ALIBABA CLOUD

阿里云

云原生分布式数据库 PolarDB-X

最佳实践

文档版本：20220117

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.如何选择实例规格 05
2.如何选择拆分键 07
3.如何选择分片数 09
4.何时选择升配 10
5.何时选择平滑扩容 11
6.如何选择应用连接池 14
7.PolarDB-X 1.0实例中的连接 15
8.如何处理DDL异常 17
9.如何高效扫描数据 21
10.在IN查询中如何选择Values个数 23

1.如何选择实例规格

PolarDB-X 1.0计算资源实例与PolarDB-X 1.0存储资源实例按照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。本文介绍如何选择实例规格。

PolarDB-X 1.0计算资源实例类型与规格的选择

PolarDB-X 1.0计算资源实例均为专享实例，每个实例最少提供2个节点保证高可用。PolarDB-X 1.0计算资源实例提供的是计算密集型的服务，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PolarDB-X 1.0计算资源实例访问的最大QPS为50000，那么应选择规格为标准版16核64 GB实例，不同PolarDB-X 1.0计算资源实例系列规格对比请参考实例规格。

系列	实例规格（CPU和内存）	特点
入门版	8核32 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	16核64 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。
	32核128 GB	
	48核192 GB	
	64核256 GB	
	96核384 GB	
	128核512 GB	
企业版	32核128 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。
	48核192 GB	
	64核256 GB	
	96核384 GB	
	128核512 GB	
	192核768 GB	
	256核1024 GB	
	348核1536 GB	
	512核2048 GB	
	768核3072 GB	
	1024核4096 GB	

PolarDB-X 1.0存储资源实例规格的选择

- 预估1~2年内的业务增长量，判断需要的最大磁盘空间。
- 估算一个PolarDB-X 1.0存储资源实例需要的最大IOPS。

> **说明** 建议购买多个中小规格的PolarDB-X 1.0存储资源实例，方便后期当遇到存储瓶颈时可对PolarDB-X 1.0存储资源实例进行快速升配以及PolarDB-X 1.0计算资源实例的平滑扩容。

2.如何选择拆分键

本文将介绍如何在PolarDB-X 1.0中选择合适的拆分键。

背景信息

拆分键即分库或分表字段，是水平拆分过程中用于生成拆分规则的数据表字段。PolarDB-X 1.0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私有定制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。

示例

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（以此类推，其它应用场景也能找到合适的业务逻辑实体），其标识型字段可用来做拆分键。

> **说明** 通常情况下，不建议将允许存在空值的列作为拆分键。

- 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，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操作，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。

例如某面向卖家的电商应用，需要对如下单表进行水平拆分：

```
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，则您可以使用如下分布式DDL语句建表：

```
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，那么可以考虑以下方法来选择拆分键。

- 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，PolarDB-X 1.0推出了全局强一致二级索引和Parallel Query能够提高在此场景下SQL并发度并缩短响应时间。
- 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了用户的所有操作，现需要对如下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
)
```

此时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，则您可以使用如下分布式DDL语句建表：

```
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，请参见CREATE TABLE和拆分函数概述。

3.如何选择分片数

本文将介绍如何为PolarDB-X 1.0中选择分片数（即水平拆分时的物理分表数）。

背景信息

PolarDB-X 1.0中的水平拆分包含了分库和分表两个层次。若您在创建数据库时，选择拆分模式为**水平拆分**，则PolarDB-X 1.0为默认为每个私有定制RDS实例创建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，而分表数通常也被称为分片数。

计算公式

一般情况下，建议单个物理分表的总容量范围在500万~5000万行数据（若单行记录超过4KB，建议总容量范围不超过500万），同时控制B+树的深度为3~4层。

您可以先预估1~2年内的数据增长量，用估算出的总数据量除以总的物理分库数，再除以建议的单个物理分表的最大数据量（本文以500万为例），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/（私有定制RDS实例数 x 8）/ 5,000,000）
```

因此，若计算出的物理分表数等于1时，当前分库即可满足需求，您无需再进一步分表，保持当前每个物理分库上一个物理分表即可。若计算结果大于1，则建议既分库又分表，即每个物理分库上再创建多个物理分表。

示例

- 假设预估一张表在2年后的总数据量约为1亿行，您已购买了4个私有定制RDS实例，那么按照分片数公式进行如下计算：

  ```
  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，那么您只需要分库而无需分表，即保持当前每个物理分库上1张物理分表即可。

