云原生分布式数据库 PolarDB-X

最佳实践

最佳实践

如何选择实例规格

PolarDB-X的计算资源DRDS与存储资源私有RDS按照CPU的处理能力、内存容量和磁盘空间等来划分实例的规格，并提供多种不同规格的实例供选择，规格越高代表实例的处理能力越强。

DRDS实例系列对比

DRDS提供至少两个服务节点起步的专项实例保证高可用及稳定性，不同实例系列对比如下：

入门版：单节点规格 4 Core 16 GB。面向初期的业务开发、测试场景，不具备复杂查询加速能力

标准版：单节点规格 8 Core 32 GB。规格丰富，性价比高。面向具备超高并发、复杂查询及轻量分析的在线业务场景，默认提供Parallel Query并行查询能力，可有效提升对于在线业务的多表关联、聚合排序等复杂查询的执行效率

企业版：单节点规格 16 Core 64 GB。大规格资源，面向具备企业级超高并发、大规模数据复杂查询、加速分析的业务场景，默认提供Parallel Query并行查询能力，可大幅提升海量数据下复杂查询、报表分析的执行效率

DRDS实例类型与规格的选择

DRDS属于计算密集型的服务，处理能力主要与CPU相关，并以QPS为衡量指标。选择实例时，应参照不同规格支持的最大QPS，并结合估算的业务最大QPS来进行选择。例如，某用户估算其业务应用对DRDS访问的最大QPS为50000，那么应选择规格为 标准版16Core 96 GB实例，不同DRDS系列规格对比请参考实例规格。

RDS 实例规格的选择

- 预估1到2年的数据增长量，判断需要的最大磁盘空间
- 估算一个RDS实例需要的最大IOPS

建议购买多个中小规格的RDS实例，有利于后期遇到存储瓶颈时可对RDS进行快速升配以及DRDS平滑扩容。

如何选择拆分键

拆分键即分库/分表字段，是水平拆分过程中用于生成拆分规则的数据表字段。DRDS 将拆分键值通过拆分函数计算得到一个计算结果，然后根据这个结果将数据分拆到 RDS 实例上。

数据表拆分的首要原则是尽可能找到数据所归属的业务逻辑实体，并确定大部分（或核心的）SQL 操作或者具备一定并发的 SQL 都是围绕这个实体进行，然后可使用该实体对应的字段作为拆分键。

业务逻辑实体通常与应用场景相关，下面的一些典型应用场景都有明确的业务逻辑实体，其标识型字段可用来做拆分键：

- 面向用户的互联网应用，围绕用户维度来做各种操作，那么业务逻辑实体就是用户，相关联的表便可使用用户 id 作为拆分键
- 侧重于卖家的电商应用，围绕卖家维度来做各种操作，那么业务逻辑实体就是卖家，可使用卖家 id 作为拆分键
- 游戏类在线应用，围绕玩家维度来做各种操作，那么业务逻辑实体就是玩家，可使用玩家 id 作为拆分键
- 车联网在线应用，围绕车辆维度来做各种操作，那么业务逻辑实体就是车辆，可使用车辆 id 作为拆分键
- 税务类在线应用，围绕纳税人来进行前台业务，那么业务逻辑实体就是纳税人，可使用纳税人 id 作为拆分键

以此类推，其它应用场景也能找到合适的业务逻辑实体。

例如，某面向卖家的电商应用，需要对下面的一张单表进行水平拆分：

```
CREATE TABLE sample_order (
id INT(11) NOT NULL,
sellerId INT(11) NOT NULL,
trade_id INT(11) NOT NULL,
buyer_id INT(11) NOT NULL,
buyer_nick VARCHAR(64) DEFAULT NULL,
PRIMARY KEY (id)
)
```

确定业务逻辑实体为卖家，那么选择字段 sellerId 作为拆分键，分布式 DDL 建表语句为：

```
CREATE TABLE sample_order (
id INT(11) NOT NULL,
sellerId INT(11) NOT NULL,
trade_id INT(11) NOT NULL,
buyer_id INT(11) NOT NULL,
```

```
buyer_nick VARCHAR(64) DEFAULT NULL,
PRIMARY KEY (id)
) DBPARTITION BY HASH(sellerId)
```

如果确实找不到合适的业务逻辑实体作为拆分键，特别是传统企业级应用，那么可以考虑下面的方法来选择拆分键：

根据数据分布和访问的均衡度来考虑拆分键，尽量将数据表中的数据相对均匀地分布在不同分表中，DRDS 后续推出的全局强一致二级索引和 Parallel Query 能够改善在此场景下 SQL 并发度和响应时间

按照数字（字符串）类型与时间类型字段相结合作为拆分键，进行分库和分表，适用于日志检索类的应用场景

例如，某日志系统记录用户的所有操作，需要对下面的日志单表进行水平拆分：

```
CREATE TABLE user_log (
userId INT(11) NOT NULL,
name VARCHAR(64) NOT NULL,
operation VARCHAR(128) DEFAULT NULL,
actionDate DATE DEFAULT NULL
)
```

可以选择用户标识与时间字段相结合作为拆分键，并按照一周七天进行分表，则分布式 DDL 建表语句为：

```
CREATE TABLE user_log (
userId INT(11) NOT NULL,
name VARCHAR(64) NOT NULL,
operation VARCHAR(128) DEFAULT NULL,
actionDate DATE DEFAULT NULL
) DBPARTITION BY HASH(userId) TBPARTITION BY WEEK(actionDate) TBPARTITIONS 7
```

