

```
package com.example.accessingdatamysql;

import org.springframework.data.repository.CrudRepository;

import com.example.accessingdatamysql.User;

// This will be AUTO IMPLEMENTED by Spring into a Bean called userRepository
// CRUD refers Create, Read, Update, Delete

public interface UserRepository extends CrudRepository<User, Integer> {

}
~
```

7. 创建一个Controller类，处理对应用程序的HTTP请求。

i. 执行如下命令，创建一个MainController类。

```
vim src/main/java/com/example/accessingdatamysql/MainController.java
```

ii. 将如下代码复制粘贴到MainController类中。

```
package com.example.accessingdatamysql;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
@Controller // This means that this class is a Controller
@RequestMapping(path="/demo") // This means URL's start with /demo (after Applicati
on path)
public class MainController {
  @Autowired // This means to get the bean called userRepository
         // Which is auto-generated by Spring, we will use it to handle the data
  private UserRepository userRepository;
  @PostMapping(path="/add") // Map ONLY POST Requests
  public @ResponseBody String addNewUser (@RequestParam String name
      , @RequestParam String email) {
    // @ResponseBody means the returned String is the response, not a view name
    // @RequestParam means it is a parameter from the GET or POST request
    User n = new User();
    n.setName(name);
    n.setEmail(email);
    userRepository.save(n);
    return "Saved";
  }
  @GetMapping(path="/all")
  public @ResponseBody Iterable<User> getAllUsers() {
    // This returns a JSON or XML with the users
    return userRepository.findAll();
  }
}
```

iii. 修改完成后的文件内容如下图所示。按下Esc键后，输入:wq后按下Enter键保存并退出。

```
package com.example.accessingdatamysql;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller // This means that this class is a Controller
@RequestMapping(path="/demo") // This means URL's start with /demo (after Application path)
public class MainController {
  @Autowired // This means to get the bean called userRepository
         // Which is auto-generated by Spring, we will use it to handle the data
  private UserRepository userRepository;

  @PostMapping(path="/add") // Map ONLY POST Requests
  public @ResponseBody String addNewUser (@RequestParam String name
      , @RequestParam String email) {
    // @ResponseBody means the returned String is the response, not a view name
    // @RequestParam means it is a parameter from the GET or POST request

    User n = new User();
    n.setName(name);
    n.setEmail(email);
    userRepository.save(n);
    return "Saved";
  }

  @GetMapping(path="/all")
  public @ResponseBody Iterable<User> getAllUsers() {
    // This returns a JSON or XML with the users
    return userRepository.findAll();
  }
}
```

8. （可选）创建一个Application。

> **说明**
>
> Spring Boot样例工程中已为您创建好AccessingDataMysqlApplication类，您可跳过此步骤。

i. 执行如下命令，创建一个AccessingDataMysqlApplication类。

```
vim src/main/java/com/example/accessingdatamysql/AccessingDataMysqlApplication.java
```

ii. 按i键进入编辑模式，将如下代码复制粘贴到User类中。

```
package com.example.accessingdatamysql;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class AccessingDataMysqlApplication {
  public static void main(String[] args) {
    SpringApplication.run(AccessingDataMysqlApplication.class, args);
  }
}
```

iii. 修改完成后的文件内容如下图所示。按下Esc键后，输入:wq后按下Enter键保存并退出。

```
package com.example.accessingdatamysql;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AccessingDataMysqlApplication {

        public static void main(String[] args) {
                SpringApplication.run(AccessingDataMysqlApplication.class, args);
        }

}
```

9. 执行如下命令，运行Spring Boot样例工程。

```
./gradlew bootRun
```

请您耐心等待大约两分钟，返回结果如下，表示您成功运行。

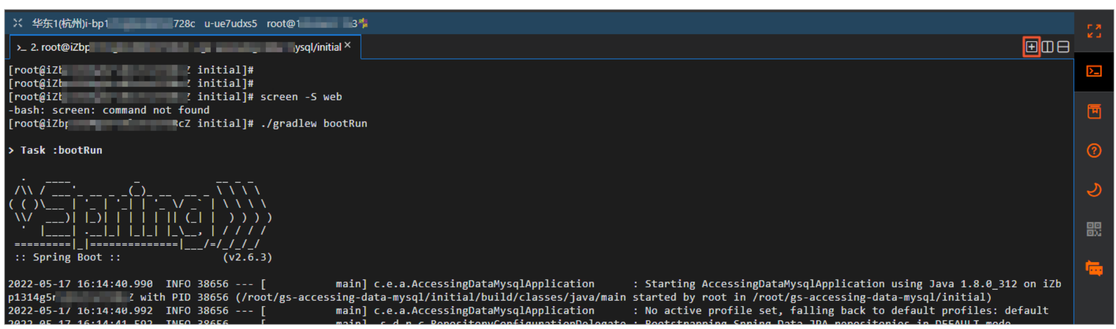
```
> Task :bootRun

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.3)

2022-05-17 16:00:33.769  INFO 20099 --- [           main] c.e.a.AccessingDataMysqlApplication      : Starting AccessingDataMysqlApplication using Java 1.8.0_312 on iZb
p1314g5rv8lctn2728cZ with PID 20099 (/root/gs-accessing-data-mysql/initial/build/classes/java/main started by root in /root/gs-accessing-data-mysql/initial)
2022-05-17 16:00:33.771  INFO 20099 --- [           main] c.e.a.AccessingDataMysqlApplication      : No active profile set, falling back to default profiles: default
2022-05-17 16:00:34.311  INFO 20099 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2022-05-17 16:00:34.349  INFO 20099 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 30 ms. Found 1 JPA rep
ository interfaces.
2022-05-17 16:00:34.837  INFO 20099 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-05-17 16:00:34.852  INFO 20099 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-05-17 16:00:34.852  INFO 20099 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.56]
2022-05-17 16:00:34.920  INFO 20099 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-05-17 16:00:34.921  INFO 20099 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1097 ms
2022-05-17 16:00:35.074  INFO 20099 --- [           main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2022-05-17 16:00:35.110  INFO 20099 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate ORM core version 5.6.4.Final
2022-05-17 16:00:35.250  INFO 20099 --- [           main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
2022-05-17 16:00:35.319  INFO 20099 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2022-05-17 16:00:39.084  INFO 20099 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2022-05-17 16:00:39.244  INFO 20099 --- [           main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.MySQL55Dialect
2022-05-17 16:00:42.170  INFO 20099 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine
.transaction.jta.platform.internal.NoJtaPlatform]
2022-05-17 16:00:42.179  INFO 20099 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default
'
2022-05-17 16:00:42.470  WARN 20099 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database
 queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2022-05-17 16:00:42.748  INFO 20099 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-05-17 16:00:42.757  INFO 20099 --- [           main] c.e.a.AccessingDataMysqlApplication      : Started AccessingDataMysqlApplication in 9.342 seconds (JVM runnin
g for 9.696)
<===========---> 80% EXECUTING [2m 28s]
> IDLE
> IDLE
> :bootRun
> IDLE
```

10. 测试。

i. 在实验页面，单击右上角的[+]图标，创建新的终端窗口。



ii. 在新的终端窗口中，执行如下命令，增加一条记录。

```
curl localhost:8080/demo/add -d name=First -d email=someemail@someemailprovider.com
```

返回结果如下，表示您成功增加一条记录。

```
[root@iZ...Z ~]# curl localhost:8080/demo/add -d name=First -d email=someemail@someemailprovider.com
Saved root@iZ...Z ~]#
```

iii. 执行如下命令，查询记录。

```
curl 'localhost:8080/demo/all'
```

返回结果如下，您可以查询到刚刚增加的记录信息。

```
[root@iZ...Z ~]# curl 'localhost:8080/demo/all'
[{"id":1,"name":"First","email":"someemail@someemailprovider.com"},
```

iv. 执行如下命令，登录PolarDB-X数据库。

```
mysql -h127.0.0.1 -P8527 -upolardbx_root -p123456
```

v. 执行如下SQL语句，使用数据库。

```
use db_example;
```

vi. 执行如下SQL语句，查询user表。

```
select * from user;
```

返回如下结果，您可以在user表查询到刚刚增加的记录。

```
mysql> select * from user;
+------+---------------------------------+-------+
| id   | email                           | name  |
+------+---------------------------------+-------+
|    1 | someemail@someemailprovider.com | First |
+------+---------------------------------+-------+
1 row in set (0.01 sec)
```

vii. 退出数据库。

```
exit
```

3.3. 体验WordPress+PolarDB-X部署博客站点

本节将介绍如何使用Wordpress的Docker镜像和PolarDB-X搭建一个博客站点。

前提条件

请确保您已安装PolarDB-X，详情请参见安装PolarDB-X和登录PolarDB-X。

操作步骤

> **说明**
> Wordpress提供了Docker镜像，方便其快速安装，详情请参见WordPress的Docker Hub主页。

1. 执行如下命令，安装WordPress。

```
docker run --name some-wordpress -p 9090:80 -d wordpress
```

2. 创建WordPress的数据库。

   i. 执行如下命令，登录PolarDB-X数据库。

   ```
   mysql -h127.0.0.1 -P8527 -upolardbx_root -p123456
   ```

   ii. 执行如下SQL语句，创建数据库wordpress。

   ```
   create database wordpress;
   ```

   iii. 退出数据库。

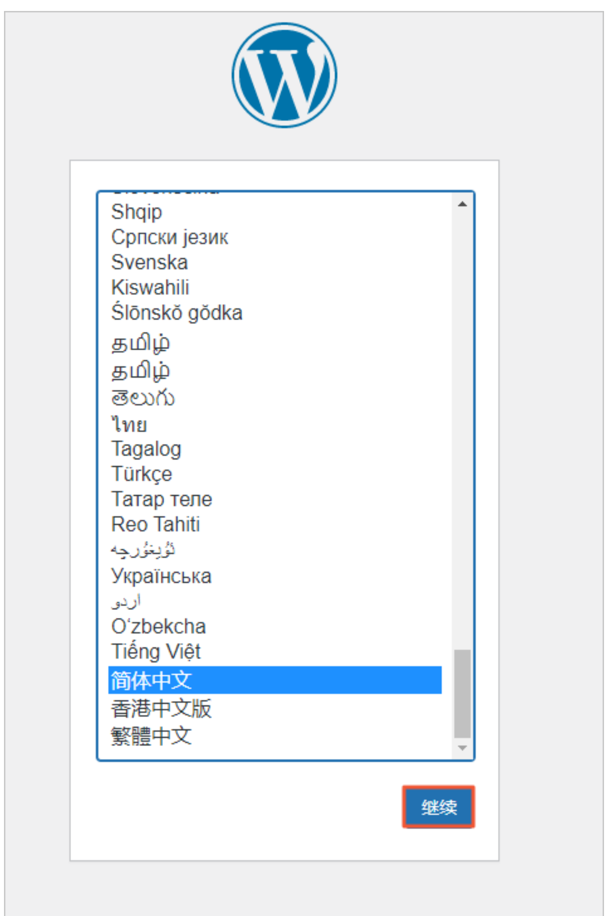
   ```
   exit
   ```

3. 配置WordPress。

   i. 在您的本机浏览器中，打开新页签，访问*http://<ECS的弹性IP>:9090*。

   > **说明**
   > 您需要将<ECS的弹性IP>替换为云产品资源列表中的ECS的弹性IP。

ii. 在初始化页面，选择**简体中文**，单击**继续**。



iii. 在准备页面，单击**现在就开始**。

iv. 在数据库配置页面，参考说明配置数据库信息，单击**提交**。

数据库名

默认为wordpress。

用户名

输入polardbx_root。

密码

输入123456。

数据库主机

输入<ECS的弹性IP>:8527。您需要将<ECS的弹性IP>替换为云产品资源列表中的ECS的弹性IP。

表前缀

默认为wp_。

v. 在数据库配置完成页面，单击**运行安装程序**。

vi. 在信息配置页面，参考说明配置相关信息，单击安装WordPress。

站点标题

输入站点标题，例如myblog。

用户名

输入用户名，例如admin。

密码

输入密码。

您的电子邮箱地址

输入邮箱地址。建议使用真实有效的邮箱地址，若没有，可以填写虚拟邮箱地址，但将无法接收信息，例如admin@admin.com。

vii. 在成功页面，单击登录。

viii. 在登录页面，依次输入您的用户名和密码，单击登录。

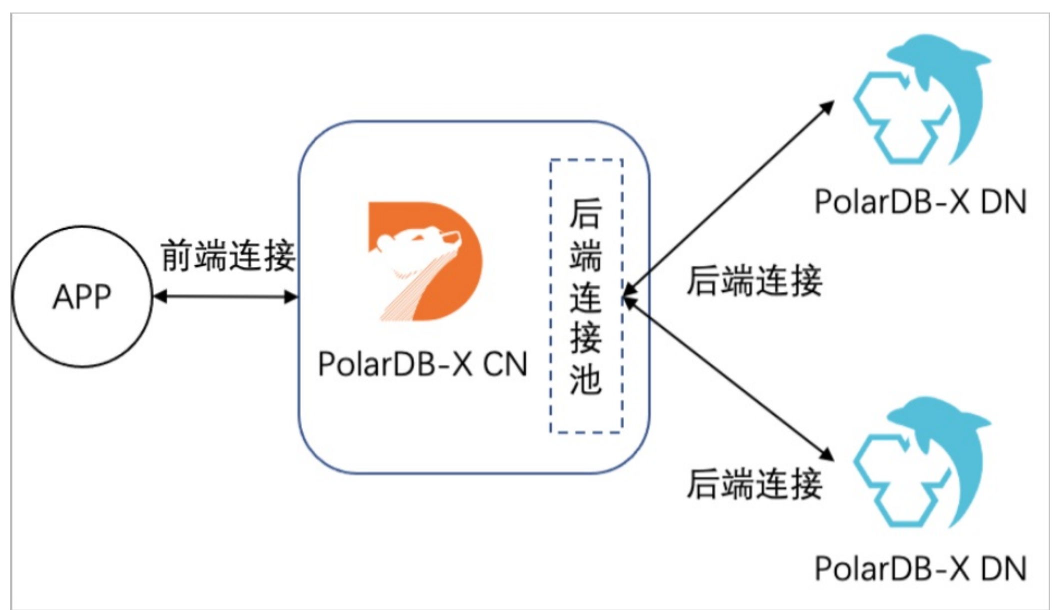
3.4. PolarDB-X应用开发最佳实践

3.4.1. 如何选择应用端连接池

背景

当应用程序连接PolarDB-X实例执行操作时，从PolarDB-X实例的角度看，会有如下两种类型的连接：

- **前端连接**：由应用程序建立的，到PolarDB-X计算节点（CN）中逻辑库的连接。
- **后端连接**：由PolarDB-X计算节点建立的，到后端数据节点（DN）中物理库的连接。



其中后端连接由CN管理，通过私有协议实现TCP连接与后端连接解绑，对用户透明。前端连接由用户创建和管理，本文主要讨论前端连接管理的最佳实践。

说明

为了简化描述，下文中的 "连接" 均代指 "前端连接"。

QPS/RT与连接数的关系

每秒查询请求数 (Query Per Second，QPS) 和响应时间 (Response Time，RT) 是衡量应用对数据库性能需求的基本指标，QPS代表应用并发访问数据库的需求，RT代表单条语句的处理性能。RT的高低与执行的SQL是否复杂和扫描的数据量紧密相关，OLTP系统中查询RT较低，通常以毫秒为单位。

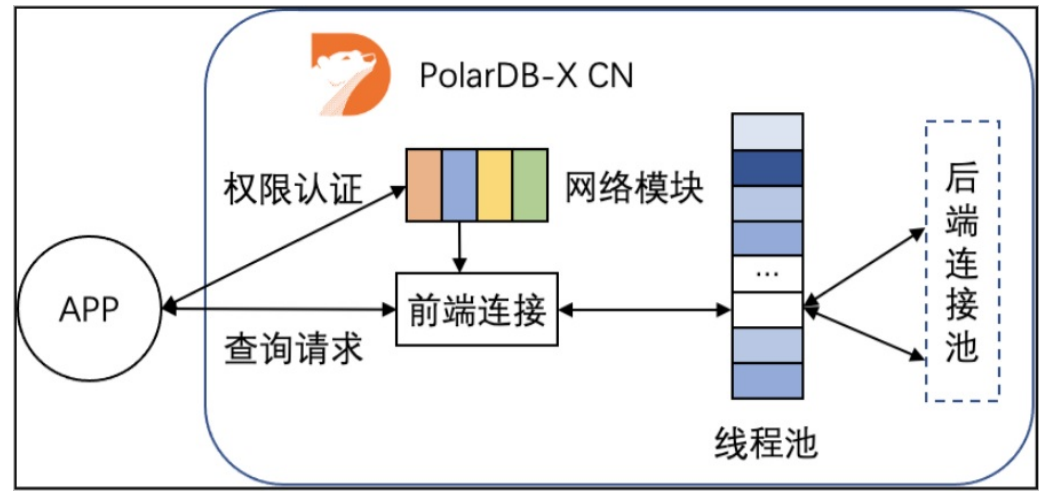
PolarDB-X兼容MySQL协议，请求在单个连接上串行执行，不同连接上的请求可以并行执行，由此得到以下公式：

- 单个连接的QPS上限 = 1000/RT
- 应用访问单个CN的QPS上限 = 单个连接的QPS上限×连接数

按照平均RT为5ms计算，单个连接的QPS上限为200，假设应用预估需要的QPS为5000，则至少需要建立25个连接。

连接数限制

PolarDB-X中前端连接仅与网络连接绑定，连接的数量理论上仅受限于CN可用的内存大小和网络连接数。但在实际的场景中，应用创建连接是为了执行查询，连接数需要与执行线程的数量相匹配才能达到最佳性能。



如上图所示，应用发起连接请求后，首先由网络模块进行权限认证，认证通过则创建连接对象。
与 MariaDB 相似，PolarDB-X计算节点收到后续查询请求后，会尝试从线程池中分配一个执行线程，用于处理查询请求。单个CN节点默认维护一个1024大小的线程池，如果并发查询数量超过线程池大小，后续请求会排队等待。由此可以得到以下公式：

- 应用访问单个CN的QPS上限 = 单个连接的QPS上限×MIN（连接数, 线程池大小）
- 应用访问数据库的QPS上限 = 单个连接的QPS上限×MIN（连接数, 线程池大小）×CN数量

我们通过以下两个示例来进行公式的实际应用：

- 示例一：

  问：查询的平均RT为10ms，理想情况下两节点CN能提供多少QPS？

  答：平均RT为10ms，则单个连接的QPS上限为100。理想情况下（CPU不成为瓶颈），包含2个CN节点的PolarDB-X实例默认能够提供的QPS上限为 100×1024×2 = 204800。这里需要注意，CN节点能够同时处理的查询数量与CN规格和查询复杂度有关，实际场景下通常不能做到1024个线程完全并行，QPS上限会低于204800。

- 示例2

  问：在CN规格为16C的PolarDB-X实例上压测，CPU刚好跑满时，某查询的平均RT为5ms。 仅考虑CN节点，支撑40w的QPS应当如何选择实例和设置应用端连接池？

  答：平均RT为5ms，则单个连接的QPS上限为200。应用端连接池大小设置为400000/200 = 2000可以将额外开销降到最低。保持单个CN节点上的并行度不超过1024，需要两个16C节点构成的32C的 PolarDB-X实例。

连接池

数据库连接池是对数据库连接进行统一管理的技术，主要目的是提高应用性能，减轻数据库负载。

- 提高系统响应效率：连接的初始化工作完成后，所有请求可以直接利用现有连接，避免了连接初始化和释放的开销，提高了系统的响应效率。
- 资源复用：连接可以重复利用，避免了频繁创建、释放连接引入的性能开销。在减少系统消耗的基础上，增强了系统的平稳性。
- 避免连接泄漏：连接池可根据预设的回收策略，强制回收连接，从而避免了连接资源泄漏。

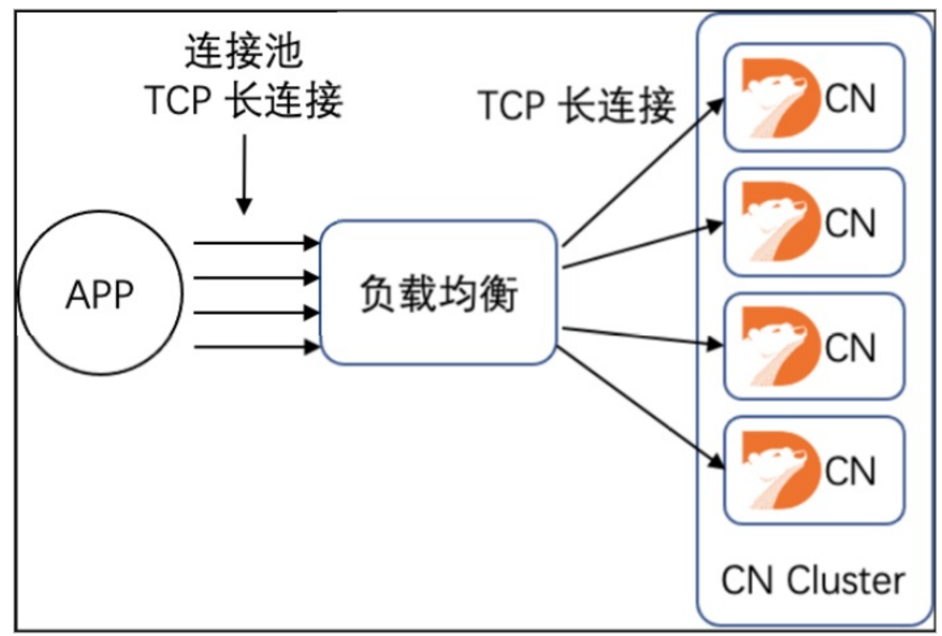
如果是Java程序，推荐使用 Druid 连接池 ，版本要求1.1.11及以上。

Druid的Spring标准配置如下：

```
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init"
destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- 基本属性 URL、user、password -->
        <property name="url" value="jdbc:mysql://ip:port/db?autoReconnect=true&rewriteBatch
edStatements=true&socketTimeout=30000&connectTimeout=3000" />
        <property name="username" value="root" />
        <property name="password" value="123456" />
        <!-- 配置初始化大小、最小、最大 -->
        <property name="maxActive" value="20" />
        <property name="initialSize" value="3" />
        <property name="minIdle" value="3" />
        <!-- maxWait 获取连接等待超时的时间 -->
        <property name="maxWait" value="60000" />
        <!-- timeBetweenEvictionRunsMillis 间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是
毫秒 -->
        <property name="timeBetweenEvictionRunsMillis" value="60000" />
        <!-- minEvictableIdleTimeMillis 一个连接在池中最小空闲的时间，单位是毫秒-->
        <property name="minEvictableIdleTimeMillis" value="300000" />
        <!-- 检测连接是否可用的 SQL -->
        <property name="validationQuery" value="select 'z' from dual" />
        <!-- 是否开启空闲连接检查 -->
        <property name="testWhileIdle" value="true" />
        <!-- 是否在获取连接前检查连接状态 -->
        <property name="testOnBorrow" value="false" />
        <!-- 是否在归还连接时检查连接状态 -->
        <property name="testOnReturn" value="false" />
        <!-- 是否在固定时间关闭连接。增加此参数可以均衡后端服务节点参数 -->
        <property name="phyTimeoutMillis" value="600000" />
        <!-- 是否在固定SQL使用次数之后关闭连接，增加此参数可以均衡后端服务节点参数-->
        <property name="phyMaxUseCount" value="10000" />
    </bean>
```

注意事项

- 连接池与负载均衡



连接池模式（TCP 长连接）效率更高，但部分场景下对分布式负载均衡不友好，可能导致CN负载不均匀

- 突发创建连接，导致分布不均

  如果应用存在突发创建大量连接的情况，负载均衡设备无法及时刷新统计信息，可能出现部分CN上连接较多，结合连接池化，最终导致部分CN压力高于其他CN，影响系统总体性能。

- 负载均衡探活异常，导致分布不均

  负载均衡通过主动探活来判断CN节点是否正常，当探活出现偶发异常时，可能导致部分CN上连接较少，结合连接池化，最终导致部分CN压力低于其他CN，影响系统总提性能。

Druid连接池增加了phyTimeoutMillis/phyMaxUseCount参数，定期（例如执行10000次或者10分钟）刷新连接池中的连接，可以在解决上述问题的同时保持性能基本不变，建议默认添加这两个配置。

- 应用线程数与连接池

  应用程序访问数据库的一种常见模式，是在应用程序中创建多个线程，每个线程获取一个到数据库的连接并执行查询。为了减少创建/释放线程的开销，通常会使用"线程池"来管理线程，线程池的一个重要参数是"最大线程数"，需要根据实际情况调整。

  理想情况下，查询的RT波动不大，可以应用上文介绍的公式，根据RT计算出合理的连接池大小，并按照"每个线程一个数据库连接"的思路确定最大线程数。实际场景中，查询RT受到热点、锁、数据倾斜等多种因素的影响，可能出现突发RT增长，甚至部分连接失去响应。如果完全按照理想情况连接池/线程池，可能由于部分慢查询耗尽连接池/线程池，导致应用失去响应，影响关联系统。因此，建议按照"理想情况"的1.5到2倍来设置最大连接数/线程数。

3.4.2. 透明分布式最佳实践

常见的分布式数据库都需要用户设置分库分表键，即需要用户手动管理分库分表规则。这为用户使用分布式数据库带来了门槛，用户需要对数据库的数据分布以及每一张表的结构都有非常清晰的认知，才能用好分布式数据库。PolarDB-X提出了透明分布式的概念，分布式的细节对用户透明，用户只需要以使用单机数据库的方式来使用分布式数据库即可，从而极大降低用户使用分布式数据库的门槛。

从PolarDB-X 5.4.13版本开始，新增支持AUTO模式的数据库（也称为自动分区数据库）。AUTO模式的数据库支持自动分区，即创建表时无需指定分区键，数据即可自动在集群内均匀分布；同时也支持使用标准的MySQL分区表语法，对表进行手动分区。可以让您便捷地享受到分布式数据库的透明式分布、弹性伸缩和分区管理等诸多红利。

PolarDB-X 5.4.13版本之前的数据库称为DRDS模式的数据库。这种模式的数据库不支持自动分区，创建表时需使用DRDS专用的分库分表语法，指定分库分表键，否则创建的是一张单表。

AUTO模式数据库和DRDS模式数据库在5.4.13版本后都支持，并且可以在一个数据库实例中共存。

> 注意
>
> - 创建AUTO模式数据库必须在CREATE DATABASE语法中显示指定MODE='AUTO'。
> - 如果在CREATE DATABASE语法中不指定MODE参数的值，默认创建DRDS模式的数据库。
> - AUTO模式数据库下不支持使用DRDS分库分表的语法创建表，仅支持创建分区表。
> - DRDS模式数据库下不支持使用分区表的语法创建表，仅支持创建分库分表。

通过MODE参数指定数据库模式

PolarDB-X在创建数据库时引入了MODE参数，以决定创建的数据库是AUTO模式还是DRDS模式。关于MODE参数的作用及其描述，如下表所示：

> 说明 数据库创建完成后，MODE不允许修改。

参数	取值类型	作用	建库语法	建表语法
MODE	'AUTO'	创建的数据库为AUTO模式。	示例： `CREATE DATABASE auto_db MODE='AUTO';` 详情请参见CREATE DATABASE。	AUTO模式数据库下创建的表称为**分区表**，采用MySQL标准语法，详情请参见MySQL分区表语法。

	'DRDS'（默认值）
	说明 若不指定MODE参数，则默认创建DRDS模式数据库。
		创建的数据库为DRDS模式。	示例：
			`CREATE DATABASE drds_db MODE='DRDS';`
			`CREATE DATABASE drds_db;`
			详情请参见CREATE DATABASE。
				DRDS模式数据库下创建的表称为**分库分表**，详情请参见DRDS分库分表语法。

自动分区与手动分区

自动分区

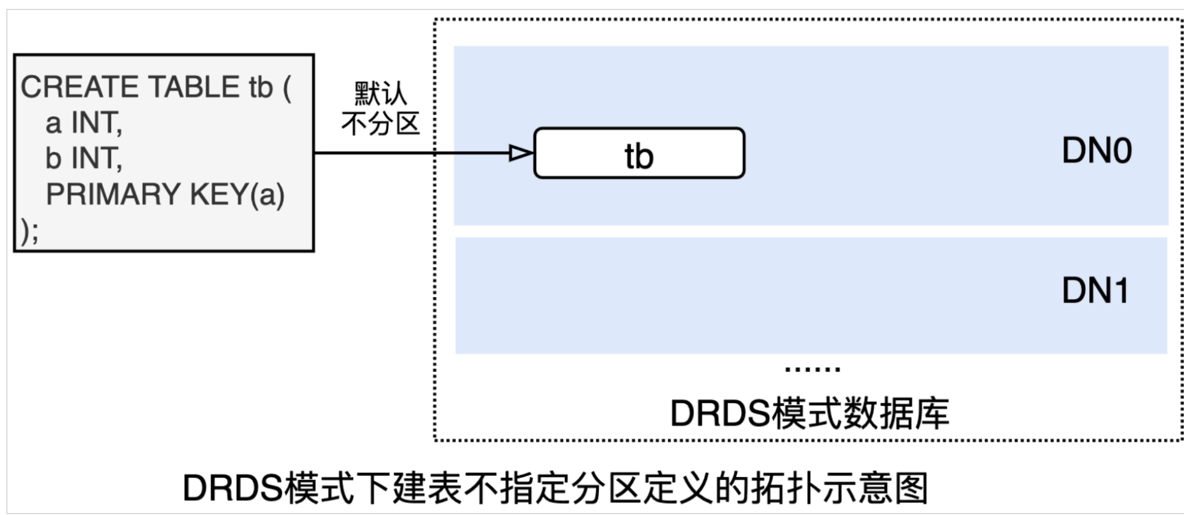
自动分区，指创建表时不指定任何分区定义（如分区键、分区策略等），PolarDB-X能够自动选择分区键并对表及其索引进行水平分区的功能。AUTO模式数据库支持自动分区，而DRDS模式数据库不支持。

示例如下：

使用标准的MySQL语法创建tb表，语法上不包含任何分区定义：

```
CREATE TABLE tb(a INT, b INT, PRIMARY KEY(a));
```

- 上述DDL在DRDS模式数据库中创建出来的表是一张单表（如下图所示，默认不分区）：

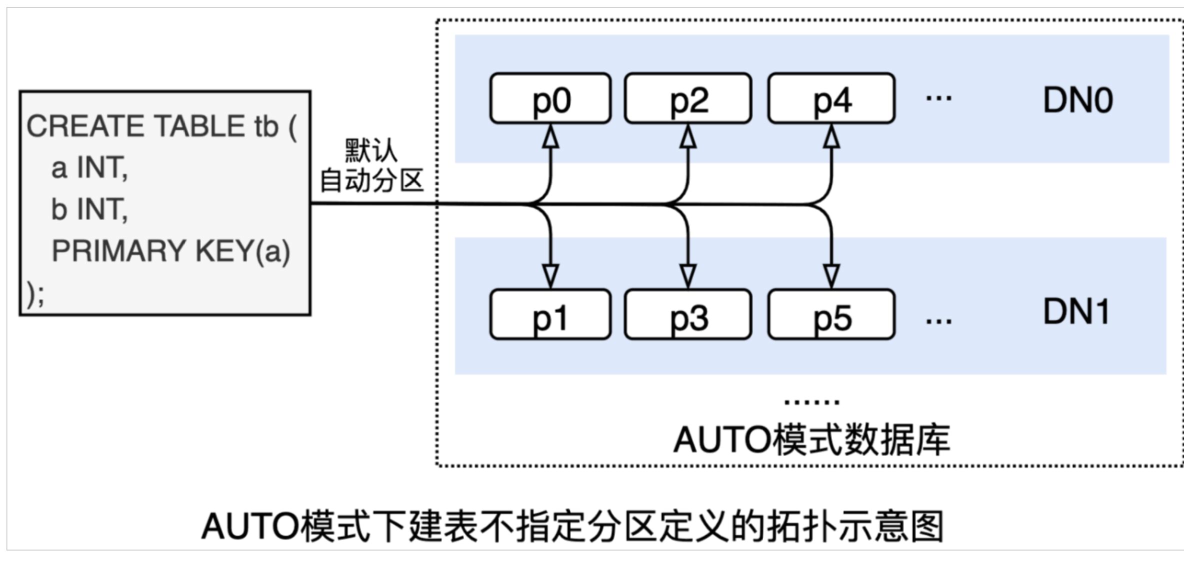


DRDS模式下建表不指定分区定义的拓扑示意图

执行 `SHOW` 语句，查看完整建表语句：

```
SHOW FULL CREATE TABLE tb \G
*************************** 1. row ***************************
       Table: tb
Create Table: CREATE TABLE `tb` (
    `a` int(11) NOT NULL,
    `b` int(11) DEFAULT NULL,
    PRIMARY KEY (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
1 row in set (0.02 sec)
```

- 上述DDL在AUTO模式数据库建来的表将是一分区表（如下图所示，默认按主键自动分区）：



AUTO模式下建表不指定分区定义的拓扑示意图

执行 `SHOW` 语句，查看完整建表语句：

```
SHOW FULL CREATE TABLE tb \G
*************************** 1. row ***************************
       TABLE: tb
CREATE TABLE: CREATE PARTITION TABLE `tb` (
    `a` int(11) NOT NULL,
    `b` int(11) DEFAULT NULL,
    PRIMARY KEY (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`a`)
PARTITIONS 16
1 row in set (0.01 sec)
```

因此，在AUTO模式数据库下，只需要使用标准的MySQL建表语法（包括建索引语法等）创建表，PolarDB-X的自动分区功能可以让应用便捷地享受到分布式数据库所带来的弹性伸缩、分区管理等诸多红利。

手动分区

手动分区，即创建表时显式指定分区定义（如分区键、分区策略等）。AUTO模式数据库与DRDS模式数据库采用手动分区时的建表语法不同。

- AUTO模式数据库：创建表使用标准的MySQL分区表语法，并支持HASH/RANGE/LIST等多种分区策略。

  如下示例，创建tb表时使用 `PARTITION BY HASH(a)` 语法，指定了分区键a列及HASH的分区策略：

```
CREATE TABLE tb (a INT, b INT, PRIMARY KEY(a))
    -> PARTITION by HASH(a) PARTITIONS 4;
Query OK, 0 rows affected (0.83 sec)
SHOW FULL CREATE TABLE tb\G
*************************** 1. row ***************************
       TABLE: tb
CREATE TABLE: CREATE TABLE `tb` (
    `a` int(11) NOT NULL,
    `b` int(11) DEFAULT NULL,
    PRIMARY KEY (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
PARTITION BY KEY(`a`)
PARTITIONS 4
1 row in set (0.02 sec)
```

- DRDS模式数据库：创建表使用DRDS专用的分库分表语法，仅支持使用HASH策略。

  如下示例，创建tb表时使用 `DBPARTITION BY HASH(a) TBPARTITION BY HASH(a)` 语法，指定分库分表键为a列：

```
CREATE TABLE tb (a INT, b INT, PRIMARY KEY(a))
    -> DBPARTITION by HASH(a)
    -> TBPARTITION by HASH(a)
    -> TBPARTITIONS 4;
Query OK, 0 rows affected (1.16 sec)
SHOW FULL CREATE TABLE tb\G
*************************** 1. row ***************************
       Table: tb
Create Table: CREATE TABLE `tb` (
    `a` int(11) NOT NULL,
    `b` int(11) DEFAULT NULL,
    PRIMARY KEY (`a`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4  dbpartition by hash(`a`) tbpartition by hash
(`a`) tbpartitions 4
1 row in set (0.02 sec)
```

分区表和分库分表的路由算法对比

分区表与分库分表最重要的区别，即它们的分区会使用完全不同的路由算法。如下图所示：

- 分库分表的路由算法是HASH值按物理分表数目取模，如果要变更分区数目（例如分表数目由4个变成5个），所有数据都需要进行rehash，因此，DRDS模式的分库分表无法提供分区级的变更能力；
- 分区表的默认路由算法是基于range的一致性HASH算法，这种算法天然支持通过分裂、合并操作等变更分区，并且无须rehash所有数据，因此，AUTO模式的分区表具备分区级的变更能力。

分区表与分库分表的路由算法对比

AUTO模式核心特性及其典型场景

热点分裂——有效解决数据热点

对于热点数据，PolarDB-X支持两种处理方式：

- 第一种方案是将热点数据所在的分区数据迁移到特定的数据节点，让热点数据以独享存储资源的方式服务业务，能够实现热点数据不影响非热点数据的业务。具体操作步骤如下：

  i. 执行以下语句，将特定的热点数据提取到一个单独的分区。

  ```
  ALTER TABLEGROUP #tgName EXTRACT to PARTITION #hotPartitionName BY HOT VALUE(#keyVal)
  ```

  ii. 执行以下语句，将这个单独的分区调度到指定的物理资源。

  ```
  ALTER TABLEGROUP #tgName MOVE PARTITIONS #hotPartitionName TO #dn
  ```

- 如果热点数据突破了机器的单点性能，在PolarDB-X中可以对其采用第二种处理方法，通过以下命令将热点数据散列，更好的支持业务的线性扩展：

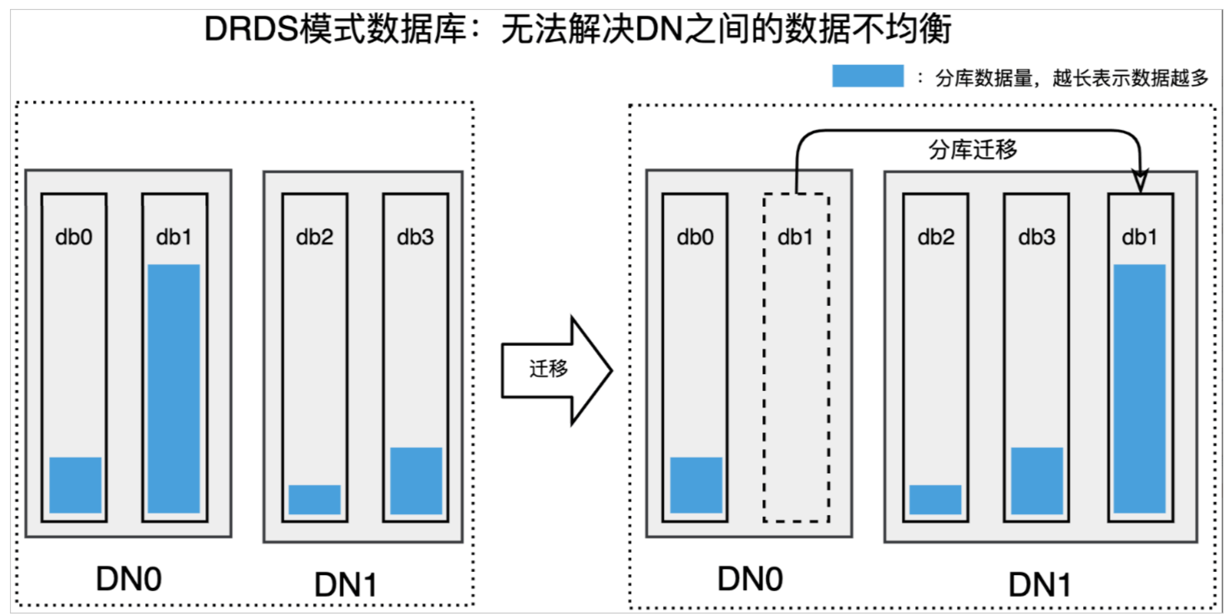
```
ALTER TABLEGROUP #tgName SPLIT INTO PARTITIONS #hotPartitionNamePrefix #N BY HOT VALUE(#keyVal);
```