- 假设预估一张表在2年后的总数据量约为1亿行，但您只购买了1个私有定制RDS实例，那么按照分片数公式进行如下计算：

  ```
  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，那么建议您既分库又分表，即需要在每个物理分库上再创建3张物理分表。

4.何时选择升配

本文介绍如何查看PolarDB-X 1.0实例的性能指标并通过升配来解决性能不足的问题。

背景信息

数据库性能主要受响应时间（RT）和容量（QPS）两个指标的影响。

- 响应时间（RT）：RT指标反映的是单个SQL的性能，这类性能问题可以通过SQL调优方法等方法进行解决。
- 容量（QPS）：容量瓶颈问题可以通过PolarDB-X 1.0实例升配来解决，通过升配来扩充容量的方式适用于低延时高QPS类型的数据库访问业务场景。

PolarDB-X 1.0性能同时受到计算层和存储层节点性能的影响，任一计算层或者存储层节点性能不足都会导致整体性能出现瓶颈。关于如何查看存储层节点性能，请参见存储监控。

判断是否出现实例性能瓶颈

PolarDB-X 1.0实例的QPS和CPU性能是正相关的。当PolarDB-X 1.0性能出现瓶颈时，主要表现为实例的CPU使用率居高不下。如果发现CPU使用率超过90%或持续超过80%，则意味着当前实例性能出现了性能瓶颈。

关于如何查看计算层的性能指标，请参见实例监控。

在存储层不存在瓶颈的情况下，可以判断当前的计算层实例规格无法满足业务的QPS性能需求，需要通过升配解决。

PolarDB-X 1.0升配

QPS是衡量PolarDB-X 1.0实例规格的重要指标。每种实例规格对应一定的QPS参考值，详情请参见实例规格。

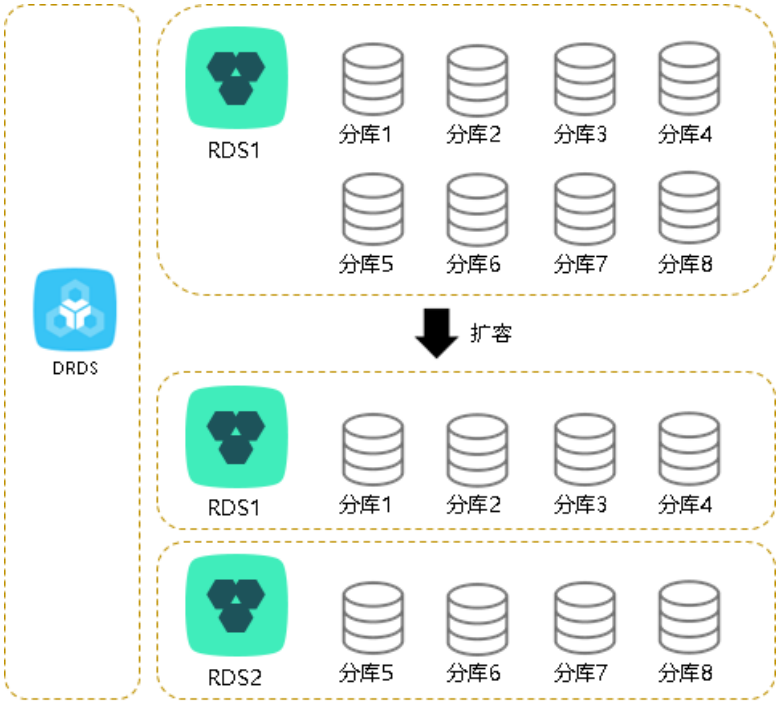
> **说明** 有些特殊的SQL语句在PolarDB-X 1.0计算层面需要更多的计算（如临时表排序、聚合计算等），此时每个PolarDB-X 1.0实例可以支撑的QPS相比规格中的标准值会有所下降。

关于如何升配，详细操作请参见实例变配。

5.何时选择平滑扩容

PolarDB-X 1.0平滑扩容是指通过增加私有定制RDS实例的数量以提升整体性能。当私有定制RDS的IOPS、CPU、磁盘容量等指标到达瓶颈，并且通过SQL优化或私有定制RDS升配已无法解决瓶颈（例如磁盘已升至顶配）时，可通过PolarDB-X 1.0水平扩容功能增加私有定制RDS实例的数量，提升PolarDB-X 1.0数据库的容量。

PolarDB-X 1.0平滑扩容通过迁移分库到新私有定制RDS来降低原私有定制RDS的压力。例如，扩容前8个库的压力集中在一个私有定制RDS实例上，扩容后8个库分别部署在两个私有定制RDS实例上，单个私有定制RDS实例的压力就明显降低。



说明 平滑扩容多次后，如果出现私有定制RDS数量和分库数量相等的情况，需要创建另外一个PolarDB-X 1.0和预期容量私有定制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PolarDB-X 1.0数据库时要考虑未来2~3年数据的增长预期，做好私有定制RDS数量规划。

判断是否需要平滑扩容

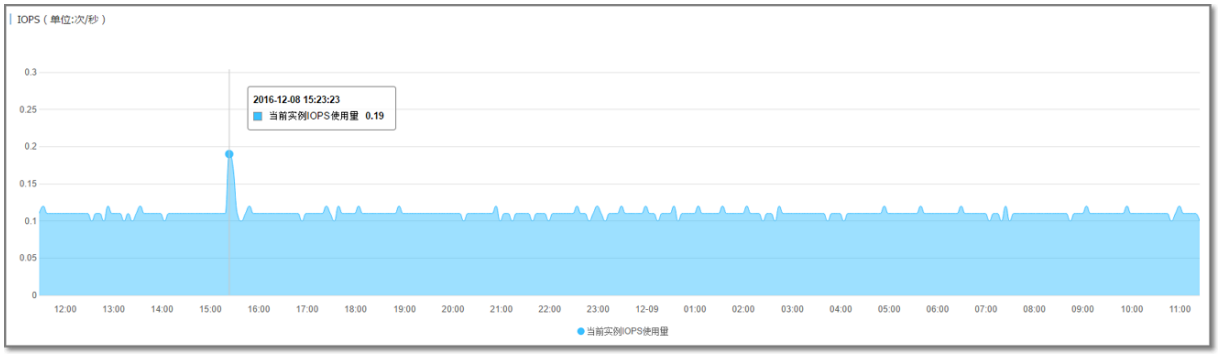
您可以通过观察私有定制RDS的IOPS、CPU、磁盘空间这三个指标，来判断是否需要对PolarDB-X 1.0进行平滑扩容。

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：

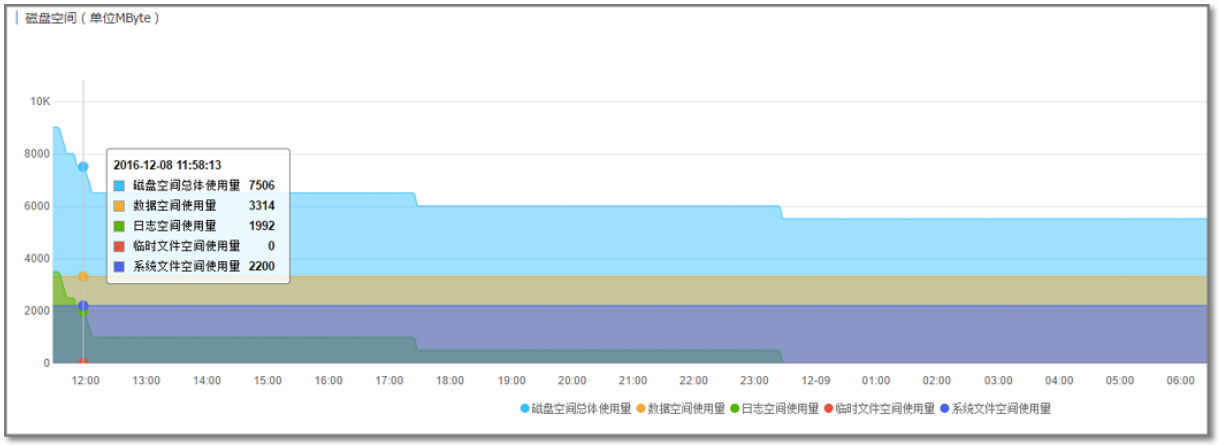
1. 尝试SQL优化，CPU使用率过高的问题通常都可以通过这一步解决。
2. SQL优化无法解决问题时，可以升高RDS的相关规格
3. 当CPU和IOPS超标时，可以通过设置只读库的方式来分担主库负荷。注意读写分离会影响读一致性。
4. 当以上步骤无法很好地解决问题时，请考虑进行PolarDB-X 1.0扩容。

磁盘空间

RDS的磁盘空间有以下几种：

1. 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。
2. 系统文件空间：包括共享表空间、错误日志文件等。
3. 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

PolarDB-X 1.0扩容流程分为**配置**>**迁移**>**切换**>**清理**四个步骤。

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PolarDB-X 1.0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PolarDB-X 1.0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。

6.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。

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&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。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>
```