更多拆分键的选择和分表形式，请参考 DRDS Create Table 文档和 DRDS 拆分函数 文档

如何选择分片数

DRDS 中的水平拆分有两个层次：分库和分表。每个 RDS 实例上默认会创建8个物理分库，每个物理分库上可以创建一个或多个物理分表。分表数通常也被称为分片数。

一般情况下，建议单个物理分表的容量不超过500万行数据。通常可以预估1到2年的数据增长量，用估算出的总数据量除以总的物理分库数，再除以建议的最大数据量500万，即可得出每个物理分库上需要创建的物理分表数：

```
物理分库上的物理分表数 = 向上取整(估算的总数据量 / (RDS 实例数 * 8) / 5,000,000)
```

因此，当计算出的物理分表数等于1时，分库即可，无需再进一步分表，即每个物理分库上一个物理分表；若计算结果大于1，则建议既分库又分表，即每个物理分库上多个物理分表。

例如，某用户预估一张表在2年后的总数据量大概是1亿行，购买了4个 RDS 实例，那么按照上述公式计算：

```
物理分库上的物理分表数 = CEILING(100,000,000 / ( 4 * 8 ) / 5,000,000) = CEILING(0.625) = 1
```

结果为1，那么只分库即可，即每个物理分库上1个物理分表。

若上述例子中仅购买了1个 RDS 实例，那么按照上述公式计算：

```
物理分库上的物理分表数 = CEILING(100,000,000 / ( 1 * 8 ) / 5,000,000) = CEILING(2.5) = 3
```

结果为3，那么建议既分库又分表，即每个物理分库上3个物理分表。

何时选择升配

数据库性能主要可以从响应时间（RT）和容量（QPS）两个指标进行衡量。RT 指标反映的是单个 SQL 的性能，这类性能问题可以通过 SQL 优化等方法进行解决。DRDS 升配则主要通过扩充容量来提升性能，适用于低延时高 QPS 类型的数据库访问业务。

DRDS 实例性能取决于 DRDS 本身和 RDS 的性能表现，任一 DRDS 或者 RDS 节点性能不足都会导致整体性能出现瓶颈。本文主要说明如何观察 DRDS 实例的性能指标，并通过升配来解决性能不足的问题。RDS 的性能判断及升配方法请参考 RDS 文档。

判断 DRDS 实例性能瓶颈

DRDS 实例的 QPS 和 CPU 性能是正相关的。当 DRDS 性能出现瓶颈时，主要表现为实例的 CPU 利用率居高不下。

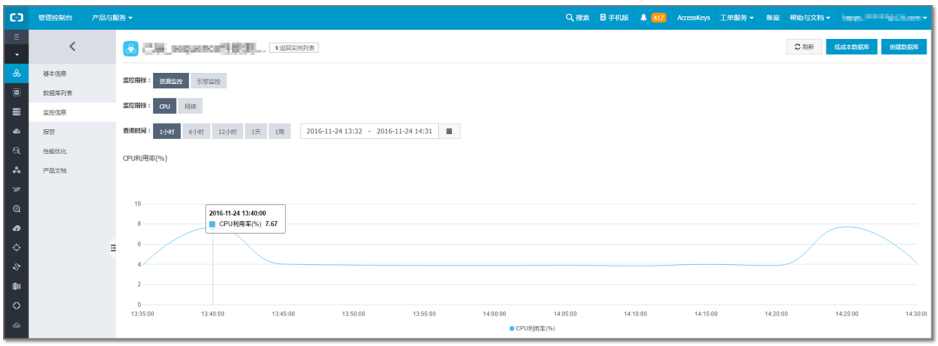
观察 CPU 利用率监控项

在 DRDS 控制台左侧菜单栏选择**实例列表**。

单击需要查看的实例名称进入实例基本信息页。

在左侧菜单栏选择**监控信息**。

如果发现 CPU 利用率**超出90%或持续超出80%**，则意味着当前实例性能出现瓶。在 RDS 不存在瓶颈的情况下，可以判断当前的 DRDS 实例规格无法满足业务的 QPS 性能需求，需要通过升配解决。