以上命令可以将keyVal对应的热点数据分裂成N份，并且将分裂后的分区名字加上指定的前缀，然后将新分裂后的分区均匀的调度到不同的数据节点，从而将热点数据在不同数据节点中线性分布，消除数据热点。

分区调度——更灵活的数据均衡

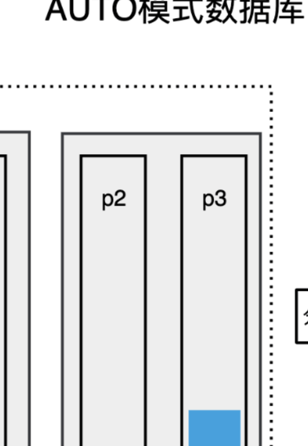
DRDS模式的分库分表使用的是按哈希取模的路由方式，分库与分表位置是强绑定的（即第n号分库必须包含第m号分表，这些库表对应关系不能修改）。这意味着，除非对全表数据进行rehash，否则所有分表都不能被分裂、合并与迁移。

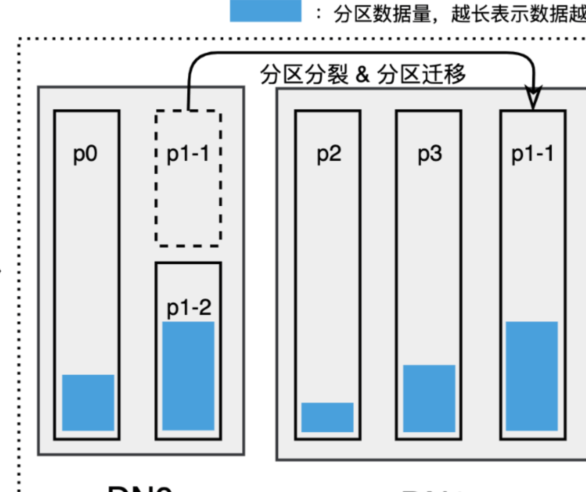
由于分表不能随意的迁移到其他DN（数据节点），调度的基本单位就是一个分库，要调度必须整个分库一起调度，粒度很大，如果某个分库的数据量很大，不管怎么调度，DN之间都无法做到数据均衡（如下图所示）。



但是，AUTO模式的分区表由于采用了一致性哈希的路由算法，它的分区可以更灵活的进行合并、分裂与迁移，分区可以按需调度到指定的DN中，并且不会影响到不相关的分区数据。通过分区的合并分离等操作并结合分区调度，PolarDB-X可以将数据均匀的分布到各个DN中（如下图所示），从而实现DN间的数据均衡。

AUTO模式数据库：能够解决DN之间的数据不均衡





TTL（Time To Live）——自动清理历史数据

某些业务场景下，业务数据增长的很快，并且业务数据的热度随着时间推移会有明显的降低。此时如果数据一直存储在PolarDB-X中，既会占用存储空间，也会降低正常业务查询的效率，此种场景很多业务会选择将历史数据归档后在PolarDB-X中删除，能快速的删除历史数据而不影响到现有业务，一直是客户所追求的。

如果PolarDB-X能通过DDL的方式删除历史数据，将极大的提高清理数据的速度，基于这种思路PolarDB-X在AUTO模式下开发出了TTL的功能，可以快速的删除历史数据，详情请参见什么是TTL功能。

若在建表时使用TTL相关语法，则将创建一张TTL表。如下所示，将t_order表按照gmt_modified列进行时间分区，每个分区间隔一个月，每个分区12个月后过期，提前创建3个分区：

```
CREATE TABLE t_order (
  id bigint NOT NULL AUTO_INCREMENT,
  gmt_modified DATETIME NOT NULL,
  PRIMARY KEY (id, gmt_modified)
)
PARTITION BY HASH(id)
PARTITIONS 16
-- 以下为TTL的相关语法
LOCAL PARTITION BY RANGE (gmt_modified)  -- 按照gmt_modified列划分
INTERVAL 1 MONTH -- 每个月一个分区
EXPIRE AFTER 12 -- 数据在12个月后过期
PRE ALLOCATE 3;  -- 提前3个月创建分区
```

Locality——按需定制数据存储位置

在AUTO模式数据库中提供多个维度的Locality属性，可以手动定制数据的存储位置。

- 可以通过设定Database的Locality属性，指定Database的存储位置，例如：

  ```
  CREATE DATABASE db1 MODE='AUTO' LOCALITY='dn=pxc-xdb-s-pxcexample'
  ```

- 可以通过设定表级的Locality属性，指定表的存储位置。例如，默认情况下单表都存在一个DN上，而通过指定Locality属性，可以指定单表的物理位置。

  ```
  CREATE TABLE tb (a INT, b INT, PRIMARY KEY(a)) LOCALITY='dn=pxc-xdb-s-pxcexample'
  ```

- （即将在5.4.14版本支持）同时也支持分区级的Locality属性。例如，在涉及多个地域的业务场景下，结合AUTO模式下List分区，可以使一个地域的数据存在一个DN上：

```
CREATE TABLE orders_region(
 id int,
 country varchar(64),
 city varchar(64),
 order_time datetime not null)
PARTITION BY LIST COLUMNS(country,city)
(
  PARTITION p1 VALUES IN (('China','Hangzhou'), ('China','Beijing')) LOCALITY='dn=pxc-xdb
-s-pxcexample1',
  PARTITION p2 VALUES IN (('United States','NewYork'),('United States','Chicago')) LOCALI
TY='dn=pxc-xdb-s-pxcexample2',
  PARTITION p3 VALUES IN (('Russian','Moscow')) LOCALITY='dn=pxc-xdb-s-pxcexample3'
);
```

功能对比

与DRDS模式数据库相比，AUTO模式数据库新增了自动分区、热点分裂、分区调度和TTL表等新特性，并在很多其它方面（如分区管理、拆分变更等）做了大量工作以优化分布式体验。

AUTO模式数据库与DRDS模式数据库主要功能对比：

功能项		AUTO模式数据库	DRDS模式数据库
透明分布式	默认主键分区	支持。若建表时不指定分区定义，将自动按主键进行分区。	不支持。
	默认全局二级索引	支持。索引不指定分区列时，将自动索引列分区。	不支持。
	负载均衡调度	支持。	不支持。
	热点散列能力	支持。	不支持。
分区策略	Hash分区 & Key分区	支持。采用一致性哈希的路由算法，并支持热点散列。	支持采用按分区数取模的路由算法，不支持热点散列。
	Range分区 & Range Columns分区	支持，支持热点散列。	不支持。
	List分区 & List Columns分区	支持。	不支持。
向量分区键（使用多个列作为分区键）		支持。分区键支持按向量分区，例如 `PARTITION BY KEY(c1,c2,c3)` 。	不支持。
分区键字符校验集		支持。	不支持。
单表 & 广播表		支持。	支持。
	创建、删除、修改分区	支持。	不支持。

功能项		AUT O模式数据库	DRDS模式数据库
分区管理	分裂、合并分区	支持。	不支持。
	迁移分区	支持。	不支持。
	截断分区	支持。	不支持。
	分区透视	即将上线将支持自动分析热点分区。	不支持。
拆分变更	调整表类型（单表、广播表与分区表互转）	支持。	支持。
	调整分区定义（包括分区数目、分区键类型、分区策略等）	支持。	支持。
弹性扩（缩）容	是否有停写阶段	否。	是（短暂的停写）。
	是否允许其他DDL	是。	否。
Locality	静态隔离	支持，创建库、表和分区时指定物理存储资源。	支持，创建库、表时指定物理存储资源。
	动态隔离	支持，动态调整库表所在的物理存储资源。	不支持。
	是否与扩缩容兼容	是。	否。
分区裁剪	前缀分区裁剪	支持。 例如，按 `PARTITION BY KEY(a,b,c)` 进行分区，向量分区键使用a、b、c这3个列。那么a=100 and b=100或a=100都能命中分区裁剪。	不支持。
	计算表达式常量折叠	支持。 例如，对含计算表达式的条件 `pk = POW(2, 4)` 进行分区裁剪。	不支持。分区键条件要求必须是常量（如pk = 123）；如果分区键是计算表达式如 `pk = POW(2, 4)` ，将执行全表扫描。
	分区路由大小写敏感及忽略行尾空格	支持。 支持通过指定分区键的字符校验集（Collation）来决定分区路由是否需要区分大小写以及是否需要忽略行尾空格。	不支持。分区列不支持使用Collation，Hash算法只支持大小写敏感，不支持忽略行尾空格。

功能项		AUTO模式数据库	DRDS模式数据库
	JOIN计算下推	支持。 支持在分区的分裂、合并与迁移等操作期间，JOIN计算下推不受影响。	支持。
	分区选择	支持。 支持分区选择语法查询特定分区，例如 `SELECT * FROM tb PARTITIONS (p1)` 。	不支持。
TTL（分区的生命周期管理）		支持。	不支持。

性能对比

由于DRDS模式分库分表与AUTO模式分区表使用了不同的路由算法，为了评估这两种路由算法的性能差异，将使用Sysbench对PolarDB-X进行基准测试，观察它们在不同的Sysbench测试场景下的吞吐（单位：QPS）差异。

测试环境

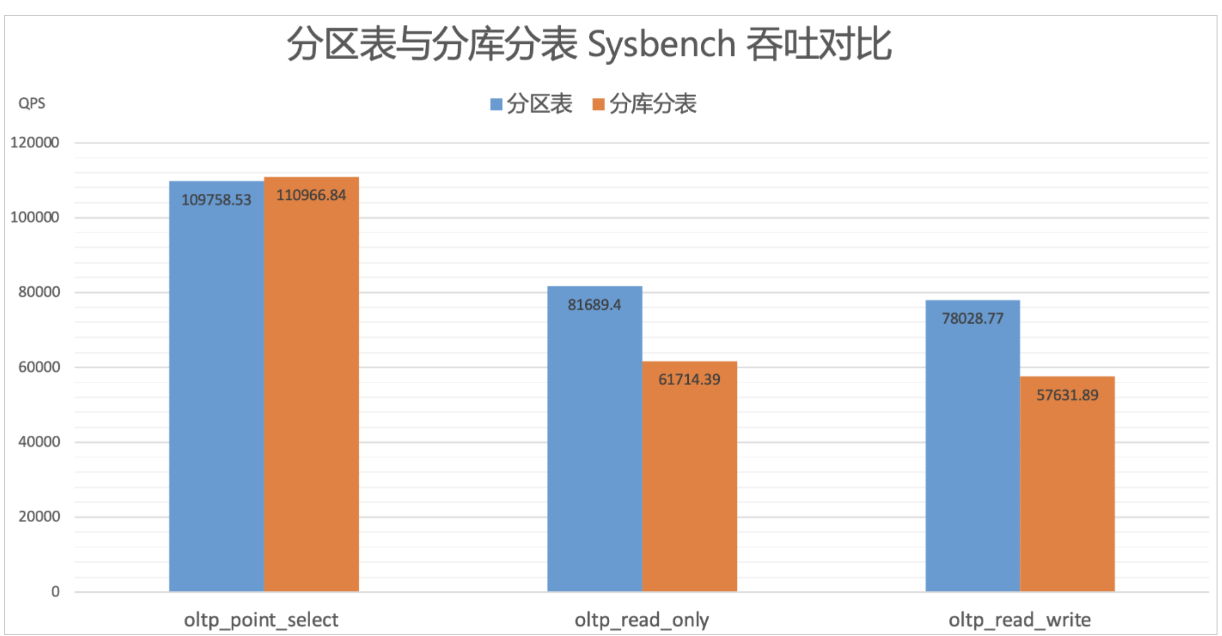
- PolarDB-X实例规格：polarx.x4.2xlarge.2e
  - CN (16C64G) × 2
  - DN (16C64G) × 2
- 版本：5.4.13-16415631
- 分区表和分库分表配置：
  - 分区表：
    - 32个分区
    - 分区语句：partition by hash(id) partitions 32
    - 表数据总量：16000W
  - 分库分表：
    - 32个物理分表
    - 分库分表语句：dbpartition by hash(id) tbpartition by hash(id) tbpartition 2
    - 表数据总量：16000W

测试场景

Sysbench细分场景说明：

- oltp_point_select：仅含分区键的单点等值查询。
- oltp_read_only：事务中同时混合分区键的单点查询与小范围查询（例如Between）。
- oltp_read_write：事务中同时混合分区键的单点与小范围的查询与写入。

测试结果



分析以上测试结果，可以得出以下结论：

- 分区表使用的一致性HASH路由算法虽然比原来分库分表中按HASH取模的路由算法更为复杂，但在oltp_point_select场景中吞吐并没有下降太多，基本与原来的持平。
- 在oltp_read_only&oltp_read_write场景中，由于这些场景会出现小范围的查询，SQL的查询条件表达式会比oltp_point_select复杂，得益于分区表的裁剪优化，整体吞吐比原来提升约33%。

4.数据导入与导出

> **说明**
>
> 您可以通过阿里云官方网站云起实验室的在线实验如何将PolarDB-X与大数据等系统互通，体验PolarDB-X通过全局Binlog与其他系统互联互通的能力。

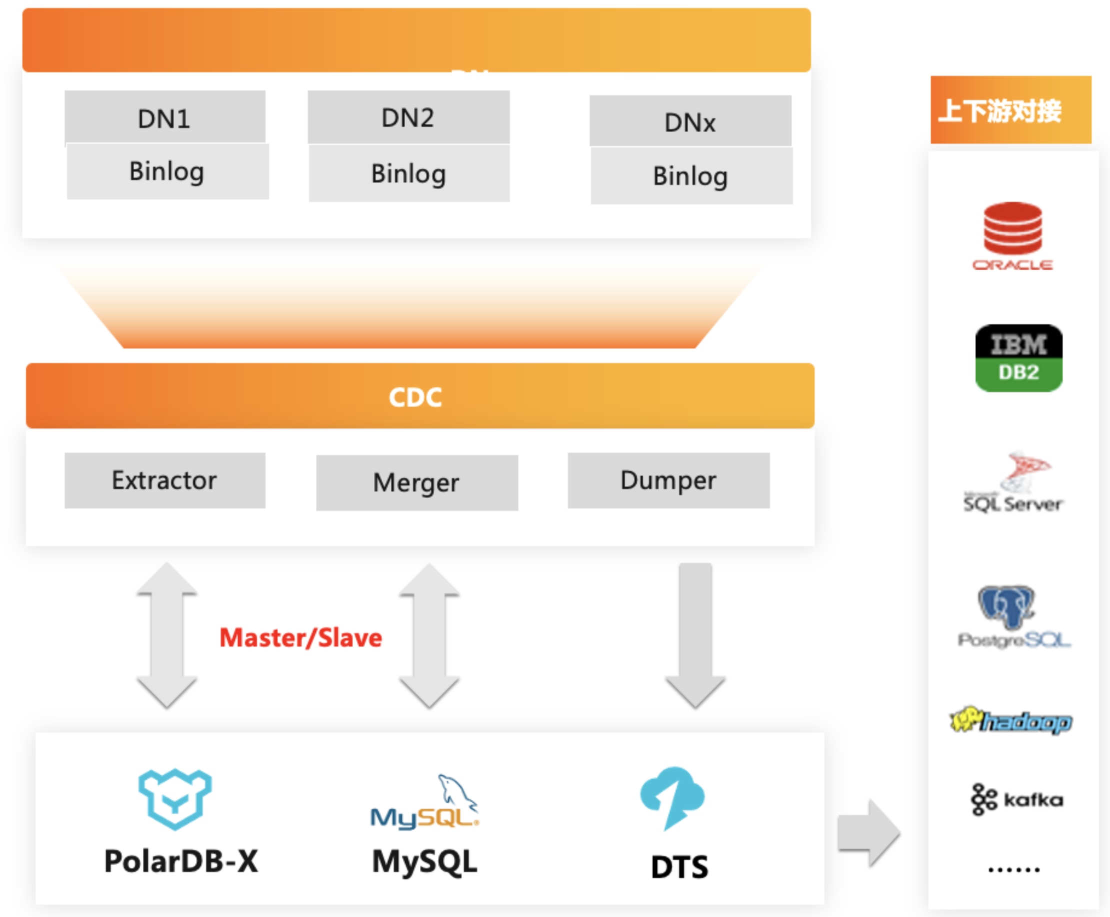
4.1. PolarDB-X CDC

PolarDB-X的日志节点（CDC, Change Data Capture）是由两个节点组成的，这两个节点互为主备，共同组成了PolarDB-X的这一个子集群。 在使用PolarDB-X数据库的时候，CDC是一个可选的能力。也就是说，在部署一个PolarDB-X实例的时候，如果没有数据导入或者导出这种需求，是可以不去选装CDC的，由此也可以节省一些成本。

核心特性

- **数据流入**：可以上游对接兼容MySQL的任何一个数据库，只要它有Binlog的输出能力，比如说原生的MySQL、RDS、PolarDB-X都可以作为数据流入的源。
- **生态兼容**：数据流入和数据流出是与MySQL完全兼容的。如果以前用一些中间件或者一些技术架构是与MySQL对接的，后来想要用PolarDB-X的话，之前所有遗留的架构或者产品，都可以无缝迁移到PolarDB-X数据库上，操作成本非常低。
- **数据流出**：由于提供了全局Binlog，下游只要可以消费原生MySQL的任何中间件或者任何的数据库产品，都可以来对接PolarDB-X的数据流出能力。

4.2. Global Binlog



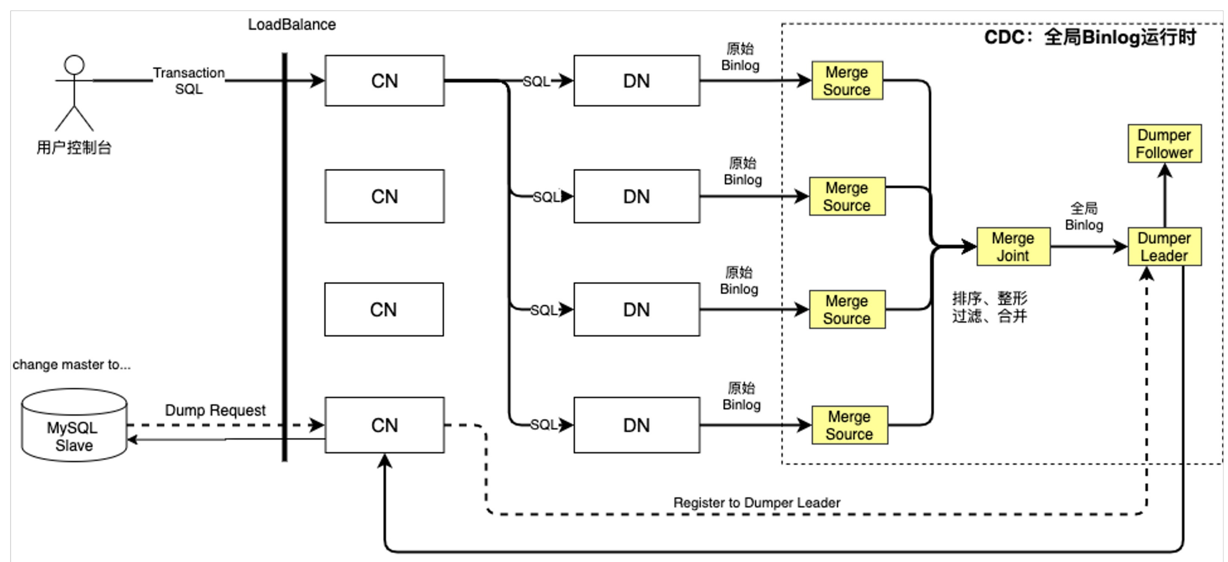
PolarDB-X会有多个DN节点，每个DN上有原始的物理Binlog（也称为原始Binlog）。CDC的全局Binlog组件有一个流水线，核心的阶段包括：Extractor解析DN的Binlog，解析完之后会进行一些排序操作；Merger把所有DN上经过排序的Binlog做全局合并，然后排序输出到Dumper；Dumper再对接下游（可以是PolarDB-X、MySQL、DTS等等），来提供输出能力。

核心特性

- 兼容事务（分布式事务全局排序）：全局Binlog中所有通过CN节点执行的事务操作，在全局Binlog中是可以完全还原成一个完整的事务的。基于TraceId、TSO信息对Binlog全局排序。
- 兼容分布式DDL：在执行DDL的时候，每个DN节点上都有对应的物理DDL，可以做到透明输出，屏蔽掉内部的一些DDL变更的细节。
- 兼容分布式扩缩容：内部执行DN节点的增加或删除操作时，对于下游而言是一个完全透明的操作，下游可以完全感知不到内部的一些变更细节。

演示视频：开源PolarDB-X升级发布会（01:24:51开始）

内部架构



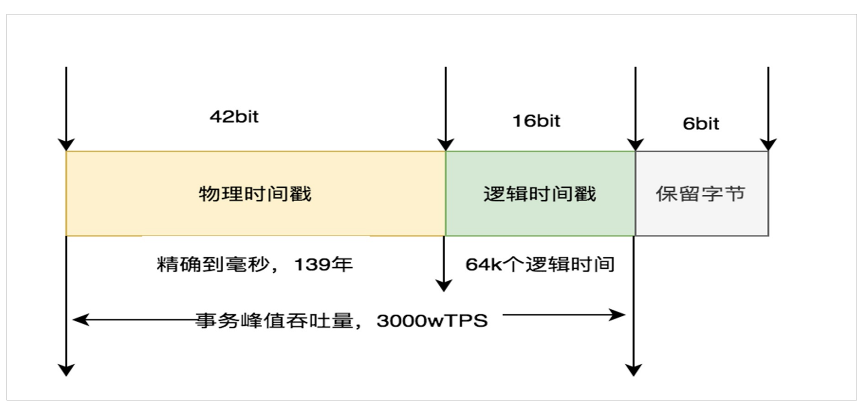
当从控制台执行一个事务时，首先到了CN节点，CN节点会执行事务里面各种DML的SQL，这些SQL最终会下发到所有DN节点执行。DN节点执行完SQL以后，等用户最终提交时，DN节点就会产生物理Binlog（原始Binlog）。

右侧虚线部分是整个全局Binlog的内部总体结构。对接每个DN，就会有一个Merge Source。Merge Source内部会做一系列数据处理。Merge Joint做全局排序处理。Dump Leader是主节点。如果主节点挂掉，Dump Follower可以接管并成为新的Leader。

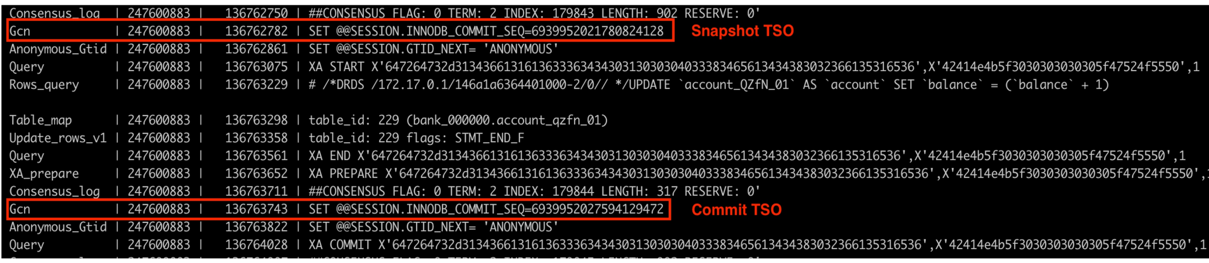
当下游执行命令时，CN就会收到命令，然后给CDC的Dumper发送对应的请求，最终Dumper的Leader会把数据源源不断推送给消费端进行消费。

事务排序和合并

TSO格式



TSO长64bit，单调递增。当在PolarDB-X上执行完一个事务提交的时候，在DN会产生Binlog，TSO会被持久化到DN的Binlog中，以名为GCN的event来持久化。



```
Consensus_log  | 247600883 |   136762750 | ##CONSENSUS FLAG: 0 TERM: 2 INDEX: 179843 LENGTH: 902 RESERVE: 0'
Gcn            | 247600883 |   136762782 | SET @@SESSION.INNODB_COMMIT_SEQ=6939952021780824128   Snapshot TSO
Anonymous_Gtid | 247600883 |   136762861 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'
Query          | 247600883 |   136763075 | XA START X'647264732d313436613161363336343430313030304033383465613434383032366135316536',X'42414e4b5f3030303030305f47524f5550',1
Rows_query     | 247600883 |   136763229 | # /*DRDS /172.17.0.1/146a1a6364401000-2/0// */UPDATE `account_QZfN_01` AS `account` SET `balance` = (`balance` + 1)

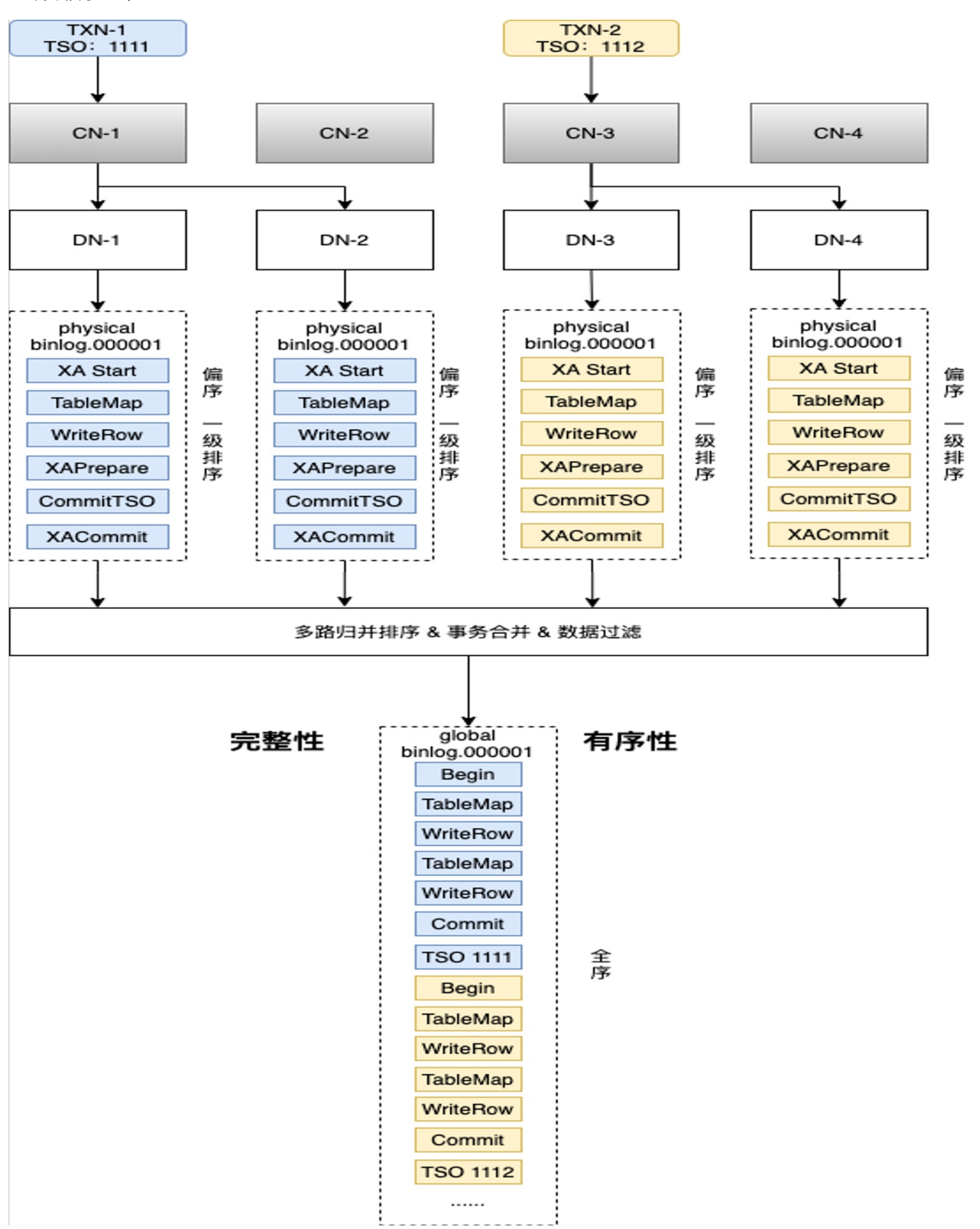
Table_map      | 247600883 |   136763298 | table_id: 229 (bank_000000.account_qzfn_01)
Update_rows_v1 | 247600883 |   136763358 | table_id: 229 flags: STMT_END_F
Query          | 247600883 |   136763561 | XA END X'647264732d313436613161363336343430313030304033383465613434383032366135316536',X'42414e4b5f3030303030305f47524f5550',1
XA_prepare     | 247600883 |   136763652 | XA PREPARE X'647264732d313436613161363336343430313030304033383465613434383032366135316536',X'42414e4b5f3030303030305f47524f5550',1
Consensus_log  | 247600883 |   136763711 | ##CONSENSUS FLAG: 0 TERM: 2 INDEX: 179844 LENGTH: 317 RESERVE: 0'
Gcn            | 247600883 |   136763743 | SET @@SESSION.INNODB_COMMIT_SEQ=6939952027594129472   Commit TSO
Anonymous_Gtid | 247600883 |   136763822 | SET @@SESSION.GTID_NEXT= 'ANONYMOUS'
Query          | 247600883 |   136764028 | XA COMMIT X'647264732d313436613161363336343430313030304033383465613434383032366135316536',X'42414e4b5f3030303030305f47524f5550',1
```

如上图所示，Commit TSO会作为CDC内部对事务进行全局排序的依据。

说明

有关全局时间戳的详情，请参见《PolarDB-X全局时间戳服务的设计》。

总体排序过程



事务提交完成后，在每一个对应DN的Merge Source内部做一级排序，保证每个Merge Source输出的都是局部有序的序列。然后再按照TSO做全局排序，保证整个事务是全局有序的。同时，还会做事务合并，并将分布式事务中的XA Start等事件去掉。最终呈现出MySQL单机事务的形态。

事务空洞

定义：对于一个2PC事务 (Prepare+Commit)，如果在其Prepare和Commit之间，穿插了其它事务的Prepare或Commit，我们称这些事务间存在空洞。由于多个分布式事务时并发提交的，所以空洞现象不可避免。

举例：假设有事务T1和T2，Prepare阶段简称为P，Commit阶段简称为C

有空洞：P1 P2 C1 C2、P1 P2 C2 C1、P2 P1 C1 C2、P2 P1 C2 C1

无空洞：P1 C1 P2 C2、P2 C2 P1 C1

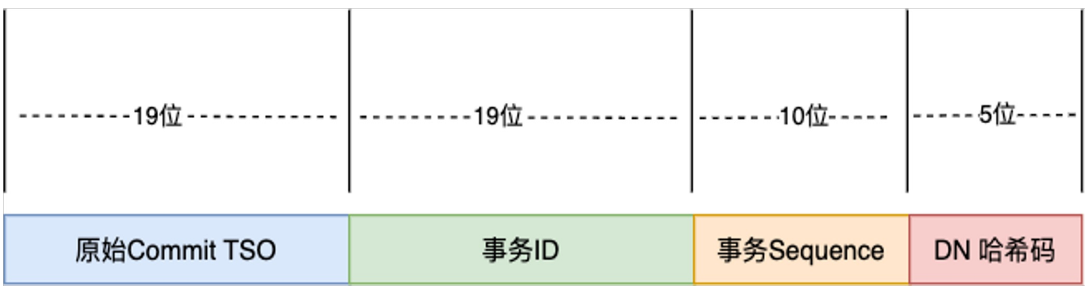
排序

一级排序：解决事务空洞场景下TSO天然乱序问题

二级排序：多路归并排序，按TSO有序输出，并做事务合并

虚拟TSO

排序完成后，输出到全局Binlog的内容，与原生MySQL会有一些差异。PolarDB-X的全局Binlog，每一个Commit后面会有一个CTS（Commit Transaction Snapshot）。CTS是一串数字，是CDC基于64位原始TSO生成的虚拟TSO。该虚拟TSO的格式如下：



- 原始TSO：高19位记录物理Binlog中的原始TSO，如果是非TSO事务或TSO一阶段提交事务，则复用maxCTS
- 事务ID：后续19位记录XA事务的transactionID，如果是单机事务，则复用maxTransactionId
- 事务Sequence：后续10位记录事务的序列号，用于区分复用同一maxTransactionId的多个单机事务，按序自增
- DN哈希码：后续6位用来标识局部事务所属的DN节点

为什么会有虚拟TSO呢？因为PolarDB-X在事务提交的时候，会有很多优化，比如：如果发现事务里只在一个DN上面有操作，就优化为一阶段提交；如果没有开启事务，就是auto commit。在这种场景下，Binlog里其实是没有记录TSO的。所以会在整个排序的过程中，进行虚拟构造来保证全局有序。

```
| binlog.000001 | 15114100 | Query      | 1465637011 |    15114156 | BEGIN                                              |
| binlog.000001 | 15114156 | Rows_query | 1465637011 |    15114225 | /*DRDS /192.168.0.226/146b4abd6fc01000/0// */      |
| binlog.000001 | 15114225 | Table_map  | 1465637011 |    15114280 | table_id: 7 (ddltest.t1)                           |
| binlog.000001 | 15114280 | Write_rows | 1465637011 |    15114334 | table_id: 7 flags: STMT_END_F                      |
| binlog.000001 | 15114334 | Xid        | 1465637011 |    15114365 | COMMIT /* xid=29734 */                             |
| binlog.000001 | 15114365 | Rows_query | 1465637011 |    15114448 | CTS::694028663993139206414713518554171351050000000001416581 |
```

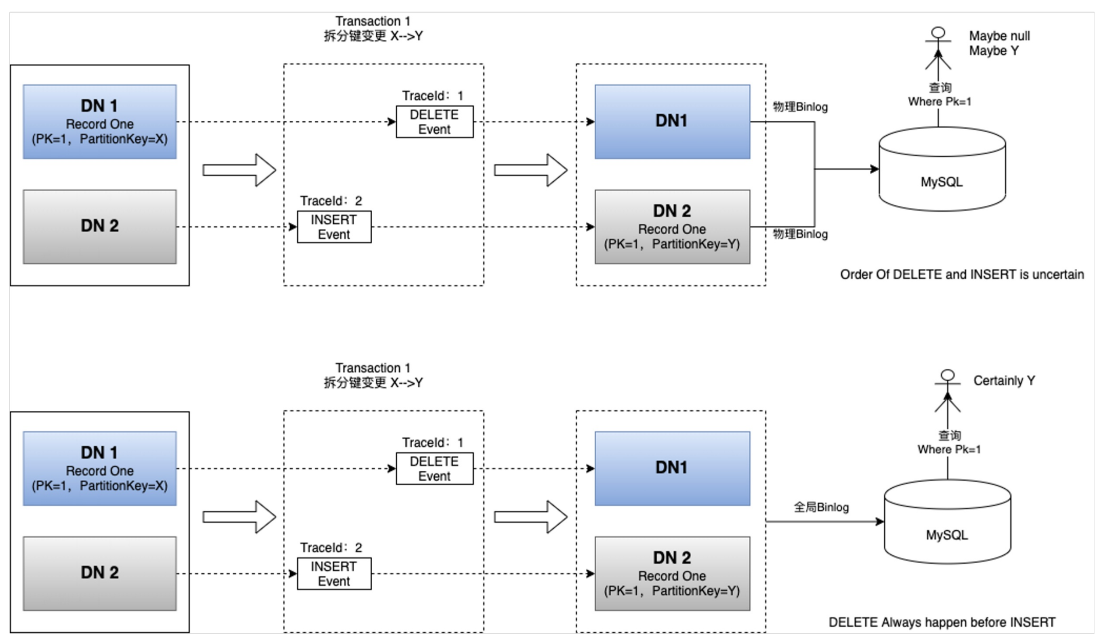
> 说明
>
> 详情请参阅《PolarDB-X全局Binlog解读之理论篇》。

拆分键变更

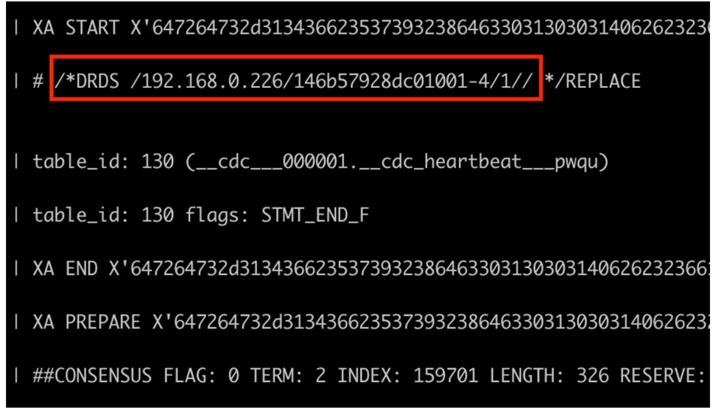
任何一个分布式数据库都有分区，数据分布在各个分区中。但分布完后，会有分区变更（比如：有一条数据之前在分区1；对该数据做变更后，起分区键发生变化，该条数据就可能从分区1编导分区2），该行为称为“数据漂移”。 PolarDB-X设计了一个TraceId（有序自增的序列号），来实现事务内event的排序。

总结：PolarDB-X依靠TSO来实现事务之间的排序，依靠TraceId来保证事务内不同操作之间的顺序，从而保证事务内的有序性，最终实现数据的一致性。

拆分键变更导致数据漂移的示例



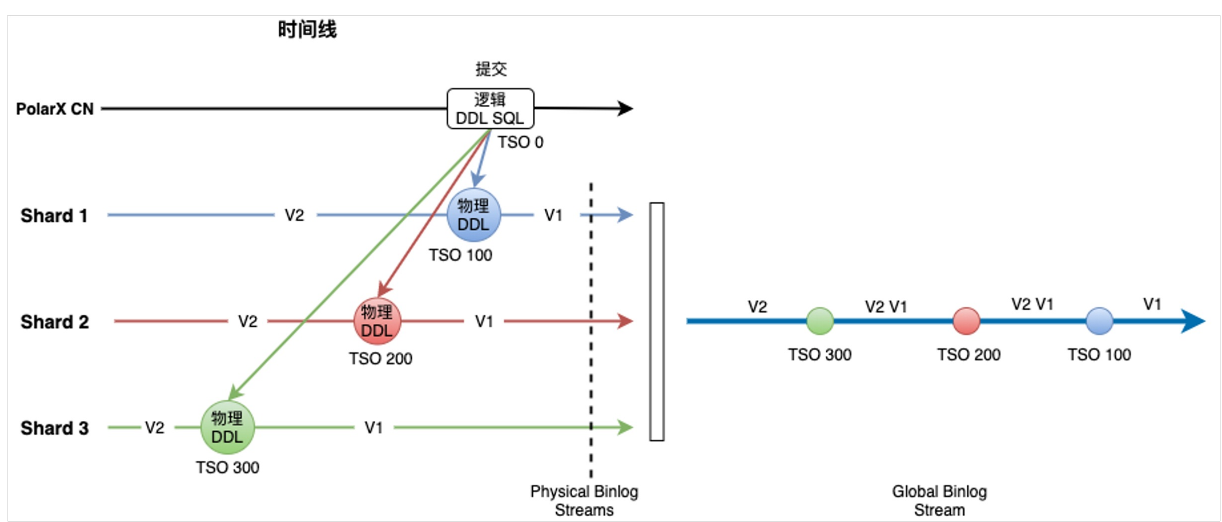
TraceId的组成



Online DDL和Event Reformat

对于分布式数据库而言，DDL变更是一个非常复杂的操作。一方面是因为用户从控制台提交完一个DDL之后，数据库内核内部在执行用户提交的DDL时，在众多个节点之间执行的是异步操作（因为分布式场景没有办法做到原子性）。 另一方面，在DDL变更过程中，PolarDB-X所有的DDL都是Online DDL（Online DDL意味着在执行DDL变更过程中，同时不能阻塞DML的操作），也就是在变更过程中，DML的流量还在持续不断地操作，这样就会生成多版本的数据。