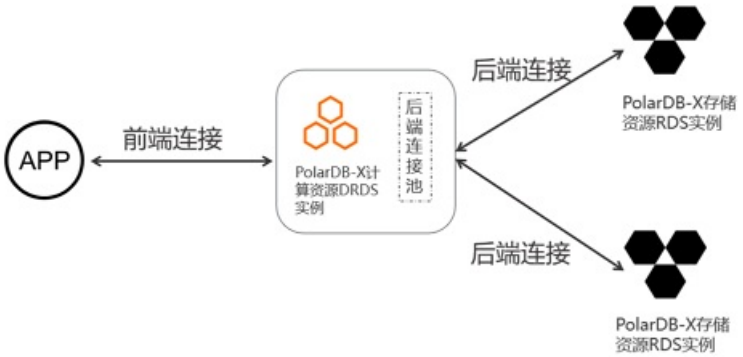
7.PolarDB-X 1.0实例中的连接

本文将介绍PolarDB-X 1.0实例中的前、后端连接的相关信息。

背景信息

当应用程序连接PolarDB-X 1.0实例执行操作时，从PolarDB-X 1.0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 1.0计算资源实例中逻辑库的连接。
- 后端连接：由PolarDB-X 1.0计算资源实例中的节点建立的，到后端PolarDB-X 1.0存储资源实例（即私有定制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 1.0计算资源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 1.0计算资源实例时，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（如数万个并发的长连接），因此可认为PolarDB-X 1.0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 1.0实例的场景。

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 1.0计算资源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 1.0实例处理请求的整体并发度是有限的。

后端连接

PolarDB-X 1.0计算资源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定制RDS实例中物理库的后端连接。因此，PolarDB-X 1.0实例中后端连接池的最大连接数与私有定制RDS实例支持的最大连接数直接相关。您可以参照以下公式来计算PolarDB-X 1.0实例中后端连接池的最大连接数。

```
PolarDB-X 1.0实例后端连接池的最大连接数 = 向下取整（私有定制RDS实例最大连接数 / 私有定制RDS实例物理分库数 / PolarDB-X 1.0实例节点数）
```

例如，某用户搭配购买了以下规格的私有定制RDS实例和PolarDB-X 1.0实例。

- 1个私有定制RDS实例，包含8个物理分库，规格为通用型4核16 GB，最大连接数为4000。
- 1个PolarDB-X 1.0计算资源实例，规格为标准版16核64 GB（每4核16 GB为1个节点，即该实例包含4个节点）。