更多性能相关的业务监控场景及配置 DRDS CPU 利用率报警的方法请参考 DRDS 实例监控。

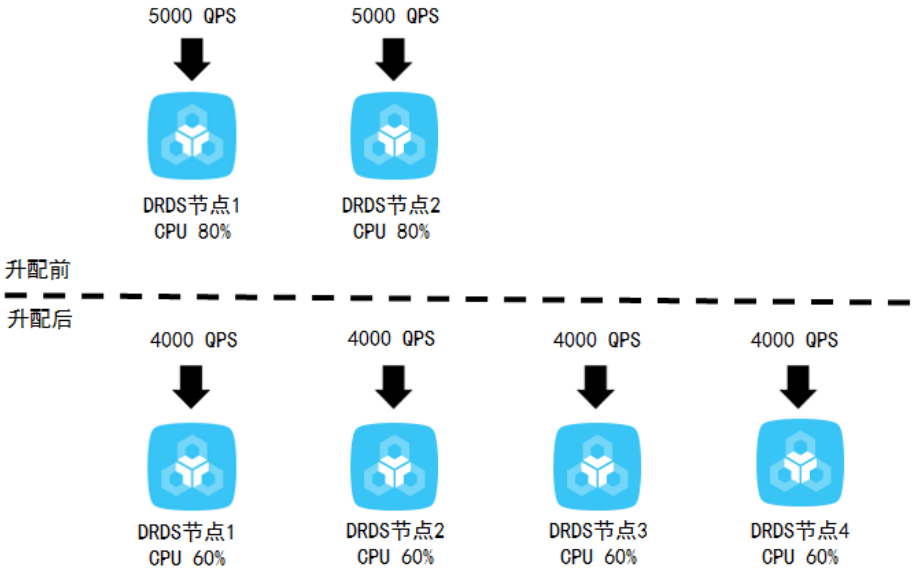
DRDS 升配

QPS 是衡量 DRDS 实例规格的重要指标。每种实例规格对应一定的 QPS 参考值，具体请参考文档 DRDS 实例规格。

注意：有些特殊的 SQL 语句在 DRDS 层面需要更多的计算（如临时表排序、聚合计算等），此时每个 DRDS 实例可以支撑的 QPS 相比规格中的标准值会有所下降。

DRDS 升配以增加处理节点，均摊 QPS 的方式来提高实例的处理性能。由于 DRDS 节点本身是无状态的，因此这种升配方式对 DRDS 实例的性能会有线性的提升。

例如业务 A 需要1.5万左右的 QPS 性能，当前 DRDS 实例规格为 4C4G，两个节点， QPS 只能达到1万。通过观察发现 DRDS 的 CPU 占用一直处于高位后，升配到 8C8G，升配后实例节点约各承担4000的 QPS。此时性能满足了用户的需求，同时 CPU 利用率也下降到合理位。如下图所示：



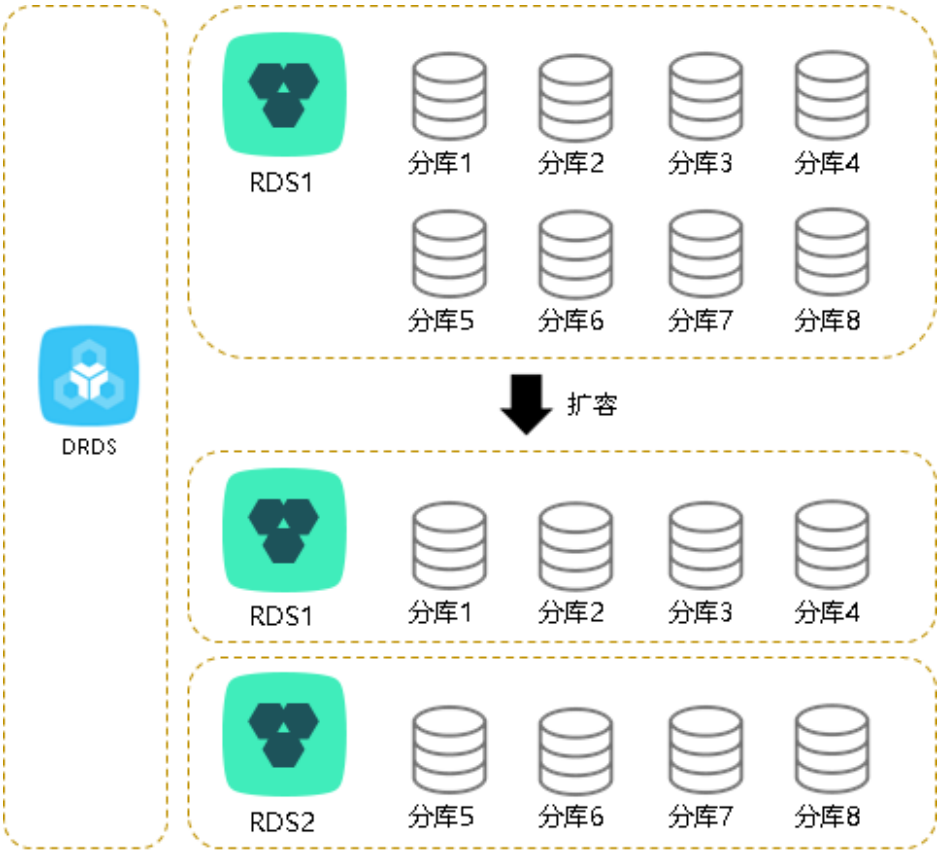
实例升配的具体操作请参考 DRDS 实例变配。

何时选择平滑扩容

什么是平滑扩容

DRDS 平滑扩容是指通过增加 RDS 的数量以提升整体性能。当 RDS 的 IOPS、CPU、磁盘容量等指标到达瓶颈，并且 SQL 优化、RDS 升配已无法解决瓶颈（例如磁盘已升至顶配）时，可通过 DRDS 水平扩容增加 RDS 数量，提升 DRDS 数据库的容量。

DRDS 平滑扩容通过迁移分库到新 RDS 来降低原 RDS 的压力。例如，扩容前8个库的压力集中在一个 RDS 实例上，扩容后8个库分别部署在两个 RDS 实例上，单个 RDS 实例的压力就明显降低。如下图所示：



说明：平滑扩容多次后，如果出现 RDS 数量和分库数量相等的情况，需要创建另外一个 DRDS 和预期容量 RDS 的数据库，再进行数据迁移以达到更大规模数据容量扩展的目标。此过程较复杂，推荐创建 DRDS 数据库时要考虑未来2-3年数据的增长预期，做好 RDS 数量规划。