PolarDB-X在DDL变更过程中，受其分布式特性的影响，不同分片（Shard）的Schema变更无法做到完全同步，即同一个时刻，数据库的Schema元数据会存在两个版本，例如加列操作，新列对部分分片已经可见，对部分分片还未可见，此时会同时产生两个版本的数据。

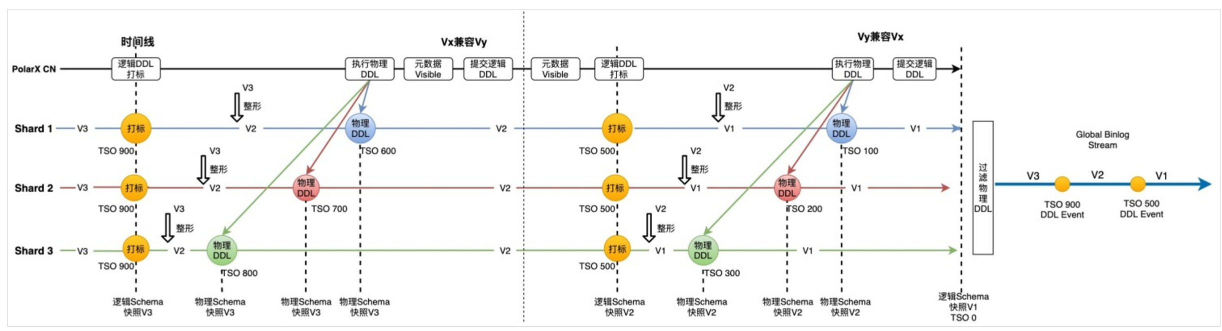


为解决这一问题，PolarDB-X引入一套方案：维护Schema版本快照的历史。在CDC内部会对逻辑表维护一套快照，然后对每一个DN上所有的物理表也维护一份快照。在消费各个DN的Binlog时，在某一个点拿到了DN的数据后，就会去查找当前在该时间线上逻辑Schema的形态以及对应的物理Binlog的Schema形态。如果两个Schema一致，就可以对下游输出；如果Schema不一致，就要以逻辑Schema为依据来进行数据整形。

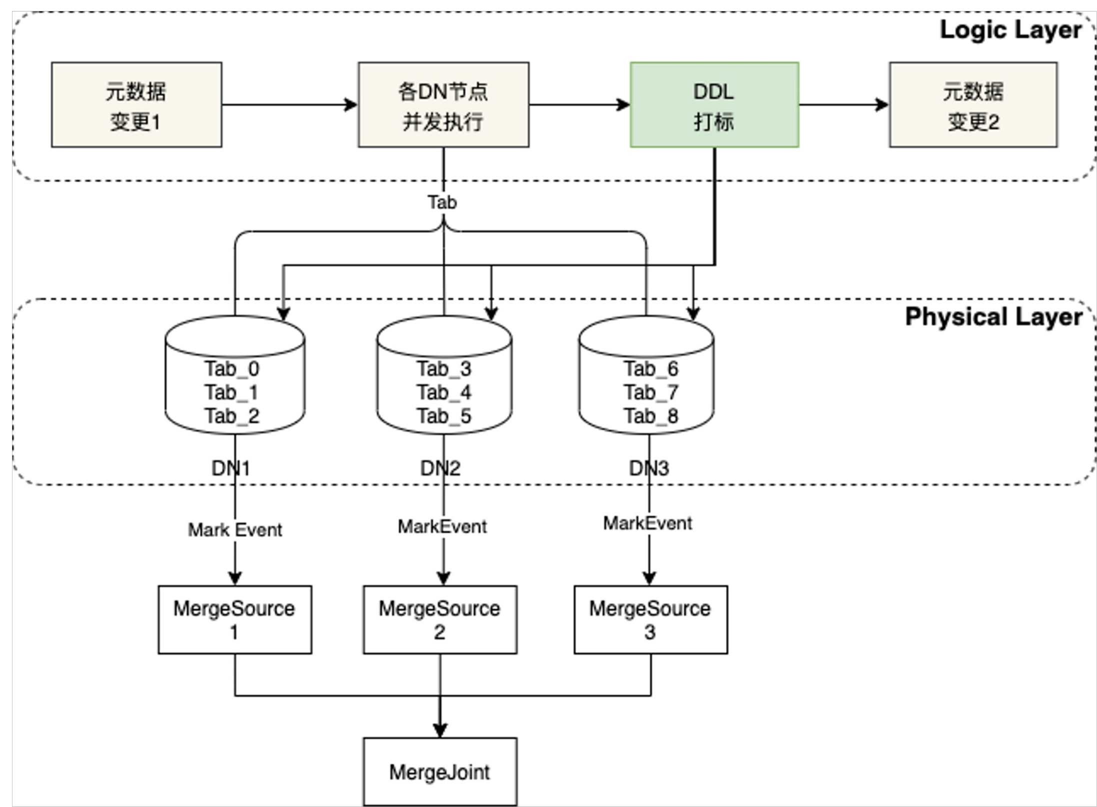
DDL版本：一个时刻的Schema结构（包含表结构等）称为一个版本，DDL前的版本称为Vx，DDL操作后的版本称为Vy。

Vy兼容Vx的DDL类型：比如加列，列的长度变长（例如varchar(128)→varchar(1024)），增加列的精度（例如double(10)→double(20)）。

Vx兼容Vy的DDL类型：比如删除列，列的长度变短（例如varchar(1024)→varchar(128)），删表。



DDL变更的总体操作流程



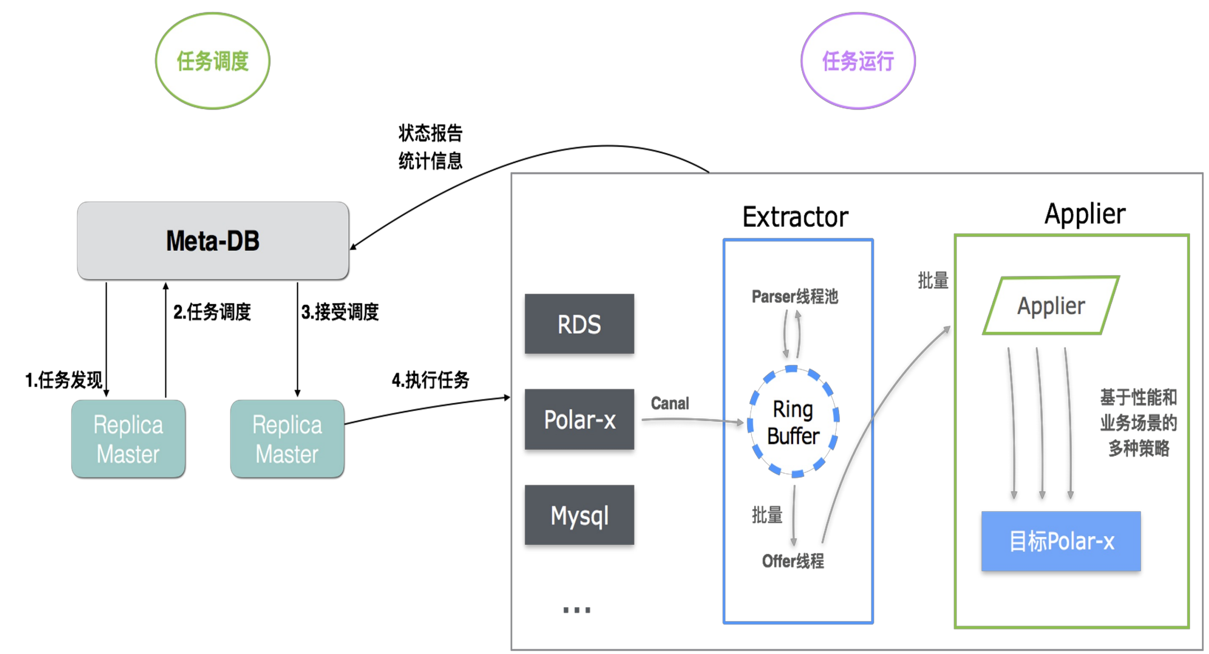
说明

详情请参阅《PolarDB-X全局Binlog解读之DDL》。

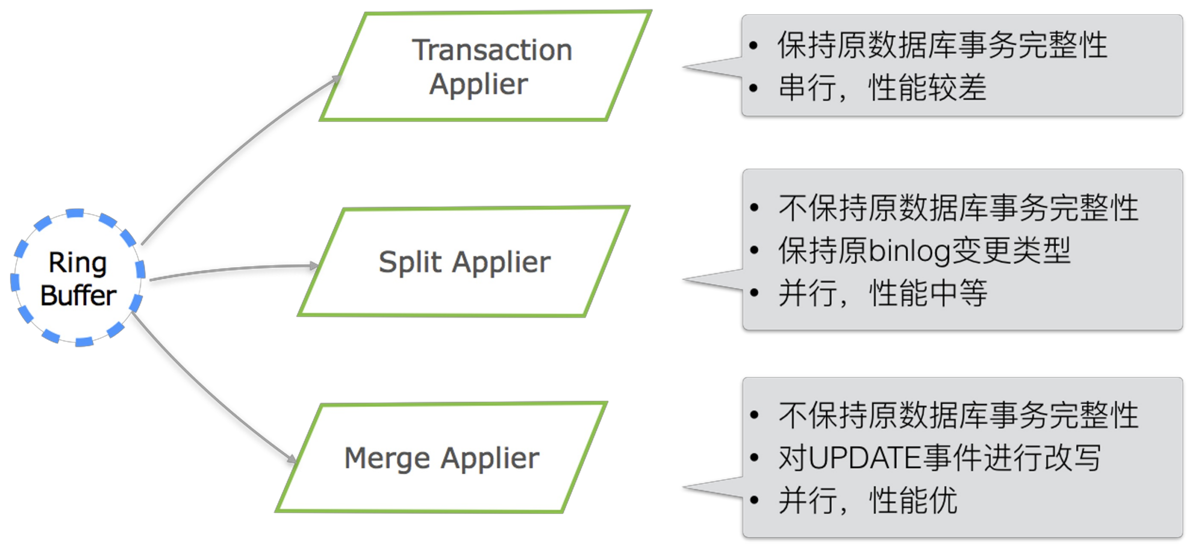
4.3. PolarDB-X Replica

Replica可以实现数据的流入，可以与MySQL之间实现双向同步，也可以和PolarDB-X之间实现双向同步。

Replica的架构如下图所示。



写入模式有以下三种：



- 保持原数据库事务完整性的写入，即源端MySQL Binlog的事务是怎样的，在消费时就串行消费。这样在写入时能够保证事务的完整性。但这种模式性能较差。
- 牺牲事务，按照唯一键进行Hash，然后并发向目标端写入。这种模式性能较好。
- 按唯一键进行Hash，并合并相同事件类型。比如：在Hash完成后会组装队列，有一条数据ID唯一的数据有连续10次的更新，内部会将这10次更新合并成1次，这样在向目标端写入时，就不会写入10次而是只写入1次。这种模式可以大大提升性能。

演示视频：开源PolarDB-X升级发布会（01:30:46开始）。该视频演示了PolarDB-X作为MySQL的Slave进行数据消费，并且验证了数据一致性。

4.4. 相关资料

- PolarDB-X全局Binlog解读
- PolarDB-X全局Binlog解读之 DDL

- PolarDB-X全局Binlog解读之理论篇
- PolarDB-X 如何兼容MySQL Binlog协议和参数
- PolarDB-X 源码解读（三）：CDC 代码结构
- PolarDB-X 源码解读（八）：Global Binlog 的一生

5.PolarDB-X集群运维

5.1. 扩容和缩容

PolarDB-X是从阿里巴巴淘宝业务起家，之前主要面向互联网业务。互联网业务的特点之一就是通常会有高并发和海量数据的存储需求。通常，随着业务的发展，单机数据库存在性能瓶颈，例如并发能力有限或者存储空间达到了瓶颈。在这种情况下，通常需要对数据库进行扩展。

传统的扩展方式目前有两种，一种是垂直扩展，也就是升降配（可以理解为在已有的机器中增加更多的资源来提高数据库的处理能力。例如，对于部署在物理机上的数据库，可能采用更多核的CPU或者采用更大或更快地内存条的方式，来提升数据库的性能）。对于云数据库而言，由于底层资源已经通过虚拟化的技术进行了池化，这种情况下，通常只需要修改相关资源的一些配置，就可以实现快速垂直的升降配，提升数据库性能。这种方式的优势是扩展过程基本不需要迁移数据，但问题在于性能始终局限于单机（单机性能有上限）。

第二种方式时水平扩展，即扩缩容。这种方式相当于加入了更多的机器来解决数据库的扩展性问题，不再局限于单机的性能，但需要对数据进行一定的迁移来做到水平扩展。同时，这种方式对于数据库的水平扩展能力有更多的要求，需要数据库能够具备很好的水平线性扩展能力，这样才能将新增加进来的机器的性能充分发挥出来。

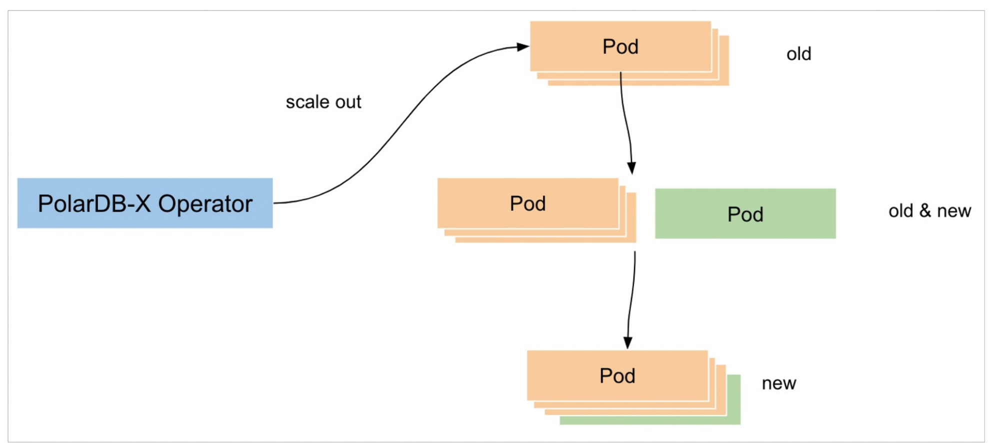
说明

- 本节实验操作部分主要通过阿里云官方网站的云起实验室进行，详情可登录阿里云官方网站，访问如何对PolarDB-X集群做动态扩缩容了解。
- 了解更多有关PolarDB-X的水平扩展，请参见谈谈PolarDB-X 的水平扩展。

扩容的基本原理

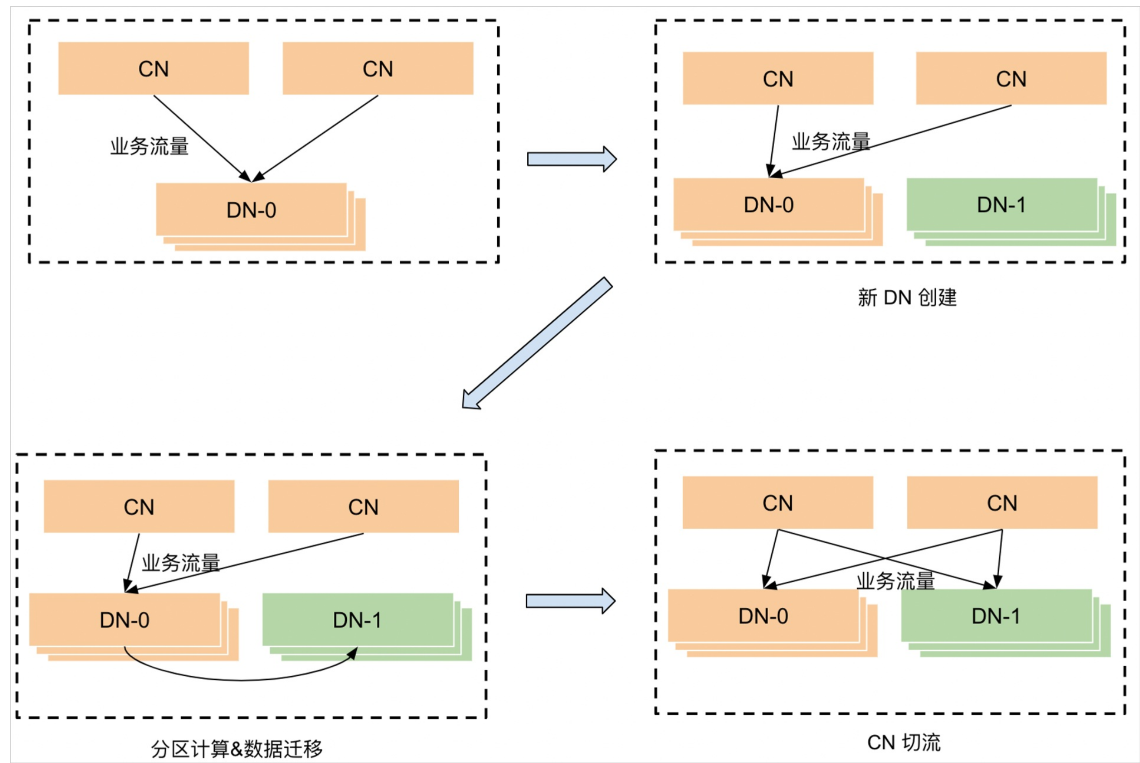
对于扩容而言，可以分为两部分来看：

第一部分是CD和CDC这种无状态的组件。举例来说，假设现在有3个CN的节点，需要扩容到4个，这时只需要新建1个新的CN节点，将它加入到CN集群中，就完成了扩容操作。



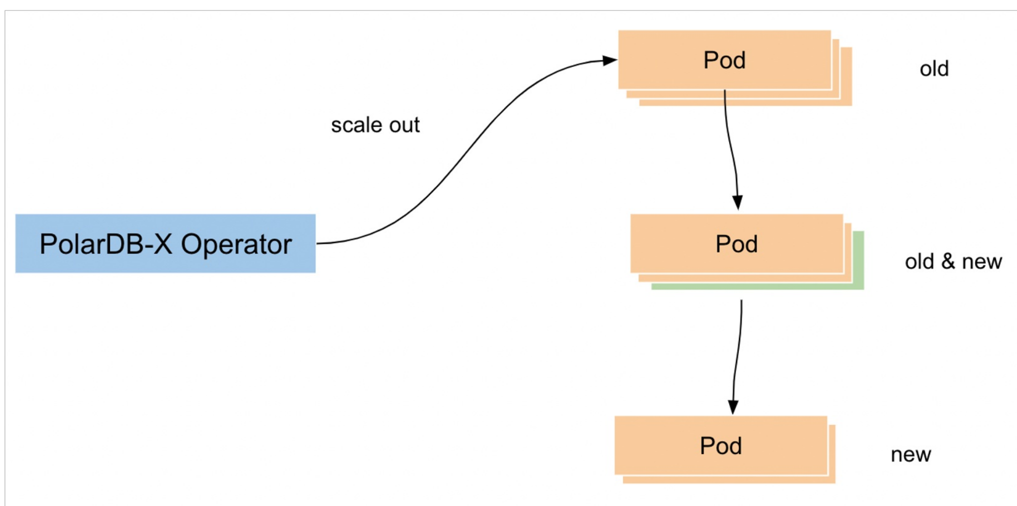
另一部分是DN这种有状态的组件。 首先需要创建一个新的DN（原来只有一个DN0在工作，现在新创建一个DN1）。DN1创建完成后，需要对DN0上已有的数据的分区进行计算，以便了解DN0上的哪些数据是需要移动到新建的DN1上去，同时需要将这一部分数据迁移到DN1上（在迁移过程中，所有CN的访问流量还是访问原来的DN0，DN1不参与到实际的业务请求中）。等到DN1所有的数据都迁移完成并达到与DN0数据完全一致的状态，就发生CN切流的过程，将CN的数据请求流量引到DN1上，达到新的数据均衡状态。

由此，整个扩容流程完成。

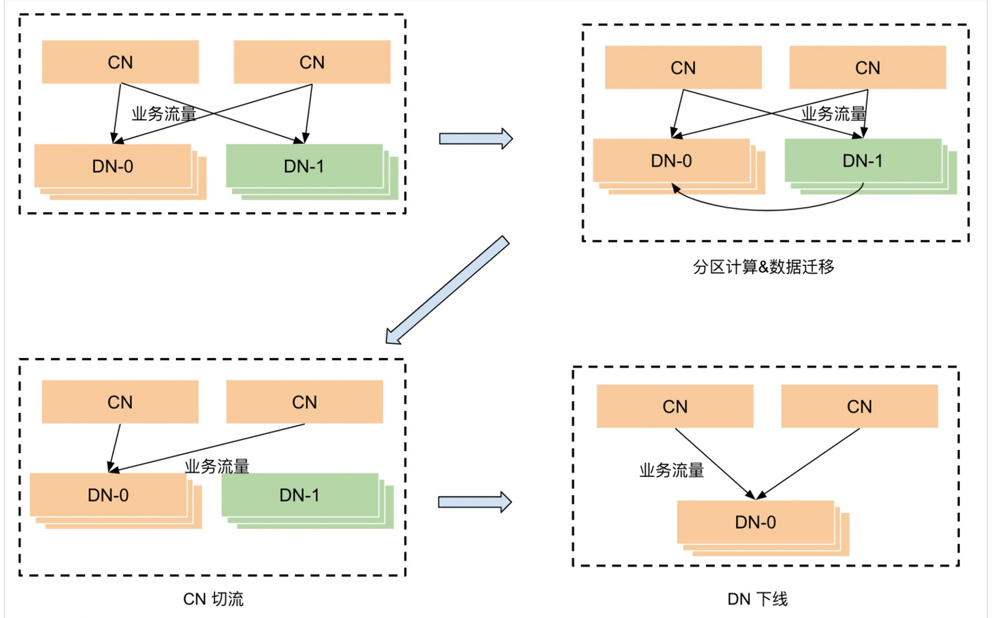


缩容的基本原理

对于CN和CDC这种无状态的组件，缩容原理与扩容类似。举例来说，假设原有3个CN节点，现在需要缩容到2个。只需要对多余的1个CN的节点执行下线的操作，就可以达到CN缩容的效果。



对于DN这种有状态的组件而言，假设原有DN0和DN1两个DN节点，现在需要将DN1下线，仅保留DN0这一个DN节点。首先，需要将DN1上的数据全部迁移到已有的DN0上（该过程涉及分区的计算和数据的迁移）。全部迁移完成后，将执行切流，将所有访问DN1的流量全部切换到访问DN0上。切流完成后，当DN1上没有任何业务流量后，就可以对DN1执行下线操作。由此达到缩容的目的。



5.1.1. 第1步：安装环境

本节将介绍如何安装Docker、kubectl、minikube和Helm3。

操作步骤

1. 安装Docker。
   i. 执行如下命令，安装Docker。

   ```
   curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun
   ```

   ii. 执行如下命令，启动Docker。

   ```
   systemctl start docker
   ```

2. 安装kubectl。
   i. 执行如下命令，下载kubectl文件。

   ```
   curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
   ```

   ii. 执行如下命令，赋予可执行权限。

   ```
   chmod +x ./kubectl
   ```

   iii. 执行如下命令，移动到系统目录。

   ```
   mv ./kubectl /usr/local/bin/kubectl
   ```

3. 执行如下命令，下载并安装minikube。

   ```
   curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
   sudo install minikube-linux-amd64 /usr/local/bin/minikube
   ```

4. 安装Helm3。

i. 执行如下命令，下载Helm3。

```
wget https://labfileapp.oss-cn-hangzhou.aliyuncs.com/helm-v3.9.0-linux-amd64.tar.gz
```

ii. 执行如下命令，解压Helm3。

```
tar -zxvf helm-v3.9.0-linux-amd64.tar.gz
```

iii. 执行如下命令，移动到系统目录。

```
mv linux-amd64/helm /usr/local/bin/helm
```

5. 安装MySQL。

```
yum install mysql -y
```

5.1.2. 第2步：使用PolarDB-X Operator安装PolarDB-X

本节将介绍如何创建一个简单的Kubernetes集群并部署PolarDB-X Operator，然后使用Operator部署一个完整的PolarDB-X集群。

操作步骤

> **说明**
>
> 有关通过Kubernetes部署PolarDB-X集群的详情，请参见通过Kubernetes部署集群。

1. 使用minikube创建Kubernetes集群。

> **说明**
>
> minikube是由社区维护的用于快速创建Kubernetes测试集群的工具，适合测试和学习Kubernetes。使用minikube创建的Kubernetes集群可以运行在容器或是虚拟机中。
>
> 本实验场景以CentOS 8.5上创建Kubernetes为例。如果您使用其他操作系统部署minikube，例如macOS或Windows，部分步骤可能略有不同。

i. 执行如下命令，新建账号galaxykube，并将galaxykube加入docker组中。

> **说明**
>
> minikube要求使用非root账号进行部署，所以您需要新建一个账号。

```
useradd -ms /bin/bash galaxykube
usermod -aG docker galaxykube
```

ii. 执行如下命令，切换到账号galaxykube。

```
su galaxykube
```

iii. 执行如下命令，进入到home/galaxykube目录。

```
cd
```

iv. 执行如下命令，启动一个minikube。

```
minikube start --cpus 4 --memory 12288 --image-mirror-country cn --registry-mirror=
https://docker.mirrors.sjtug.sjtu.edu.cn
```

> **说明**
>
> 这里我们使用了阿里云的minikube镜像源以及USTC提供的docker镜像源来加速镜像的拉取。

返回结果如下，表示minikube已经正常运行，minikube将自动设置kubectl的配置文件。

```
[galaxykube[illegible]ot]$ minikube start --cpus 4 --memory 7960 --image-mirror-country cn --registry-mirror=https://docker.mirrors.ustc.edu.cn
😄 minikube v1.25.2 on Centos 8.5.2111 (amd64)
✨ Automatically selected the docker driver
❗ Your cgroup does not allow setting memory.
    ▪ More information: https://docs.docker.com/engine/install/linux-postinstall/#your-kernel-does-not-support-cgroup-swap-limit-capabilities
✅ Using image repository registry.cn-hangzhou.aliyuncs.com/google_containers
👍 Starting control plane node minikube in cluster minikube
🚜 Pulling base image ...
    > registry.cn-hangzhou.aliyun...: 379.06 MiB / 379.06 MiB  100.00% 6.55 MiB
🔥 Creating docker container (CPUs=4, Memory=7960MB) ...
🐳 Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
    > kubelet.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubectl.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubeadm.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
    > kubeadm: 43.12 MiB / 43.12 MiB [--------------] 100.00% 4.71 MiB p/s 9.4s
    > kubectl: 44.43 MiB / 44.43 MiB [---------------] 100.00% 4.40 MiB p/s 10s
    > kubelet: 118.75 MiB / 118.75 MiB [-------------] 100.00% 7.43 MiB p/s 16s
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎 Verifying Kubernetes components...
    ▪ Using image registry.cn-hangzhou.aliyuncs.com/google_containers/storage-provisioner:v5
🌟 Enabled addons: storage-provisioner, default-storageclass
🏄 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

v. 执行如下命令，使用kubectl查看集群信息。

```
kubectl cluster-info
```

返回如下结果，您可以查看到集群相关信息。

```
[galaxyku[illegible]x9dZ ~]$ kubectl cluster-info
Kubernetes control plane is running at https://19[illegible]8443
CoreDNS is running at https://[illegible]/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

2. 部署 PolarDB-X Operator。

i. 执行如下命令，创建一个名为polardbx-operator-system的命名空间。

```
kubectl create namespace polardbx-operator-system
```

ii. 执行如下命令，安装PolarDB-X Operator。

```
helm repo add polardbx https://polardbx-charts.oss-cn-beijing.aliyuncs.com
helm install --namespace polardbx-operator-system polardbx-operator polardbx/polard
bx-operator
```

iii. 执行如下命令，查看PolarDB-X Operator组件的运行情况。

```
kubectl get pods --namespace polardbx-operator-system
```

返回结果如下，请您耐心等待2分钟，等待所有组件都进入Running状态，表示PolarDB-X Operator已经安装完成。

```
[galaxykub[illegible]tZ ~]$ kubectl get pods --namespace polardbx-operator-system
NAME                                        READY   STATUS    RESTARTS   AGE
polardbx-controller-manager-7978bc7bd5-sr5qf   1/1     Running   0          2m11s
polardbx-hpfs-br7wh                          1/1     Running   0          2m11s
polardbx-tools-updater-f2zpw                 1/1     Running   0          2m11s
```

3. 部署 PolarDB-X 集群。

i. 执行如下命令，创建polardb-x.yaml。

```
vim polardb-x.yaml
```

ii. 按i键进入编辑模式，将如下代码复制到文件中，然后按ECS退出编辑模式，输入:wq后按下Enter键保存并退出。

```
apiVersion: polardbx.aliyun.com/v1
kind: PolarDBXCluster
metadata:
  name: polardb-x
spec:
  config:
    dn:
      mycnfOverwrite: |-
        print_gtid_info_during_recovery=1
        gtid_mode = ON
        enforce-gtid-consistency = 1
        recovery_apply_binlog=on
        slave_exec_mode=SMART
  topology:
    nodes:
      cdc:
        replicas: 1
        template:
          resources:
            limits:
              cpu: "1"
              memory: 1Gi
            requests:
              cpu: 100m
              memory: 500Mi
      cn:
        replicas: 2
        template:
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 1Gi
      dn:
```

```
        replicas: 1
        template:
          engine: galaxy
          hostNetwork: true
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 500Mi
      gms:
        template:
          engine: galaxy
          hostNetwork: true
          resources:
            limits:
              cpu: "1"
              memory: 1Gi
            requests:
              cpu: 100m
              memory: 500Mi
          serviceType: ClusterIP
  upgradeStrategy: RollingUpgrade
```

iii. 执行如下命令，创建PolarDB-X集群。

```
kubectl apply -f polardb-x.yaml
```

iv. 执行如下命令，查看PolarDB-X集群创建状态。

```
kubectl get polardbxCluster polardb-x -o wide -w
```

返回结果如下，请您耐心等待七分钟左右，当PHASE显示为Running时，表示PolarDB-X集群已经部署完成。

```
[galaxykube@i                    ~]$ kubectl get polardbxCluster polardb-x -o wide -w
NAME        PROTOCOL   GMS   CN    DN    CDC   PHASE      DISK     STAGE   REBALANCE   VERSION                              AGE
polardb-x   8.0        0/1   0/1   0/1   0/1   Creating                                                                      2m28s
polardb-x   8.0        1/1   0/1   1/1   0/1   Creating                                                                      3m33s
polardb-x   8.0        1/1   0/1   1/1   1/1   Creating                                                                      5m3s
polardb-x   8.0        1/1   1/1   1/1   1/1   Creating                                                                      6m32s
polardb-x   8.0        1/1   1/1   1/1   1/1   Running    7.2 GiB                      8.0.3-PXC-5.4.13-16534775/8.0.18   6m44s
polardb-x   8.0        1/1   1/1   1/1   1/1   Running    7.2 GiB                      8.0.3-PXC-5.4.13-16534775/8.0.18   7m44s
```

v. 按Ctrl+C键，退出查看PolarDB-X集群创建状态。

5.1.3. 第3步：体验PolarDB-X集群扩容

本节将介绍如何对PolarDB-X集群进行扩容。

操作步骤

1. 执行如下命令，编辑polardb-x.yaml文件。

```
vim polardb-x.yaml
```

2. 按i键进入编辑模式，将CN、DN和CDC的replicas参数改为2，进行扩容操作。

```yaml
apiVersion: polardbx.aliyun.com/v1
kind: PolarDBXCluster
metadata:
  name: polardb-x
spec:
  topology:
    nodes:
      cdc:
        replicas: 2
        template:
          resources:
            limits:
              cpu: "1"
              memory: 1Gi
            requests:
              cpu: 100m
              memory: 500Mi
      cn:
        replicas: 2
        template:
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 1Gi
      dn:
        replicas: 2
        template:
          engine: galaxy
          hostNetwork: true
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 500Mi
      gms:
        template:
          engine: galaxy
          hostNetwork: true
          resources:
```

3. 按ECS退出编辑模式，输入:wq后按下Enter键保存并退出。
4. 执行如下命令，将修改后的polardb-x.yaml文件应用到已经创建的PolarDB-X集群中。

```
kubectl apply -f polardb-x.yaml
```

5. 执行如下命令，观察集群的变化情况。

```
kubectl get polardbxCluster polardb-x -o wide -w
```

返回结果如下，您可以看到PolarDB-X集群扩容过程中各个节点的变化。请您耐心等待两分钟左右，当PHASE显示为Running时，表示PolarDB-X集群已经扩容完成。

```
[galaxykube@iZ...  ~]$ kubectl get polardbxCluster polardb-x -o wide -w
NAME        PROTOCOL   GMS   CN    DN    CDC   PHASE       DISK       STAGE    REBALANCE    VERSION                               AGE
polardb-x   8.0        1/1   1/2   1/2   1/2   Upgrading   7.2 GiB                          8.0.3-PXC-5.4.13-16534775/8.0.18      9m38s
polardb-x   8.0        1/1   1/2   1/2   2/2   Upgrading   7.2 GiB                          8.0.3-PXC-5.4.13-16534775/8.0.18      9m56s
polardb-x   8.0        1/1   2/2   1/2   2/2   Upgrading   7.2 GiB                          8.0.3-PXC-5.4.13-16534775/8.0.18      10m
polardb-x   8.0        1/1   2/2   1/2   2/2   Upgrading   7.2 GiB                          8.0.3-PXC-5.4.13-16534775/8.0.18      10m
polardb-x   8.0        1/1   2/2   2/2   2/2   Upgrading   7.2 GiB    RebalanceStart                 8.0.3-PXC-5.4.13-16534775/8.0.18   11m
polardb-x   8.0        1/1   2/2   2/2   2/2   Upgrading   7.2 GiB    RebalanceWatch                 8.0.3-PXC-5.4.13-16534775/8.0.18   11m
polardb-x   8.0        1/1   2/2   2/2   2/2   Upgrading   7.2 GiB    RebalanceWatch   0.0%          8.0.3-PXC-5.4.13-16534775/8.0.18   11m
polardb-x   8.0        1/1   2/2   2/2   2/2   Upgrading   7.2 GiB    Clean            100.0%        8.0.3-PXC-5.4.13-16534775/8.0.18   11m
polardb-x   8.0        1/1   2/2   2/2   2/2   Running     7.2 GiB                                   8.0.3-PXC-5.4.13-16534775/8.0.18   11m
polardb-x   8.0        1/1   2/2   2/2   2/2   Running     10.8 GiB                                   8.0.3-PXC-5.4.13-16534775/8.0.18   11m
```

6. 按Ctrl+C键，退出查看PolarDB-X集群状态。
7. 执行如下命令，获取PolarDB-X集群登录密码。

```
kubectl get secret polardb-x -o jsonpath="{.data['polardbx_root']}" | base64 -d - | xargs echo "Password: "
```

返回结果如下，您可以查看到PolarDB-X集群登录密码。

8. 执行如下命令，将PolarDB-X集群的端口转发到本地的3306端口。

```
kubectl port-forward svc/polardb-x 3306
```

9. 在实验页面，单击右上角的图标，创建新的终端二窗口。

10. 执行如下命令，连接PolarDB-X集群。

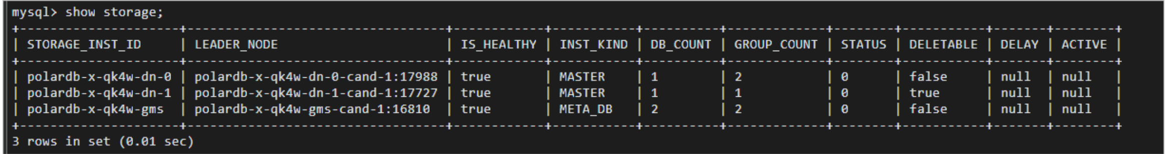
```
mysql -h127.0.0.1 -P3306 -upolardbx_root -p<PolarDB-X集群登录密码>
```

> **说明**
>
> 您需要将<PolarDB-X集群登录密码>替换为实际获取到的PolarDB-X集群登录密码。

11. 执行如下SQL语句，检查扩容后的状态。

```
show storage;
```

返回结果如下，您可查看到PolarDB-X集群扩容后的状态。



mysql> show storage;

STORAGE_INST_ID	LEADER_NODE	IS_HEALTHY	INST_KIND	DB_COUNT	GROUP_COUNT	STATUS	DELETABLE	DELAY	ACTIVE
polardb-x-qk4w-dn-0	polardb-x-qk4w-dn-0-cand-1:17988	true	MASTER	1	2	0	false	null	null
polardb-x-qk4w-dn-1	polardb-x-qk4w-dn-1-cand-1:17727	true	MASTER	1	1	0	true	null	null
polardb-x-qk4w-gms	polardb-x-qk4w-gms-cand-1:16810	true	META_DB	2	2	0	false	null	null

3 rows in set (0.01 sec)

5.1.4. 第4步：体验PolarDB-X集群缩容

本节将介绍如何对PolarDB-X集群进行缩容。

操作步骤

1. 切换至终端一，按Ctrl+C键，停止PolarDB-X集群端口转发。
2. 执行如下命令，编辑polardb-x.yaml文件。

```
vim polardb-x.yaml
```

3. 按i键进入编辑模式，将CN、DN和CDC的replicas参数改为1，进行缩容操作。

```yaml
apiVersion: polardbx.aliyun.com/v1
kind: PolarDBXCluster
metadata:
  name: polardb-x
spec:
  topology:
    nodes:
      cdc:
        replicas: 1
        template:
          resources:
            limits:
              cpu: "1"
              memory: 1Gi
            requests:
              cpu: 100m
              memory: 500Mi
      cn:
        replicas: 1
        template:
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 1Gi
      dn:
        replicas: 1
        template:
          engine: galaxy
          hostNetwork: true
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 500Mi
      gms:
        template:
          engine: galaxy
          hostNetwork: true
          resources:
```

4. 按ECS退出编辑模式，输入:wq后按下Enter键保存并退出。
5. 执行如下命令，将修改后的polardb-x.yaml文件应用到已经创建的PolarDB-X集群中。

```
kubectl apply -f polardb-x.yaml
```