按照上述公式可计算出PolarDB-X 1.0实例中后端连接池的最大连接数。

```
PolarDB-X 1.0实例后端连接池的最大连接数 = FLOOR(4000 / 8 / 4) = FLOOR(125) = 125
```

说明

- 上述公式计算的结果为PolarDB-X 1.0实例中后端连接池的最大连接数的上限。实际使用中，为了减轻私有定制RDS实例的连接压力需要留出一定的缓冲余地，PolarDB-X 1.0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。
- 建议每个PolarDB-X 1.0实例中的数据库都创建在独立的私有定制RDS实例上，而不要和其它用或其它PolarDB-X 1.0实例的数据库共享1个私有定制RDS实例。

前、后端连接的关系

在应用程序与PolarDB-X 1.0计算资源实例建立了前端连接并发出SQL语句的执行请求后，PolarDB-X 1.0计算资源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。

PolarDB-X 1.0计算资源实例节点内部为异步的处理流程，前端连接和后端连接之间并无绑定关系。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 1.0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 1.0计算资源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（原因见后端连接），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事务（亦称为大事务），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 1.0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 1.0计算资源实例或私有定制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。严重时可能还会因为执行超时导致SQL执行失败率上升。

说明 若在正常使用连接和执行查询的情况下，后端连接数仍然达到了PolarDB-X 1.0实例中后端连接池的最大连接数，请提交工单联系技术支持。

8.如何处理DDL异常

本文介绍如何处理使用PolarDB-X 1.0时出现的DDL异常情况。

DDL原理简介

PolarDB-X 1.0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：

- 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。

  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或磁盘空间不足等。

- 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。

  长时间无响应一般是由分库的执行时间过长导致的。以MySQL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Online 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。示例如下：

- 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（需要一点时间，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，可以通过以下步骤判定这是一个快速还是慢速的操作：

1. 复制表结构生成一张克隆表。
2. 插入一些数据。
3. 在克隆表上执行目标DDL操作。
4. 检查操作完成后 `rows affected` 值是否为0。非0的值表示该操作需要重建整张表，这时可能需要考虑在业务低峰期执行该操作。

PolarDB-X 1.0 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，PolarDB-X 1.0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。只需保证最终所有分表结构一致即可。

DDL执行失败处理步骤

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。此时执行步骤2，否则执行步骤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，否则继续执行步骤3。
3. 使用SHOW PROCESSLIST指令查看所有当前执行的SQL状态。如有仍在执行的DDL操作，请等候其执行完成后再执行步骤1、2，检查表结构是否符合预期，否则执行步骤4。
4. 在PolarDB-X 1.0上重新执行DDL操作。如果出现 `Lock conflict` 的报错信息请执行步骤5，否则执行

步骤3。

5. 使用RELEASE DBLOCK指令释放DDL操作锁，然后执行步骤4。

详细操作如下：

1. 检查表结构一致性。

   使用CHECK TABLE指令检查表结构，示例如下：

```
check table `xxxx`;
```

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

   若返回结果只有一行且显示状态OK时，表明表结构一致。示例如下：

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

2. 检查表结构。

   使用SHOW CREATE TABLE指令检查表结构，示例如下：

```
mysql> show create table `xxxx`;
```

   若表结构一致且表结构无误时，可认为DDL已执行成功，返回结果示例如下：

```
+---------+-------------------------------------------------------------------------------
-------------------------------------+
| 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)
```

3. 查看当前正在执行的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指令查看所有当前执行的SQL状态，示例如下：

```
mysql> SHOW PROCESSLIST WHERE COMMAND != 'Sleep';
```

   返回结果示例如下：

```
+---------------+-----------+--------------------+-------------+---------+-------------
-------------------------------------------------------------+---------------------------
-----------------------------------------------------------------------------+-----------+
---------------+-----------+
| 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`.`pr
ice`) from `t_i |      NULL |          NULL |      NULL |
| 0-0-352864311 | cowxhthg0 | NULL               | Binlog Dump |      13 | Master has s
ent 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。发现耗时较长的指令（如示例中ID为 `0-0-402714795` 的指令）后，您可使用**KILL '0-0-402714795'**命令来取消该指令。

> **说明** PolarDB-X 1.0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。您可以从SHOW PROCESSLIST结果集的INFO列判断该指令归属的逻辑SQL。

4. `Lock conflict` 报错处理。

   PolarDB-X 1.0执行DDL操作会加库级锁，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会出现以下报错：

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

   此时执行RELEASE DBLOCK命令释放该锁即可。指令取消及锁释放后，您可以选择在业务低谷或者停止期间，重新执行该DDL。

常见问题

- Q：为什么在DMS或其它客户端上无法显示修改后的表结构？

  A：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，PolarDB-X 1.0在您的0分库RDS里建立了一个影子库，影子库名与PolarDB-X 1.0逻辑库名一致，存储了逻辑库里所有的表结构等信息。

DMS会从影子库系统表中获取PolarDB-X 1.0的表结构。在处理异常DDL过程中，由于种种原因可能会出现库表结构已经修改，但是影子库中的表结构却未修改的现象。此时您需连接到影子库，在该库中重新对表进行一次DDL操作即可。

> **说明** CHECK TABLE不会检测影子库表结构与PolarDB-X 1.0逻辑库表结构是否一致。

- Q：执行DDL语句时出现错误码，该如何解决？

  A：关于PolarDB-X 1.0返回的常见错误码及解决方法，请参见错误代码。

9.如何高效扫描数据

PolarDB-X 1.0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。本文介绍如何高效扫描PolarDB-X 1.0数据。

常见的扫描场景

- 没有分库分表：PolarDB-X 1.0会把原SQL传递到后端MySQL执行。这种情况下PolarDB-X 1.0支持任何聚合函数。
- 非全表扫描：SQL经过PolarDB-X 1.0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和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。

通过HINT进行表遍历

1. 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)
```