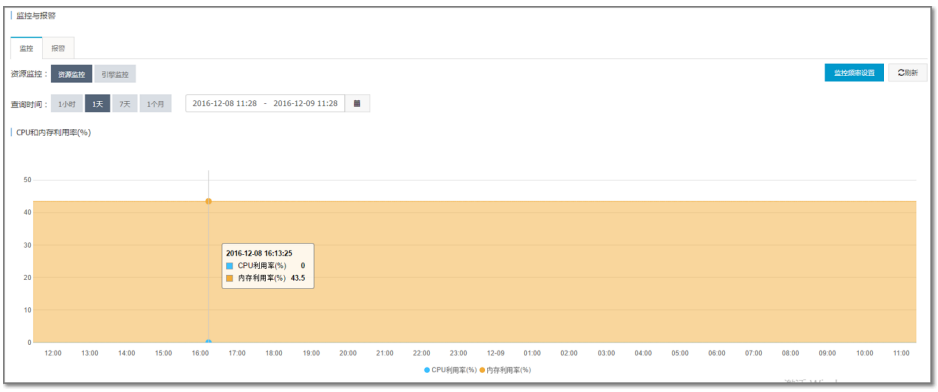
判断是否需要平滑扩容

DRDS 是否需要进行平滑扩容，可以通过观察 RDS 的三个指标进行判断：IOPS、CPU、磁盘空间。在 RDS 控制台可以查看这些指标，详情请参见 RDS 相关文档。

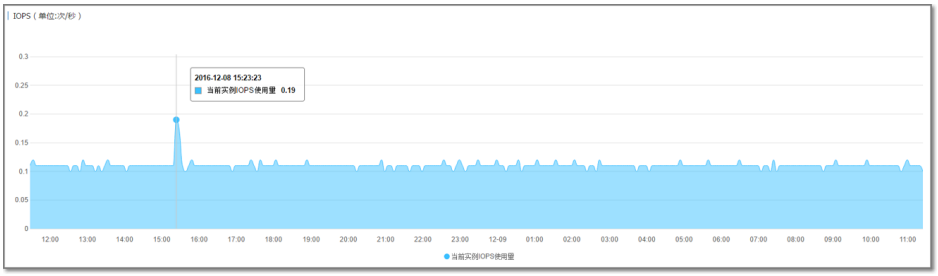
在 RDS 控制台也可以针对这些指标设置专门的报警项，报警策略及默认阈值见下图。

CPU 及 IOPS 指标

RDS 平台 CPU 监控信息。



RDS 对 IO 的频率做了限制，体现在 IOPS 这个指标上。



对于以上两个指标，如果发现任何一个指标长期保持在80%以上或频繁收到报警信息，请考虑通过以下步骤来解决：

尝试 SQL 优化。CPU 利用率过高的问题通常都可以通过这一步解决，详情请参考 MySQL CPU 使用率高的解决方法及MySQL IOPS 使用率高的解决方法。