6. 执行如下命令，观察集群的变化情况。

```
kubectl get polardbxCluster polardb-x -o wide -w
```

返回结果如下，您可以看到PolarDB-X集群缩容过程中各个节点的变化。请您耐心等待两分钟左右，当PHASE显示为Running时，表示PolarDB-X集群已经缩容完成。

```
[galaxykube@iZbp[illegible] ~]$ kubectl get polardbxCluster polardb-x -o wide -w
NAME        PROTOCOL   GMS   CN    DN    CDC   PHASE       DISK       STAGE            REBALANCE   VERSION                                AGE
polardb-x   8.0        1/1   1/1   2/1   1/1   Upgrading   10.8 GiB   RebalanceWatch   0.0%        8.0.3-PXC-5.4.13-16534775/8.0.18   16m
polardb-x   8.0        1/1   1/1   2/1   1/1   Upgrading   10.8 GiB   RebalanceWatch   0.0%        8.0.3-PXC-5.4.13-16534775/8.0.18   17m
polardb-x   8.0        1/1   1/1   2/1   1/1   Upgrading   10.8 GiB   RebalanceWatch   50.0%       8.0.3-PXC-5.4.13-16534775/8.0.18   17m
polardb-x   8.0        1/1   1/1   2/1   1/1   Upgrading   10.8 GiB   RebalanceWatch   0.0%        8.0.3-PXC-5.4.13-16534775/8.0.18   17m
polardb-x   8.0        1/1   1/1   2/1   1/1   Upgrading   10.8 GiB   RebalanceWatch   0.0%        8.0.3-PXC-5.4.13-16534775/8.0.18   18m
polardb-x   8.0        1/1   1/1   2/1   1/1   Upgrading   10.8 GiB   Clean            100.0%      8.0.3-PXC-5.4.13-16534775/8.0.18   18m
polardb-x   8.0        1/1   1/1   2/1   1/1   Upgrading   10.8 GiB   Clean                        8.0.3-PXC-5.4.13-16534775/8.0.18   18m
polardb-x   8.0        1/1   1/1   1/1   1/1   Upgrading   10.8 GiB   Clean                        8.0.3-PXC-5.4.13-16534775/8.0.18   18m
polardb-x   8.0        1/1   1/1   1/1   1/1   Running     10.8 GiB                                8.0.3-PXC-5.4.13-16534775/8.0.18   18m
polardb-x   8.0        1/1   1/1   1/1   1/1   Running     7.3 GiB                                 8.0.3-PXC-5.4.13-16534775/8.0.18   19m
```

7. 按Ctrl+C键，退出查看PolarDB-X集群状态。
8. 执行如下命令，将PolarDB-X集群的端口转发到本地的3306端口。

```
kubectl port-forward svc/polardb-x 3306
```

9. 切换至终端二，执行如下SQL语句，检查缩容后的状态。

```
show storage;
```

返回结果如下，您可查看到PolarDB-X集群缩容后的状态。

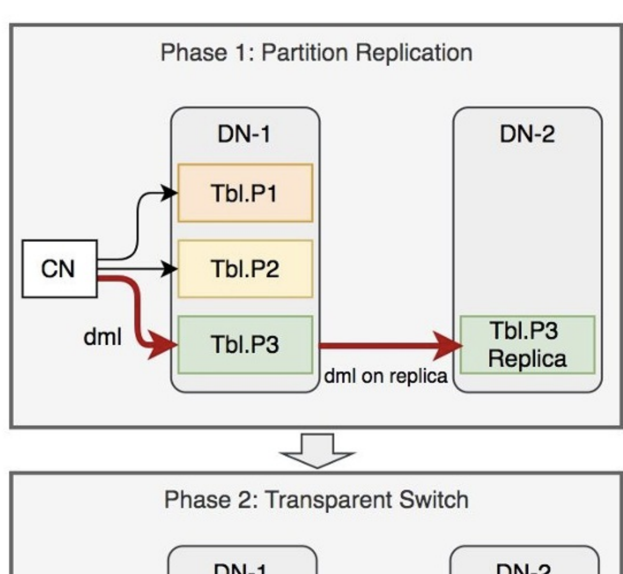
```
mysql> show storage;
+---------------------+------------------------------------+------------+-----------+----------+-------------+--------+-----------+-------+--------+
| STORAGE_INST_ID     | LEADER_NODE                        | IS_HEALTHY | INST_KIND | DB_COUNT | GROUP_COUNT | STATUS | DELETABLE | DELAY | ACTIVE |
+---------------------+------------------------------------+------------+-----------+----------+-------------+--------+-----------+-------+--------+
| polardb-x-qk4w-dn-0 | polardb-x-qk4w-dn-0-cand-1:17988   | true       | MASTER    | 1        | 2           | 0      | false     | null  | null   |
| polardb-x-qk4w-gms  | polardb-x-qk4w-gms-cand-1:16810    | true       | META_DB   | 2        | 2           | 0      | false     | null  | null   |
+---------------------+------------------------------------+------------+-----------+----------+-------------+--------+-----------+-------+--------+
2 rows in set (0.00 sec)
```

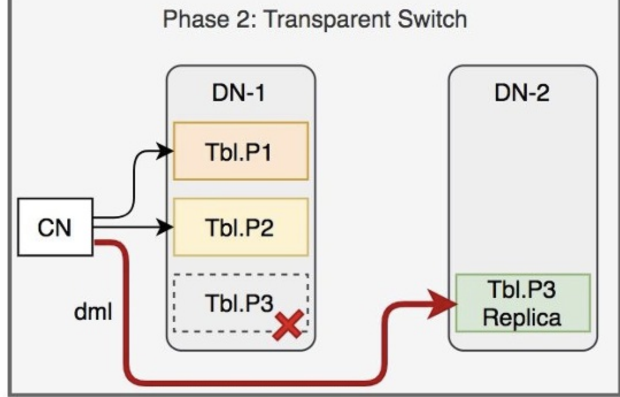
5.2. 升配和降配

分区迁移

分区迁移主要的过程，就是将老DN上的数据，迁移到新的DN上。由于PolarDB-X是一款分布式数据库，存储的数据量往往较大，迁移的数据量往往也是比较大的。因此在整个迁移过程中，我们尽可能采用并行的策略，来提高迁移的效率。

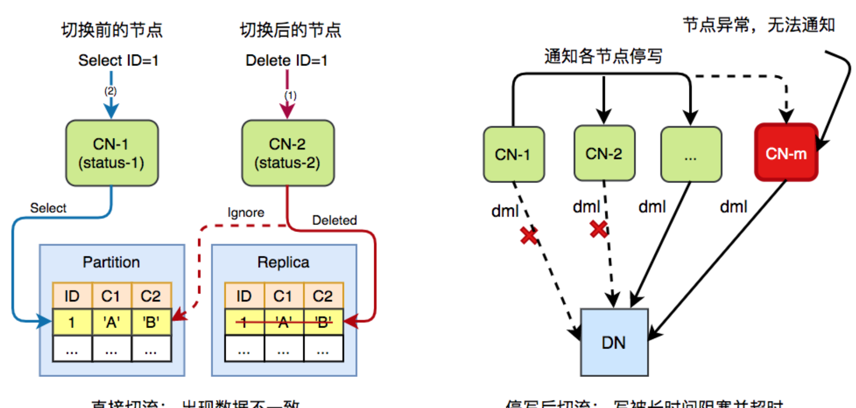
当然，迁移过程中，也是比较消耗CPU和I/O资源的。如果迁移过快，也会对正常的业务请求造成一定影响。因此PolarDB-X支持自适应的流控策略，一方面能够允许用户自定义地控制迁移任务的启停和迁移速率的上限，另一方面也会根据实际节点的资源情况，自适应地控制迁移速率，从而既保证迁移效率，也减少对业务的影响。





透明切换

当完成数据迁移，需要告知CN从访问老DN切换到访问新DN时，由于PolarDB-X是一款分布式数据库，会有多个CN节点存在。如果采用直接切流的方案（下图左侧），那切换期间分布式系统中便会因为节点状态不兼容（因为所有节点不可能在同一时间完成切换）而导致出现数据不一致；如果采用停写后再切流的方案（下图右侧），那需要通知分布式系统所有节点来阻塞业务所有的写操作，但通知过程本身时间不可控（或网络故障或节点自身不可用），应用有可能导致被长时间阻塞写，从而使应用侧会产生超时异常。可见，无论是直接切流还是停写后再切流的方案，都不能满足保持透明的要求。



PolarDB-X为了实现透明的切换效果，其思路是将目标端分区看作是主表，将源端分区看作是索引表，并对主表进行一次标准的删除索引的Online DDL操作(其状态过程是ReadyPublic-->Writ eOnly-->Delet eOnly-->Absent，与Online Schema Change定义的一致)。

Transparent Switching

如上图所示，在Online DDL期间，由于新索引表会被逐渐下线，CN节点的SQL引擎根据不同DDL状态，将不同类型的流量分步骤地从源端（索引表）切换到目标端（主表）：先切Select流量（WriteOnly状态，索引表不可读），再切Insert流量（DeleteOnly状态，索引表不可插入增量），最后才切Update/Delete/加锁操作等流量（Absent状态，索引表不再接收写入）。基于Online DDL，整个切换会有以下几点的优势：

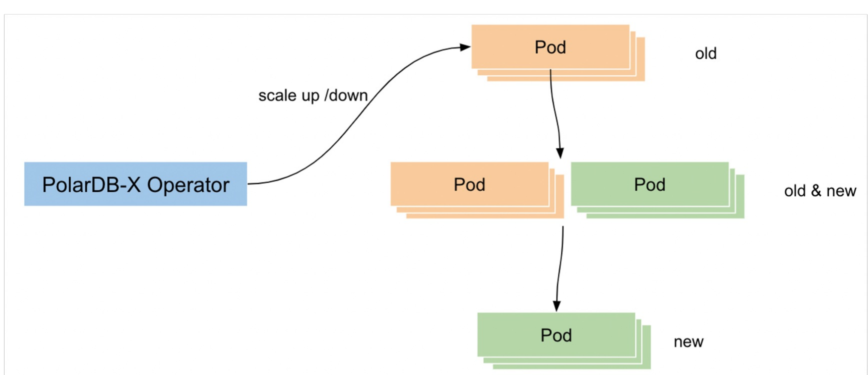
- 切期期间业务读写不会因被阻塞(不需要禁写);
- 切换期间不会产生死锁（因为在WriteOnly与DeleteOnly状态的节点，加锁顺序都是一致）；
- 切换期间不会产生数据不一致的异常（Online Schema Change论文已证明）。

这些优势使得流量切换全程能对应用做到了透明。

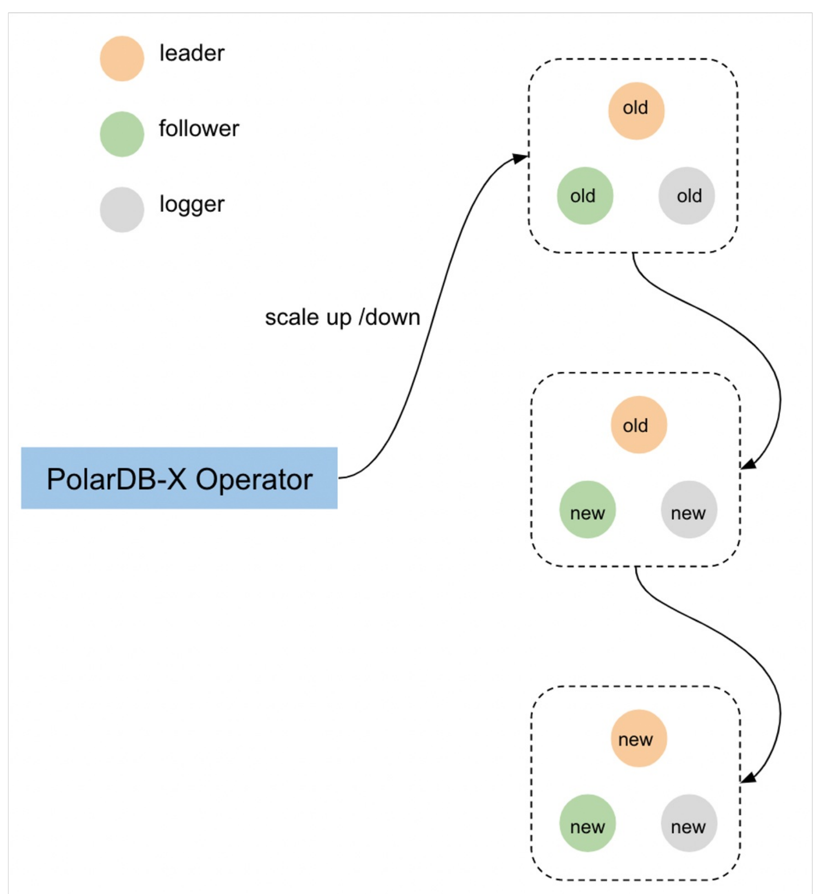
> 说明
>
> 更多技术细节，请参见《谈谈PolarDB-X的水平扩展》和《PolarDB-X Online Schema Change》。

升降配整体过程

对于无状态的CN和CDC组件而言，升降配就是将这类组件的节点资源（如CPU、内存等）进行更改，需要用新的Pod来替换老的Pod。为了保证相对的平滑，PolarDB-X采用Rolling Upgrade的方式。首先按照要求的CN节点配置，创建好一批节点，然后将老的CN下线，来用新的CN节点替换老的CN节点。



对于有状态的DN节点，每个DN内部有leader、follower和logger三种状态，其中，leader是对外响应业务请求的，follower和logger是做数据强一致保证的。首先会对follower和logger进行升降配，更改其配置，换成新的Pod；然后将原来的leader切换到follower，由已经升配完成的follower作为leader，再对老的leader进行升降配，从而最终完成升配过程，并且尽量减少对业务的影响。



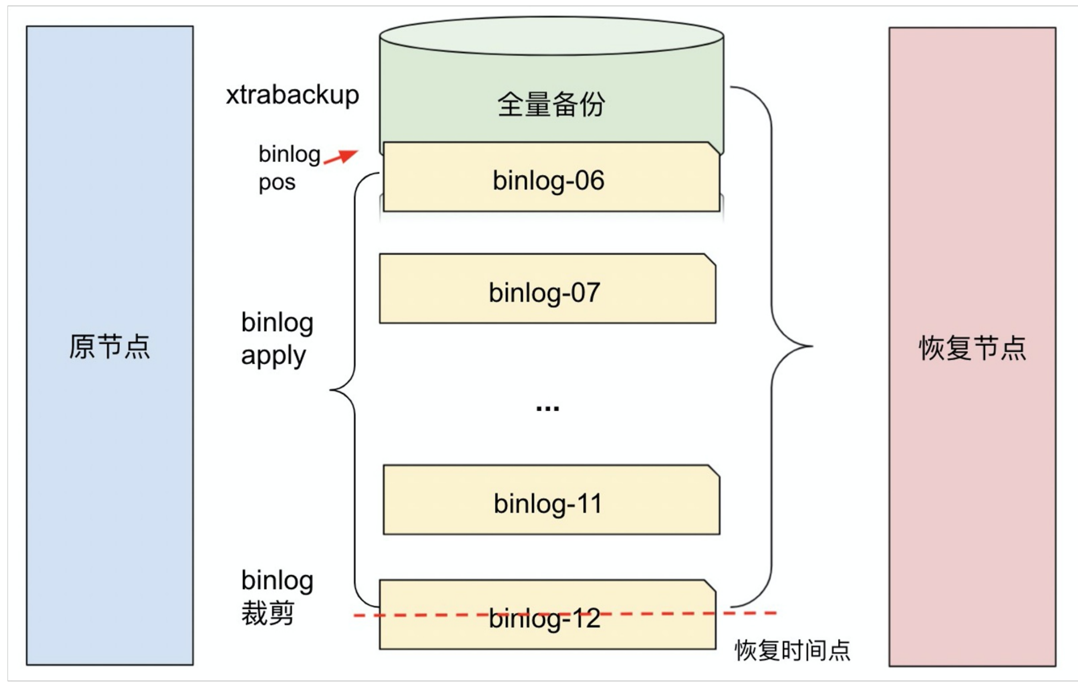
升配和降配演示

演示视频(44分38秒开始)

5.3. 备份恢复

单机MySQL的备份恢复（秒级）

以PITR为例，可以将其备份恢复过程看做两个部分，第一部分是全量备份的过程，另一部分是基于Binlog增量备份的过程。假如需要对MySQL进行任意时间点恢复，首先找到该时间点之前最近的一个全量备份（通常采用xtrabackup工具执行该任务），然后找到xtrabackup全量备份完成的时间点所对应的Binlog到所需要恢复的时间点（一般都是秒级时间点）这段时间内所有的Binlog，进行增量Apply。这样就可以保证单机MySQL的数据恢复到期望的时间点。

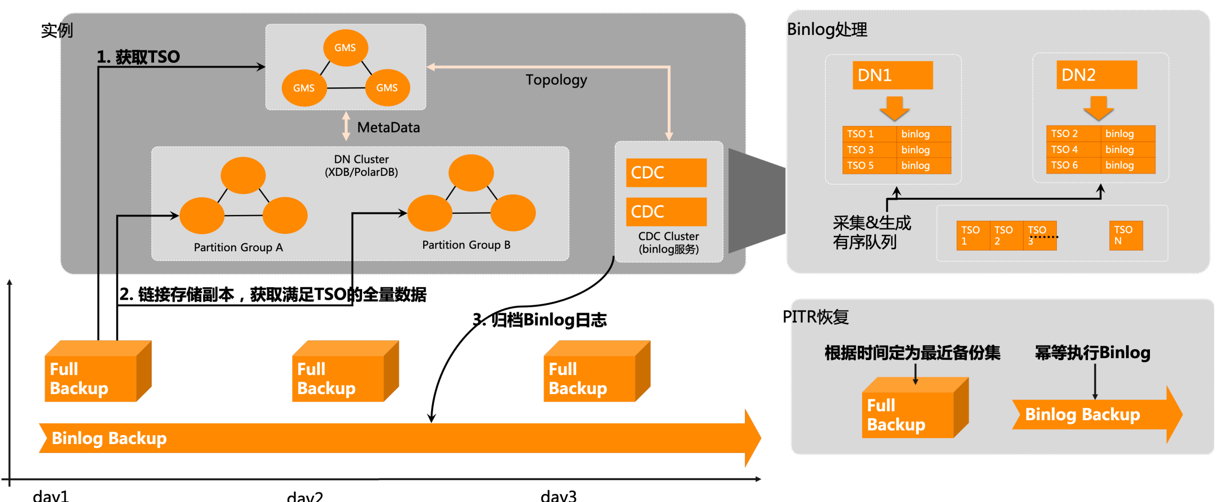


PolarDB-X的DN节点是用来存储数据的。我们可以将每个DN节点看做MySQL来使用。如果对每个DN，通过上述方式进行备份恢复，由于PolarDB-X是一款分布式数据库，有分布式事务的存在，会出现数据一致性的问题。

PolarDB-X的备份恢复

PolarDB-X的分布式事务目前都是采用基于TSO的事务模型，每一个分布式事务都会在Prepare和Commit阶段去GMS获取一个TSO。在此基础上，我们对DN的Binlog进行改造，对于每一个Binlog的event，会将TSO也写入DN的Binlog中。基于以上过程，PolarDB-X的整个备份恢复过程，与单机MySQL类似，也是基于全量备份+基于Binlog增量备份的一个过程。全量备份是在每个DN上，基于xtrabackup执行备份操作；增量备份的Binlog，会按照DN实际的产出进行实时备份。

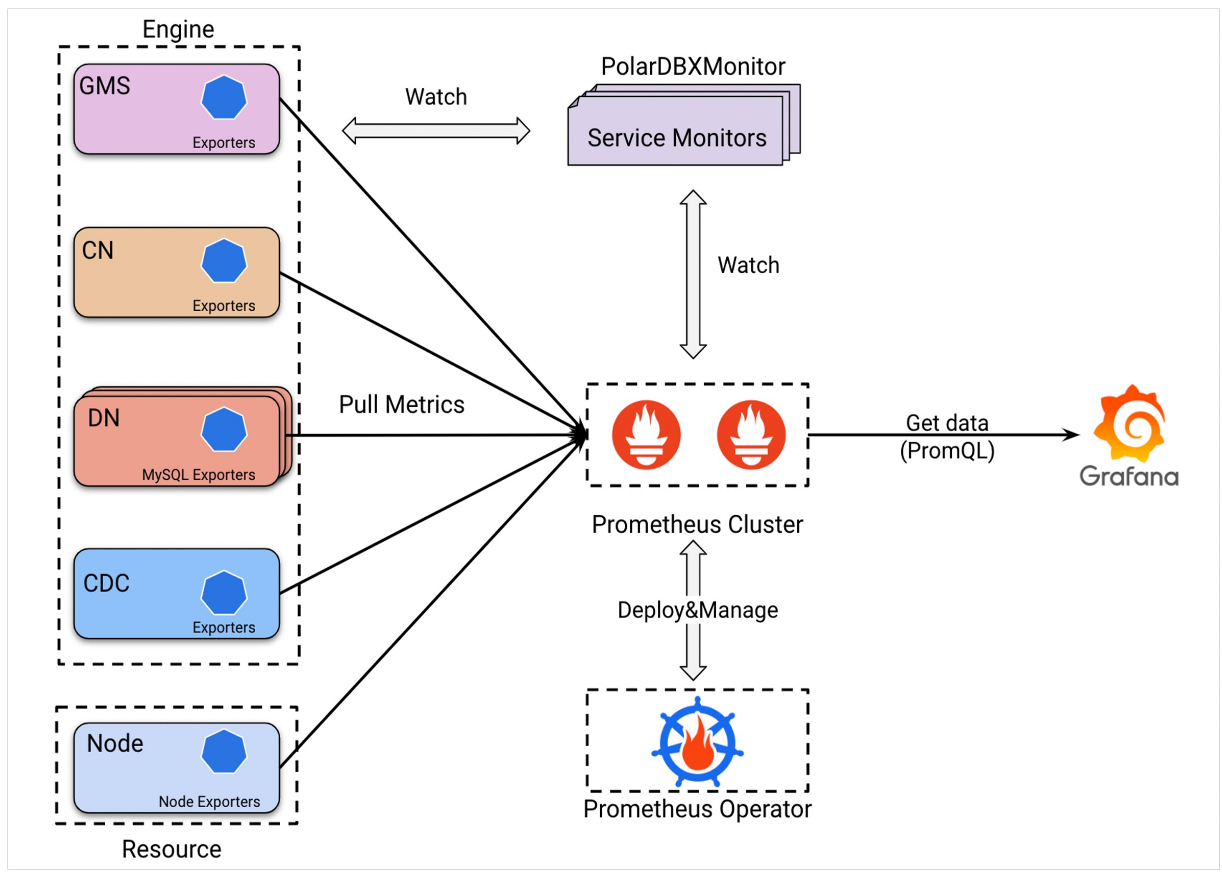
在恢复过程中，我们会根据Binlog中的TSO，对现有的增量Binlog进行裁剪，从而保证恢复出来的数据是全局一致的。



5.4. 数据库监控

PolarDB-X监控架构

PolarDB-X目前在Kubernetes上构建的监控体系的基本架构如下：



监控指标的采集层中，监控指标分为两种类型。

一种是引擎相关指标（例如QPS、RT等）。对于这些指标而言，PolarDB-X针对每一个不同的组件，构建了一个Exporter的服务，它通过HTTP的方式来暴露监控指标。Prometheus会定期获取数据（周期就是在PolarDBXMonitor对象中限定的monitorInterval参数值，如下图所示）。

```
apiVersion: polardbx.aliyun.com/v1
kind: PolarDBXMonitor
metadata:
  name: pxc-demo-monitor
spec:
  clusterName: pxc-demo
  monitorInterval: 15s
  scrapeTimeout: 10s
```

另一种是资源层面的指标（CPU、内存等基本指标）。对于这些指标而言，PolarDB-X通过开源的Node Exporter去暴露（如下图所示，可以看到在polardbx-monitor这个命名空间下，创建好了一些node-exporter的Pod来负责监控的采集）。

```
~/classes/class-10 » kc get pods -n polardbx-monitor
                                        polardbx@polardbx-demo
NAME                                   READY   STATUS    RESTARTS      AGE
grafana-66b65bb44d-9cmhj               1/1     Running   0             25m
kube-state-metrics-864bcbb4b8-7rwpq    3/3     Running   0             25m
node-exporter-4t8vt                    2/2     Running   0             25m
node-exporter-djn9j                    2/2     Running   0             25m
node-exporter-jvd9m                    2/2     Running   0             25m
node-exporter-jzvxb                    2/2     Running   0             25m
prometheus-adapter-64d9799cc9-s8q72    1/1     Running   0             25m
prometheus-k8s-0                       2/2     Running   1 (24m ago)   24m
prometheus-operator-66947d5df-7ksf7    2/2     Running   0             25m
```

采集完监控指标后，Prometheus会将监控指标存储到内部的一个时序数据库里面。然后，Grafana的监控报表会通过PromQL的方式从Prometheus中获取监控数据，并构建相应的监控图表，以供用户了解实例的运行状况。此外，对于Prometheus Cluster，我们是通过Prometheus Operator组件去管理，它可以帮助我们有效地部署和管理Prometheus集群。

一个Kubernetes集群内，哪些PolarDB-X实例需要监控、以何种频率监控，都由PolarDBXMonitor对象所控制。PolarDBXMonitor会告诉Prometheus，需要采集哪些实例的监控指标。

监控安装及访问演示

以下视频演示如何在Kubernetes集群中，为PolarDB-X集群安装监控，然后通过Grafana访问监控报表。

视频链接：演示视频（08分09秒开始）

相关文档：数据库监控

5.5. SQL限流和SQL Advisor

本节介绍当一个PolarDB-X集群遇到一些问题SQL的时候，如何通过已有的一些手段对其进行优化处理，以保障系统的稳定运行。本节将介绍两个能力：SQL限流和SQL Advisor（索引推荐）。

> **说明**
>
> 本节实验操作部分主要通过阿里云官方网站的云起实验室进行，详情可登录阿里云官方网站，访问如何在PolarDB-X中优化慢SQL了解。

PolarDB-X SQL限流（应急处理）

PolarDB-X提供的SQL限流能力，能够帮助我们快速地将问题SQL拦截，不再占用系统的任何资源，来保证已有业务的稳定运行。同时，SQL限流支持丰富的SQL匹配方法：支持针对某一个数据库的某一张具体的表来进行限流（只针对某一张有问题的表，去限流关于这张表的SQL，其他表的SQL不会进行限流）；支持按照用户名和执行的主机来限流；支持对select或者insert语句进行限流；等等。

此外，PolarDB-X还针对SQL的内容，提供了两种匹配方式，一种是基于关键字（keyword）的匹配方式，它能够根据SQL里面的一个关键字去进行规则匹配，从而对其进行限流；另一种是基于SQL模板的匹配方式，它能够根据模板ID将某一类SQL（基于一个模板生成的不同SQL，不同用户产生的SQL只是参数不同而已）进行限流。

```
CREATE CCL_RULE [ IF NOT EXISTS ] `ccl_rule_name`
ON `database`.`table`
TO '<usename>'@'<host>'
FOR { UPDATE | SELECT | INSERT | DELETE }
[ filter_options ]
with_options filter_options:

    [ FILTER BY KEYWORD('KEYWORD1', 'KEYWORD2',…) ]
    [ FILTER BY TEMPLATE('template_id') ]

with_options:
    WITH MAX_CONCURRENCY = value1 [ , WAIT_QUEUE_SIZE = value2 ] [ , WAIT_TIMEOUT = value3 ] [ ,FAST_MATCH = { 0 , 1 }]
```

```
CREATE CCL_RULE IF NOT EXISTS `selectrule`
ON *.*
TO 'ccltest'@'%'
FOR SELECT
FILTER BY KEYWORD('cclmatched')
WITH MAX_CONCURRENCY=0;
```

基于 SQL 关键字

```
CREATE CCL_RULE IF NOT EXISTS `selectrule`
ON *.*
TO 'ccltest'@'%'
FOR SELECT
FILTER BY TEMPLATE('0b722ad1')
WITH MAX_CONCURRENCY=0;
```

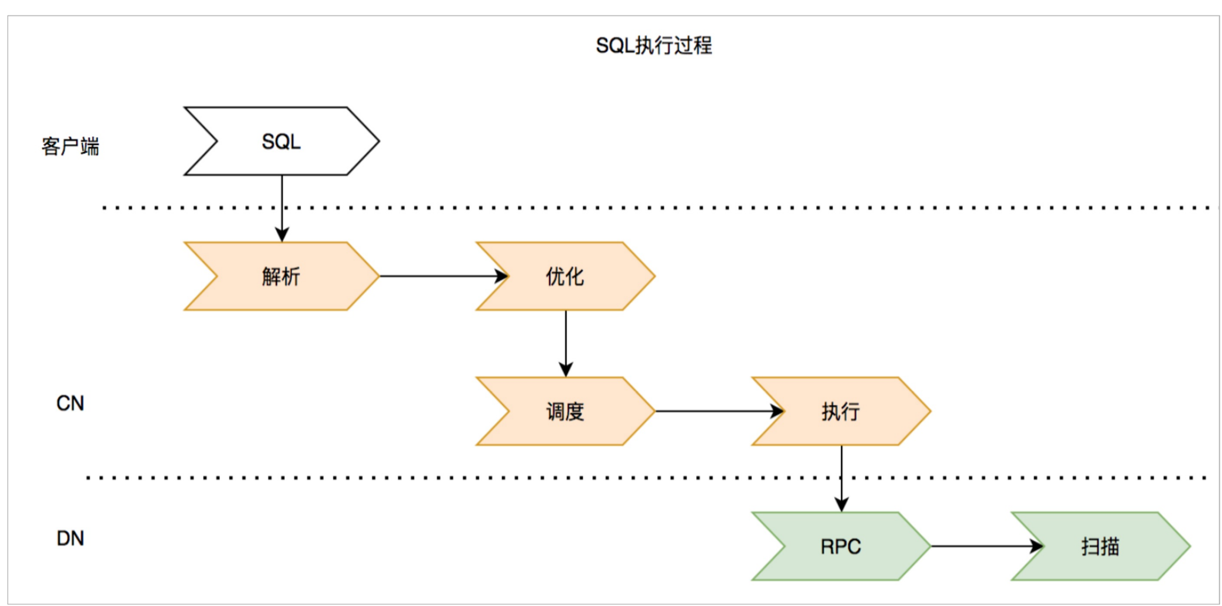
基于 SQL 模板

说明

有关SQL限流的更多详情，请参见SQL限流。

SQL执行的过程

首先，客户端会将业务SQL发给PolarDB-X的一个CN，CN对其进行解析优化，然后生成执行计划并调度执行。对于中间可以下推的部分，PolarDB-X会直接通过网络的方式下推到每个DN节点上执行，并收集返回的结果。对于不可以下推的部分，PolarDB-X会等待DN的结果返回，在CN中进行计算，最终将结果返回到用户的客户端。

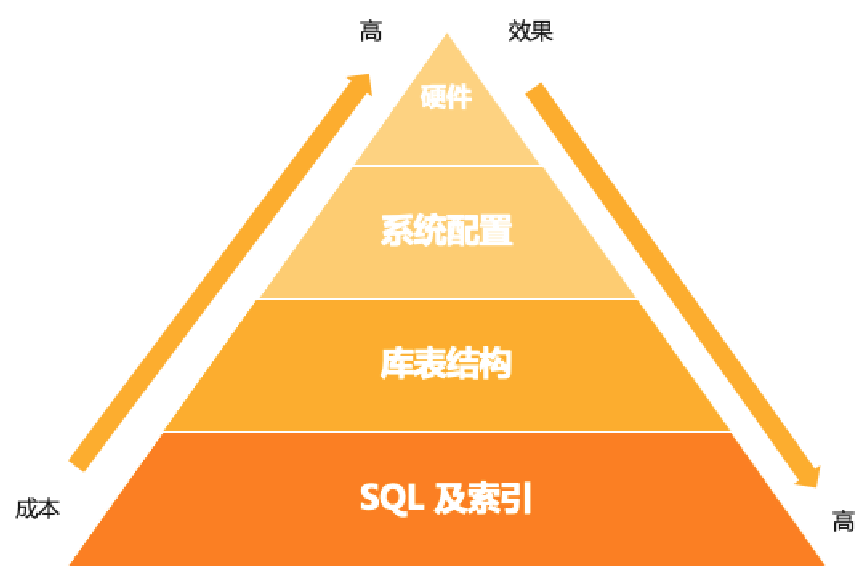


在这个过程中，我们主要关注的就是CN和DN两部分的资源使用和执行耗时。针对这几种场景，我们将可能存在慢SQL的原因，分为如下三类：

- **业务问题：明显的数据倾斜、不合理的分片策略设置、数据返回过多等和业务使用相关问题。**
  - 在不合理分片策略的情况下，有可能出现某一个分片里的数据量非常多，某些分片的数据量非常少。一旦某一个SQL落到数据非常多的分片，就可能占用比较多的查询时间，或者产生较大的数据扫描IO，成为整个SQL执行的瓶颈点。 这种情况下，我们需要对整个表结构进行优化调整。
  - 对于一些业务使用方面的问题，比如基于某个列进行查询但该列缺少全局二级索引，导致很多不必要的分片扫描，占用了较多的CN的CPU资源。这种情况下，需要进行SQL的优化或者创建一些索引来对该问题进行优化。
  - 对于业务或SQL本身需要返回过多数据的情况，我们推荐设置一个合理的限流规则（比如限定并发度），避免过多的同类SQL占用过多的资源。
- **系统问题：流量太大，资源成为瓶颈或者网络抖动造成的问题。**
  - 对于资源瓶颈的情况，我们推荐采用弹性扩展的方式来增加更多机器，对问题进行优化。
  - 对于网络抖动导致延迟变大的情况，需要检查网络配置是否正确，或者网络中间的一些组件是否存在问题。
- **执行问题：如选错索引，选错Join类型或顺序等问题。**

  由于业务SQL到了PolarDB-X的CN节点后，会对SQL进行解析优化，生成相应的执行计划。在这个过程中，PolarDB-X会根据统计信息，选择一个合适的执行计划。如果执行统计信息过期或不准确，就可能出现选错索引、选错Join类型或顺序等问题，也会导致慢SQL。在这种情况下，我们需要对SQL进行进一步分析优化或改写，以解决慢SQL的问题。

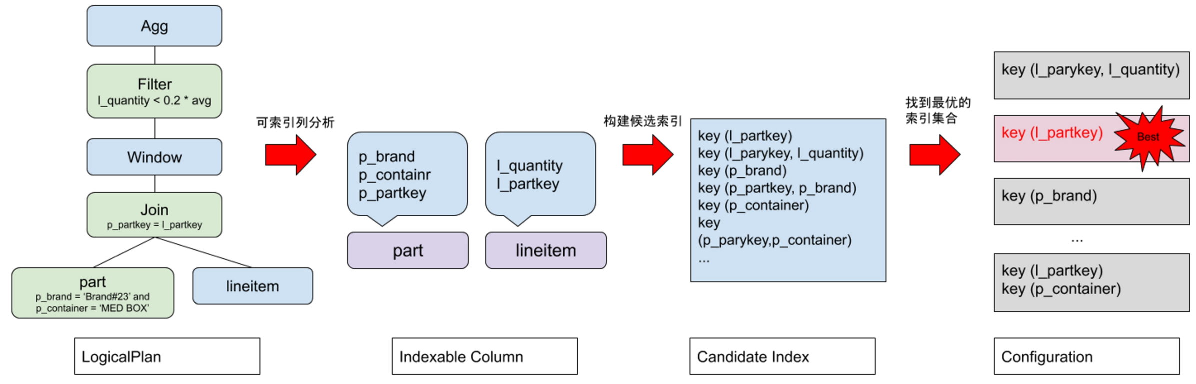
几种慢SQL产生的原因以及对应的处理方式，如下图所示。在该图中，从下往上优化，成本是逐步提高的，但效果反而越来越差。采用SQL及索引调优的方式，往往不需要花费过多成本，就可以取得显著的效果。



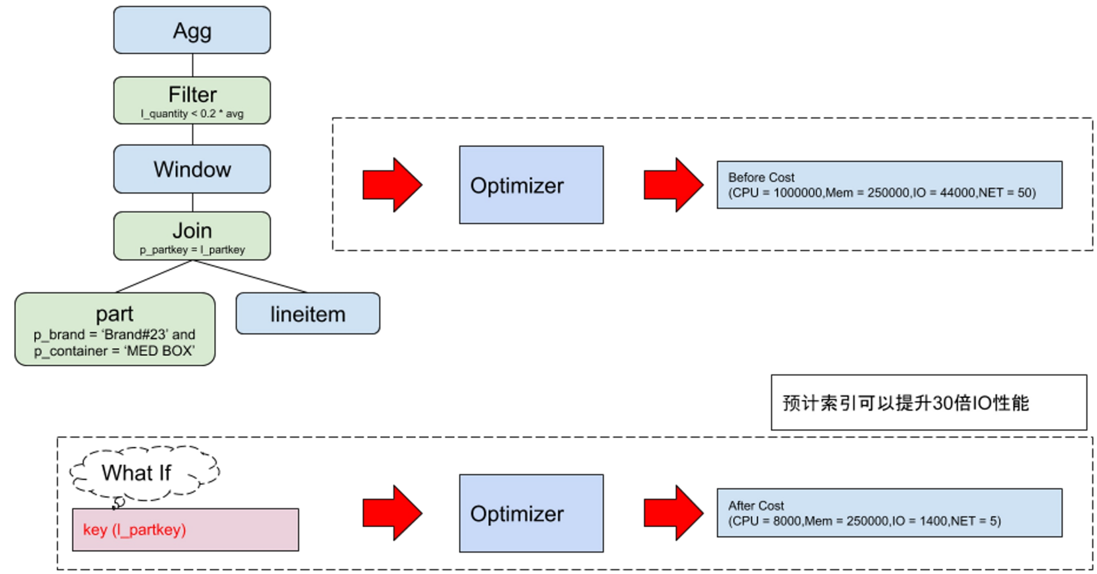
PolarDB-X SQL Advisor基本原理

SQL Advisor的过程大致可以分为三个部分：

1. 第一个部分是可索引列分析，即找出哪些列是可以添加索引的（这些列可能包括while条件里的列、join条件里的列、group by或order by的列，等等）。
2. 第二个部分是构建候选索引集合。候选索引集合的构建，基于如下假设：对于一张表而言，产生关键业务影响的索引数不超过两个。因此，我们限定索引的长度（即组合索引的长度）不超过2。在此条件下，我们将所有条件索引进行美剧，生成Candidate Index（所有候选索引的集合）。
3. 第三个部分是在所有的候选索引集合中，寻找最优的集合，并将其推荐出来，得到最优的索引推荐结果。



如何找到最优的索引集合？我们采用了PolarDB-X的优化器。它是一个基于代价的优化器，它会为每一条SQL估算其执行代价。在这种情况下，优化器还提供了一个叫What if的能力，即告知优化器索引的存在但并不实际创建它，由优化器来给出这条SQL的执行代价。基于What if的能力，我们只需要将刚刚构建出来的所有候选索引的集合，分别输入到优化器中，并让优化器提供每个索引创建后的执行代价是多少。在执行代价枚举中，我们只需要找到执行代价最小的索引组合，就是最终的推荐结果，也就可以达到推荐出最优索引的效果。