> **说明**
> - 非分库分表的表默认存储在第0个分库。
> - 推荐每次扫描前执行SHOW TOPOLOGY FROM TABLE_NAME获取最新的表拓扑结构。

2. 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;
```

并行扫描

PolarDB-X 1.0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;
```

说明

- HINT中的TDDL5_00_GROUP与SHOW TOPOLOGY指令结果中的GROUP_NAME列相对应。
- 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。

10.在IN查询中如何选择Values个数

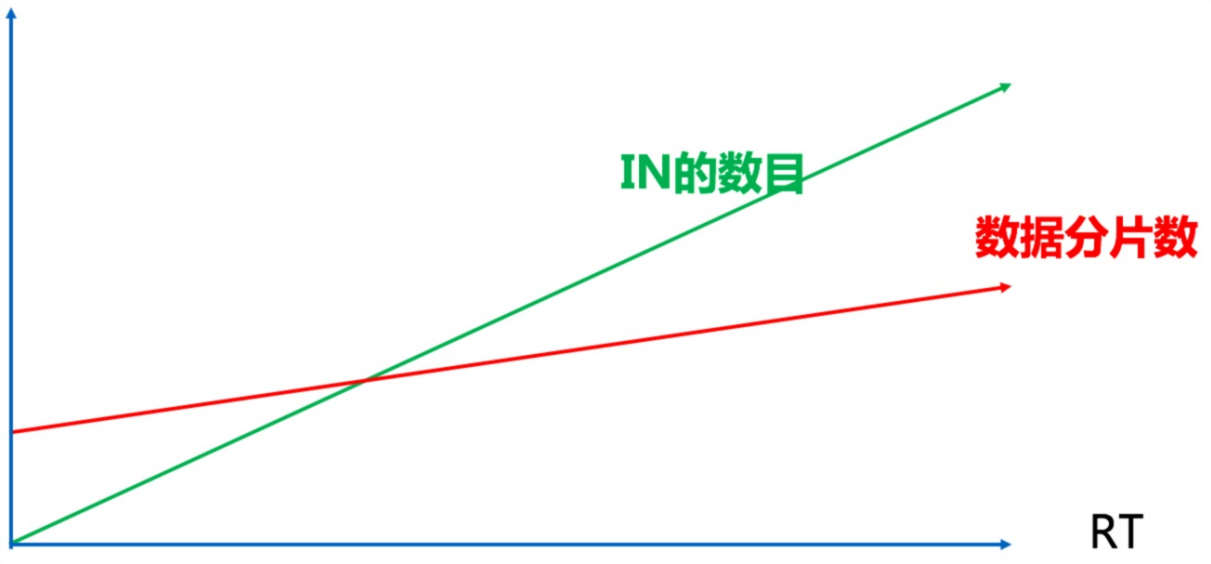
本文将介绍如何在PolarDB-X 1.0中做IN查询时，选择最佳的Values个数。

功能介绍

实际场景中经常需要根据一些常量指标做IN查询，其中IN的字段是分区键。例如在电商场景中，所有订单都会记录到订单表Order，此表按照订单ID进行拆分，一个买家经常会根据已购买的订单列表，查询这些订单的具体信息。假设用户已购买的订单数是2，那么会产生2个值的IN条件查询，理论上查询会路由到两个2分片。查询SQL示例：

```
SELECT * FROM ORDER WHERE ORDER_ID IN (id1,id2)
```

随着用户购买的订单数增加，查询订单信息的IN值数量也会增加，这样一次查询很可能会路由到所有的分片，导致RT变高。下图展示了IN值数量、扫描分片数和RT之间的关系。



为了尽可能避免随着IN值数量增加，导致物理SQL膨胀和扫描压力，PolarDB-X 1.0在内核版本5.4.8-16069335（包含）之后引入了基于IN值做动态分区裁剪的能力。

继续以上述场景为例假设Order表分片数量是128，IN查询的数量128个，那么一次查询的SQL为：

```
SELECT * FROM ORDER WHERE ORDER_ID IN (id1,id2,id3....id128)
```

如果ID足够离散，可能会分散到所有的分片，需要查询最多128个分片，每个分片的物理查询没有做IN值的裁剪，每个物理查询都会携带128个IN值条件下推给MySQL，过多的IN条件也会加大MySQL执行压力。查询示例如下：

```
SELECT * FROM ORDER WHERE ORDER_ID_1 IN (id1,id2,id3....id128);
SELECT * FROM ORDER WHERE ORDER_ID_2 IN (id1,id2,id3....id128);
SELECT * FROM ORDER WHERE ORDER_ID_3 IN (id1,id2,id3....id128);
.....
SELECT * FROM ORDER WHERE ORDER_ID_128 IN (id1,id2,id3....id128);
```