SQL 优化无法解决问题时，可以升高 RDS 的相关规格。详情请参考 RDS 升配文档。

当 CPU 和 IOPS 超标时，可以通过设置只读库的方式来分担主库负荷。注意读写分离会影响读一致性。详情请参见设置读写分离文档。

当以上步骤无法很好地解决问题时，请考虑进行 DRDS 扩容。

磁盘空间

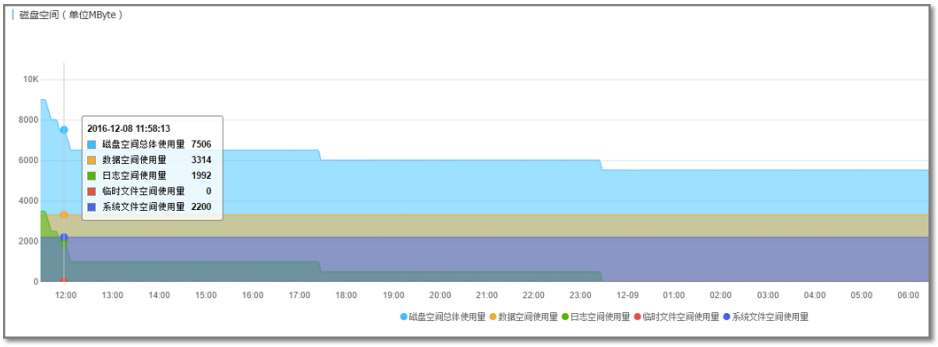
RDS 的磁盘空间有以下几种：

数据空间：数据所占用的空间。随着数据的插入，空间占用量会一直增长。磁盘存储容量的余量建议保持在30%以上。

系统文件空间：包括共享表空间、错误日志文件等。

Binlog 文件空间：这是数据库运行过程中产生的空间。更新事务越多，空间占用量就越大。

判断是否需要扩容主要关注**数据空间**即可。当数据空间将要或预期要超出磁盘容量时，可以通过扩容的方式将数据分散到多个 RDS。



扩容风险及注意事项

DRDS 扩容流程分为 **配置**>**迁移**>**切换**>**清理** 四个步骤。具体请参考**扩容操作**文档。

在扩容前请注意以下事项：

如果需要新增5个或5个以上 RDS 实例，需要事先提工单，以防后端迁移资源不足造成迁移不成功。

源 RDS 实例扩容过程中会有读压力，请尽量在源 RDS 低负载时操作。

扩容期间请勿在控制台提交 DDL 任务或连接 DRDS 直接执行 DDL SQL，否则会导致扩容任务失败。

扩容需要源库表中有主键，如果没有需要事先加好主键。

扩容的切换动作会将读写流量切换到新增的 RDS 上，切换过程大约持续3~5分钟，建议在停业务的情况下进行切换。

在执行切换前，扩容动作不会对 DRDS 产生任何影响。因此在切换前都可以通过回滚来放弃本次扩容

。

扩容操作对数据库有一定压力，建议在业务低谷期操作。

如何选择应用连接池

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。

资源复用：连接可以重复利用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，同时增进了系统的平稳性。

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。

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。

连接池推荐

将应用和数据库连接进行业务操作，建议使用连接池。如果是 Java 程序，推荐使用 Druid 连接池，最低要求版本1.1.11。

Druid 的 Spring 标准配置

```
    <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&rewriteBatchedStatements=true&socketTimeout=30000&con
nectTimeout=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" />
```

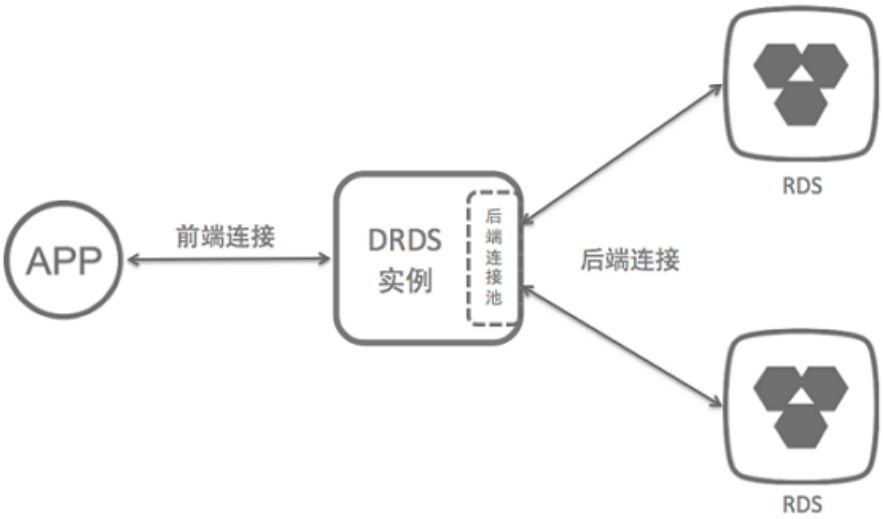
```
<!-- minEvictableIdleTimeMillis 一个连接在池中最小空闲的时间，单位是毫秒-->
<property name="minEvictableIdleTimeMillis" value="300000" />
<!-- 检测连接是否可用的 SQL -->
<property name="validationQuery" value="select 'z' from dual" />
<!-- 是否开启空闲连接检查 -->
<property name="testWhileIdle" value="true" />
<!-- 是否在获取连接前检查连接状态 -->
<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。此参数默认可以不加，但是增加此参数可以均衡后端服务节点参数 -->
<property name="phyTimeoutMillis" value="1800000" />
<!-- 是否在固定SQL使用次数之后关闭连接，此参数默认可以不加，但是增加此参数可以均衡后端服务节点参数-->
<property name="phyMaxUseCount" value="10000" />
</bean>
```

PolarDB-X实例中的连接

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，会有两种类型的连接：

前端连接：由应用程序建立的，到PolarDB-X实例中逻辑库的连接；

后端连接：由PolarDB-X实例中的节点建立的，到后端私有制RDS实例中物理库的连接。



前端连接

前端连接的数量理论上仅受限于PolarDB-X实例节点可用的内存大小和网络连接数。但在实际的应用场景中

，应用程序连接到PolarDB-X实例时，通常会管理有限数量的连接来执行请求的操作，并不会维持很高并发量的持久化长连接（例如，数万个并发的长连接），因此可认为PolarDB-X实例能接受的前端连接数量是无限制的。

由于前端连接数量不受限制，可以允许有大量空闲连接存在，因此适用于业务端部署应用程序的服务器数量较多，需要同时保持连接到PolarDB-X实例的场景。

注意：虽然前端连接的数量可被认为是无限制的，但从前端连接获取的操作请求是由PolarDB-X实例的内部线程通过后端连接实际执行，而内部线程和后端连接的数量有限，因此PolarDB-X实例处理请求的整体并发度是有限的。

后端连接

PolarDB-X实例的每个节点内部都会创建后端连接池，自动管理和维护到私有制RDS实例中物理库的后端连接。因此，PolarDB-X实例中后端连接池的最大连接数与私有制RDS实例支持的最大连接数直接相关。可参照以下公式来计算PolarDB-X实例中后端连接池的最大连接数：

```
PolarDB-X实例后端连接池的最大连接数 = 向下取整（私有制RDS实例最大连接数 / 私有制RDS实例物理分库数 / PolarDB-X实例节点数）
```

例如，某用户搭配购买了如下规格的私有制RDS实例和PolarDB-X实例：

- 1个私有制RDS实例，包含8个物理分库，规格为4Core16G，最大连接数为4000；
- 1个PolarDB-X专享实例，规格为 32Core32G（每2Core2G为1个PolarDB-X节点，即该实例包含16个PolarDB-X节点）。