相关推荐

如果您想了解更多有关PolarDB-X调优知识，详情请参见如下内容。

- PolarDB-X智能索引推荐技术尝鲜
- PolarDB-X CBO优化器技术内幕
- PolarDB-X如何限流慢SQL
- PolarDB-X调优指南

5.5.1. 第1步：连接PolarDB-X集群

本节将介绍如何连接通过Kubernetes部署的PolarDB-X集群。

前提条件

已安装完成通过Kubernetes部署的PolarDB-X集群，详情请参见使用Kubernetes安装PolarDB-X。

操作步骤

1. 执行如下命令，切换到账号galaxykube。

```
su galaxykube
```

2. 执行如下命令，启动一个minikube。

```
minikube start --cpus 4 --memory 12288 --image-mirror-country cn --registry-mirror=http
s://docker.mirrors.sjtug.sjtu.edu.cn
```

> 说明
>
> 这里我们使用了阿里云的minikube镜像源以及USTC提供的Docker镜像源来加速镜像的拉取。

返回结果如下，表示minikube已经正常运行。

```
[galaxykube@i... oot]$ minikube start --cpus 4 --memory 12288 --image-mirror-country cn --registry-mirror=https://docker.mirrors.sjtug.sjtu.edu.cn
😄 minikube v1.25.2 on Centos 8.5.2111 (amd64)
✨ Using the docker driver based on existing profile
❗ Your cgroup does not allow setting memory.
   ▪ More information: https://docs.docker.com/engine/install/linux-postinstall/#your-kernel-does-not-support-cgroup-swap-limit-capabilities
❗ Your cgroup does not allow setting memory.
   ▪ More information: https://docs.docker.com/engine/install/linux-postinstall/#your-kernel-does-not-support-cgroup-swap-limit-capabilities
👍 Starting control plane node minikube in cluster minikube
🚜 Pulling base image ...
🔄 Restarting existing docker container for "minikube" ...
🐳 Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
   ▪ kubelet.housekeeping-interval=5m
🔎 Verifying Kubernetes components...
   ▪ Using image registry.cn-hangzhou.aliyuncs.com/google_containers/storage-provisioner:v5
🌟 Enabled addons: storage-provisioner, default-storageclass
🏄 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

3. 执行如下命令，查看PolarDB-X集群登录密码。

```
kubectl get secret polardb-x -o jsonpath="{.data['polardbx_root']}" | base64 -d - | xar
gs echo "Password: "
```

返回结果如下，您可以查看到PolarDB-X集群登录密码。

```
[galaxyk... /]$ kubectl get secret polardb-x -o jsonpath="{.data['polardbx_root']}" | base64 -d - | xargs echo "Password: "
Password: wh... sn
```

4. 执行如下命令，将PolarDB-X集群端口转发到3306端口。

> **说明**
>
> 使用MySQL Client方式登录通过k8s部署的PolarDB-X集群前，您需要进行获取PolarDB-X集群登录密码和端口转发。

```
kubectl port-forward svc/polardb-x 3306
```

5. 在实验页面，单击右上角的⊞图标，创建新的终端二。

```
[galaxykube@iZb... /]$ kubectl port-forward svc/polardb-x 3306
Forwarding from 127.0.0.1:3306 -> 3306
Forwarding from [::1]:3306 -> 3306
```

6. 执行如下命令，连接PolarDB-X集群。

```
mysql -h127.0.0.1 -P3306 -upolardbx_root -p<PolarDB-X集群登录密码>
```

> **说明**
>
> - 您需要将<PolarDB-X集群登录密码>替换为实际获取到的PolarDB-X集群登录密码。
> - 如遇到mysql: [Warning] Using a password on the command line interface can be insecure.ERROR 2013 (HY000): Lost connection to MySQL server at 'reading initial communication packet', system error: 0报错，请您稍等一分钟，在终端一中重新执行转发端口命令，在终端二中重新执行连接PolarDB-X集群命令即可。

5.5.2. 第2步：启动业务

本节将介绍如何使用Sysbench Select场景模拟业务流量。

操作步骤

1. 准备压测数据。

i. 执行如下SQL语句，创建压测数据库sysbench_test。

```
CREATE DATABASE sysbench_test;
```

ii. 执行如下SQL语句，使用压测数据库sysbench_test。

```
USE sysbench_test;
```

iii. 在实验页面，单击右上角的图标，创建新的终端三。

iv. 执行如下命令，切换到账号galaxykube。

```
su galaxykube
```

v. 执行如下命令，进入到*/home/galaxykube*目录。

```
cd
```

vi. 执行如下命令，创建准备压测数据的sysbench-prepare.yaml文件。

```
vim sysbench-prepare.yaml
```

vii. 按**i**键进入编辑模式，将如下代码复制到文件中，然后按**ECS**退出编辑模式，输入**:wq**后按下**Enter**键保存并退出。

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: sysbench-prepare-data-test
  namespace: default
spec:
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: sysbench-prepare
          image: severalnines/sysbench
          env:
            - name: POLARDB_X_USER
              value: polardbx_root
            - name: POLARDB_X_PASSWD
              valueFrom:
                secretKeyRef:
                  name: polardb-x
                  key: polardbx_root
          command: [ 'sysbench' ]
          args:
            - --db-driver=mysql
            - --mysql-host=$(POLARDB_X_SERVICE_HOST)
            - --mysql-port=$(POLARDB_X_SERVICE_PORT)
            - --mysql-user=$(POLARDB_X_USER)
            - --mysql_password=$(POLARDB_X_PASSWD)
            - --mysql-db=sysbench_test
            - --mysql-table-engine=innodb
            - --rand-init=on
            - --max-requests=1
            - --oltp-tables-count=1
            - --report-interval=5
            - --oltp-table-size=160000
            - --oltp_skip_trx=on
            - --oltp_auto_inc=off
            - --oltp_secondary
            - --oltp_range_size=5
            - --mysql_table_options=dbpartition by hash(`id`)
            - --num-threads=1
            - --time=3600
            - /usr/share/sysbench/tests/include/oltp_legacy/parallel_prepare.lua
            - run
```

viii. 执行如下命令，运行准备压测数据的sysbench-prepare.yaml文件，初始化测试数据。

```
kubectl apply -f sysbench-prepare.yaml
```

ix. 执行如下命令，获取任务进行状态。

```
kubectl get jobs
```

返回结果如下，请您耐心等待大约1分钟，当任务状态COMPLETIONS为1/1时，表示数据已经初始化完成。

```
[galaxykube@iZ[illegible]spfZ ~]$ kubectl get jobs
NAME                          COMPLETIONS   DURATION   AGE
sysbench-prepare-data-test    1/1           56s        74s
```

2. 启动压测流量。

   i. 执行如下命令，创建启动压测的sysbench-select.yaml文件。

```
vim sysbench-select.yaml
```

   ii. 按i键进入编辑模式，将如下代码复制到文件中，然后按ECS退出编辑模式，输入:wq后按下Enter键保存并退出。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sysbench-point-select-k-test
  namespace: default
spec:
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: sysbench-point-select-k
          image: severalnines/sysbench
          env:
            - name: POLARDB_X_USER
              value: polardbx_root
            - name: POLARDB_X_PASSWD
              valueFrom:
                secretKeyRef:
                  name: polardb-x
                  key: polardbx_root
          command: [ 'sysbench' ]
          args:
            - --db-driver=mysql
            - --mysql-host=$(POLARDB_X_SERVICE_HOST)
            - --mysql-port=$(POLARDB_X_SERVICE_PORT)
            - --mysql-user=$(POLARDB_X_USER)
            - --mysql_password=$(POLARDB_X_PASSWD)
            - --mysql-db=sysbench_test
            - --mysql-table-engine=innodb
            - --rand-init=on
            - --max-requests=0
            - --oltp-tables-count=1
            - --report-interval=5
            - --oltp-table-size=32000000
            - --oltp_skip_trx=on
            - --oltp_auto_inc=off
            - --oltp_secondary
            - --oltp_range_size=5
            - --mysql-ignore-errors=all
            - --num-threads=8
            - --time=3600
            - --random_points=1
            - /usr/share/sysbench/tests/include/oltp_legacy/select_random_points.lu
a
            - run
```

iii. 执行如下命令，运行启动压测的sysbench-select.yaml文件，开始压测。

```
kubectl apply -f sysbench-select.yaml
```

iv. 执行如下命令，查找压测脚本运行的POD。

```
kubectl get pods
```

返回结果如下， 以sysbench-point-select-k-test-开头的POD即为目标POD。

```
[galaxykube@i[illegible]rspfZ ~]$ kubectl get pods
NAME                                              READY   STATUS      RESTARTS        AGE
polardb-x-rz2l-cdc-default-579b4b7549-qdrk8       2/2     Running     3 (4m57s ago)   2d23h
polardb-x-rz2l-cn-default-55778b9d9f-2878p        3/3     Running     3 (10m ago)     2d23h
polardb-x-rz2l-dn-0-cand-0                        3/3     Running     3 (10m ago)     2d23h
polardb-x-rz2l-dn-0-cand-1                        3/3     Running     3 (10m ago)     2d23h
polardb-x-rz2l-dn-0-log-0                         3/3     Running     3 (10m ago)     2d23h
polardb-x-rz2l-gms-cand-0                         3/3     Running     3 (10m ago)     2d23h
polardb-x-rz2l-gms-cand-1                         3/3     Running     3 (10m ago)     2d23h
polardb-x-rz2l-gms-log-0                          3/3     Running     3 (10m ago)     2d23h
sysbench-point-select-k-test-zqlbc                1/1     Running     0               6s
sysbench-prepare-data-test-7bjzw                  0/1     Completed   0               4m12s
```

v. 执行如下命令，查看QPS等信息。

```
kubectl logs -f 目标POD
```

> **说明**
>
> 您需要将命令中的目标POD替换为以sysbench-point-select-k-test-开头的POD。

5.5.3. 第3步：体验SQL限流和SQL Advisor

本节将介绍体验SQL限流和SQL Advisor的方法和步骤。

操作步骤

> **说明**
>
> 本步骤有关SQL语句的操作都在终端二窗口中操作，查看QPS等信息在终端三窗口中操作，终端一窗口只负责PolarDB-X集群端口转发。

1. 体验SQL限流。

> **说明**
>
> SQL限流是PolarDB-X提供的对符合特定规则的SQL进行限制的功能。
>
> 在本实验场景中假设第2步：启动业务中发起的Sysbench Select流量严重影响了其他业务，所以我们首先用SQL限流对Select SQL进行限流。

i. 执行如下SQL语句，查看当前正在运行的请求。

```
show full processlist where info is not null
```

返回结果如下，您可查看到有如下SQL正在执行。

```
mysql> mysql> show full processlist where info is not null;
+-----+---------------+------------------+--------------+---------+------+-------+----------------------------------
--------------+
| Id  | User          | Host             | db           | Command | Time | State | Info
L_Template_Id |
+-----+---------------+------------------+--------------+---------+------+-------+----------------------------------
--------------+
| 273 | polardbx_root | 172.17.0.1:55279 | sysbench_test | Query   |    0 |       |         SELECT id, k, c, pad
          FROM sbtest1
          WHERE k IN (?)
         | 1471932e1f401001 | be880505         |
| 275 | polardbx_root | 172.17.0.1:46164 | sysbench_test | Query   |    0 |       |         SELECT id, k, c, pad
          FROM sbtest1
          WHERE k IN (?)
         | 1471932e23401000 | be880505         |
| 277 | polardbx_root | 172.17.0.1:65227 | sysbench_test | Query   |    0 |       |         SELECT id, k, c, pad
          FROM sbtest1
          WHERE k IN (?)
```

ii. 执行如下SQL语句，创建针对这条SQL的限流规则。

```
create ccl_rule block_select on sysbench_test.* to 'polardbx_root'@'%' for select f
ilter by keyword('pad') with max_concurrency=0;
```

在终端二中执行对select sql进行拦截的SQL语句后，在终端三您可查看到出现大量的SQL报错统计。

```
[ 445s ] thds: 8 tps: 881.86 qps: 881.86 (r/w/o: 881.86/0.00/0.00) lat (ms,95%): 15.00 err/s: 0.00 reconn/s: 0.00
[ 450s ] thds: 8 tps: 843.60 qps: 843.60 (r/w/o: 843.60/0.00/0.00) lat (ms,95%): 16.12 err/s: 0.00 reconn/s: 0.00
[ 455s ] thds: 8 tps: 878.33 qps: 878.33 (r/w/o: 878.33/0.00/0.00) lat (ms,95%): 14.46 err/s: 0.00 reconn/s: 0.00
[ 460s ] thds: 8 tps: 781.57 qps: 781.57 (r/w/o: 781.57/0.00/0.00) lat (ms,95%): 18.28 err/s: 0.00 reconn/s: 0.00
[ 465s ] thds: 8 tps: 386.05 qps: 386.05 (r/w/o: 386.05/0.00/0.00) lat (ms,95%): 23.10 err/s: 612.29 reconn/s: 0.00
[ 470s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 1826.19 reconn/s: 0.00
[ 475s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 2348.84 reconn/s: 0.00
[ 480s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 2634.92 reconn/s: 0.00
[ 485s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 2753.47 reconn/s: 0.00
```

iii. 执行如下SQL语句，查看SQL限流具体拦截情况。

```
show ccl_rules;
```

返回结果如下，您可查看SQL限流具体拦截情况。

```
mysql> show ccl_rules;
+-----+--------------+---------+---------+--------+-----------------+-------------+-------------------+-------------------------+--------------------------+----------
----+--------------+------------+----------+----------------+----------------+----------+-------------+-------+---------------------+
| NO. | RULE_NAME    | RUNNING | WAITING | KILLED | MATCH_HIT_CACHE | TOTAL_MATCH | ACTIVE_NODE_COUNT | MAX_CONCURRENCY_PER_NODE | WAIT_QUEUE_SIZE_PER_NODE | WAIT_TIME
OUT | FAST_MATCH | LIGHT_WAIT | SQL_TYPE | USER           | TABLE          | KEYWORDS | TEMPLATE_ID | QUERY | CREATED_TIME        |
+-----+--------------+---------+---------+--------+-----------------+-------------+-------------------+-------------------------+--------------------------+----------
----+--------------+------------+----------+----------------+----------------+----------+-------------+-------+---------------------+
|   1 | block_select |       0 |       0 | 535116 |          535115 |      535116 |                 1 |                        0 |                        0 |
600 |            1 |          1 | SELECT   | polardbx_root@% | sysbench_test.* | ["pad"]  | NULL        | NULL  | 2022-06-10 14:36:53 |
+-----+--------------+---------+---------+--------+-----------------+-------------+-------------------+-------------------------+--------------------------+----------
----+--------------+------------+----------+----------------+----------------+----------+-------------+-------+---------------------+
1 row in set (0.00 sec)
```

2. 用SQL Advisor优化慢SQL。

> **说明**
>
> 在对慢SQL进行限制后，我们的系统就可以恢复正常状态了，那么接下来就可以对SQL进行优化。PolarDB-X提供内置的SQL Advisor功能，可以针对某条SQL给出具体的优化建议。

i. 执行如下，使用SQL Advisor功能分析SQL语句。

```
explain advisor SELECT id, k, c, pad from sbtest1 where k in(10)\G
```

返回结果如下，在ADVISE_INDEX部分，就是SQL Advisor给出的建议。

```
mysql> explain advisor SELECT id, k, c, pad from sbtest1 where k in(10)\G
*************************** 1. row ***************************
IMPROVE_VALUE: 37.1%
  IMPROVE_CPU: -60.7%
  IMPROVE_MEM: -100.0%
   IMPROVE_IO: -83.3%
  IMPROVE_NET: 37.5%
 BEFORE_VALUE: 1.37550055E7
   BEFORE_CPU: 5.5
   BEFORE_MEM: 0
    BEFORE_IO: 1
   BEFORE_NET: 2.7
  AFTER_VALUE: 1.00300141E7
    AFTER_CPU: 14
    AFTER_MEM: 105.6
     AFTER_IO: 6
    AFTER_NET: 2
 ADVISE_INDEX: ALTER TABLE `sysbench_test`.`sbtest1` ADD GLOBAL INDEX `__advise_index_gsi_sbtest1_k`(`k`) DBPARTITION BY HASH(`k`);
     NEW_PLAN:
Project(id="id", k="k", c="c", pad="pad")
  BKAJoin(condition="_drds_implicit_id_ = _drds_implicit_id_ AND id <=> id", type="inner")
    IndexScan(tables="SYSBENCH_TEST_000002_GROUP.sbtest1__what_if_gsi_k_57bz", sql="SELECT `id`, `k`, `_drds_implicit_id_` FROM `sbtest1__what_if_gsi_k` AS `sbtest1__w
hat_if_gsi_k` WHERE (`k` IN(?))")
    Gather(concurrent=true)
      LogicalView(tables="[000000-000007].sbtest1_sInX", shardCount=8, sql="SELECT `id`, `c`, `pad`, `_drds_implicit_id_` FROM `sbtest1` AS `sbtest1` WHERE ((`k` IN(?
) AND (`_drds_implicit_id_` IN (...)))")

         INFO: GLOBAL_INDEX
1 row in set (0.74 sec)
```

ii. 执行SQL Advisor给出的建议SQL语句。

```
ALTER TABLE `sysbench_test`.`sbtest1` ADD GLOBAL INDEX `__advise_index_gsi_sbtest1_
k`(`k`) DBPARTITION BY HASH(`k`);
```

iii. 执行如下SQL语句，解除SQL限流。

```
drop ccl_rule block_select;
```

终端二中执行解除SQL限流的SQL语句后，在终端三您可查看到QPS在优化后进行了大幅度的提升。

```
[ 1385s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 849.73 reconn/s: 0.00
[ 1390s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 1410.80 reconn/s: 0.00
[ 1395s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 1593.07 reconn/s: 0.00
[ 1400s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 1955.48 reconn/s: 0.00
[ 1405s ] thds: 8 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,95%): 0.00 err/s: 2035.29 reconn/s: 0.00
[ 1410s ] thds: 8 tps: 242.95 qps: 242.95 (r/w/o: 242.95/0.00/0.00) lat (ms,95%): 41.10 err/s: 266.75 reconn/s: 0.00
[ 1415s ] thds: 8 tps: 903.37 qps: 903.37 (r/w/o: 903.37/0.00/0.00) lat (ms,95%): 28.16 err/s: 0.00 reconn/s: 0.00
[ 1420s ] thds: 8 tps: 1777.57 qps: 1777.57 (r/w/o: 1777.57/0.00/0.00) lat (ms,95%): 8.58 err/s: 0.00 reconn/s: 0.00
[ 1425s ] thds: 8 tps: 2099.08 qps: 2099.08 (r/w/o: 2099.08/0.00/0.00) lat (ms,95%): 6.67 err/s: 0.00 reconn/s: 0.00
[ 1430s ] thds: 8 tps: 2128.47 qps: 2128.47 (r/w/o: 2128.47/0.00/0.00) lat (ms,95%): 6.43 err/s: 0.00 reconn/s: 0.00
[ 1435s ] thds: 8 tps: 2132.50 qps: 2132.50 (r/w/o: 2132.50/0.00/0.00) lat (ms,95%): 6.21 err/s: 0.00 reconn/s: 0.00
[ 1440s ] thds: 8 tps: 2164.78 qps: 2164.78 (r/w/o: 2164.78/0.00/0.00) lat (ms,95%): 6.21 err/s: 0.00 reconn/s: 0.00
```

6.分布式事务与数据分区

PolarDB-X是一个支持计算/存储水平扩展、数据高可用，基于存储计算分离、Shared-Nothing架构和一致性复制协议的分布式数据库。其中，存储计算分离和Shared-Nothing架构，是PolarDB-X为了提供水平扩展能力，与单机数据库相比，增加的两个新的设计：

- 第一个是资源层面的设计，将集群内的节点分为计算节点和数据节点两类，称为存储计算分离。计算节点是无状态节点，通过增加计算节点，提供算力的水平扩展能力。
- 第二个是数据层面的设计，数据按照分区键切分为多个分区，通过将分区迁移到新增的数据节点上，提供存储容量的水平扩展能力。由于分区后数据节点之间不需要共享存储资源，称为Shared-Nothing架构。

本节将介绍关于PolarDB-X在分布式事务和数据分区方面的探索。

6.1. 分布式事务

数据库是用于数据存储的系统，我们使用数据库最主要的需求就是读写数据。设计良好的数据库系统，应该尽可能的屏蔽实现细节。

但是，数据库又是一个非常复杂的系统，需要面对各种情况，比如：代码bug或者硬件故障，可能导致数据库在数据写入的任何阶段，意外崩溃；客户的应用程序可能在一系列连续操作中，突然退出；为了提高吞吐，通常要允许多个客户端并发地修改同一条记录；等等。

出现异常的时候，数据库需要保证失败的操作不会导致数据问题，并且引导用户继续完成业务需求。也就是需要有一个关于数据可靠性的保证，用户按照约定来使用数据库，就可以规避数据问题。那么“事务”就是这个关于数据可靠性的“保证”。

事务的定义

具体来说，事务是一组用来表达业务含义的读写操作的集合，且满足ACID。

示例如下：

```
-- 事务1：转账
BEGIN;
UPDATE account SET balance = balance - 30 WHERE id = "Alice";
UPDATE account SET balance = balance + 30 WHERE id = "Bob";
COMMIT;
```

这一个用来完成转账的事务，包含4条SQL，第一行的BEGIN和最后一行的COMMIT圈定了事务的范围，事务中包含两个UPDATE语句，第一条语句在account表上为id等于Alice的账户减30，第二条为id等于Bob的账户加30。这样就表达了Alice向Bob转账30的业务语义。

对于这样的一个事务，数据库提供ACID四个方面的保障。

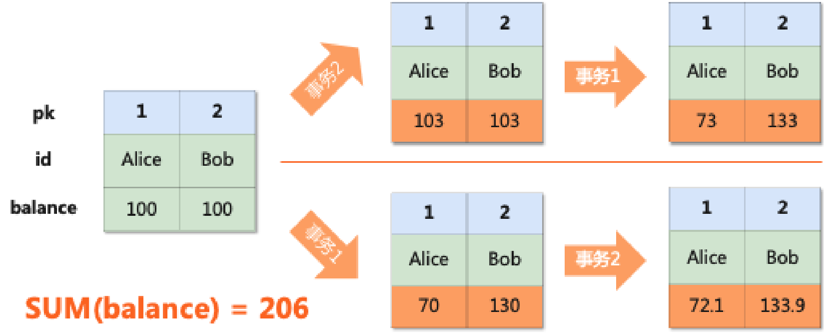
- A是原子性，代表如果出现异常，用户可以通过重试整个事务来解决，无需担心执行失败的事务会对数据产生影响。
- C是一致性，代表用户无需担心数据操作会违反预定义的约束，比如唯一约束、外键等。
- I是隔离性，代表用户可以认为只有自己在操作数据库，无需担心多个客户端并发读写相同数据会导致异常。
- D是持久性，代表一旦事务提交成功，用户无需担心事务产生的变更会因为其它异常而丢失。

原子性和隔离性对比

原子性强调事务提交后，事务内的变更必须一起生效。比如事务1提交后，必须是Alice的账户减30且Bob的账户加30，不能出现Alice减了30而Bob没加，或者Bob减了而Alice没加。

隔离性强调的是多个事务并发执行时，数据库必须使执行结果看起来像没有并发一样。比如数据库可能按照任意顺序收到事务1和事务2中的四条UPDATE，但执行结果应该看起来像是先执行了事务1再执行事务2，或者是先执行事务2再执行事务1，事务执行后，两个账户的总额一定是206。

```
-- 事务1：转账
BEGIN;
UPDATE account SET balance = balance - 30 WHERE id = "Alice";
UPDATE account SET balance = balance + 30 WHERE id = "Bob";
COMMIT;
-- 事务2：结息事务
BEGIN;
UPDATE account SET balance = balance * 1.03 WHERE id = "Alice";
UPDATE account SET balance = balance * 1.03 WHERE id = "Bob";
COMMIT;
```



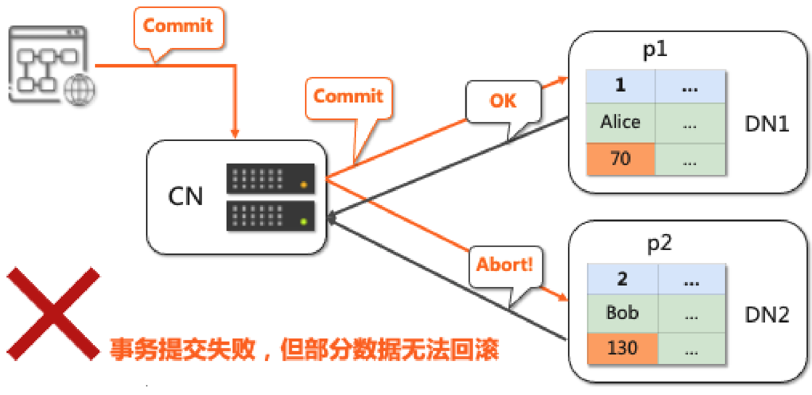
分布式事务中，由于存在多个分区，原子性和隔离性需要重新实现。

2000年前后出现的第一代NoSQL系统，Google的Bigtable和Amazon的Dynamo，都不支持跨分片事务。但后来发现缺少跨分区事务，不符合用户的使用习惯。Google在论文中总结到，发现许多工程师将过多的精力放在处理数据一致性上，原本封装在数据库内部的逻辑，溢出到了应用代码中，大幅提高了应用代码的复杂度。所以目前市面上OLTP类的分布式数据库，都将分布式事务作为一个必选项。

原子性的实现

单分片事务的原子性由日志来保证，写入数据前先记录一个回滚日志，如果事务异常，通过日志来将数据恢复到先前的版本。跨分片事务，还需要协调所有分区在提交阶段的行为，保证一起提交或者一起回滚。

按顺序提交每个分片是否可行呢？以事务1的转账为例。收到用户的提交请求后，向每个分片下发Commit语句，可能出现分片1已经提交成功，但是分片2出现异常回滚，此时整个分布式事务提交失败，但是分片1上的数据无法回滚，产生了不一致。

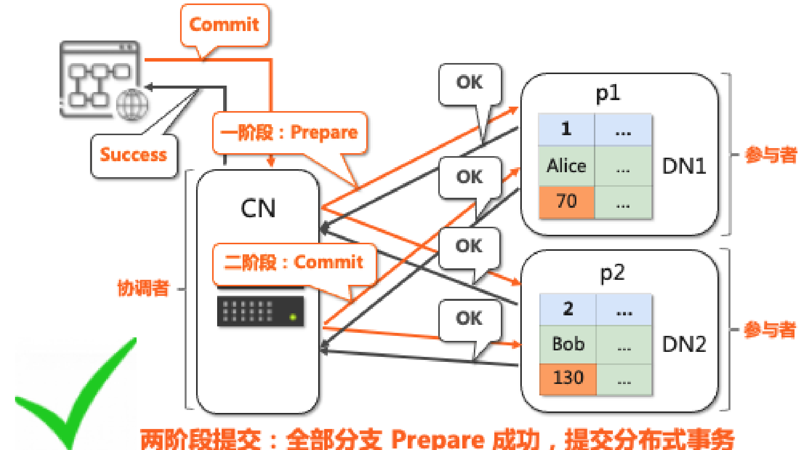


业界通常使用两阶段提交协议解决这个问题。两阶段提交协议中包含两种角色：协调者和参与者。协调者负责判断事务是否能够继续提交（PolarDB-X中使用计算节点作为协调者），参与者代表分支事务涉及的数据节点。

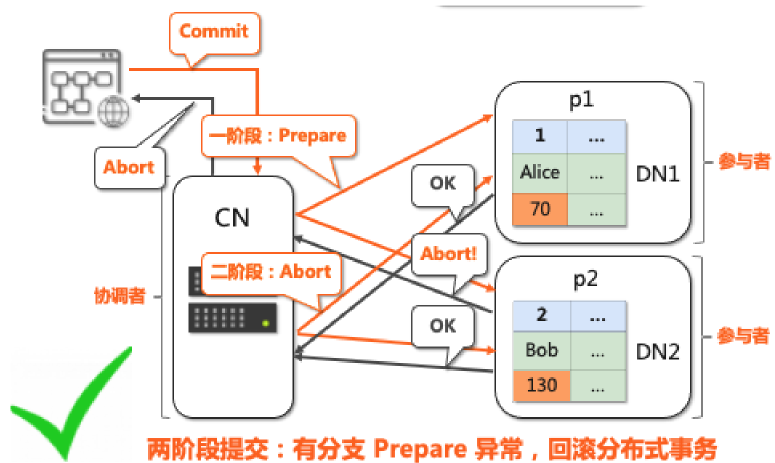
提交过程的第一个阶段称为Prepare阶段。协调者询问每个事务参与者：“分支事务是否能够提交？”事务参与者执行一些校验和日志操作之后，如果能够提交则返回OK。

协调者收到所有OK后，进入第二个阶段——Commit阶段。协调者通知每个参与者：“所有分支都具备提交的条件，可以继续提交了。”然后和一阶段一样，所有参与者提交完成后返回OK。最后，协调者确认所有参与者都完成提交之后，就可以告知用户，分布式事务提交完成。

这里面有一个细节是，一阶段成功后，通常会保存一条日志，用来记录一阶段已经完成可以进入二阶段。这样如果在进入二阶段之前协调者宕机，重启之后依然可以继续执行提交。



再来看一个提交失败的例子。Prepare阶段，依然是协调者首先询问每个参与者：“是否能够提交？”这次有一个分片返回了失败，于是二阶段协调者通知所有参与者回滚事务，收到参与者的成功回复后，返回用户事务已回滚。

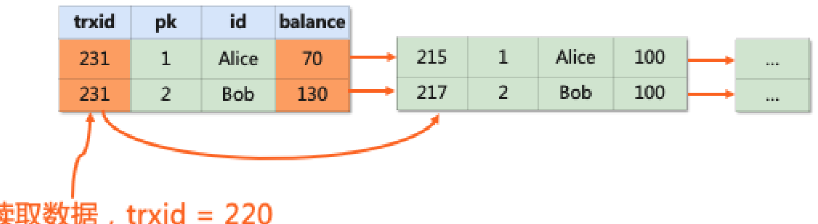


常见的实现有Percolator和XA协议。Percolator在提交阶段延迟较高，只在提交阶段汇报写入冲突，且仅支持乐观锁场景，与传统的关系数据库基于悲观事务的模型有比较大的区别。因此**PolarDB-X选择通过XA协议实现两阶段提交**。

隔离性的实现

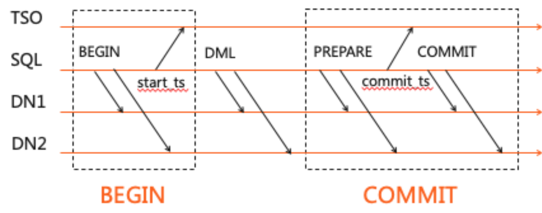
对于单分片事务而言，目前基本所有数据库在做单分片事务的时候，都会采用MVCC（多版本并发控制）方案。举例而言，当更新了Alice和Bob的账户，将Alice的账户余额设置为70，将Bob的账户余额设置为130。在做这个操作的时候，数据库会将老的数据（在转账之前，各自账户都是100元时的数据），会为其生成一个历史版本的快照，并且把每一行数据的快照和写入这行数据的事务ID进行关联。

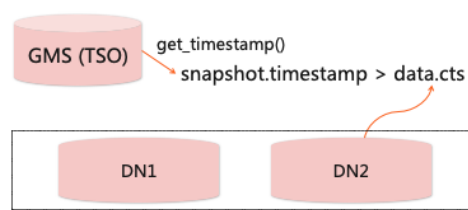
当有一个事务要来读取数据的时候，它会首先根据每一行记录上的事务ID，去看一下这个事务是否提交了。如果是没有提交的事务，那它会认为这个数据一定读不到。所以这个时候就可以继续沿着这个版本链往前找，找到一条已经提交的事务。找到一条已经提交的事务写入的数据之后，它又会根据自己的事务ID，和这个数据上的事务ID去做一个对比，判断一下这条数据是在这个事务开始之前写入的，还是在开始之后写入的。如果是在开始之前写入的，表示能够看到这个版本；如果是在开始之后写入的，那就看不到这个版本。所以这样就可以达到确定一行数据是否可见。



对于分布式事务而言，目前业界主要有两种方案。一种与单机数据库一样，基于活跃事务列表GPM，区别在于将分发事务ID的地点，从每个分片上，改为从一个公共服务上。每次读写事务开始和写事务提交的时候，都会去活跃事务列表里面做一些更新和读取。这种方案对GPM本身的压力较大，它会非常依赖中心化的数据管理器，相对比较容易出现系统瓶颈。

另一种是基于TSO的MVCC方案来支持隔离性，这种方案对于读写操作都有一定的改善。在数据写入的过程中，依然采用两阶段提交。但在一阶段结束之后，会读取提交时间戳，然后将提交时间戳在COMMIT阶段一起下发到每个分片上。这样就将提交时间戳和数据关联到了一起。这也是PolarDB-X所选择的方案。

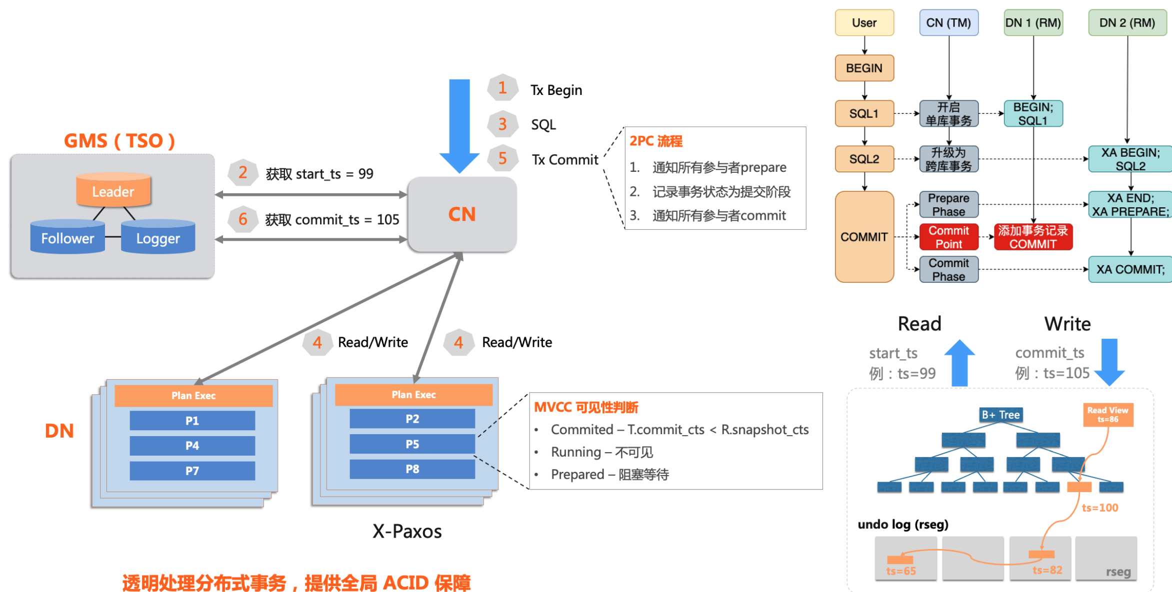




在数据读取的时候，同样也需要获取时间戳，主要用来判断数据是否可见。具体过程为：首先到GMS（TSO）服务获取时间戳，然后作为自己的snapshot timestamp，再将其与和数据相关的数据时间戳进行对比。如果获取到的时间戳比数据时间戳大，并且数据所属的事务已提交，就认为该数据可见；反之，如果该数据所属的事务没有提交，或者获取到的时间戳比数据时间戳小，表示该条数据不可见。

这就是PolarDB-X在实现分布式事务隔离性方面所采用的设计方案。

分布式事务的具体执行过程



1. 分布式事务启动后，首先向TSO获取一个start_ts作为读取操作的快照。
2. 然后接收并处理用户的读写请求，过程中根据数据对应的事务状态、快照的时间戳和数据对应的提交时间戳来判断数据是否可见，保证隔离性。
3. 最后在提交节点，采用两阶段提交，首先通知所有参与写操作的分区执行prepare，然后记录事务状态，最后通知所有参与者commit。
4. 在记录事务状态成功前产生异常都会导致事务回滚，保证原子性。

采用2PC和TSO+MVCC方案实现的分布式事务，经常被质疑的问题是提交阶段延迟增加和TSO单点问题，针对这两个问题PolarDB-X都进行了工程实现上的优化。

- **一阶段提交优化解决两阶段提交延迟问题**：两阶段提交由于增加了prepare的阶段，提交阶段的延迟高于单分片事务。但实际上对于单分区写入事务无论读是否跨分片，我们依然可以使用一阶段提交保证原子性，PolarDB-X支持自动识别这种情况，显著减少这类场景下的提交延迟。
- **GMS性能优化解决TSO单点问题**：TSO方案由于采用单点授时，一个潜在的问题是可能存在单点故障和单点性能瓶颈风险。PolarDB-X的GMS服务部署在三节点集群上，通过X-Paxos协议保证服务高可用。同时PolarDB-X多种场景进行了优化，使得带拆分条件的点查、点写，可不依赖GMS，提升查询性能同时也降低GMS的压力。另外，对于单个CN进程，默认采用grouping的方式将同一时间发生的多个TSO请求合并为batch操作，一次性获取，保证CN和GMS之间交互的指令包不会过多，进一步保证GMS不会成为系统瓶颈。

> **说明**
>
> 详情可参考演示视频（29分03秒开始）。

6.2. 数据分区

当数据库采用了Shared-Nothing架构后，数据全部按照分区键进行分区，这样对数据查询是有一定影响的。本节将介绍具体的影响以及对应的解决方案。

分区键、二级索引和全局二级索引

where id=14

Hash(14)=p3

	p1			p2			p3			p4			p5		
ID	10	12	15	24	1	19	22	3	14	23	9	16	25	7	18
NAME	Hugh	Albert	Clark	Milo	Ward	August	Omar	Morg	Megan	Isaac	Emily	Yedda	Berg	Bennet	Rose
ADDR															