在支持IN分区裁剪的版本上，首先计算层会根据条件计算分片，引入IN值的动态分片裁剪，下发给MySQL的物理查询上就会只包含属于该分片的ID条件，裁剪掉了多余的IN值条件，因此IN查询的RT和吞吐都会有一定的提升。查询示例如下：

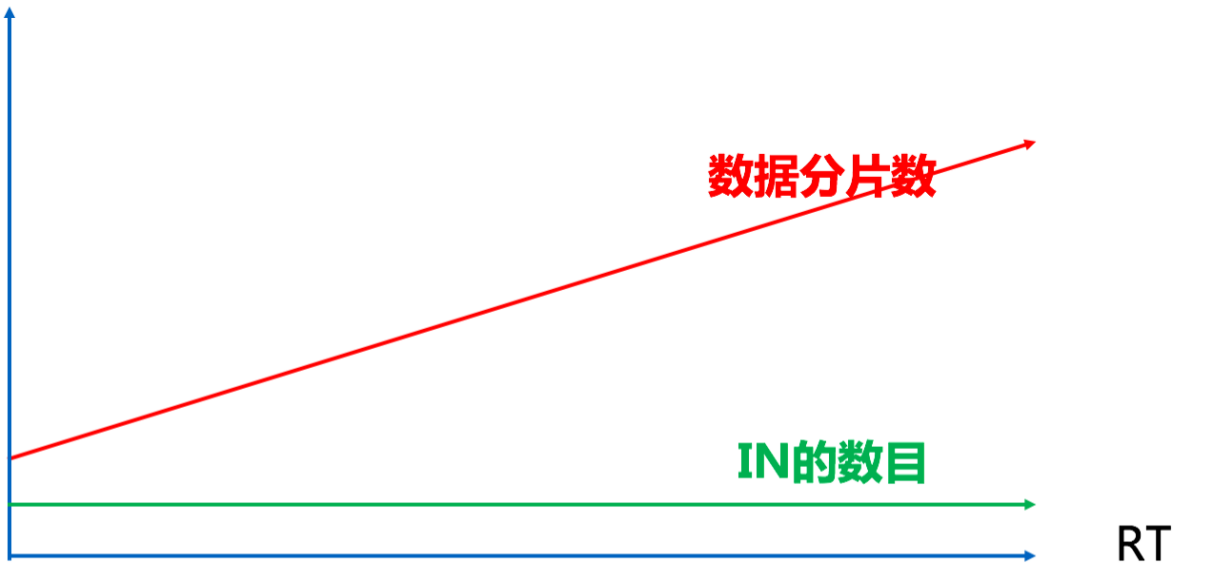
```
SELECT * FROM ORDER WHERE ORDER_ID_1 IN (id1);
SELECT * FROM ORDER WHERE ORDER_ID_2 IN (id2,id12);
SELECT * FROM ORDER WHERE ORDER_ID_3 IN (id3,id4,id5);
.....
SELECT * FROM ORDER WHERE ORDER_ID_32 IN (id100....id128);
```

另外，PolarDB-X 1.0内部针对跨分片的查询会有一个Parallel Query执行，例如涉及32个分片，针对每个用户查询，会有节点CPU内核数大小的并发度，例如分布式下单个节点规格为16core时，默认并发数就是16个，即32个分片会分成2批才能执行完成。

结合以往经验，在IN查询的业务中，阿里云建议：

- IN的值的数目远小于分片数，这样可以避免每次都做全分片查询；
- IN的值的数目不会随着业务的发展而增长，这样可以避免随着业务变化而导致性能下降；
- 兼顾RT和吞吐的话，建议IN的值的数量在8~32之间。

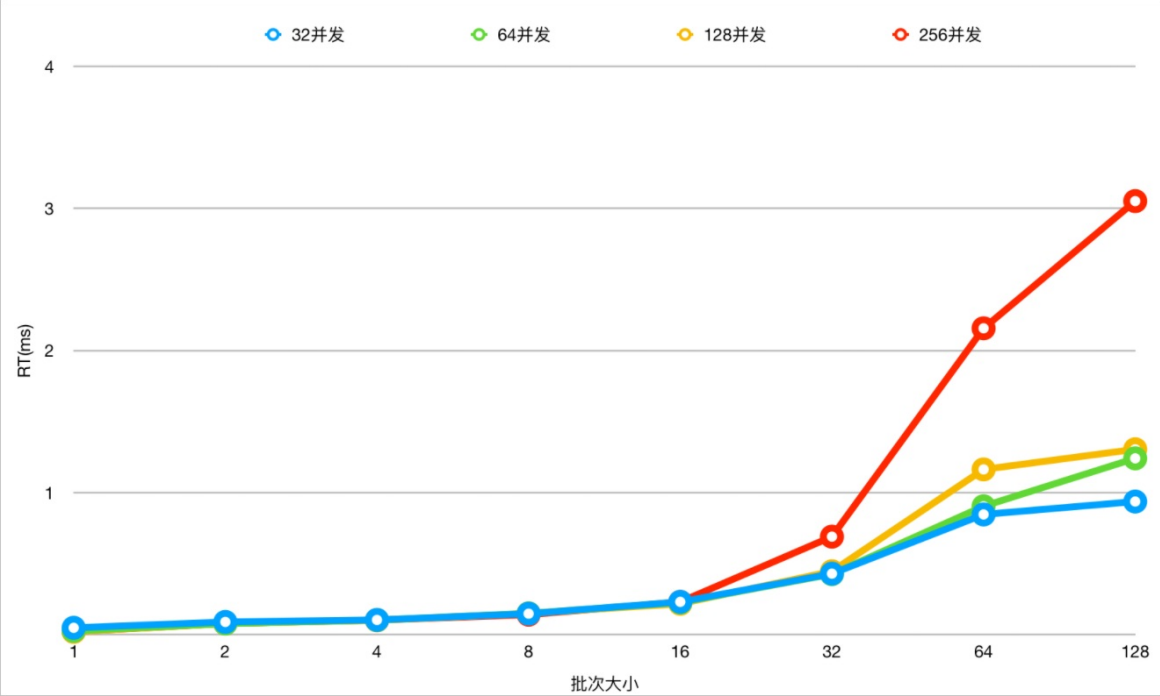
满足上述最佳经验后，涉及到IN查询的业务可以做到面向并发场景下的线性扩展，而RT也不会有明显抖动。线性扩展举例：例如分布式16core能跑1万个IN并发，扩展到32core之后就能跑2万个并发。