按照上述公式可计算出PolarDB-X实例中后端连接池的最大连接数：

```
PolarDB-X实例后端连接池的最大连接数=FLOOR(4000 / 8 / 16) = FLOOR(31.25) = 31
```

注意：

上述公式计算的结果为PolarDB-X实例中后端连接池的最大连接数的上限。实际使用中，为了减轻RDS 实例的连接压力，留出一定的缓冲余地，PolarDB-X实例会适当调整后端连接池的最大连接数，使其小于上限值。

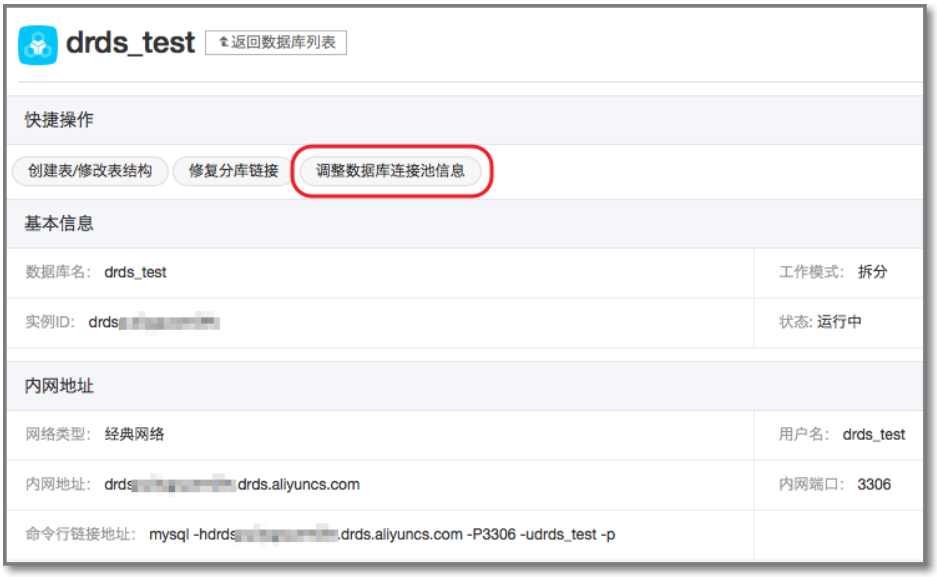
建议PolarDB-X实例中的数据库创建在私有制RDS实例上，即私有制RDS实例上不应创建其它应用或其它PolarDB-X实例的数据库。

调整后端连接数

若增大私有制RDS实例的最大连接数后，想调整PolarDB-X实例中后端连接池的最大连接数，请按以下步骤操作。

1. 登录云原生分布式数据库控制台，在数据库列表页选择相应的数据库。

在数据库**基本信息**页，单击**调整数据库连接池信息**。控制台会自动进行调整。



前后端连接的关系

在应用程序与PolarDB-X实例建立了前端连接并发出SQL语句的执行请求后，PolarDB-X实例节点会异步地处理请求，并通过内部后端连接池获取后端连接，在一个或多个物理库上执行经过优化处理的SQL语句。

由于PolarDB-X实例节点内部为异步的处理流程，前端连接和后端连接并无绑定关系，因此在通常的短事务和简单查询情况下，少量的后端连接有能力处理并发量较高的前端连接所带来的大量请求。这也是在PolarDB-X中应关注QPS指标，而非并发连接数的原因。

虽然前端连接数量可被认为近乎无限制，但PolarDB-X实例节点内部的后端连接池所维护的最大连接数是有限的（参见上文“后端连接”一节的说明），因此在实际的应用场景中，需要注意以下几点：

应用程序中应尽量避免长的（或称为大的）事务，以免其长时间未提交或回滚而占用后端连接，造成后端连接池紧张或达到上限，降低整体的并发处理能力，增加响应时间；

应监控并优化或消除在PolarDB-X中执行的慢查询，以免其长时间执行而占用后端连接，造成后端连接池紧张或达到上限。这种情况下PolarDB-X实例或私有制RDS实例会面临更大的处理压力，导致整体的并发处理能力下降，增加响应时间，还可能因为执行超时导致SQL执行失败率上升。对于慢查询的排查和优化，请参考 排查慢SQL 和 SQL优化；

若在正常使用连接和执行查询的情况下，对后端连接的使用仍然达到了PolarDB-X实例中后端连接池的最大连接数，请联系客服协助解决。

如何处理 DDL 异常

DDL 原理简介

DRDS 的 DDL 指令会在所有分表上执行对应的 DDL 操作。失败的情况可以分为两类：

1. DDL 在分库执行失败。DDL 在任意分库执行出错都可能导致各分表结构不一致。
2. 执行长时间无响应。在对大表执行 DDL 操作时，有可能由于分库的执行时间过长导致 DDL 长时间无响应。

分库执行报错的原因多种多样，如建表时表已存在、加列时列已存在等各类冲突、磁盘空间不足等。

长时间无响应一般是由分库的执行时间过长导致的。以 MySQL（RDS 原理与之类似） 为例，DDL 的耗时大部分取决于该操作是 In-Place（直接在源表修改）还是 Copy Table（拷贝表数据）。In-Place 只需要修改元数据就可以了，而 Copy Table 需要重建整张表数据，此外还涉及日志和 buffer 操作。