```
CREATE TABLE t1(
    id INT,
    name CHAR(32),
    addr TEXT,
    PRIMARY KEY (id)
) PARTITION BY HASH(id)
```

考虑一张按照主键ID作为分区键拆分的表——t1表（如上图所示，PARTITION BY HASH(ID)）。当我们提供的查询是ID上的等值查询时，例如ID=14，我们计算出14的哈希值之后，就可以快速的定位到P3这个分片。如果我们要按照WHERE NAME来查，要怎么办？如果扫描全部的分片，代价会变高。

单机数据库的解决方案

在MySQL中，同样的表结构实际上是一颗B+树。这颗B+树按照主键id有序。当我们以id为条件进行查询时，实际上数据库会在B+树上做二分查找，从而快速定位到数据所在的叶子节点。同样的，当我们以name为条件进行查询时，就无法做二分查找了，只能将整颗B+树都遍历一遍，这个代价是比较高的，这就叫全表扫描。

在MySQL中，为了避免全表扫描，我们会在name上创建**一个二级索引**。二级索引实际上就是创建了另一颗B+树。这颗B+树按照索引列（也就是name列）有序，其叶子节点记录了name以及其对应的主键id。对于name上的等值查询，数据库会先在二级索引这颗B+树上进行二分查找，找到对应的id后，再使用这个id到主键那颗B+树上进行查找，就是常见的回表操作。

二级索引的理念，在计算机中很常见，是典型的空间换时间。在MySQL中，我们设计主键更多只考虑其业务上的唯一特性，而不必在意这个列是否是查询最多的那个列，就是因为我们可以非常低成本的去创建二级索引，甚至想按什么查，就在哪个列上创建一个索引。这是使用单机数据库一个非常常见的思路。

分布式数据库的解决方案

参考空间换时间的思路，以name为分区键，再将数据做一份冗余，记录name到id的映射。这份冗余的数据就称为**全局二级索引**。全局是指索引的数据和主表的数据没有保存在同一个分片上，通常是因为拆分维度的不同（比如此处一个按照name拆分，一个按照ID拆分，这样通常就会保存在不同的分区上）。

当按照name=megan做查询时，先通过全局二级索引，定位到megan所在的分片，找到megan的id值，再根据id值到主表上查到整条记录所在的分区。这个过程与单机数据库相同，称之为回表。

如果我们在分布式数据库上创建全局二级索引，能够做得像在单机数据库上创建索引一样，那我们就不用在关注分区键了。一切的关键，都在全局索引。

> 说明
>
> 有关使用全局索引优化查询性能的示例，请参见演示视频（38分31秒开始）。

全局索引与单机索引的兼容性

全局索引要尽可能做到与单机索引一样的兼容性。兼容度越高，也就越透明，开发者就更能够使用单机数据库的使用经验来使用分布式数据库。兼容性体现在很多方面，这里列举了比较重要的几个方面以及PolarDB-X是如何解决这类问题的。

兼容项	MySQL中的索引	不透明的“全局索引”
一致性	强一致	最终一致、弱一致、XX一致
创建方式	DDL语句	第三方工具，订阅Binlog
使用方式	优化器自动选择	手动选择
兼容其他DDL	支持	不支持
前缀查询	支持	不支持
BigKey	无太大影响	单节点瓶颈

- **数据的一致性**：PolarDB-X支持强一致分布式事务，通过事务来保证主表与索引表数据强一致。
- **创建全局索引**：创建全局索引的过程中是不会阻塞用户读写的。

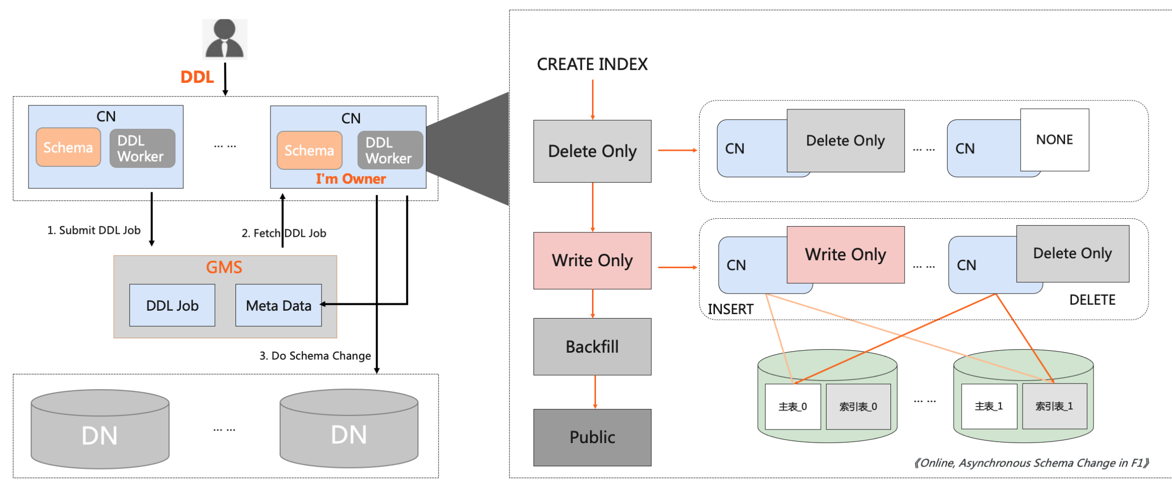
  分布式数据库中，实现Online Schema Change的一个主要挑战是，Schema变更过程中不同CN节点上的事务，看到的元数据版本可能不一致。

  单机数据库解决这个问题的办法是，通过对元数据加锁，保证任意时刻所有事务都只能看到一个元数据版本。

  但分布式系统中，由于网络延迟存在不确定性，实现跨节点的加锁操作可能导致明显的读写卡顿，不符合Online的定义。

  PolarDB-X参考实现了Google F1的Online, Asynchronous Schema Change，通过增加两个相互兼容的中间状态，允许系统中同时存在两个元数据版本，使得DDL过程无需加锁，不会阻塞读写请求。

  有了Online Schema Change的支撑，PolarDB-X中，使用CREATE INDEX语句即可轻松创建全局索引，无需依赖第三方组件。



- **索引变更（DDL兼容性）**：在各种DDL操作中维护索引表数据一致是一项工作量比较大的内容，PolarDB-X支持无锁化的Online DDL，根据DDL类型自动选择合适的执行方式，并且能够自动维护全局索引。
- **基于CBO的自动索引选择**：索引创建完成后，在使用索引时，还需要让它能够自动应用在查询中。由于索引本身也是一张表，且回表操作可以理解为索引表和主表在主键上做JOIN，可以大部分复用估算分区表上执行计划代价的方法，来估算使用索引扫描的执行计划的代价。难点在于，两个表做Join，各自有3个global index和没有global index，执行计划空间差异很大，需要更复杂的枚举算法。PolarDB-X支持基于代价的优化器，自动完成索引选择。
- **前缀索引**：单机数据库中，索引是可以很好的支持前缀查询的，比如MySQL在创建多列索引的时候，是将每一列索引的数据按顺序拼在一起，所以当只有前面几列数据的时候，依然可以拼出一个前缀，然后在树上查找。

  对于全局索引而言，当创建多列索引时，PolarDB-X会分别对每个列进行hash，这样在匹配到前缀的情况下，依然能做一定的分区裁剪。
- **BigKey**：在单机数据库中，创建索引相对是比较随意的。比如我们在性别这一列上创建了一个索引，这个索引区分度比较低，一般不会被用到，这样在写入的时候确实会有一定的争抢，但也不会有太大的问题。

  但是如果是分布式数据库，这个索引由于只有两个值，按照哈希进行分区的话，会只分布在两个节点上。假如业务是以写入操作为主，那么我们会发现，无论集群有多少台机器，最后的瓶颈都在这两个节点上，这就失去了使用分布式数据库提升扩展性的初衷。因此全局索引一定要能解决这种BigKey问题才能降低使用的门槛。

  PolarDB-X通过将主键和索引Key一起作为分区键，使得当热点出现时，能够按照主键进一步进行分裂，从而消除热点。

对于拆分算法和分区分布相同的两张表或者索引，PolarDB-X支持将Join下推到存储节点执行，在OLTP场景下可以显著降低网络开销，提升查询性能。

但如果不对分区操作进行限制，可能会由于分区分裂、合并或者分区迁移，导致两张表的分区分布变得不同，进而导致Join无法下推。从用户角度来看，可能出现一种迁移分区之后，反而导致查询性能下降的困惑。

为了解决这个问题，PolarDB-X在后台引入了两个概念，Table Group和Partition Group。

- Table Group是一组Global Index的集合，系统会确保同一个Table Group中的索引具备相同的数据分布。当发生分区分裂/合并时，Table Group中的索引会一起进行分裂合并，保证分区数量一致。
- 同一个Table Group中的Global Index，区间相同的分区组成Partition Group。PolarDB-X会确保同一个Partition Group中的分区，始终落在相同的DN上，保证分区迁移操作不会影响Join的下推。

总结：合理划分Table Group，可以降低分区迁移等操作的代价；PolarDB-X使用Table Group与Partition Group来对全局索引上的Join进行下推优化。

7.X-Paxos三副本与高可用

7.1. DN高可用方案

在PolarDB-X的系统架构中，DN是提供存储服务的。一个DN就是一个逻辑节点。在当前架构中，DN是有多个副本的，即由多个MySQL节点组成一个X-Paxos集群，从而保证高可用。

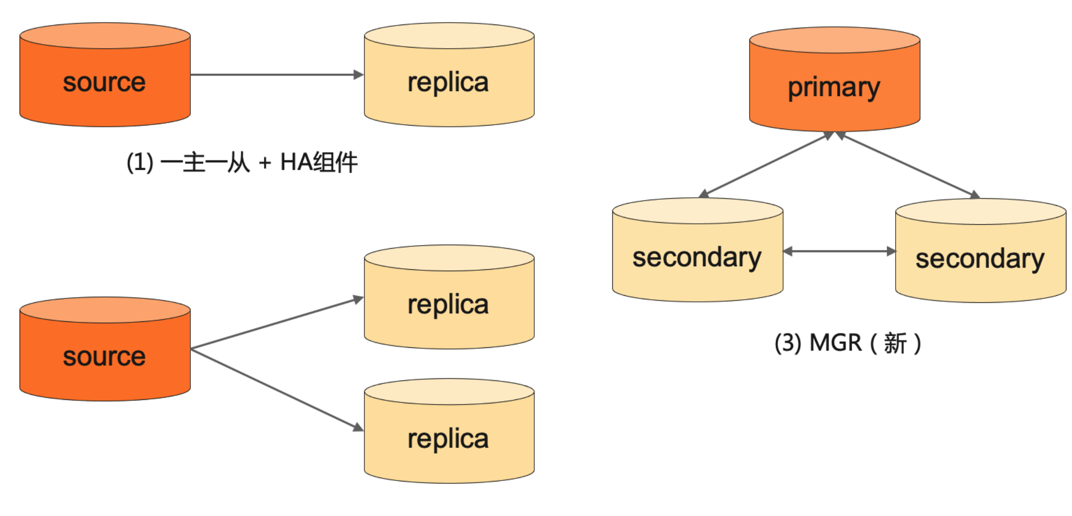
存储服务的要求

存储服务的要求，主要包括以下三个方面：

- **多副本**：某个副本出现问题，其他几个副本仍然能够保存原来的数据。
- **高可用**：通过副本的数据传输方式，可以实现其他备份节点继续提供服务的能力，即高可用。
- **自管理**：存储组件在运行的时候，不希望引入其他组件来管理集群，也就是存储组件自身是一个自闭环的存储服务。

MySQL经典高可用模式

MySQL家族常见的几种高可用方式如下图所示，包括一主一从、一主多从、MySQL社区的MGR、基于共享存储的部署方案等。不同的高可用模式都有各自的应用场景。



总体而言，每一种高可用模式需要关注的问题包括：

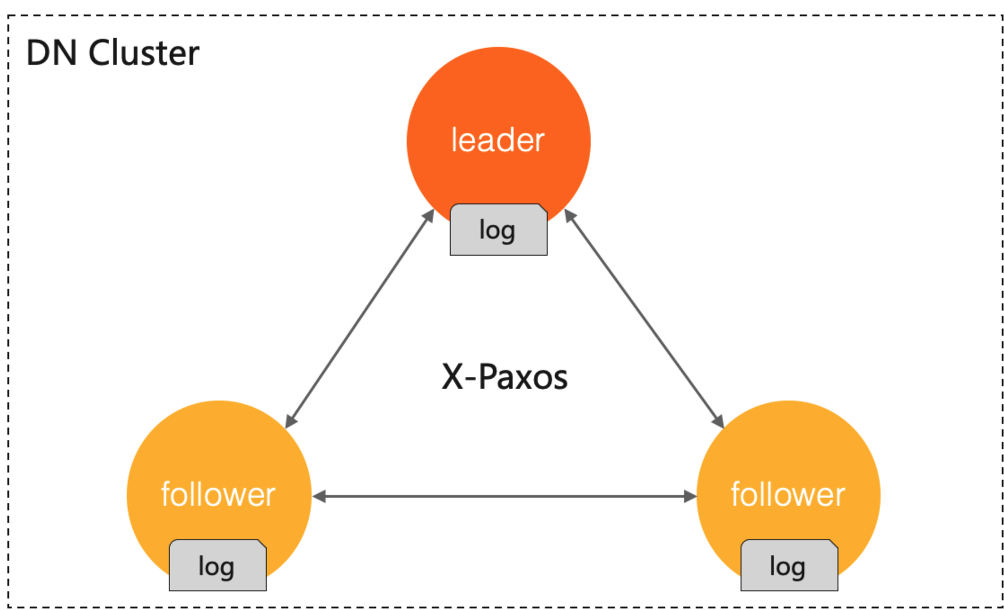
- 分区故障可用性（副本数）
- 全局数据一致性（RPO）
- 服务探活和切换（管理复杂度）
- 性能

DN集群强一致方案

最初，PolarDB-X的DN集群就是最基础的一主一从结构，用semisync半同步构建的主备模式。这种模式最大的问题在于，在对数据一致性要求较大的场景（比如金融业务），仍然存在数据丢失的风险。比如在运行过程中，主库宕机备库接管，主备库之间的数据是没有一定标准的。如果备库接管后，以备库自己现有的数据为标准，再继续提供服务，对于数据用户而言，之前提交的事务是否存储在该备库上无法确定，即RPO不等于0。

现在DN集群采用基于一致性协议组织的多副本。在该模式中，有2N+1个副本，可以容忍最多N个节点故障。也就是只要不超过N个节点故障，整个集群还是可以正常提供服务的。但如果超过N个节点故障，整个集群就拒绝服务。当节点数大于N+1个的时候，节点副本间的数据传输是用一致性协议来控制的，从而可以达到RPO为0。目前采用的X-Paxos一次性协议的模式，有一个leader节点提供读写服务。一旦leader节点挂了，故障检测和leader节点快速切换，就成为非常关键的问题。

X-Paxos协议基本结构图如下。

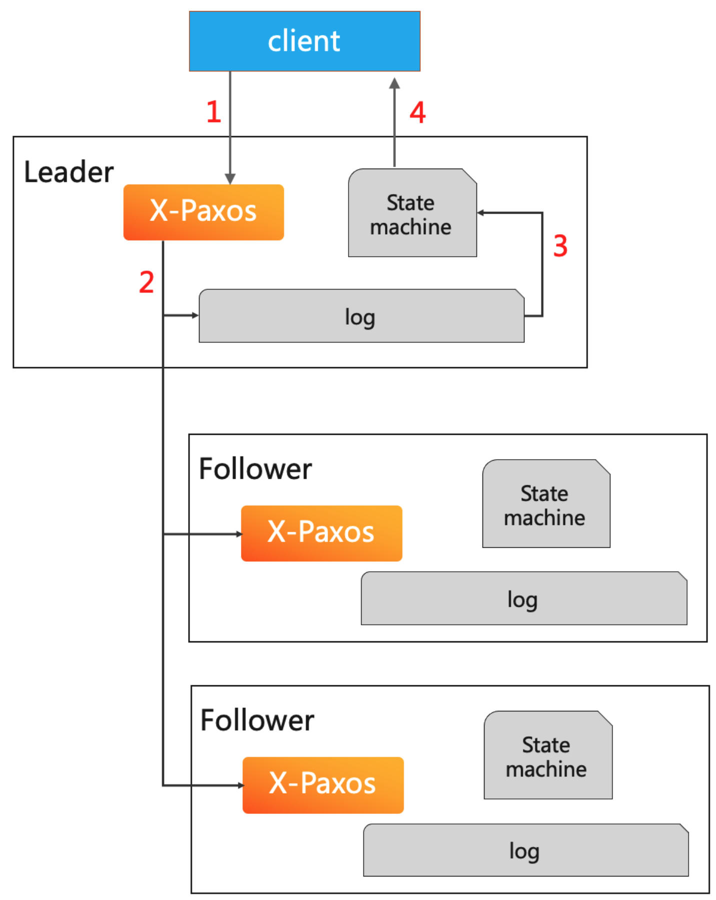


7.2. X-Paxos协议

协议简介

X-Paxos是阿里巴巴自研的Paxos协议实现。具体而言，我们将其抽象为如下过程。

一个节点的数据就是一个状态机。对这个节点的数据的一次修改，将其抽象为一条日志。将该日志应用到状态机上，就说明该节点的数据就变化了一次。当集群正常运行的时候，有一个Leader。集群初始化的时候，状态机是初始化的状态，日志都是空的，可以随机选举出一个节点作为Leader。接下来系统在运行的时候，就围绕Leader运转。



当client对节点数据执行一次修改，这次修改会形成一条日志，Leader节点将日志写到本地日志中，同时将该日志广播到其他所有Follower节点，在收到其他所有Follower节点的应答之后，说明这条日志在集群里提交了，接下来就可以应用状态机，向client返回应答。

这就是正常系统的运行过程。在这个过程中：

- Leader是主节点，提供读写服务，并参与选主。
- Follower接收主节点的日志。当Leader挂掉之后，Follower也会发起选主请求参与选主。
- Learner节点（图中未展示）也接收Leader的日志，但不参与选主；换言之，Learner节点就是集群数据的订阅者。可以使用Learner构建只读节点，它不参与集群的运作，只订阅集群数据。

在该模型中，关键有三个部分：

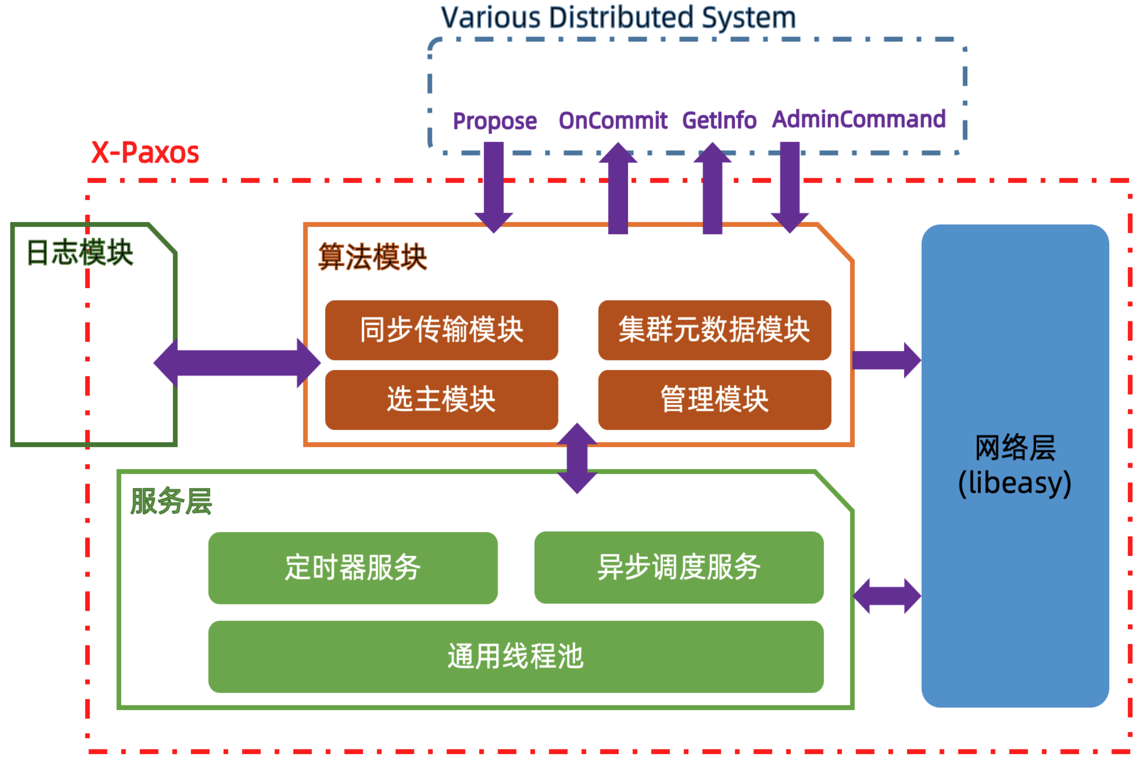
- **选主原则**：集群初始化时，状态机为空，日志为空，可以随机选主。当系统运作一段时间后，如果当前的Leader挂了，需要从剩余节点中选主的话，剩余节点都可以发起选主请求，但只有拥有集群已经达到一致性状态的日志的节点，才有可能选为新的主节点。
- **日志复制**：消息从client发出后，在集群中广播。
- **节点维护**：向集群中添加或删除节点。

工程框架

X-Paxos协议模块的工程框架如下图所示，共分为四个部分：

- 网络层（libeasy）：阿里内部实现的异步网络通信框架。
- 服务层：提供定时器服务、消息异步传输服务和线程池。服务层是协议运转的驱动齿，驱动协议模块的运行。

- 算法层：算法层的逻辑就是前面介绍的选主、日志传输和DN集群管理部分的内容。
- 日志模块：日志模块本身属于算法模块中的一部分（因为算法在日志传输过程的部分，所维护的就是日志，所以在工程中将日志模块单独抽出来，以实现性能优化）。



7.3. DN高可用体系

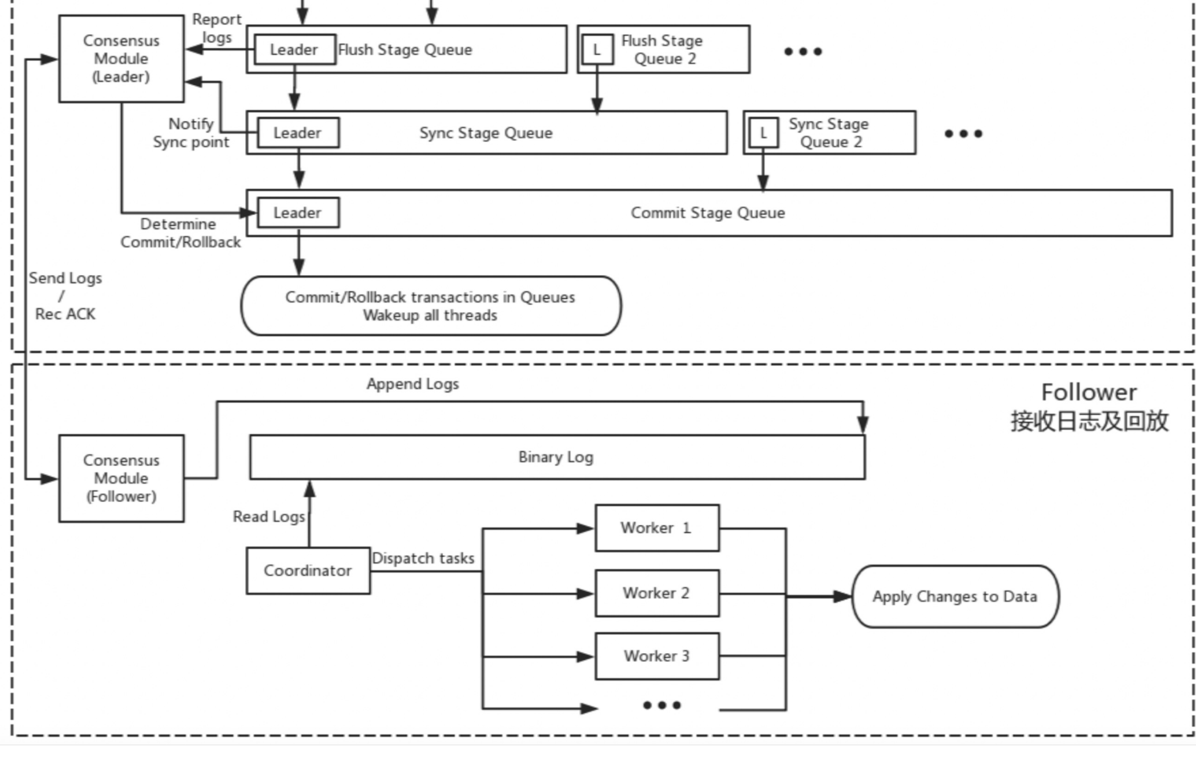
数据一致性

内置X-Paxos协议模块的设计要点如下：

首先，在DN集群里，节点之间传输的还是Binlog。MySQL的事务提交，在有Binlog参与的情况下，是分为两阶段的。第一阶段是引擎Prepare，最常见的就是InnoDB Prepare。第二阶段进入有Binlog协调的Commit阶段。

所有事务都会进入队列。Commit阶段分为三个阶段：第一阶段是Flush，将日志写入到Binlog；第二个阶段是Sync，将队列中所有已经写入Binlog的日志一起刷到盘上；第三个阶段是引擎的Commit。 引入X-Paxos协议后，Commit阶段的过程会有所变化。

- 在第一阶段（Flush阶段），在Leader节点，日志写入到本地Binlog同时，会同时广播到所有Follower节点（通过网络传输）。
- 在第二阶段（Sync阶段），这一组事务的Binlog都已持久化到盘上了，会形成一个持久化的位点，并与协议层同步一下位点。
- 在第三阶段（引擎Commit阶段），协议层的X-Paxos模块，需要等待这一批Binlog在其他大多数节点（多数派）应答之后，才能够在Leader进行这一组事务的提交（即，最终这一批事务的提交点，要根据这批日志在其他节点上的接收情况来决定）。



集群选主

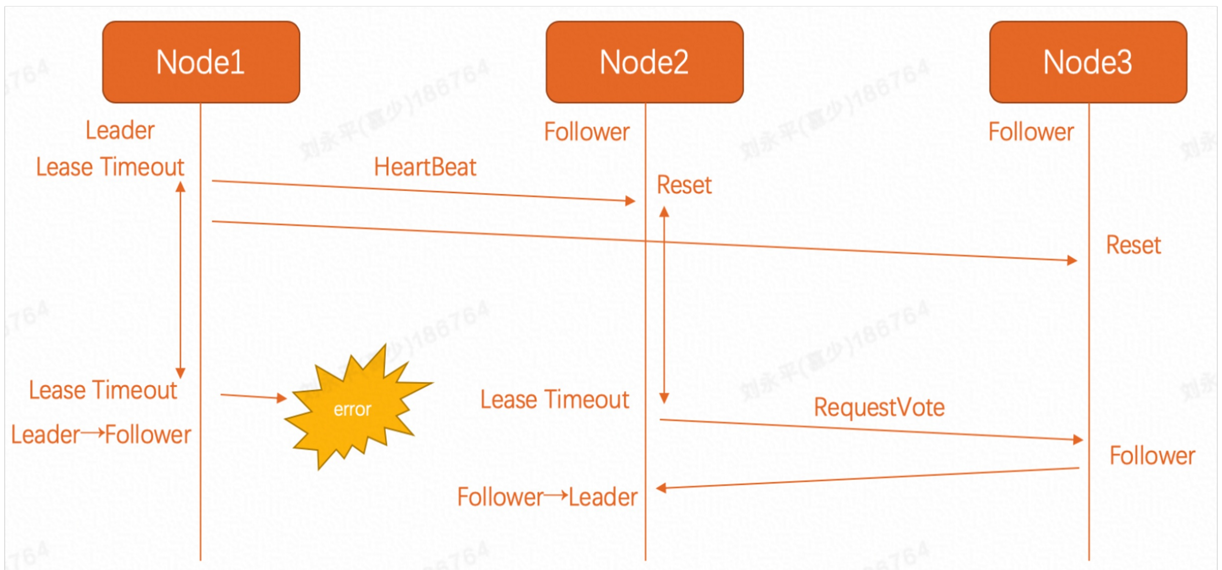
在上面这个模型中，Leader是一个很重要的角色。在集群中，同一时刻不允许有多个Leader存在。Leader节点挂掉之后，需要迅速选出新的Leader来提供服务。

选主租约

对于Leader而言，它自己是Leader。对于Follower而言，它认为集群中有一个Leader。每一个Leader都有一个任期。在这个任期内，所有Follower节点不会再接收其他节点的选主请求。所以在正常情况下，Leader节点通过向其他Follower节点发送心跳，来保证不断延长其任期。这是绝大部分情况下的工作状态。

对于Follower节点而言，一旦Leader发生异常，每个Follower就会检测到Leader不存在或服务异常了。此时每个Follower都会发起选主请求，请求选择自己为Leader。如果其他大多数节点都还健康，就能够选出新的Leader节点。

对于Leader节点而言，如果在它的一个任期内，它发现大部分Follower都失去联系了，那么它就会认为这个集群不能够再提供服务了，它就会主动发起降级请求把自己降级为Follower，然后从Follower再重新发起选主请求。这个过程中保证了集群中同时只有一个Leader的存在，而当Leader发生异常时，集群会自发进行选择，选出一个新的Leader。

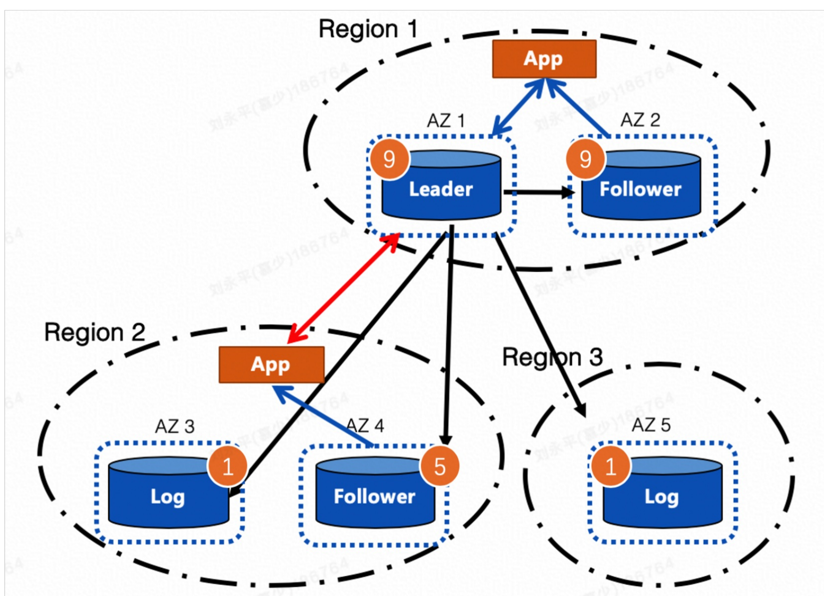


权重选主

默认情况下，集群中的每个节点当选为Leader的概率是相等的。

但在有些情况下，我们希望集群中的某些节点或某个区域的节点，具有更高的优先级能当选为Leader。此时，我们可以给每个节点配置一个选主权重。权重高的节点，有更高的概率当选为Leader。 但这只是使得当选Leader的概率比较高，并不能保证这个节点百分之百就是高权重节点、就一定会当选为Leader。因为选主的最基本原则是这个节点要拥有集群当时已经达成共识的最新的日志，如果它的日志不够，是不能当选为Leader的。

权重选主是部署方面的优化。



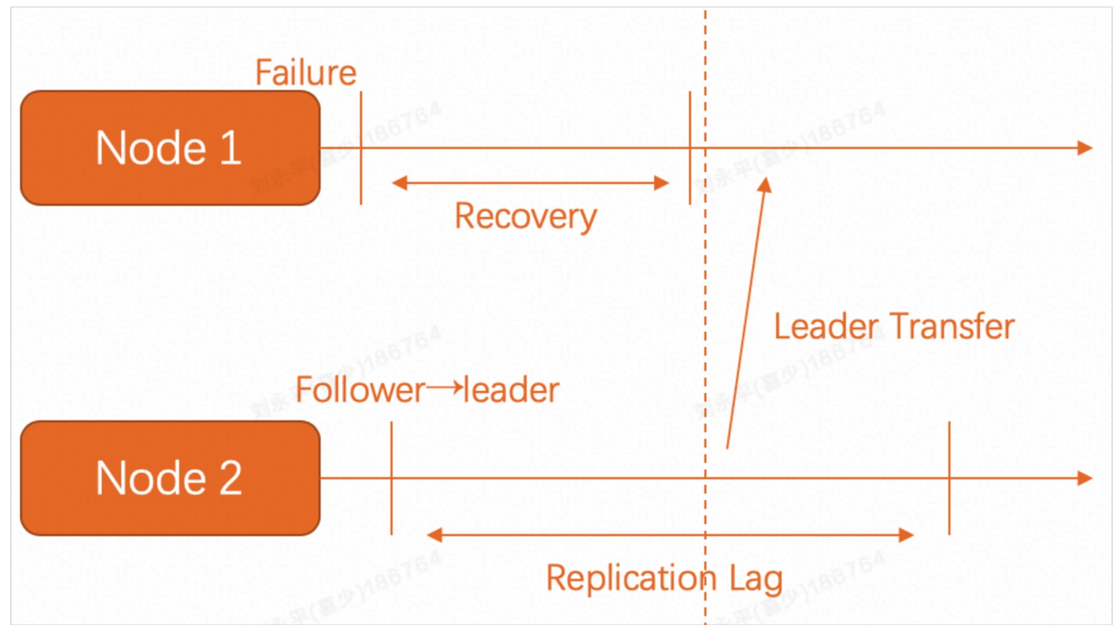
状态机诊断

复制延迟场景快速恢复服务

集群节点间复制的是Binlog日志。主节点将Binlog日志传输到Follower之后，Follower表示收到了该日志，这样集群中对这个日志就达成了一个共识，也就是这个日志在集群中都存在。

但是对于Follower节点而言，并不会要求它一定要把当前日志应用到一次性状态位点后才能继续。所以集群正常运转的时候，只有Leader节点保持了集群中最新的数据。Follower节点有最新的日志，但它的状态机（MySQL集群的数据）不一定是最新的，因为它的Binlog可能还有一段没有应用（存在间隔）。

所以当原Leader异常之后，新的Leader被选出来。新Leader在对外提供读写服务之前，首先要把Binlog应用到最新的位点（追平日志）才能服务。在一些特殊的情况下，Binlog延迟可能比较大，这时原来的Leader又可以重新提供服务了，这时原来的Leader提供服务是更合适的，因为它的日志是最新的，也不需要重新再去回放。综上，被选为新Leader的Follower节点，在它还不能及时提供服务的时候，它也会不断探测其他已经接到集群的节点。如果发现有比它更合适的节点（应用日志更快、更新，权重更大，等待），它就会把它的Leader主动让出去，从而更快地恢复服务。

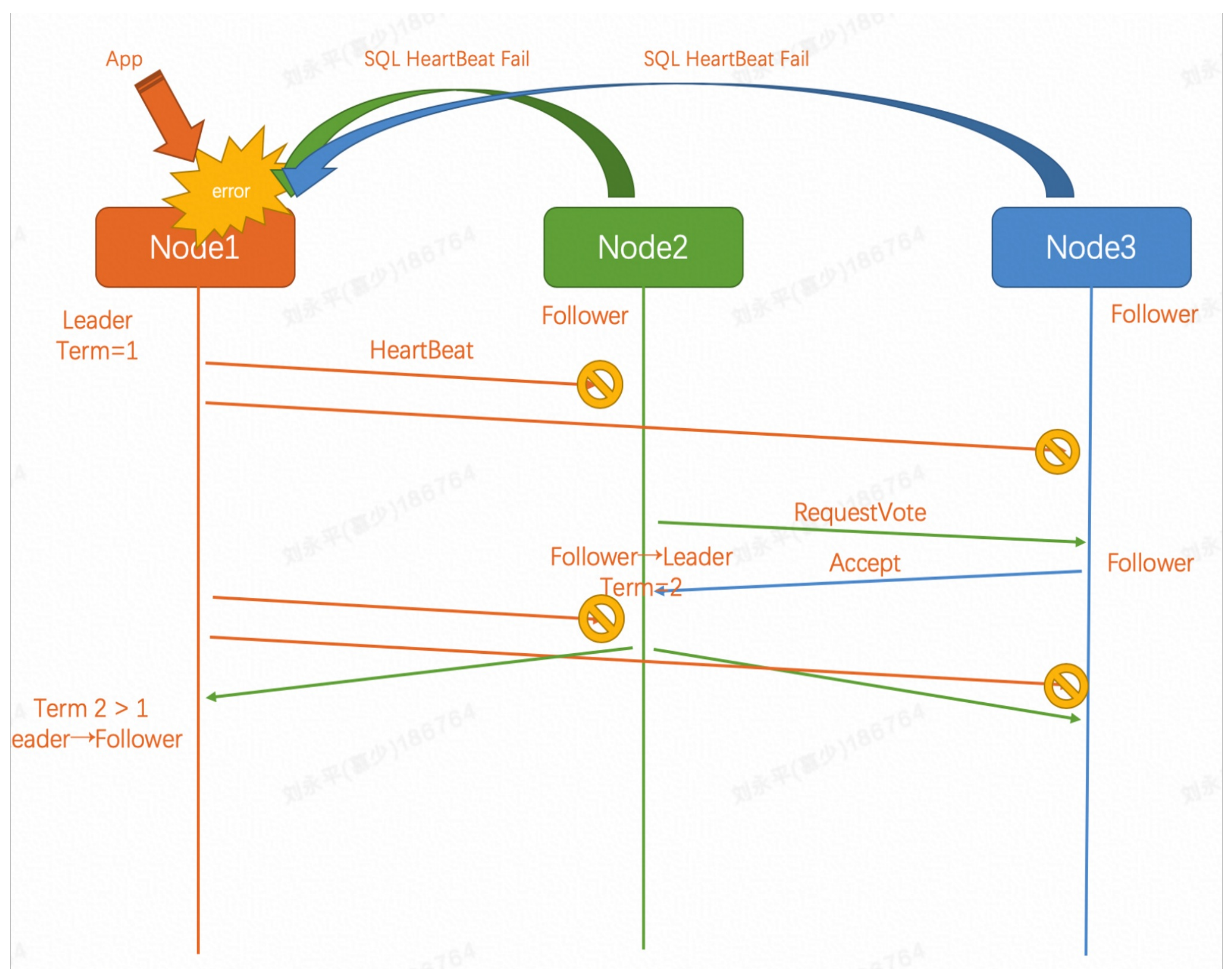


反向心跳

集群能否对外提供服务（Leader节点能否对外提供服务），就是MySQL能否对外提供服务。从集群或协议层来看，集群是健康的，它每个节点的心跳、日志复制都是健康的。但很可能由于Leader节点由于其他原因，不能对外提供服务（如磁盘满、系统问题等），这种情况下，如果没有外力干扰的话，集群就可能一直僵持（协议没有问题，Leader节点因为其他系统原因无法服务，异常状态无法终结）。

因此，在开源Galaxy引擎里面，协议中加入了反向心跳的功能。Follower节点会定期向Leader节点发起探测请求。这个探测请求其实是从把自己视为正常的客户端或App的视角，来探测Leader节点的数据服务是否可用。如果服务不可用，原Leader节点会自动进行角色降级，然后整个集群重新选主。

反向心跳功能是为了整个集群高可用的一个自闭环的功能。



7.4. DN部署和优化

> 说明
>
> 本节实验操作部分主要通过阿里云官方网站的云起实验室进行，详情可登录阿里云官方网站，访问用PolarDB-X搭建一个高可用系统了解。

场景优化

Logger节点

从协议层面看，Logger节点不是一种新的角色，就是一个正常的节点。大部分情况下，它是一个Follower，只从Leader节点接收了Binlog日志，但不应用（没有状态机），只保留一份日志，没有MySQL实例的数据。

这样设计是为了节省成本。因为在这种模式下，只有Leader和Follower节点有真正数据，Logger是没有数据的。引入Logger节点是为了在使用两份数据拷贝的情况下，达到整个集群数据的强一致，为这种特殊场景所做的优化。

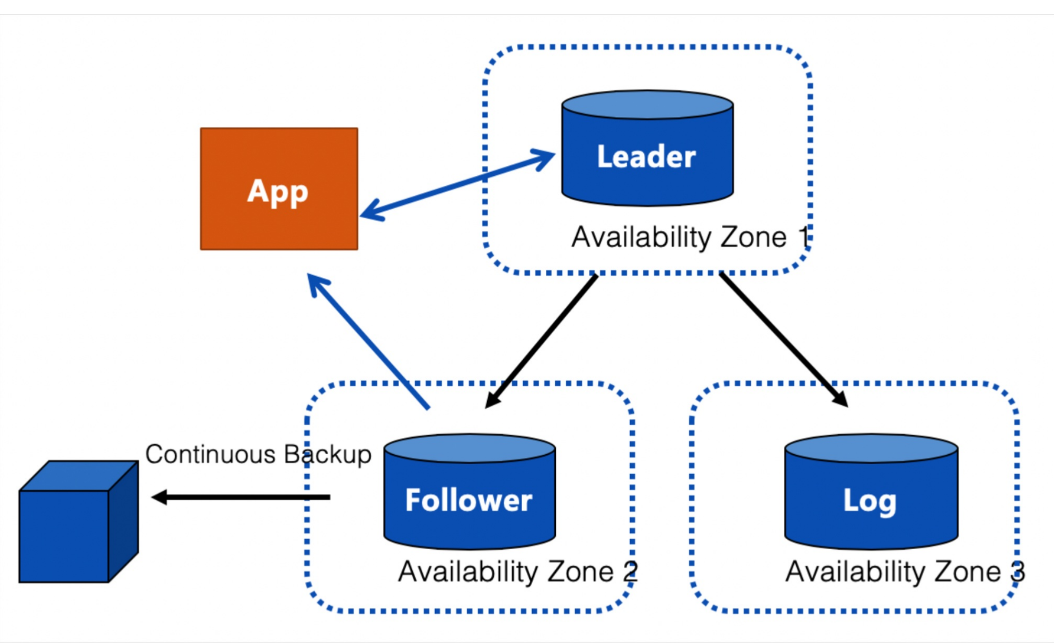
Learner节点

Learner节点是只读节点，会从集群中同步Binlog，也有自己的状态机，也要应用Binlog。Learner节点是一份正常的数据拷贝，但它可以对外提供只读服务，这是为了进行计算能力的扩展（原来只有一个Leader提供读写服务，这个Learner可以提供计算服务）。

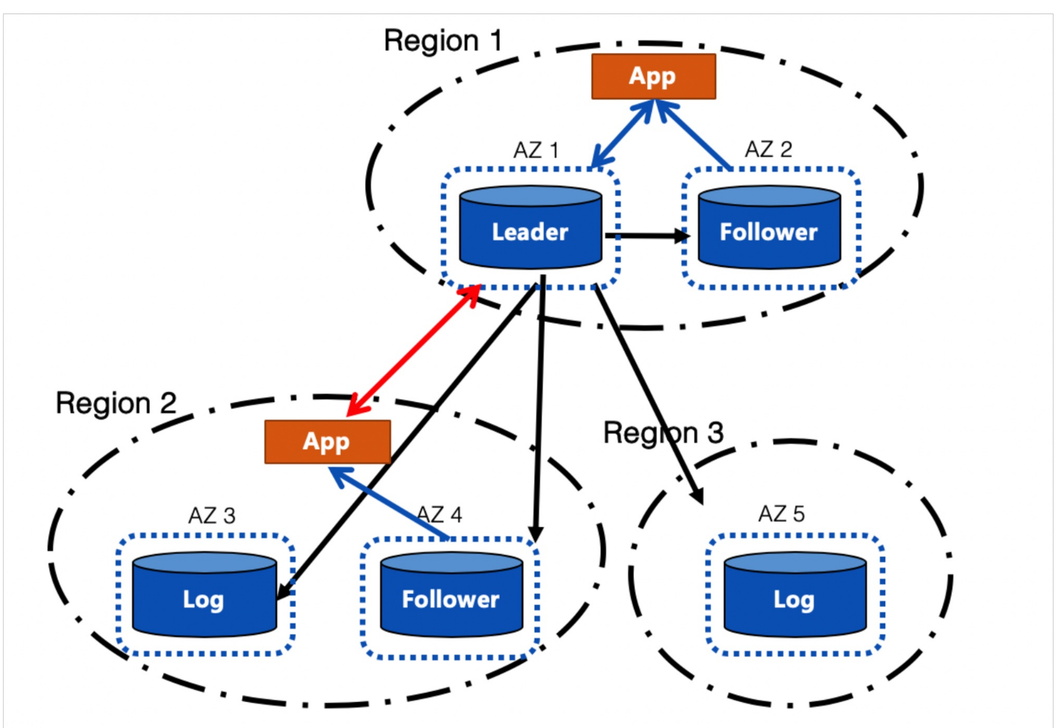
部署模式

同城三副本在三个可用区各有一个节点（即两个实体节点，一个Logger节点）。这种简化的模式相比于主备模式，基本上不增加存储成本，但可以达到强一致性状态。在公共云很多场景下，同城三副本能够达到使用要求。如果想要数据一致性更强一点，可以再做跨城五副本。

同城三副本



跨城五副本



7.4.1. 第1步：安装环境

本节将介绍如何安装Docker、kubectl、minikube和Helm3。

操作步骤

1. 安装Docker。
   i. 执行如下命令，安装Docker。

   ```
   curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun
   ```

   ii. 执行如下命令，启动Docker。

   ```
   systemctl start docker
   ```

2. 安装kubectl。
   i. 执行如下命令，下载kubectl文件。

   ```
   curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
   ```

   ii. 执行如下命令，赋予可执行权限。

   ```
   chmod +x ./kubectl
   ```

   iii. 执行如下命令，移动到系统目录。

   ```
   mv ./kubectl /usr/local/bin/kubectl
   ```

3. 执行如下命令，下载并安装minikube。

   ```
   curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
   sudo install minikube-linux-amd64 /usr/local/bin/minikube
   ```

4. 安装Helm3。
   i. 执行如下命令，下载Helm3。

   ```
   wget https://labfileapp.oss-cn-hangzhou.aliyuncs.com/helm-v3.9.0-linux-amd64.tar.gz
   ```

   ii. 执行如下命令，解压Helm3。

   ```
   tar -zxvf helm-v3.9.0-linux-amd64.tar.gz
   ```

   iii. 执行如下命令，移动到系统目录。

   ```
   mv linux-amd64/helm /usr/local/bin/helm
   ```

5. 安装MySQL。

   ```
   yum install mysql -y
   ```

7.4.2. 第2步：使用PolarDB-X Operator安装PolarDB-X

本节将介绍如何创建一个简单的Kubernetes集群并部署PolarDB-X Operator，然后使用Operator部署一个完整的PolarDB-X集群。

操作步骤

说明

有关通过Kubernetes部署PolarDB-X集群的详情，请参见通过Kubernetes部署集群。

1. 使用minikube创建Kubernetes集群。

   说明

   minikube是由社区维护的用于快速创建Kubernetes测试集群的工具，适合测试和学习Kubernetes。使用minikube创建的Kubernetes集群可以运行在容器或是虚拟机中。

   本实验场景以CentOS 8.5上创建Kubernetes为例。如果您使用其他操作系统部署minikube，例如macOS或Windows，部分步骤可能略有不同。

   i. 执行如下命令，新建账号galaxykube，并将galaxykube加入docker组中。

      说明

      minikube要求使用非root账号进行部署，所以您需要新建一个账号。