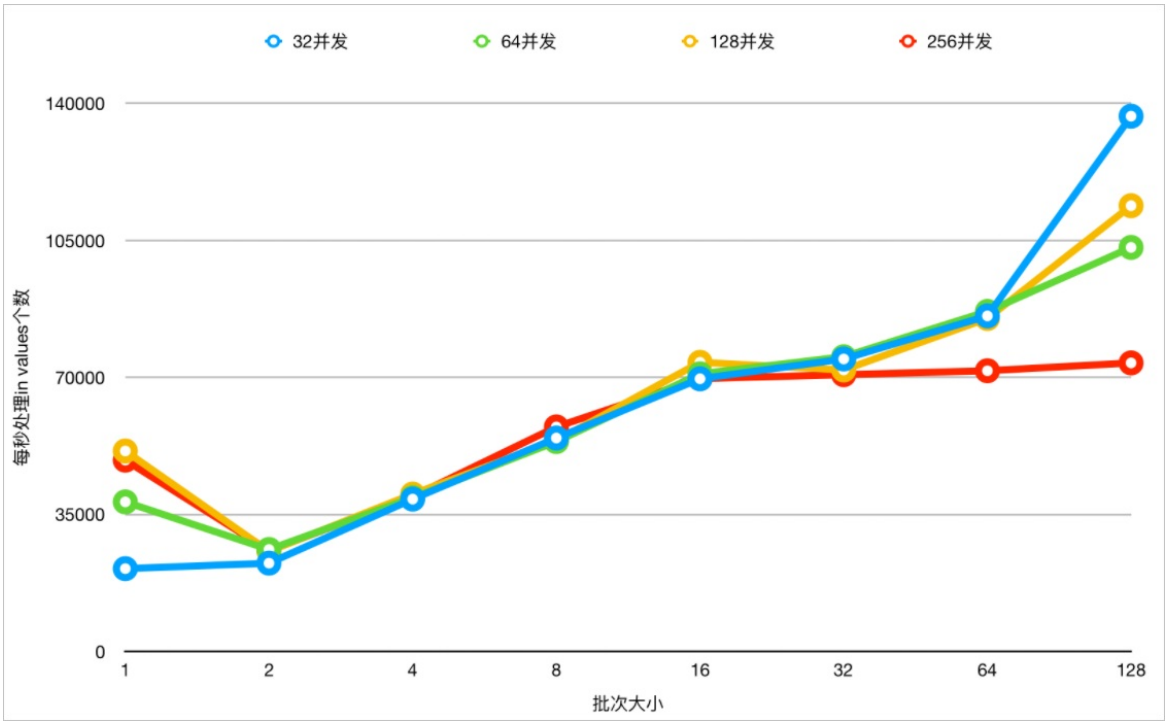
比对测试

在兼顾RT和吞吐的场景下，确定合理的IN查询的值的数量。在规格2×16C64G的节点，针对一张分表数量为64，分表记录数为百万级别的表在不同值数量、不同并发下做测试。在内核版本5.4.8-16069335（包含）之后针对IN查询进一步完善了动态裁剪分表的能力，下发的物理SQL也会裁剪掉多余的Values，下面是比对测试的结果。