各类操作与这两项因素的关系详见 MySQL 官方文档 Summary of Online Status for DDL Operations。

判断 DDL 操作是 In-place 还是 Copy Table 操作，可以查看操作结束后 “rows affected” 这一项的返回值。

示例：

改变某列的默认值（非常快，完全不会影响表数据）：

Query OK, 0 rows affected (0.07 sec)

增加一个索引（需要花些时间 ，但是 0 rows affected 说明表数据没有被复制）：

Query OK, 0 rows affected (21.42 sec)

改变某列的数据类型（耗费大量时间并且需要重建表中的所有数据行）：

Query OK, 1671168 rows affected (1 min 35.54 sec)

因此，执行一个大表 DDL 操作前，可以先通过以下步骤判定这是一个快速或慢速操作：

1. 复制表结构生成一张克隆表。
2. 插入一些数据。
3. 在克隆表上执行这个 DDL 操作。
4. 检查操作完成后 “rows affected” 值是否是0。一个非0的值意味着该操作需要重建整张表，这时可能需要考虑在流量低谷去执行该操作。

失败处理

DRDS DDL 操作会将所有 SQL 分发到所有分库上并行执行。任一分库上执行失败不会影响其他分库。另外，DRDS 还提供了 CHECK TABLE 指令来检测分表结构的一致性。因此，失败的 DDL 操作可以重新执行，已经执行成功的分库上失败报错并不会影响其他分库。只要保证最终所有分表结构一致即可。

DDL 失败处理步骤

1. 使用 **CHECK TABLE** 指令检查表结构。如果返回结果只有一行且为状态正常则可认为**表状态一致**。此时进行步骤2，否则进行步骤3。
2. 使用 **SHOW CREATE TABLE** 指令检查表结构。如果显示的表结构符合 DDL 执行后的预期则可认为**DDL 执行成功**，否则继续进行步骤3。
3. 使用 **SHOW PROCESSLIST** 指令观察所有当前执行的 SQL 状态。如有仍在执行的 DDL 操作，请等候其执行完成后再进行步骤1、2，检查表结构是否符合预期，否则进行步骤4。
4. 在 DRDS 上重新执行 DDL 操作。如果出现 **Lock conflict** 的报错请进行步骤5，否则进行步骤3。
5. 使用 **RELEASE DBLOCK** 指令释放 DDL 操作锁，然后进行步骤4。

详细操作如下：

检查表结构一致性

使用 CHECK TABLE 指令检查表结构，当返回结果只有一行且显示状态 OK 时，表明**表结构一致**。

注意：如果在 DMS 上执行 CHECK TABLE 没有返回结果，请在命令行下重试。

```
mysql> check table `xxxx`;
+----------------------------+-------+----------+----------+
| TABLE                      | OP    | MSG_TYPE | MSG_TEXT |
+----------------------------+-------+----------+----------+
| TDDL5_APP.xxxx             | check | status   | OK       |
+----------------------------+-------+----------+----------+
1 row in set (0.05 sec)
```

检查表结构

使用 SHOW CREATE TABLE 指令检查表结构，如果**表结构一致**且**表结构无误**时，可认为 DDL 已执行成功。

```
mysql> show create table `xxxx`;
+---------+-----------------------------------------------------------------------------------------------------------
--------+
| Table   | Create Table                                                                                              |
+---------+-----------------------------------------------------------------------------------------------------------
--------+
|  xxxx   | CREATE TABLE `xxxx` (
```

```
`id` int(11) NOT NULL DEFAULT '0',
`NAME` varchar(1024) NOT NULL DEFAULT '',
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`) tbpartition by hash(`id`) tbpartitions 3
|
+---------+-----------------------------------------------------------------------------------------------------------------
--------+
1 row in set (0.05 sec)
```

观察当前正在执行的 SQL 语句

有些 DDL 执行速度过慢，发现 DDL 长时间无响应后，可执行 SHOW PROCESSLIST 指令观察所有当前执行的 SQL 状态。

```
mysql> SHOW PROCESSLIST WHERE COMMAND != 'Sleep';
+---------------+-----------+--------------------+-------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------
----------------------+-----------+---------------+-----------+
| ID            | USER      | DB                 | COMMAND     | TIME    | STATE
| INFO                                                                                | ROWS_SENT | ROWS_EXAMINED |
ROWS_READ |
+---------------+-----------+--------------------+-------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------
----------------------+-----------+---------------+-----------+
| 0-0-352724126 | ifisibhk0 | test_123_wvvp_0000 | Query       |      15 | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */select `t_item`.`detail_url`,SUM(`t_item`.`price`) from `t_i |
NULL |          NULL |      NULL |
| 0-0-352864311 | cowxhthg0 | NULL               | Binlog Dump |      13 | Master has sent all binlog to slave;
waiting for binlog to be updated | NULL                                                                                |
NULL |          NULL |      NULL |
| 0-0-402714566 | ifisibhk0 | test_123_wvvp_0005 | Query       |      14 | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */select `t_item`.`detail_url`,`t_item`.`price` from `t_i      |
NULL |          NULL |      NULL |
| 0-0-402714795 | ifisibhk0 | test_123_wvvp_0005 | Alter       |     114 | Sending data
| /*DRDS /42.120.74.88/ac47e5a72801000/ */ALTER TABLE `Persons` ADD `Birthday` date                   |
NULL |          NULL |      NULL |
......
+---------------+-----------+--------------------+-------------+---------+-----------------------------------
-----------------------------------+-----------------------------------------------------------------------------------------
----------------------+-----------+---------------+-----------+
12 rows in set (0.03 sec)
```