```
useradd -ms /bin/bash galaxykube
usermod -aG docker galaxykube
```

   ii. 执行如下命令，切换到账号galaxykube。

```
su galaxykube
```

   iii. 执行如下命令，进入到home/galaxykube目录。

```
cd
```

iv. 执行如下命令，启动一个minikube。

```
minikube start --cpus 4 --memory 12288 --image-mirror-country cn --registry-mirror=https://docker.mirrors.sjtug.sjtu.edu.cn
```

> **说明**
> 这里我们使用了阿里云的minikube镜像源以及USTC提供的docker镜像源来加速镜像的拉取。

返回结果如下，表示minikube已经正常运行，minikube将自动设置kubectl的配置文件。

```
[galaxykube[illegible]ot]$ minikube start --cpus 4 --memory 7960 --image-mirror-country cn --registry-mirror=https://docker.mirrors.ustc.edu.cn
😄 minikube v1.25.2 on Centos 8.5.2111 (amd64)
✨ Automatically selected the docker driver
❗ Your cgroup does not allow setting memory.
  ▪ More information: https://docs.docker.com/engine/install/linux-postinstall/#your-kernel-does-not-support-cgroup-swap-limit-capabilities
✅ Using image repository registry.cn-hangzhou.aliyuncs.com/google_containers
👍 Starting control plane node minikube in cluster minikube
🚜 Pulling base image ...
  > registry.cn-hangzhou.aliyun...: 379.06 MiB / 379.06 MiB  100.00% 6.55 MiB
🔥 Creating docker container (CPUs=4, Memory=7960MB) ...
🐳 Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
  ▪ kubelet.housekeeping-interval=5m
  > kubelet.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
  > kubectl.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
  > kubeadm.sha256: 64 B / 64 B [--------------------------] 100.00% ? p/s 0s
  > kubeadm: 43.12 MiB / 43.12 MiB [-------------] 100.00% 4.71 MiB p/s 9.4s
  > kubectl: 44.43 MiB / 44.43 MiB [---------------] 100.00% 4.40 MiB p/s 10s
  > kubelet: 118.75 MiB / 118.75 MiB [-------------] 100.00% 7.43 MiB p/s 16s
  ▪ Generating certificates and keys ...
  ▪ Booting up control plane ...
  ▪ Configuring RBAC rules ...
🔎 Verifying Kubernetes components...
  ▪ Using image registry.cn-hangzhou.aliyuncs.com/google_containers/storage-provisioner:v5
🌟 Enabled addons: storage-provisioner, default-storageclass
🏄 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

v. 执行如下命令，使用kubectl查看集群信息。

```
kubectl cluster-info
```

返回如下结果，您可以查看到集群相关信息。

```
[galaxyku[illegible]x9dZ ~]$ kubectl cluster-info
Kubernetes control plane is running at https://19[illegible]8443
CoreDNS is running at https://[illegible]/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

2. 部署 PolarDB-X Operator。

i. 执行如下命令，创建一个名为polardbx-operator-system的命名空间。

```
kubectl create namespace polardbx-operator-system
```

ii. 执行如下命令，安装PolarDB-X Operator。

```
helm repo add polardbx https://polardbx-charts.oss-cn-beijing.aliyuncs.com
helm install --namespace polardbx-operator-system polardbx-operator polardbx/polardbx-operator
```

iii. 执行如下命令，查看PolarDB-X Operator组件的运行情况。

```
kubectl get pods --namespace polardbx-operator-system
```

返回结果如下，请您耐心等待2分钟，等待所有组件都进入Running状态，表示PolarDB-X Operator已经安装完成。

```
[galaxykul[illegible]tZ ~]$ kubectl get pods --namespace polardbx-operator-system
NAME                                         READY   STATUS    RESTARTS   AGE
polardbx-controller-manager-7978bc7bd5-sr5qf   1/1     Running   0          2m11s
polardbx-hpfs-br7wh                          1/1     Running   0          2m11s
polardbx-tools-updater-f2zpw                 1/1     Running   0          2m11s
```

3. 部署 PolarDB-X 集群。

i. 执行如下命令，创建polardb-x.yaml。

```
vim polardb-x.yaml
```

ii. 按i键进入编辑模式，将如下代码复制到文件中，然后按ECS退出编辑模式，输入:wq后按下Enter键保存并退出。

```
apiVersion: polardbx.aliyun.com/v1
kind: PolarDBXCluster
metadata:
  name: polardb-x
spec:
  config:
    dn:
      mycnfOverwrite: |-
        print_gtid_info_during_recovery=1
        gtid_mode = ON
        enforce-gtid-consistency = 1
        recovery_apply_binlog=on
        slave_exec_mode=SMART
  topology:
    nodes:
      cdc:
        replicas: 1
        template:
          resources:
            limits:
              cpu: "1"
              memory: 1Gi
            requests:
              cpu: 100m
              memory: 500Mi
      cn:
        replicas: 2
        template:
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 1Gi
      dn:
        replicas: 1
        template:
          engine: galaxy
          hostNetwork: true
          resources:
            limits:
              cpu: "2"
              memory: 4Gi
            requests:
              cpu: 100m
              memory: 500Mi
      gms:
```

```
        template:
          engine: galaxy
          hostNetwork: true
          resources:
            limits:
              cpu: "1"
              memory: 1Gi
            requests:
              cpu: 100m
              memory: 500Mi
          serviceType: ClusterIP
  upgradeStrategy: RollingUpgrade
```

iii. 执行如下命令，创建PolarDB-X集群。

```
kubectl apply -f polardb-x.yaml
```

iv. 执行如下命令，查看PolarDB-X集群创建状态。

```
kubectl get polardbxCluster polardb-x -o wide -w
```

返回结果如下，请您耐心等待七分钟左右，当PHASE显示为Running时，表示PolarDB-X集群已经部署完成。

```
[galaxykube@i[illegible] ~]$ kubectl get polardbxCluster polardb-x -o wide -w
NAME        PROTOCOL   GMS   CN    DN    CDC   PHASE      DISK      STAGE   REBALANCE   VERSION                               AGE
polardb-x   8.0        0/1   0/1   0/1   0/1   Creating                                                                     2m28s
polardb-x   8.0        1/1   0/1   1/1   0/1   Creating                                                                     3m33s
polardb-x   8.0        1/1   0/1   1/1   1/1   Creating                                                                     5m3s
polardb-x   8.0        1/1   1/1   1/1   1/1   Creating                                                                     6m32s
polardb-x   8.0        1/1   1/1   1/1   1/1   Running    7.2 GiB                       8.0.3-PXC-5.4.13-16534775/8.0.18   6m44s
polardb-x   8.0        1/1   1/1   1/1   1/1   Running    7.2 GiB                       8.0.3-PXC-5.4.13-16534775/8.0.18   7m44s
```

v. 按Ctrl+C键，退出查看PolarDB-X集群创建状态。

7.4.3. 第3步：连接PolarDB-X集群

本节将介绍如何连接通过Kubernetes部署的PolarDB-X集群。

前提条件

已安装完成通过Kubernetes部署的PolarDB-X集群，详情请参见使用Kubernetes安装PolarDB-X。

操作步骤

1. 执行如下命令，查看PolarDB-X集群登录密码。

```
kubectl get secret polardb-x -o jsonpath="{.data['polardbx_root']}" | base64 -d - | xar
gs echo "Password: "
```

返回结果如下，您可以查看到PolarDB-X集群登录密码。

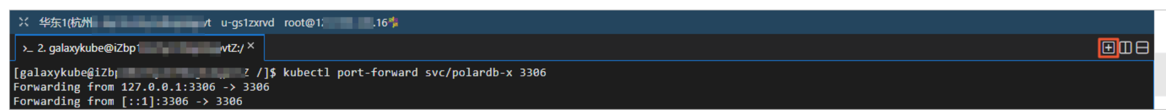
```
[galaxy[illegible] /]$ kubectl get secret polardb-x -o jsonpath="{.data['polardbx_root']}" | base64 -d - | xargs echo "Password: "
Password: wh[illegible]sn
```

2. 执行如下命令，将PolarDB-X集群端口转发到3306端口。

> **说明**
>
> 使用MySQL Client方式登录通过k8s部署的PolarDB-X集群前，您需要进行获取PolarDB-X集群登录密码和端口转发。

```
kubectl port-forward svc/polardb-x 3306
```

3. 在实验页面，单击右上角的图标，创建新的终端二。



4. 执行如下命令，连接PolarDB-X集群。

```
mysql -h127.0.0.1 -P3306 -upolardbx_root -p<PolarDB-X集群登录密码>
```

> **说明**
>
> - 您需要将<PolarDB-X集群登录密码>替换为实际获取到的PolarDB-X集群登录密码。
> - 如遇到mysql: [Warning] Using a password on the command line interface can be insecure.ERROR 2013 (HY000): Lost connection to MySQL server at 'reading initial communication packet', system error: 0报错，请您稍等一分钟，在终端一中重新执行转发端口命令，在终端二中重新执行连接PolarDB-X集群命令即可。

7.4.4. 第4步：启动业务

本节将介绍如何使用Sysbench OLTP场景模拟业务流量。

操作步骤

1. 准备压测数据。
   i. 执行如下SQL语句，创建压测数据库sysbench_test。

   ```
   CREATE DATABASE sysbench_test;
   ```

   ii. 输入**exit**退出数据库。
   iii. 执行如下命令，切换到账号galaxykube。

   ```
   su galaxykube
   ```

   iv. 执行如下命令，进入到*/home/galaxykube*目录。

   ```
   cd
   ```

   v. 执行如下命令，创建准备压测数据的sysbench-prepare.yaml文件。

   ```
   vim sysbench-prepare.yaml
   ```

vi. 按i键进入编辑模式，将如下代码复制到文件中，然后按ECS退出编辑模式，输入:wq后按下Enter键保存并退出。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sysbench-prepare-data-test
  namespace: default
spec:
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: sysbench-prepare
          image: severalnines/sysbench
          env:
            - name: POLARDB_X_USER
              value: polardbx_root
            - name: POLARDB_X_PASSWD
              valueFrom:
                secretKeyRef:
                  name: polardb-x
                  key: polardbx_root
          command: [ 'sysbench' ]
          args:
            - --db-driver=mysql
            - --mysql-host=$(POLARDB_X_SERVICE_HOST)
            - --mysql-port=$(POLARDB_X_SERVICE_PORT)
            - --mysql-user=$(POLARDB_X_USER)
            - --mysql_password=$(POLARDB_X_PASSWD)
            - --mysql-db=sysbench_test
            - --mysql-table-engine=innodb
            - --rand-init=on
            - --max-requests=1
            - --oltp-tables-count=1
            - --report-interval=5
            - --oltp-table-size=160000
            - --oltp_skip_trx=on
            - --oltp_auto_inc=off
            - --oltp_secondary
            - --oltp_range_size=5
            - --mysql_table_options=dbpartition by hash(`id`)
            - --num-threads=1
            - --time=3600
            - /usr/share/sysbench/tests/include/oltp_legacy/parallel_prepare.lua
            - run
```

vii. 执行如下命令，运行准备压测数据的sysbench-prepare.yaml文件，初始化测试数据。

```
kubectl apply -f sysbench-prepare.yaml
```

viii. 执行如下命令，获取任务进行状态。

```
kubectl get jobs
```

返回结果如下，请您耐心等待大约1分钟，当任务状态COMPLETIONS为1/1时，表示数据已经初始化完成。

```
[galaxykube@iZ...spfZ ~]$ kubectl get jobs
NAME                        COMPLETIONS   DURATION   AGE
sysbench-prepare-data-test  1/1           56s        74s
```

2. 启动压测流量。

i. 执行如下命令，创建启动压测的sysbench-oltp.yaml文件。

```
vim sysbench-oltp.yaml
```

ii. 按i键进入编辑模式，将如下代码复制到文件中，然后按ECS退出编辑模式，输入:wq后按下Enter键保存并退出。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sysbench-oltp-test
  namespace: default
spec:
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: sysbench-oltp
          image: severalnines/sysbench
          env:
            - name: POLARDB_X_USER
              value: polardbx_root
            - name: POLARDB_X_PASSWD
              valueFrom:
                secretKeyRef:
                  name: polardb-x
                  key: polardbx_root
          command: [ 'sysbench' ]
          args:
            - --db-driver=mysql
            - --mysql-host=$(POLARDB_X_SERVICE_HOST)
            - --mysql-port=$(POLARDB_X_SERVICE_PORT)
            - --mysql-user=$(POLARDB_X_USER)
            - --mysql_password=$(POLARDB_X_PASSWD)
            - --mysql-db=sysbench_test
            - --mysql-table-engine=innodb
            - --rand-init=on
            - --max-requests=0
            - --oltp-tables-count=1
            - --report-interval=5
            - --oltp-table-size=160000
            - --oltp_skip_trx=on
            - --oltp_auto_inc=off
            - --oltp_secondary
            - --oltp_range_size=5
            - --mysql-ignore-errors=all
            - --num-threads=8
            - --time=3600
            - /usr/share/sysbench/tests/include/oltp_legacy/oltp.lua
            - run
```

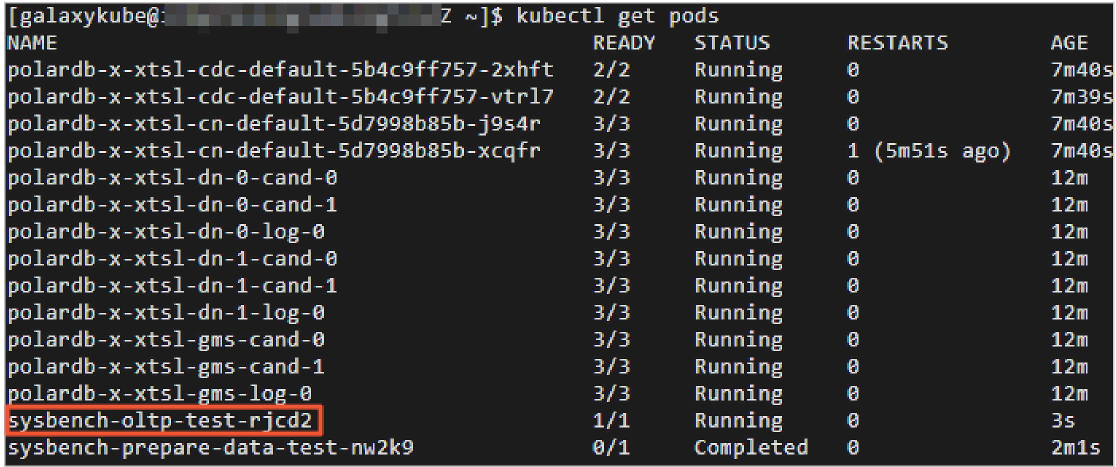
iii. 执行如下命令，运行启动压测的sysbench-oltp.yaml文件，开始压测。

```
kubectl apply -f sysbench-oltp.yaml
```

iv. 执行如下命令，查找压测脚本运行的POD。

```
kubectl get pods
```

返回结果如下， 以sysbench-oltp-test-开头的POD即为目标POD。



NAME	READY	STATUS	RESTARTS	AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-2xhft	2/2	Running	0	7m40s
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7	2/2	Running	0	7m39s
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r	3/3	Running	0	7m40s
polardb-x-xtsl-cn-default-5d7998b85b-xcqfr	3/3	Running	1 (5m51s ago)	7m40s
polardb-x-xtsl-dn-0-cand-0	3/3	Running	0	12m
polardb-x-xtsl-dn-0-cand-1	3/3	Running	0	12m
polardb-x-xtsl-dn-0-log-0	3/3	Running	0	12m
polardb-x-xtsl-dn-1-cand-0	3/3	Running	0	12m
polardb-x-xtsl-dn-1-cand-1	3/3	Running	0	12m
polardb-x-xtsl-dn-1-log-0	3/3	Running	0	12m
polardb-x-xtsl-gms-cand-0	3/3	Running	0	12m
polardb-x-xtsl-gms-cand-1	3/3	Running	0	12m
polardb-x-xtsl-gms-log-0	3/3	Running	0	12m
sysbench-oltp-test-rjcd2	1/1	Running	0	3s
sysbench-prepare-data-test-nw2k9	0/1	Completed	0	2m1s

v. 执行如下命令，查看QPS等流量数据。

```
kubectl logs -f 目标POD
```

说明

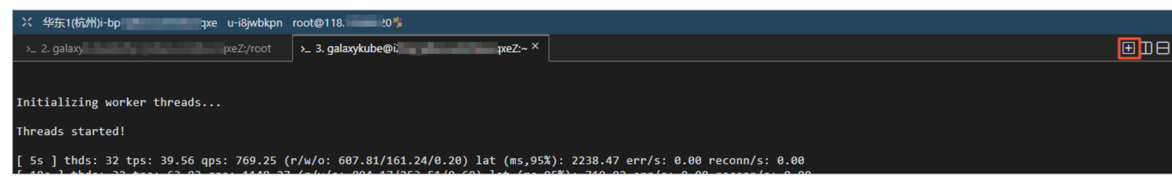
您需要将命令中的目标POD替换为以sysbench-oltp-test-开头的POD。

7.4.5. 第5步：体验PolarDB-X高可用能力

经过前面的准备工作，我们已经用PolarDB-X+Sysbench OLTP搭建了一个正在运行的业务系统。本介绍如何通过使用kill POD的方式，模拟物理机宕机、断网等导致的节点不可用场景，并观察业务QPS的变化情况。

操作步骤

1. 在实验页面，单击右上角的图标，创建新的终端三。



2. kill CN。

   i. 执行如下命令，切换到账号galaxykube。

```
su galaxykube
```

ii. 执行如下命令，获取CN POD的名字。

```
kubectl get pods
```

返回结果如下，以'polardb-x-xxxx-cn-default'开头的是CN POD的名字。

```
[galaxykube@iZ[illegible] root]$ kubectl get pods
NAME                                        READY   STATUS      RESTARTS      AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-2xhft 2/2     Running     0             8m50s
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7 2/2     Running     0             8m49s
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r  3/3     Running     0             8m50s
polardb-x-xtsl-cn-default-5d7998b85b-xcqfr  3/3     Running     1 (7m1s ago)  8m50s
polardb-x-xtsl-dn-0-cand-0                  3/3     Running     0             13m
polardb-x-xtsl-dn-0-cand-1                  3/3     Running     0             13m
polardb-x-xtsl-dn-0-log-0                   3/3     Running     0             13m
polardb-x-xtsl-dn-1-cand-0                  3/3     Running     0             13m
polardb-x-xtsl-dn-1-cand-1                  3/3     Running     0             13m
polardb-x-xtsl-dn-1-log-0                   3/3     Running     0             13m
polardb-x-xtsl-gms-cand-0                   3/3     Running     0             13m
polardb-x-xtsl-gms-cand-1                   3/3     Running     0             13m
polardb-x-xtsl-gms-log-0                    3/3     Running     0             13m
sysbench-oltp-test-rjcd2                    1/1     Running     0             73s
sysbench-prepare-data-test-nw2k9            0/1     Completed   0             3m11s
```

iii. 执行如下命令，删除任意一个CN POD。

```
kubectl delete pod <CN POD>
```

说明

您需要将命令中的<CN POD>替换为任意一个以'polardb-x-xxxx-cn-default'开头的CN POD的名字。

iv. 执行如下命令，查看CN POD自动创建情况。

```
kubectl get pods
```

返回结果如下，您可查看到CN POD已经处于自动创建中。

```
[galaxykube@iZbp1h6lnig7w6k15gf2xdZ root]$ kubectl get pods
NAME                                          READY   STATUS      RESTARTS   AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-2xhft   2/2     Running     0          11m
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7   2/2     Running     0          11m
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r    3/3     Running     0          11m
polardb-x-xtsl-cn-default-5d7998b85b-r5wdw    2/3     Running     0          8s
polardb-x-xtsl-dn-0-cand-0                    3/3     Running     0          15m
polardb-x-xtsl-dn-0-cand-1                    3/3     Running     0          15m
polardb-x-xtsl-dn-0-log-0                     3/3     Running     0          15m
polardb-x-xtsl-dn-1-cand-0                    3/3     Running     0          15m
polardb-x-xtsl-dn-1-cand-1                    3/3     Running     0          15m
polardb-x-xtsl-dn-1-log-0                     3/3     Running     0          15m
polardb-x-xtsl-gms-cand-0                     3/3     Running     0          15m
polardb-x-xtsl-gms-cand-1                     3/3     Running     0          15m
polardb-x-xtsl-gms-log-0                      3/3     Running     0          15m
sysbench-oltp-test-rjcd2                      1/1     Running     0          3m32s
sysbench-prepare-data-test-nw2k9              0/1     Completed   0          5m30s
```

经过几十秒后，被kill的CN POD自动恢复正常。

```
[galaxykube@                    Z root]$ kubectl get pods
NAME                                          READY   STATUS      RESTARTS   AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-2xhft   2/2     Running     0          11m
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7   2/2     Running     0          11m
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r    3/3     Running     0          11m
polardb-x-xtsl-cn-default-5d7998b85b-r5wdw    3/3     Running     0          48s
polardb-x-xtsl-dn-0-cand-0                    3/3     Running     0          16m
polardb-x-xtsl-dn-0-cand-1                    3/3     Running     0          16m
polardb-x-xtsl-dn-0-log-0                     3/3     Running     0          16m
polardb-x-xtsl-dn-1-cand-0                    3/3     Running     0          16m
polardb-x-xtsl-dn-1-cand-1                    3/3     Running     0          16m
polardb-x-xtsl-dn-1-log-0                     3/3     Running     0          16m
polardb-x-xtsl-gms-cand-0                     3/3     Running     0          16m
polardb-x-xtsl-gms-cand-1                     3/3     Running     0          16m
polardb-x-xtsl-gms-log-0                      3/3     Running     0          16m
sysbench-oltp-test-rjcd2                      1/1     Running     0          4m12s
sysbench-prepare-data-test-nw2k9              0/1     Completed   0          6m10s
```

v. 切换至终端二，您可查看kill CN之后业务QPS的情况。

```
[ 150s ] thds: 32 tps: 72.24 qps: 1287.29 (r/w/o: 1000.11/286.79/0.40) lat (ms,95%): 682.06 err/s: 0.00 reconn/s: 0.00
[ 155s ] thds: 32 tps: 67.42 qps: 1221.76 (r/w/o: 951.08/270.67/0.00) lat (ms,95%): 877.61 err/s: 0.00 reconn/s: 0.00
[ 160s ] thds: 32 tps: 86.17 qps: 1540.81 (r/w/o: 1197.74/343.07/0.00) lat (ms,95%): 530.08 err/s: 0.00 reconn/s: 0.00
[ 165s ] thds: 32 tps: 43.01 qps: 765.70 (r/w/o: 593.68/171.82/0.20) lat (ms,95%): 1678.14 err/s: 0.00 reconn/s: 0.00
[ 170s ] thds: 32 tps: 72.33 qps: 1304.28 (r/w/o: 1015.97/288.11/0.20) lat (ms,95%): 623.33 err/s: 0.00 reconn/s: 0.00
[ 175s ] thds: 32 tps: 58.03 qps: 1073.79 (r/w/o: 841.89/231.10/0.80) lat (ms,95%): 816.63 err/s: 0.00 reconn/s: 0.00
[ 180s ] thds: 32 tps: 10.20 qps: 170.00 (r/w/o: 123.20/46.60/0.20) lat (ms,95%): 3706.08 err/s: 0.00 reconn/s: 0.00
[ 185s ] thds: 32 tps: 7.60 qps: 142.54 (r/w/o: 116.35/26.19/0.00) lat (ms,95%): 6594.16 err/s: 0.00 reconn/s: 0.00
[ 190s ] thds: 32 tps: 5.42 qps: 101.46 (r/w/o: 76.23/25.22/0.00) lat (ms,95%): 6476.48 err/s: 0.00 reconn/s: 0.00
[ 195s ] thds: 32 tps: 6.24 qps: 120.32 (r/w/o: 95.57/24.75/0.00) lat (ms,95%): 6026.41 err/s: 0.00 reconn/s: 0.00
[ 200s ] thds: 32 tps: 3.37 qps: 78.57 (r/w/o: 66.86/11.71/0.00) lat (ms,95%): 6960.17 err/s: 0.00 reconn/s: 3.17
[ 205s ] thds: 32 tps: 41.93 qps: 750.56 (r/w/o: 584.24/166.32/0.00) lat (ms,95%): 8333.38 err/s: 0.00 reconn/s: 0.00
[ 210s ] thds: 32 tps: 53.83 qps: 978.75 (r/w/o: 759.43/219.12/0.20) lat (ms,95%): 1013.60 err/s: 0.00 reconn/s: 0.00
[ 215s ] thds: 32 tps: 47.93 qps: 864.98 (r/w/o: 674.05/190.32/0.60) lat (ms,95%): 1149.76 err/s: 0.00 reconn/s: 0.00
[ 220s ] thds: 32 tps: 52.73 qps: 938.61 (r/w/o: 729.88/208.74/0.00) lat (ms,95%): 1032.01 err/s: 0.00 reconn/s: 0.00
[ 225s ] thds: 32 tps: 48.67 qps: 867.63 (r/w/o: 673.35/194.08/0.20) lat (ms,95%): 977.74 err/s: 0.00 reconn/s: 0.00
[ 230s ] thds: 32 tps: 49.79 qps: 900.44 (r/w/o: 701.48/198.36/0.60) lat (ms,95%): 1089.30 err/s: 0.00 reconn/s: 0.00
[ 235s ] thds: 32 tps: 64.39 qps: 1165.34 (r/w/o: 910.19/254.94/0.20) lat (ms,95%): 802.05 err/s: 0.00 reconn/s: 0.00
[ 240s ] thds: 32 tps: 62.77 qps: 1125.89 (r/w/o: 873.60/251.49/0.80) lat (ms,95%): 1032.01 err/s: 0.00 reconn/s: 0.00
[ 245s ] thds: 32 tps: 82.84 qps: 1498.37 (r/w/o: 1163.40/334.37/0.60) lat (ms,95%): 530.08 err/s: 0.00 reconn/s: 0.00
[ 250s ] thds: 32 tps: 77.83 qps: 1392.28 (r/w/o: 1081.58/310.51/0.20) lat (ms,95%): 759.88 err/s: 0.00 reconn/s: 0.00
```

3. kill DN。

i. 切换至终端三，执行如下命令，获取DN POD的名字。

```
kubectl get pods
```

返回结果如下，以'polardb-x-xxxx-dn'开头的是DN POD的名字。

```
[galaxykube@iZb[illegible]2xdZ root]$ kubectl get pods
NAME                                          READY   STATUS      RESTARTS   AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-2xhft   2/2     Running     0          13m
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7   2/2     Running     0          13m
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r    3/3     Running     0          13m
polardb-x-xtsl-cn-default-5d7998b85b-r5wdw    3/3     Running     0          2m4s
polardb-x-xtsl-dn-0-cand-0                    3/3     Running     0          17m
polardb-x-xtsl-dn-0-cand-1                    3/3     Running     0          17m
polardb-x-xtsl-dn-0-log-0                     3/3     Running     0          17m
polardb-x-xtsl-dn-1-cand-0                    3/3     Running     0          17m
polardb-x-xtsl-dn-1-cand-1                    3/3     Running     0          17m
polardb-x-xtsl-dn-1-log-0                     3/3     Running     0          17m
polardb-x-xtsl-gms-cand-0                     3/3     Running     0          17m
polardb-x-xtsl-gms-cand-1                     3/3     Running     0          17m
polardb-x-xtsl-gms-log-0                      3/3     Running     0          17m
sysbench-oltp-test-rjcd2                      1/1     Running     0          5m28s
sysbench-prepare-data-test-nw2k9              0/1     Completed   0          7m26s
```

ii. 执行如下命令，删除任意一个DN POD。

```
kubectl delete pod <DN POD>
```

说明

- 您需要将命令中的<DN POD>替换为任意一个以'polardb-x-xxxx-dn'开头的DN POD的名字。
- DN每个逻辑节点为三副本架构，也就是说一个DN节点对应3个POD，可任意选择一个进行删除操作。此外，GMS节点是一个特殊角色的DN，同样具备高可用能力，可选择任一POD进行删除。

iii. 执行如下命令，查看DN POD自动创建情况。

```
kubectl get pods
```

返回结果如下，您可查看到DN POD已经处于自动创建中。

```
[galaxykube@iZbp1h6lnig7w6k15gf2xdZ root]$ kubectl get pods
NAME                                          READY   STATUS              RESTARTS   AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-2xhft   2/2     Running             0          13m
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7   2/2     Running             0          13m
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r    3/3     Running             0          13m
polardb-x-xtsl-cn-default-5d7998b85b-r5wdw    3/3     Running             0          2m39s
polardb-x-xtsl-dn-0-cand-0                    0/3     ContainerCreating   0          0s
polardb-x-xtsl-dn-0-cand-1                    3/3     Running             0          18m
polardb-x-xtsl-dn-0-log-0                     3/3     Running             0          18m
polardb-x-xtsl-dn-1-cand-0                    3/3     Running             0          18m
polardb-x-xtsl-dn-1-cand-1                    3/3     Running             0          18m
polardb-x-xtsl-dn-1-log-0                     3/3     Running             0          18m
polardb-x-xtsl-gms-cand-0                     3/3     Running             0          18m
polardb-x-xtsl-gms-cand-1                     3/3     Running             0          18m
polardb-x-xtsl-gms-log-0                      3/3     Running             0          18m
sysbench-oltp-test-rjcd2                      1/1     Running             0          6m3s
sysbench-prepare-data-test-nw2k9              0/1     Completed           0          8m1s
```

经过几十秒后，被kill的DN POD自动恢复正常。