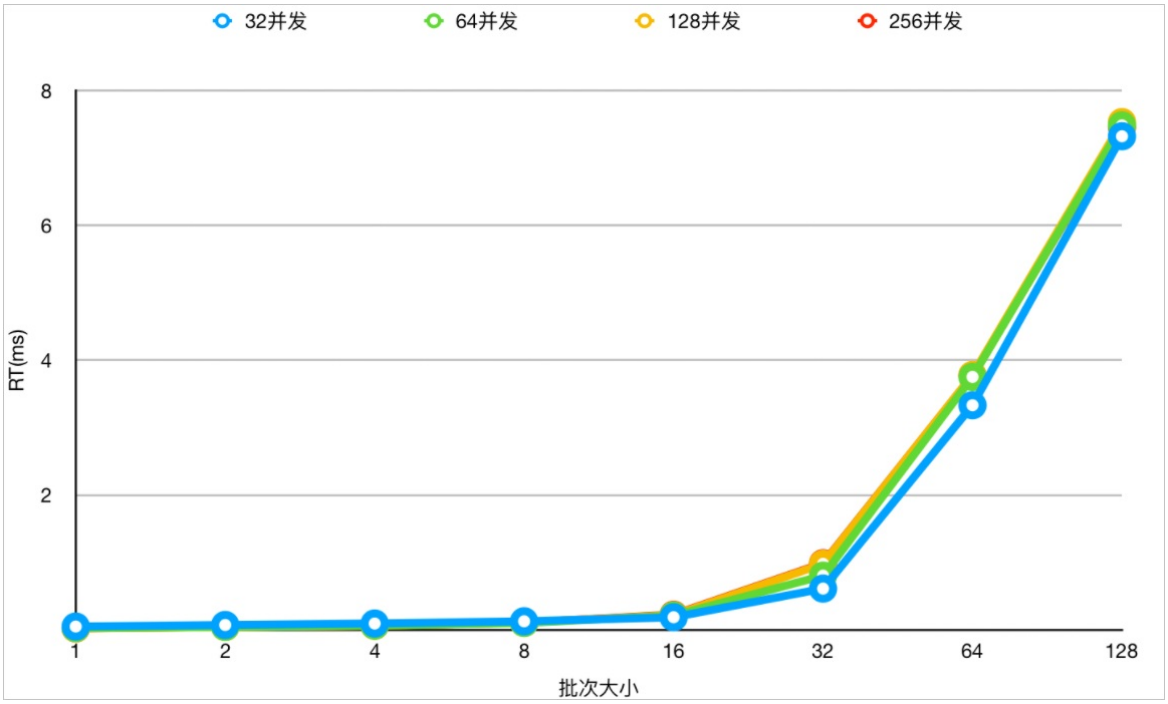
1. 在不同并发下，不同Values值数量下测试，开启IN查询动态裁剪能力下，查看RT变化。



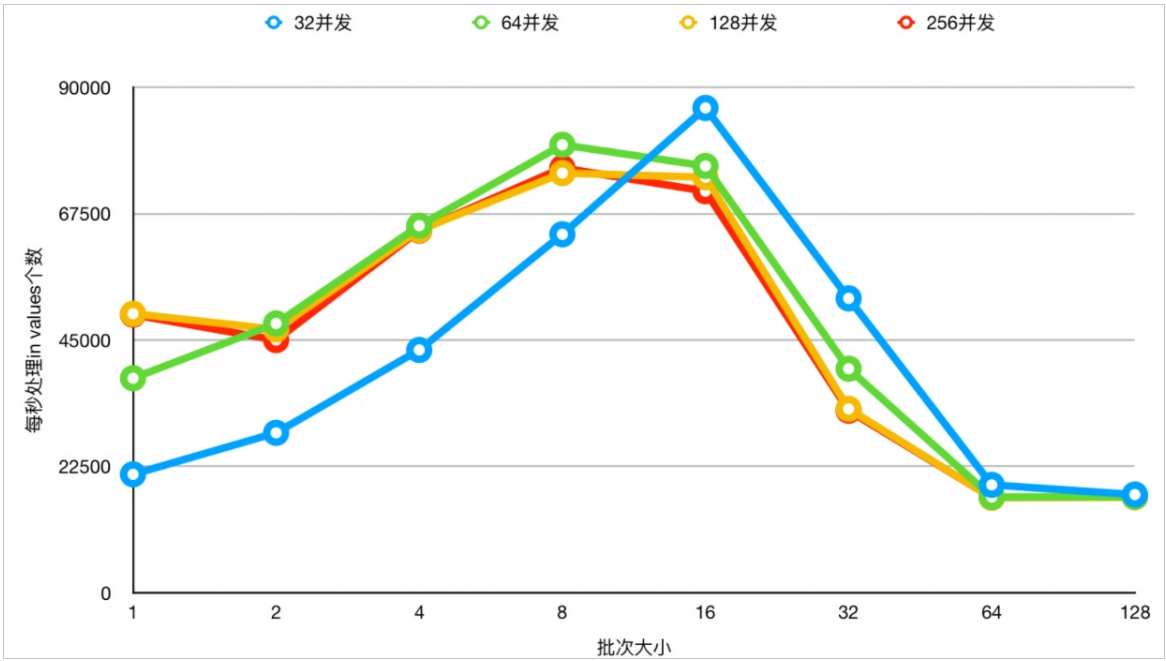
2. 在不同并发下，不同Values值数量下测试，开启IN查询动态裁剪能力下，查看吞吐变化。



3. 在不同并发下，不同Values值数量下测试，关闭IN查询动态裁剪能力下，查看RT变化。



4. 在不同并发下，不同Values值数量下测试，关闭IN查询动态裁剪能力下，查看吞吐变化。



通过测试对比，可以得到以下结论：

- 兼顾RT和吞吐时，建议IN的值的数量在8~32之间，基本对齐分布式Parallel Query的默认并发度（单节点的CPU内核数）。
- 在内核版本5.4.8-16069335（包含）之后，在开启IN查询的动态裁剪能力下，吞吐和RT都有明显的优势，推荐您将内核版本升级至5.4.8及以上版本。