TIME 列代表该指令已经执行的秒数。发现过慢指令后，例如图中，可使用 **KILL '0-0-402714795'** 指令来取消慢指令。

注意： DRDS 中一个逻辑 SQL 对应多条分库指令，因此为了停止一个逻辑 DDL 可能需要 Kill 多条指令。从 SHOW PROCESSLIST 结果集的 INFO 列判断该指令归属的逻辑 SQL。

Lock conflict 报错处理

DRDS 执行 DDL 操作先会加库级锁，操作完后再释放掉。KILL DDL 操作很可能会导致该锁没有释

放，此时再执行 DDL 会有以下报错：

```
Lock conflict , maybe last DDL is still running
```

此时执行 **RELEASE DBLOCK** 释放该锁即可。指令取消及锁释放后，选择业务低谷甚至停止期间，重新执行该 DDL。

其它问题

DMS 或其它客户端无法显示修改后的表结构

为了兼容某些客户端从系统表（如 COLUMNS 或 TABLES）中获取表结构的功能，DRDS 在用户0分库 RDS 里建立了一个影子库，影子库名与用户的 DRDS 逻辑库名一致。存储了所有用户库里的表结构等信息。

DMS 获取 DRDS 表结构是从影子库系统表中获取的。在处理异常 DDL 过程中，由于种种原因可能会出现用户库表结构正常修改，但是影子库中的表结构却未修改的现象。此时需要用户连接到影子库，在该库中重新对表进行一次 DDL 操作即可。

注意： CHECK TABLE 不会检测影子库表结构与用户库是否一致。

如何高效扫描 DRDS 数据

DRDS 支持高效的数据扫描方式，并支持在全表扫描时使用聚合函数进行统计汇总。

常见的扫描场景如下：

没有分库分表： DRDS 会把原 SQL 传递到后端 MySQL 执行。这种情况下 DRDS 支持任何聚合函数。

非全表扫描：SQL 经过 DRDS 路由后，发送到单个 MySQL 库上执行。比如说拆分键在 WHERE 中是等于关系时，就会出现非全表扫描。此时同样可以支持任何聚合函数。

全表扫描：目前支持的聚合函数有 COUNT、MAX、MIN、SUM。另外在全表扫描时同样支持 LIKE、ORDER BY 、LIMIT 以及 GROUP BY 语法。

并行的全表扫描：如果需要从所有库导出数据，可以通过 SHOW 指令查看表拓扑结构，针对分表并

行处理。详见下文。

通过 HINT 来进行表遍历

执行 SHOW TOPOLOGY FROM TABLE_NAME 指令获取表拓扑结构。

```
mysql:> SHOW TOPOLOGY FROM DRDS_USERS;
+------+------------------+--------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+------------------+--------------+
| 0 | DRDS_00_RDS | drds_users |
| 1 | DRDS_01_RDS | drds_users |
+------+------------------+--------------+
2 rows in set (0.06 sec)
```

非分库分表的表默认存储在第0个分库。

针对 TOPOLOGY 进行单表遍历。

- 第0个分库运行当前 SQL

```
/!TDDL:node='DRDS_00_RDS'*/ SELECT * FROM DRDS_USERS;
```

- 第1个分库运行当前 SQL

```
/!TDDL:node='DRDS_01_RDS'*/ SELECT * FROM DRDS_USERS;
```

注意：推荐每次扫描前执行 **SHOW TOPOLOGY FROM TABLE_NAME** 获取最新的表拓扑结构。

并行扫描

DRDS 支持 mysqldump 指令导出数据。但如果想要更快地扫描数据，可以针对每个分表开启多个会话的方式并行加速扫描。

```
mysql> SHOW TOPOLOGY FROM LJLTEST;
+------+----------------+------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+----------------+------------+
| 0 | TDDL5_00_GROUP | ljltest_00 |
| 1 | TDDL5_00_GROUP | ljltest_01 |
| 2 | TDDL5_00_GROUP | ljltest_02 |
| 3 | TDDL5_01_GROUP | ljltest_03 |
| 4 | TDDL5_01_GROUP | ljltest_04 |
| 5 | TDDL5_01_GROUP | ljltest_05 |
| 6 | TDDL5_02_GROUP | ljltest_06 |
| 7 | TDDL5_02_GROUP | ljltest_07 |
| 8 | TDDL5_02_GROUP | ljltest_08 |
```

```
| 9 | TDDL5_03_GROUP | ljltest_09 |
| 10 | TDDL5_03_GROUP | ljltest_10 |
| 11 | TDDL5_03_GROUP | ljltest_11 |
+------+----------------+------------+
12 rows in set (0.06 sec)
```

如上所示该表有四个分库，每个分库有三个分表。使用以下的 SQL 对 TDDL5_00_GROUP 库上的分表进行操作：

```
/!TDDL:node='TDDL5_00_GROUP'*/ select * from ljltest_00;
```

注意： HINT 中的 TDDL5_00_GROUP 与 SHOW TOPOLOGY 指令结果中的 GROUP_NAME 列相对应。另外 SQL 中的表名为分表名。

此时可开启最多12个会话（分别对应12张分表）并行处理数据。