```
[galaxykub                        dZ root]$ kubectl get pods
NAME                                          READY   STATUS      RESTARTS   AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-2xhft   2/2     Running     0          14m
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7   2/2     Running     0          14m
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r    3/3     Running     0          14m
polardb-x-xtsl-cn-default-5d7998b85b-r5wdw    3/3     Running     0          3m34s
polardb-x-xtsl-dn-0-cand-0                    3/3     Running     0          55s
polardb-x-xtsl-dn-0-cand-1                    3/3     Running     0          19m
polardb-x-xtsl-dn-0-log-0                     3/3     Running     0          19m
polardb-x-xtsl-dn-1-cand-0                    3/3     Running     0          19m
polardb-x-xtsl-dn-1-cand-1                    3/3     Running     0          19m
polardb-x-xtsl-dn-1-log-0                     3/3     Running     0          19m
polardb-x-xtsl-gms-cand-0                     3/3     Running     0          19m
polardb-x-xtsl-gms-cand-1                     3/3     Running     0          19m
polardb-x-xtsl-gms-log-0                      3/3     Running     0          19m
sysbench-oltp-test-rjcd2                      1/1     Running     0          6m58s
sysbench-prepare-data-test-nw2k9              0/1     Completed   0          8m56s
```

iv. 切换至终端二，您可查看kill DN之后业务QPS的情况。

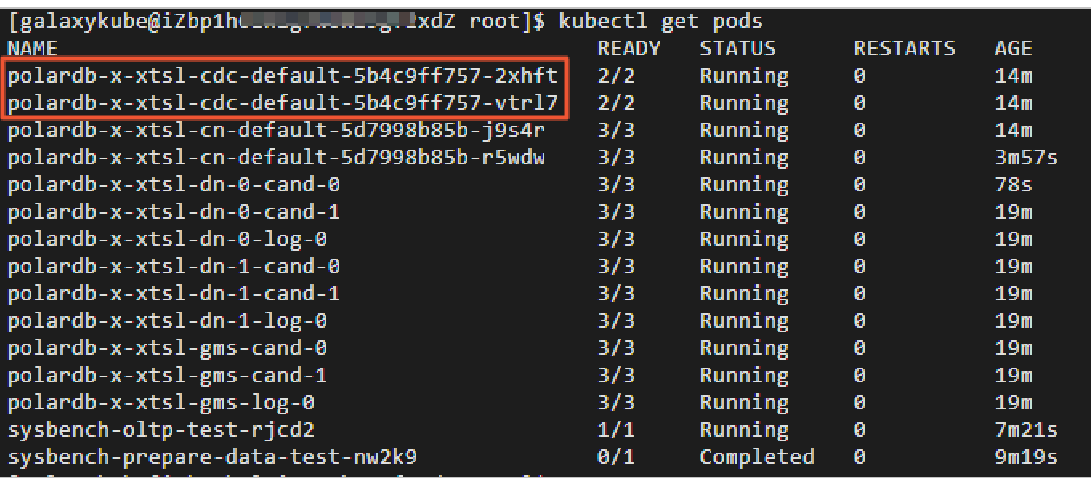
```
[ 335s ] thds: 32 tps: 69.07 qps: 1239.47 (r/w/o: 963.38/275.88/0.20) lat (ms,95%): 746.32 err/s: 0.00 reconn/s: 0.00
[ 340s ] thds: 32 tps: 85.79 qps: 1559.00 (r/w/o: 1214.24/344.36/0.40) lat (ms,95%): 484.44 err/s: 0.00 reconn/s: 0.00
[ 345s ] thds: 32 tps: 73.22 qps: 1306.68 (r/w/o: 1017.22/289.26/0.20) lat (ms,95%): 646.19 err/s: 0.00 reconn/s: 0.00
[ 350s ] thds: 32 tps: 81.00 qps: 1477.85 (r/w/o: 1149.44/328.01/0.40) lat (ms,95%): 559.50 err/s: 0.00 reconn/s: 0.00
[ 355s ] thds: 32 tps: 72.96 qps: 1277.35 (r/w/o: 990.10/287.05/0.20) lat (ms,95%): 623.33 err/s: 0.00 reconn/s: 0.00
[ 360s ] thds: 32 tps: 61.90 qps: 1130.20 (r/w/o: 881.80/247.81/0.60) lat (ms,95%): 909.80 err/s: 0.00 reconn/s: 0.00
[ 365s ] thds: 32 tps: 53.27 qps: 965.43 (r/w/o: 752.14/213.09/0.20) lat (ms,95%): 831.46 err/s: 0.00 reconn/s: 0.00
[ 370s ] thds: 32 tps: 70.00 qps: 1252.46 (r/w/o: 971.86/280.19/0.40) lat (ms,95%): 802.05 err/s: 0.00 reconn/s: 0.00
[ 375s ] thds: 32 tps: 56.25 qps: 1048.89 (r/w/o: 817.09/231.40/0.40) lat (ms,95%): 759.88 err/s: 0.00 reconn/s: 0.00
[ 380s ] thds: 32 tps: 75.99 qps: 1347.60 (r/w/o: 1047.65/299.16/0.80) lat (ms,95%): 694.45 err/s: 0.00 reconn/s: 0.00
[ 385s ] thds: 32 tps: 72.61 qps: 1320.60 (r/w/o: 1034.76/285.64/0.20) lat (ms,95%): 569.67 err/s: 0.00 reconn/s: 0.00
```

4. kill CDC。

i. 切换至终端三，执行如下命令，获取CDC POD的名字。

```
kubectl get pods
```

返回结果如下，以‘polardb-x-xxxx-cdc-def aul’开头的是CDC POD的名字。



ii. 执行如下命令，删除任意一个CDC POD。

```
kubectl delete pod <CDC POD>
```

说明

您需要将命令中的<CDC POD>替换为任意一个以‘polardb-x-xxxx-cdc-def aul’开头的CDC POD的名字。

iii. 执行如下命令，查看CDC POD自动创建情况。

```
kubectl get pods
```

返回结果如下，您可查看到CDC POD已经处于自动创建中。

```
[galaxyk                          kdZ root]$ kubectl get pods
NAME                                          READY   STATUS              RESTARTS   AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-jhrhv   0/2     ContainerCreating   0          14s
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7   2/2     Running             0          17m
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r    3/3     Running             0          17m
polardb-x-xtsl-cn-default-5d7998b85b-r5wdw    3/3     Running             0          6m29s
polardb-x-xtsl-dn-0-cand-0                    3/3     Running             0          3m50s
polardb-x-xtsl-dn-0-cand-1                    3/3     Running             0          22m
polardb-x-xtsl-dn-0-log-0                     3/3     Running             0          22m
polardb-x-xtsl-dn-1-cand-0                    3/3     Running             0          22m
polardb-x-xtsl-dn-1-cand-1                    3/3     Running             0          22m
polardb-x-xtsl-dn-1-log-0                     3/3     Running             0          22m
polardb-x-xtsl-gms-cand-0                     3/3     Running             0          22m
polardb-x-xtsl-gms-cand-1                     3/3     Running             0          22m
polardb-x-xtsl-gms-log-0                      3/3     Running             0          22m
sysbench-oltp-test-rjcd2                      1/1     Running             0          9m53s
sysbench-prepare-data-test-nw2k9              0/1     Completed           0          11m
```

经过几十秒后，被kill的CDC POD自动恢复正常。

```
[galaxykube@                          dZ root]$ kubectl get pods
NAME                                          READY   STATUS      RESTARTS   AGE
polardb-x-xtsl-cdc-default-5b4c9ff757-jhrhv   2/2     Running     0          41s
polardb-x-xtsl-cdc-default-5b4c9ff757-vtrl7   2/2     Running     0          17m
polardb-x-xtsl-cn-default-5d7998b85b-j9s4r    3/3     Running     0          17m
polardb-x-xtsl-cn-default-5d7998b85b-r5wdw    3/3     Running     0          6m56s
polardb-x-xtsl-dn-0-cand-0                    3/3     Running     0          4m17s
polardb-x-xtsl-dn-0-cand-1                    3/3     Running     0          22m
polardb-x-xtsl-dn-0-log-0                     3/3     Running     0          22m
polardb-x-xtsl-dn-1-cand-0                    3/3     Running     0          22m
polardb-x-xtsl-dn-1-cand-1                    3/3     Running     0          22m
polardb-x-xtsl-dn-1-log-0                     3/3     Running     0          22m
polardb-x-xtsl-gms-cand-0                     3/3     Running     0          22m
polardb-x-xtsl-gms-cand-1                     3/3     Running     0          22m
polardb-x-xtsl-gms-log-0                      3/3     Running     0          22m
sysbench-oltp-test-rjcd2                      1/1     Running     0          10m
sysbench-prepare-data-test-nw2k9              0/1     Completed   0          12m
```

iv. 切换至终端二，您可查看kill CDC之后业务QPS的情况。

```
[ 535s ] thds: 32 tps: 42.71 qps: 737.09 (r/w/o: 569.43/167.26/0.40) lat (ms,95%): 2585.31 err/s: 0.00 reconn/s: 0.00
[ 540s ] thds: 32 tps: 33.54 qps: 624.96 (r/w/o: 488.98/135.77/0.20) lat (ms,95%): 1506.29 err/s: 0.00 reconn/s: 0.00
[ 545s ] thds: 32 tps: 42.77 qps: 721.57 (r/w/o: 555.27/166.10/0.20) lat (ms,95%): 1170.65 err/s: 0.00 reconn/s: 0.00
[ 550s ] thds: 32 tps: 46.55 qps: 854.53 (r/w/o: 667.15/187.19/0.19) lat (ms,95%): 877.61 err/s: 0.00 reconn/s: 0.00
[ 555s ] thds: 32 tps: 46.08 qps: 853.70 (r/w/o: 666.68/186.61/0.41) lat (ms,95%): 1771.29 err/s: 0.00 reconn/s: 0.00
[ 560s ] thds: 32 tps: 28.01 qps: 522.51 (r/w/o: 405.89/116.63/0.00) lat (ms,95%): 1213.57 err/s: 0.00 reconn/s: 0.00
[ 565s ] thds: 32 tps: 6.00 qps: 108.04 (r/w/o: 84.83/23.21/0.00) lat (ms,95%): 5918.87 err/s: 0.00 reconn/s: 0.00
[ 570s ] thds: 32 tps: 7.60 qps: 139.74 (r/w/o: 109.36/30.19/0.20) lat (ms,95%): 5813.24 err/s: 0.00 reconn/s: 0.00
[ 575s ] thds: 32 tps: 7.40 qps: 119.25 (r/w/o: 87.64/31.61/0.00) lat (ms,95%): 5124.81 err/s: 0.00 reconn/s: 0.00
[ 580s ] thds: 32 tps: 8.27 qps: 129.38 (r/w/o: 99.05/30.33/0.00) lat (ms,95%): 5217.92 err/s: 0.00 reconn/s: 0.00
[ 585s ] thds: 32 tps: 10.77 qps: 228.31 (r/w/o: 184.44/43.67/0.20) lat (ms,95%): 5217.92 err/s: 0.00 reconn/s: 0.00
[ 590s ] thds: 32 tps: 28.33 qps: 497.54 (r/w/o: 384.43/112.91/0.20) lat (ms,95%): 2159.29 err/s: 0.00 reconn/s: 0.00
[ 595s ] thds: 32 tps: 58.02 qps: 1015.98 (r/w/o: 788.09/227.68/0.20) lat (ms,95%): 1191.92 err/s: 0.00 reconn/s: 0.00
[ 600s ] thds: 32 tps: 51.03 qps: 925.15 (r/w/o: 721.61/203.34/0.20) lat (ms,95%): 846.57 err/s: 0.00 reconn/s: 0.00
[ 605s ] thds: 32 tps: 52.87 qps: 941.20 (r/w/o: 729.72/211.49/0.00) lat (ms,95%): 943.16 err/s: 0.00 reconn/s: 0.00
[ 610s ] thds: 32 tps: 39.13 qps: 717.27 (r/w/o: 559.35/157.92/0.00) lat (ms,95%): 1213.57 err/s: 0.00 reconn/s: 0.00
[ 615s ] thds: 32 tps: 26.91 qps: 482.39 (r/w/o: 376.16/106.23/0.00) lat (ms,95%): 2082.91 err/s: 0.00 reconn/s: 0.00
[ 620s ] thds: 32 tps: 30.83 qps: 559.90 (r/w/o: 436.59/123.32/0.00) lat (ms,95%): 1869.60 err/s: 0.00 reconn/s: 0.00
[ 625s ] thds: 32 tps: 32.53 qps: 590.88 (r/w/o: 457.79/132.89/0.20) lat (ms,95%): 1479.41 err/s: 0.00 reconn/s: 0.00
```

8.常见问题解答

1. Q：使用PXD工具一键安装PolarDB-X时，还需要安装MySQL吗？

   A：首先，我们在安装部署和使用PolarDB-X的过程中使用的MySQL，仅仅是MySQL的Client，换言之，MySQL的Server部分是不需要安装的。安装MySQL Client也仅仅是为了使用它连接到我们所创建的PolarDB-X从而进行常规操作。

   也就是说，PolarDB-X的部署和使用是不依赖于MySQL的，安装MySQL Client仅仅是为了连接到PolarDB-X数据库，从而进行一些常规的SQL操作。

   使用PXD工具一键安装PolarDB-X时，需要安装MySQL Client。

2. Q：使用Docker镜像安装部署PolarDB-X，没有CN节点吗？

   A：有的。Docker镜像部署方式，是一种一体化的部署方式。容器启动运行之后，容器中有一个CN、一个DN和一个CND，DN同时会扮演GMS的角色。亦即，PolarDB-X的节点都在同一个容器中。

3. Q：PolarDB-X中，只有CN节点是无状态的吗？

   A：在PolarDB-X的四个组件中，CN和CDC都是无状态的，GMS和DN是有状态的。

4. Q：一条SQL会过两遍优化器吗？计算层过一遍，PolarDB-X过一遍？

   A：并不一定，目前CN和DN之间的交互通过定制的私有协议，用户的SQL优化会在CN层先完成，然后CN层会将生成的执行计划Plan通过私有协议直接发送给DN，在DN层面可以绕过对应的SQL解析和优化，从而可以直接执行。目前针对单表查询可以具备下推Plan，多表join和子查询目前还需要通过SQL下推到DN层，会经过DN层的解析和优化。

5. Q：阿里云PolarDB-X商业版本和开源版本是一致的吗？

   A：PolarDB-X开源版本和商业版本，在内核能力上是完全一致的，甚至于有些能力会优先在开源版本里提供，之后才会在商业版本中体现。在部署、运维和监控等能力方面，商业版本是基于阿里云的底座进行开发，而开源版本则是基于开源生态（比如基于Kubernetes生态安装部署、基于Prometheus+Grafana的监控系统）。

6. Q：生产环境下，一般采用什么模式部署PolarDB-X？

   A：推荐采用Kubernetes的方式来部署生产环境。

7. Q：不分区的话，数据是怎么分发到8个分片上的？

   A：所谓的不分区，不是真的不分区，而是自动分区。PolarDB-X会自动选择主键列来对数据进行分区。

8. Q：HTAP如何物理隔离OLTP和OLAP？

   A：以一个包含4个CN和4个DN的系统为例。假设现在有HTAP诉求，并希望OLTP和OLAP能够强隔离。我们可以将其中的两个CN作为OLTP的流量入口，两个CN作为OLAP的流量入口。下层的DN是三副本的结构，包含一个主节点、两个follower节点和一个learner节点。HTAP时，我们可以将follower节点或者将learner节点作为处理OLAP流量的节点。由此，就可以实现OLTP和OLAP在物理资源上的隔离。

9. Q：CN、DN、GMS的副本个数，最低是三个吗？

   A：CN是没有副本的概念的，因为它本身是个无状态的节点，最少可以只有一个，但如果想让系统具备高可用的能力，那么有两个CN会比较合适。一个DN本身就是三副本的结构（一个GMS也是个三副本的结构），那么一个DN它最少的副本就是三个。

10. Q：PolarDB-X是目前PolarDB的升级吗？

    A：PolarDB-X适合于有非常高的读写扩展需求以及有非常大的存储扩展需求的场景，包括比如有一个特别大的表需要进行拆分等这一类场景。

11. Q：Navicat之类的MySQL客户端，也可以连接PolarDB-X吗？

    A：是的，可以连接。

12. Q：扩充DN后，分表的片会被移动吗？

    A：会的。您可以通过体验扩容实验来感受该能力。在扩容实验中，通过动态添加一个DN，会将已有的分片数据移动到新加入的DN中。

13. Q：galaxyglue是用来做什么的？

    A：CN和DN一开始就在分库分表的形态下，CN可以看做是分库分表的中间件，DN可以认为就是个MySQL，CN和DN之间的交互是通过标准的JDBC来进行通信的。如果后续想提供更强的能力（比如想要具备分布式事务的能力），通过JDBC是没有办法实现的。因此，我们将中间的JDBC去掉，实现了一个从CN到DN的私有通信协议（私有RPC），这就是galaxyglue库所做的工作。

14. Q：PolarDB-X可以和MySQL搭主从吗？

    A：可以的。目前支持搭建一个以MySQL为主，PolarDB-X为备，或者PolarDB-X为主，MySQL为备的系统，这样PolarDB-X和MySQL就可以互为主备关系。

15. Q：可以直接查询自动生成的分区表吗？

    A：可以。

16. Q：MySQL的存储过程和触发器支持吗？

    A：下一个发布的版本会对MySQL存储过程进行支持。触发器的支持情况和计划暂不确定。

17. Q：DN伸缩后，会影响数据分片吗？

    A：不会影响数据分片，只是将整个分片进行移动。

18. Q：多个CN是并行写，还是读写分离？

    A：可以并行写，这也是PolarDB-X最大的特点之一，就是所有的CN节点都是无状态的、对等的。也就是说，可以将CN当成一个读写节点来使用，从而达到对系统进行读写扩展的目的。

19. Q：支持虚拟机部署吗？

    A：目前二进制的部署方式还在开发中，开发完成后就能够支持虚拟机部署了。

20. Q：CN是完全无状态的吗？新加入的CN节点是否需要从其他CN节点同步元数据之类的信息？

    A：是的，CN是完全无状态的，新加入的CN节点不需要同步信息。

21. Q：GMS相当于TiDB的PD功能吗？元数据在GMS里吗？

    A：是的，可以这样理解。

22. Q：Kubernetes上的DN是以pod的方式运行吗？

    A：是的。

23. Q：单DN上的数据持久是怎么实现的？

    A：DN就是一个MySQL的实例，目前数据是存在本地盘的。后续我们将开发基于ESSD等一些云盘的能力。

24. Q：DN的副本是块级别的，还是记录级别的？

    A：DN的三副本，可以理解为三个MySQL进程，通过Paxos协议达数据一致性。

25. Q：DN可以在线扩缩容吗？数据在迁移过程中，可以对外提供服务吗？

    A：都是可以的。

26. Q：PolarDB-X如此高可用，是不是不需要备份呢？

A：备份会有多种用途，除了防止数据丢失，还可以用于将数据回滚到历史版本，这是备份就会发挥作用（因为库中存的永远是当前的快照）。备份还可以用于跨地域级别的容灾等特殊需求。

27. Q：PolarDB-X支持读写分离吗？

    A：PolarDB-X的1.0和2.0都是支持读写分离的。

28. Q：AUTO模式下，各分区数据满了，会自动增加分区吗?还是手动？

    A：我们会提供运维手段，可以看到DN的容量。目前没有帮用户自动做分区，还是需要手动的。

29. Q：读写分离需要第三方插件吗？

    A：读写分离不需要第三方插件，阿里云官方会提供，只需要在控制台上做一个简单的操作就可以，而且还可以调整读写比例。

30. Q：有没有查看数据分布或者占比的命令？

    A：请参见分区热力图。

31. Q：写入场景中的热点是如何优化的呢？

    A：请参见PolarDB-X 热点优化系列 (一)：如何支持淘宝库存热点更新和PolarDB-X 热点优化系列（二）：如何支持淘宝大卖家分区热点。

32. Q：AUTO模式并发写入TPS能到达多少？这个性能和CN节点数成正比吗？

    A：理论上性能和CN是成正比的，CN越多，能够处理的请求数越大。

33. Q：TABLEGROUP是什么概念？

    A：请参见PolarDB-X 数据分布解读（一）。

34. Q：支持创建的数据库个数和表的个数有限制吗？

    A：有，请参见开发限制。

35. Q：全局Binlog，数据整形过程中，会导致变更列的数据丢失吗？

    A：不会导致数据丢失。因为整形是以逻辑Schema为基准来进行整形的，整形的这些数据都是在DDL变更过程中产生的，都是中间数据，是没有用的。最终打标完成之后才会返回给用户，告知DDL成功，这时用户插入的才是正常的数据。在整形的过程中，逻辑表还没有生效，用户插入的字段会报错，所以整形的数据不会导致变更，由此保证数据一致性。

36. Q：全局Binlog备份了以后，怎样连CN导入恢复到库里呢？

    A：全局Binlog兼容了开源mysql binlog的协议，可以使用原生MySQL的mysqlbinlog命令做解析，或者借助其他开源工具来把binlog转成SQL，最后的SQL通过CN进行恢复。

37. Q：Scale Up的时候，DN不是follower切换到leader吗？那为什么sysbench会QPS为0？

    A：Sysbench演示的模型是单CN+单DN，在DN做paxos leader切换时，会有RTO<30秒左右的DML不可写，但可读的状态，导致Sysbench OLTP压测时一段时间处于全失败。

38. Q：如果CN和DN都是多副本的，QPS就不会为0是吗？

    A：是的，如果CN和DN都是多节点模式，就不会出现QPS跌0的情况。

39. Q：公共云也是用Kubernetes进行管控的吗？

    A：是的，只不过在公共云是不是采用Operator的方式，采用的是云上的管控架构。

40. Q：在数据量比较大的情况下，进行DN节点的扩容，失败了会怎样 ？

A：可以从扩容的几个步骤来分析。首先，在新增DN节点时如果失败，这种情况不影响实例的使用，因为数据和分区都没有迁移。如果在迁移过程中失败，也没有问题，因为流量没有切换，而且分区计算和数据迁移也是异步的过程，同时也涉及自适应流控，不会对用户造成影响。如果真出现问题，PolarDB-X也提供了异步DDL的运维手段，可以帮助排查并解决问题。

41. Q：DN一副本只能是一主一从吗？多副本是多主模式？

    A：DN目前有两种模式，一种是一副本，也就是一个leader；另一种是我们推荐的三副本，即一个leader、一个follower、一个logger，其中leader和follower有全量数据，logger只存储Binlog日志而不会apply binlog。

42. Q：生产环境扩容CN的话，有什么办法避免失败吗？

    A：如果是整体资源不足导致如Pod处于pending状态等情况，此时我们通常会先保证新的Pod完全起来并能够完全可对外提供服务，再将老的Pod下线，从而降低对整体资源的影响（可以理解为Rolling Upgrade操作，与Kubernetes中Deployment的升级类似）。

43. Q：监控能接Zabbix，接收企业微信、发送短信吗？

    A：可以。基于Prometheus+Grafana的监控方案中，还包含一个叫Alert Manager的组件，可以通过该组件对接Prometheus的数据，再对接不同的消息源来推送报警和通知消息。

44. Q：explain advisor为什么IMPROVE_CPU、MEM、IO会是负的？

    A：MEM和IO是负的，是这部分代价会增加。但是由于减少了分片扫描，网络代价会减少。最终判断整体效率还是有所提升的，所以推荐出了该索引。可以理解为，当对一条SQL创建了一个索引，它可能网络性能有所提升，但CPU和内存代价会有所增加，这样也是合理的，综合会有一个基于优化器的代价评估。

45. Q：全局索引和局部索引可以分开建吗？

    A：可以。

46. Q：一个DN上Leader、Follower保证副本，但磁盘坏了，数据不是一样会丢失吗？做多副本的意义是什么？

    A：保证高可用，实际上保证的是一个概率问题。我们认为一个DN上得三副本，会部署在不同的机器上。如果这三台机器同时故障了，确实也没有办法保证数据不丢失。如果只是其中一台故障，我们仍有办法保证业务可用。如果其中两台故障，我们可以保证数据不丢失。

47. Q：Exporter能接入自建的Prometheus和Grafana吗？

    A：可以。创建PolarDB-X实例的时候，默认是不安装PolarDB-X Monitor的，也就是默认没有监控的。但是Exporter是已经启动了的。之哟啊Prometheus能够访问到Exporter的服务，就可以获取监控指标，构建自定义的监控报表。

48. Q：一个DN如何对应多个主机节点？

    A：一个DN会有三个节点分散在三台主机上，分别是Leader、Follower和Logger的角色。对于它们而言是一个Paxos组。最终是Leader对外提供服务，Follower和Logger保证数据的强一致和可靠。如果Leader故障，Follower会自动升为Leader并对外提供相应的响应和服务。

49. Q：两阶段提交中，一阶段的日志是记录在哪里的？

    A：每一个分片上都保留了一张事务的日志表。一阶段的日志就会根据第一个写入的分片，将日志落在该分片上。如果出现CN宕机恢复的话，就会去扫描所有分片上的数据日志，看有哪些事务实在上一次宕机的时候是处于一阶段结束但二阶段还没有开始的情况，然后进一步推进该事务的提交和回滚。

50. Q：一个分片是一台ECS吗？

    A：不一定。因为分片本身是一个相对独立的概念，可以多个分片落在同一台DN上，也可以一个分片落在一个DN上，这个没有强制绑定。

51. Q：一个表可以有多个聚簇索引吗？

    A：是的，PolarDB-X的设计支持有多个聚簇索引。

52. Q：PolarDB-X底层分布式存储，有可用区的概念吗？

    A：有的。DN本身是基于X-Paxos实现的三节点架构，其中一些节点可以在不同的可用区中。

53. Q：一个表可以有多个聚簇索引吗？

    A：可以的。这个可能与我们见到的大多数数据库不太一样，比如MySQL索引组织表只能有一个聚簇索引，Oracle和SQL Server除了索引组织表外可以允许再单独创建一个全局聚簇索引。但PolarDB-X作为分布式数据库，回表的代价是非常高的。

    而且除了聚簇索引外，还支持一种覆盖索引的方式，就是在索引里面指定一些冗余的列。这两种方式的区别在于，对于一些相对模糊的查询（比如select *），仅仅采用覆盖索引这种方式，可能由于加列，导致模糊查询性能有波动。如果想要彻底解决该问题，就可以建聚簇索引。聚簇索引的特点是会保持表结构与主表完全一致，这样无论是加列还是减列，查询性能都不会受到影响。

54. Q：可以重新指定分区列吗？

    A：可以。现在的Online DDL支持变更分区键的，其底层原理对上层完全屏蔽，而且不会影响过程当中的读写请求。

55. Q：不同服务间的多个PolarDB-X，怎么实现事务？

    A：跨服务的事务超出了数据库应该支持的范畴，通常是由一些跨服务的事务组件来支持的。

56. Q：全局索引保存在DN上，还是保存在CN上？

    A：保存在DN上。全局索引本身也是一张逻辑表，只是与主表采用了不同的分区键，并且它会跟随主表的数据更新而同步更新，但它本身还是保存在DN上的。

57. Q：重新指定分区列的过程是动态的吗？数据量大会锁吗？

    A：这个完全采取的是一个Online Schema Change的过程。重新重新指定分区键，可以将其过程理解为先建了一个不同拆分键的聚簇索引，然后再把主表的元数据和这个聚簇索引交换一下，再把主表删掉。所以整个过程是完全的Online Schema Change不加任何锁。

58. Q：索引的数量有限制吗？

    A：目前我们只有一些建议值的限制。在代码层面的话是没有限制的。但肯定索引是有代价的。第一，它会有写放大（比如聚簇索引的空间要乘以2，因为它冗余了主表上的所有列），第二是索引本身有写入的代价，每多一个索引，在写入的过程当中就要去维护多一个索引的写入。当然随着索引数量的增多，因为采取了并行的写，只要系统资源不成为瓶颈，那么可能这个RT增加不会太多。但依然建议**合理使用索引**。所以这和MySQL上索引加太多的效果是一样的。

59. Q：索引表会随着主表变化吗？

    A：这个是聚簇索引特有的功能。比如在加列的时候，聚簇索引会跟随着主表一起加列。在减列的时候，聚簇索引上也会减列。当然普通索引上也会加减，正常的普通的全局二级索引，包括全局唯一二级索引，都不会跟随着主表加列而加列，但是会跟随着主表的变更列而变更列、减列而减列。

60. Q：索引创建默认是global的还是local的？

    A：首先PolarDB-X 2.0上，库分为了两种类型：AUTO模式的库和DRDS模式的库。DRDS模式库是兼容了以前PolarDB-X 1.0的使用方式，它默认创建的是local索引，默认需要自己指定分区键，如果不指定的话，就是单表。

我们现在更推荐的是第二种方法，就是建立一个AUTO模式库。AUTO模式库的设计目标是帮助大家像使用单机数据库一样去使用PolarDB-X，所以它基本上允许用和单机数据库一样的语法去得到一个可用的系统。所以它在建表的时候不要求指定拆分键，默认会去使用主键作为分区键，而创建的索引默认会是global index（全局索引）。当然它有一些针对性的细节的优化。但大多数场景下，在AUTO模式下，创建的都是全局索引。

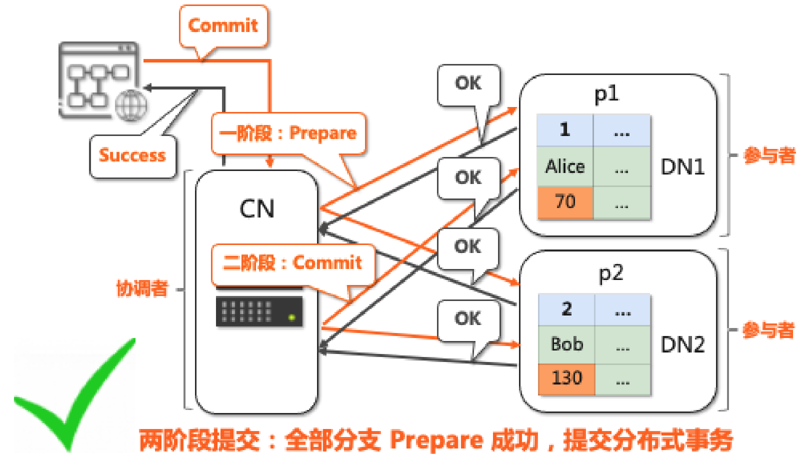
61. Q：某个DN挂了会丢数据吗？

A：所有DN本身都是一个MySQL，只要数据正常提交到DN，DN的crash不会导致数据丢失。并且PolarDB-X还基于三副本，所以有更强的数据持久性保障。

可能在一些具体的过程中，需要考虑一下数据会不会丢失。以下图为例，比如在两阶段提交的过程当中，左边描述的是一个两阶段提交全部成功的情况。首先是在一阶段提交的时候，CN向每个DN都发送了一个prepare，在prepare的过程中，所有的DN会尝试着将自己的这个redo log去进行落盘来确定其确实是可以提交的，并且记录一下这个事务状态已经是prepare 的状态了。只要这些状态都写到了日志里，那么DN的crash就不会导致状态的丢失。

如果CN上这一条commit log写入成功的话（它其实也是写在DN上的），只要这条日志成功，那么DN的crash也不会导致这条日志消失。所以当系统在各种crash拉起之后的话，依然可以看到这样一个分布式事务，它已经完成了所有DN上的prepare，并且这个commit 日志确认可以继续进行提交操作。 接下来CN就可以接着去把所有的commit 请求下发到每个DN上，DN就可以继续根据日志里面的内容，把该提交的内容全部提交掉。

所以某个DN挂了，在提交阶段是不会导致数据丢失的，而在其他的情况下，也不会导致数据丢失。



62. Q：Join下推的性能怎么样？

A：这个可以分两部分看，第一是Join下推本身算法的性能，这个依赖于Plan Cache。首先我们会对执行计划做一个参数化，然后根据参数化的SQL，当判定这个查询能够下推，那它后面所有的不同参数的查询到来之后，就可以直接从Plan Cache里面读出来，那就可以下推。

接下来可能的问题是，当把Join下推到DN上执行，会不会性能就更好？这个对于TP来说的话，大体上是成立的，因为降低网络开销，肯定带来一个RT的缩短。但是对于AP类的查询，有的时候不一定。因为DN的计算资源肯定没有CN多，有的时候把一些AP类的Join下推下去是不太合理的。所以在做这个事情的时候，也有基于这个CBO的判断，CBO本身的代价模型里会去估算一下DN执行这个Join的代价和CN执行这个Join的不同代价。如果下推到DN上代价更高的话，那就会考虑把这个数据拉到CN上来做。

所以下推性能问题，在TP场景下大多数时候下推都是可以的。如果下推性能不好，也可以通过CBO识别出来，将数据拉回到CN上执行。

63. Q：每个DN都有多副本吗？

A：是的，目前每个DN都是基于X-Paxos的三副本。

64. Q：创建全局二级索引，对应列的索引hash规则保存在GMS节点吗？

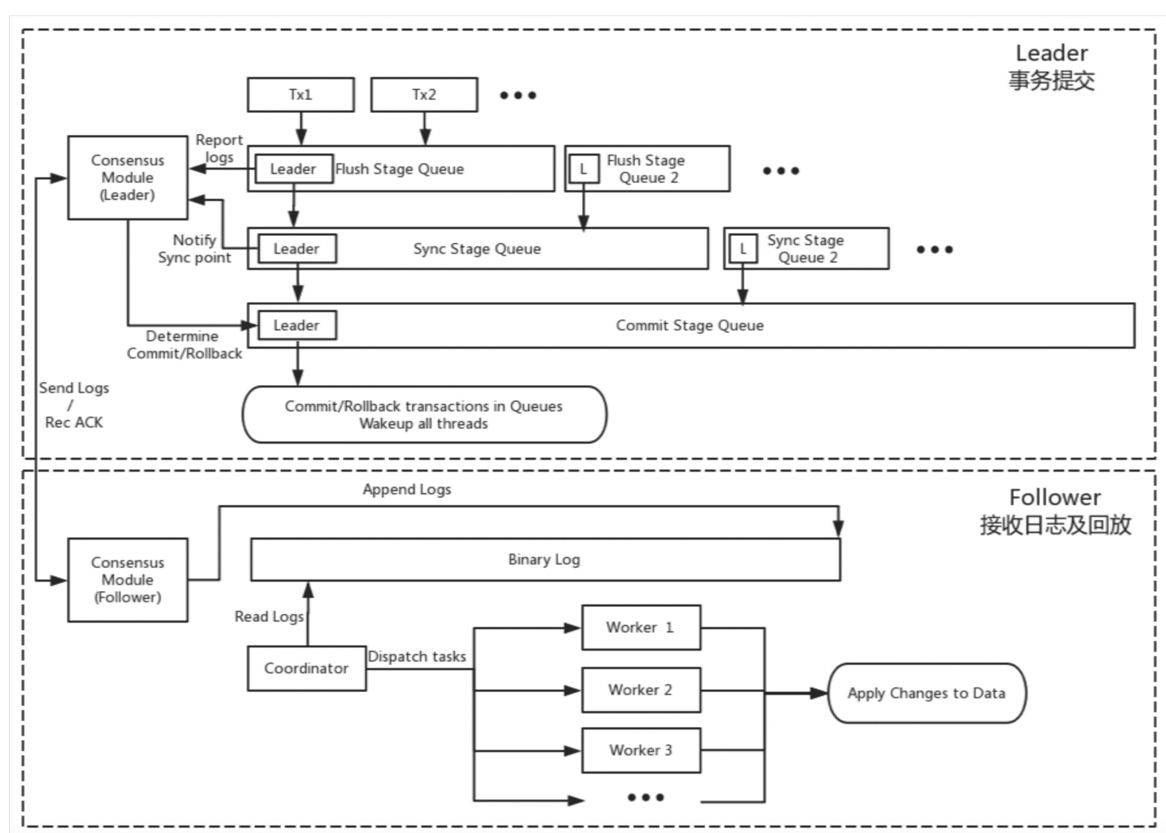
    A：是的。全局二级索引本质上也是一张表，所以它的元数据和普通表的元数据存储方式基本一致，都是存放在GMS节点上的。

65. Q：PolarDB-X怎么支持AP查询？

    A：PolarDB-X本身是基于存储计算分离的架构，支持MPP引擎，支持将数据拖到这个CN上，然后进行一个分布式的计算。

66. Q：三个节点数据如何保持同步？

    A：

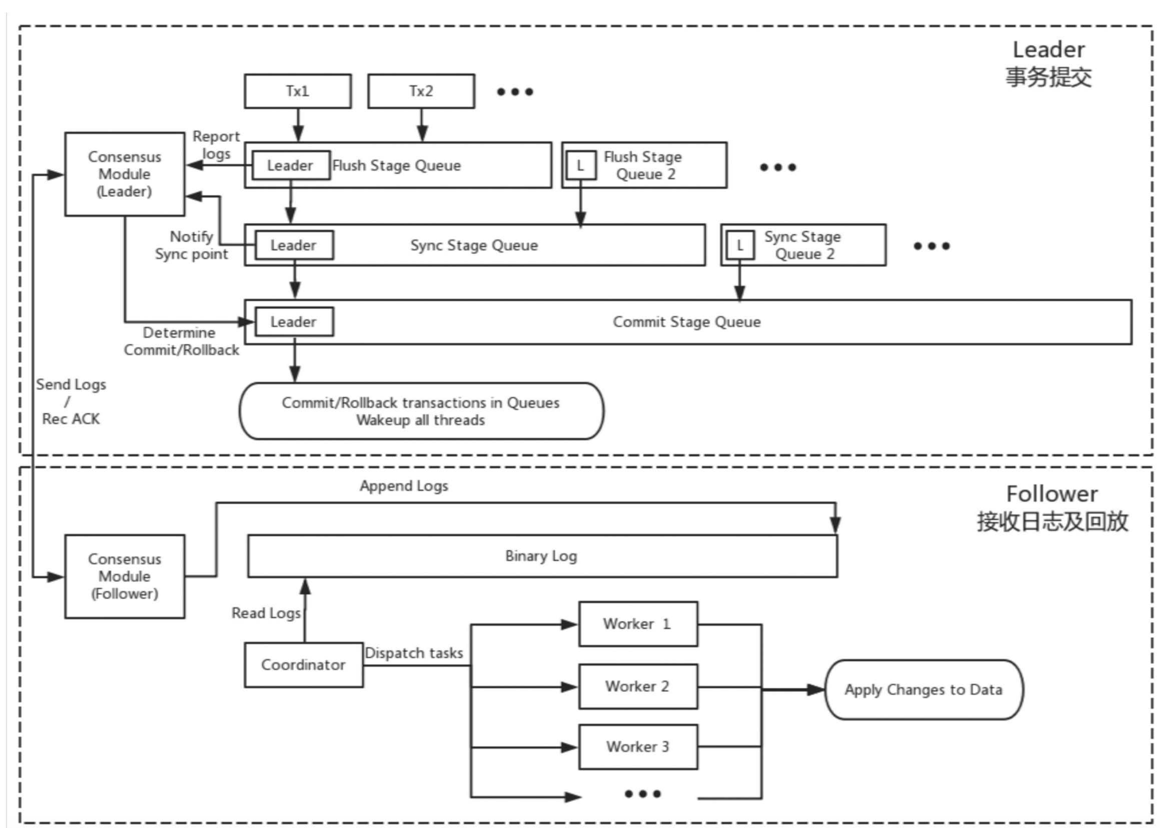


67. Q：Paxos监控Binlog传输，数据还是Binlog同步的吧？

    A：是的，Binlog还是载体，传输的日志就是Binlog。

68. Q：X-Paxos如何与InnoDB存储引擎配合？

    A：



69. Q：副本数量有上限吗？推荐的数量一般是多少？

    A：正常情况下，在生产环境中，基本使用同城三副本模式。如果要求更高，可以采用三地五中心的跨城五副本模式。

70. Q：DN上的Binlog会落盘吗？

    A：会的。

71. Q：Paxos选主的时候，如何确保新Slave追完了Binlog？

    A：选主是根据Binlog里面的日志进行选主的。也就是说，选主的时候，还不关注是否追完了Binlog。拥有集群中达到多数派最新日志的节点会成为Leader，但可能它的Binlog还没有追完。

72. Q：会出现两个Leader吗？

    A：不会。

73. Q：不使用Logger节点的话，数据的可用性是不是更高？

    A：是的。使用Logger是一种折中的选择，因为Logger节点如果换成普通Follower，就会增加一份数据存储。

74. Q：Binlog不追完，服务能起来吗？服务起不来，系统就挂了吗？

    A：有时能，有时不能。如果在Galaxy引擎上自己搭建环境，有时会发现一个节点起来后，可能登录不进去，因为这个节点在重启的时候，在客户端能够登录进去之前，首先要接入到集群里面去，这是为了确定它自己的角色以及与集群中的其他节点协商当前最新的日志是哪一个。 如果整个集群都是不健康的，在节点起来之后，没有新的Leader选出来的话，集群确实无法提供服务。

    三节点与semisync半同步有一个很大的区别：semisync半同步，在slave异常的时候，半同步会退化为异步，实际上是为了保证可用性而牺牲了数据一致性。三节点模式要保证强一致性，如果大多数节点都不健康，那整个系统就是不能用的。

75. Q：三节点之间的Binlog同步是同步模式还是异步模式？

A：同步模式。