ALIBABA CLOUD

阿里云

云原生数据仓库AnalyticDB MySQL版

云原生数据仓库AnalyticDB MySQL版公共云合集

文档版本：20220630

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.快速入门

1.1. AnalyticDB MySQL使用流程

欢迎使用云原生数据仓库AnalyticDB MySQL版入门指南。云原生数据仓库AnalyticDB MySQL版（简称ADB，原分析型数据库MySQL版）是云端托管的PB级高并发实时数据仓库，是专注于服务OLAP领域的数据仓库。本指南将指引您完成一次AnalyticDB MySQL版集群创建及使用。

如果您是首次使用AnalyticDB MySQL版的用户，我们建议您先阅读以下部分：

- 产品简介：本内容概述了AnalyticDB MySQL版的产品概念、产品优势及应用场景等内容。
- 产品定价：本内容介绍了AnalyticDB MySQL版的产品定价、计费方式等信息。
- 入门指南（本指南）：本指南提供了有关使用AnalyticDB MySQL版创建示例集群并使用示例数据的教程。

在本教程中，操作流程概览如下：

1. 准备阿里云账号
2. 创建集群
3. 创建数据库账号
4. 设置白名单
5. 连接集群
6. 创建数据库
7. 导入数据并查询

完成本教程后，您可以查通过用户指南查看了解后续步骤等更多信息，详情请参见了解更多。

1.2. 准备阿里云账号

使用云原生数据仓库AnalyticDB MySQL版前，您需要创建一个阿里云账号。本文为您介绍如何创建阿里云账号。

背景信息

云原生数据仓库AnalyticDB MySQL版的账号登录体系与阿里云保持一致，统一采用RAM主子账号登录的方式。

- 阿里云账号（即主账号）是阿里云资源的归属及使用计量计费的基本主体，负责生成本企业组织下的子账号，并对子账号进行管理、授权等操作。
- RAM子账号从属于阿里云主账号，由主账号在RAM系统中创建并进行管理。这些RAM子账号本身不拥有实际的任何资源，也没有独立的计量计费，由所属主账号统一控制和付费。

注册阿里云账号

如果您还没有阿里云账号，请单击立即注册，进入阿里云账号注册页面完成阿里云账号注册。

阿里云账号实名认证

登录阿里云账号，单击实名认证，按步骤填写信息完成账号实名认证。

> **说明**
> - 阿里云账号需要进行实名制认证后，才能购买和使用阿里云上的各种产品。为保证后续操作顺利进行，请务必完成实名认证操作。
> - 如果您是企业级用户，建议进行企业级认证，以获取更多的便利。

1.3. 创建集群

本文介绍如何在云原生数据仓库AnalyticDB MySQL版（简称ADB，原分析型数据库MySQL版）控制台上创建集群。

前提条件

- 已注册阿里云账号。具体操作请参见注册阿里云账号。
- 若您要创建按量付费的集群，请确保您的阿里云账号的余额大于等于100元。

创建数仓版（3.0）集群

1. 登录云原生数据仓库AnalyticDB MySQL控制台。
2. 在页面左上角，选择集群所在地域。
3. 在左侧导航栏，单击**集群列表**。
4. 在**数仓版（3.0）**页签中，单击右上角**创建集群**。
5. 选择**商品类型**。
   - **按量付费**：属于后付费，即按小时扣费。适合短期需求，用完可以立即释放集群，节省费用。
   - **包年包月**：属于预付费，即在新建集群时需要支付费用。适合长期需求，价格比按量付费更实惠，且购买时长越长，折扣越多。
6. 设置参数后，单击右下角**立即购买**。

参数	说明
版本	选择**数仓版（3.0）**。
地域	集群所在的地理位置，购买后无法更换地域。建议选择离业务最近的地域，以便于提升集群访问速度。
可用区	可用区是地域中的一个独立物理区域，不同可用区之间没有实质性区别。
网络类型	固定为**专有网络**，无需选择。 专有网络，也称为VPC（Virtual Private Cloud）。VPC是一种隔离的网络环境，安全性较高。

参数	说明
专有网络（VPC） **专有网络交换机**	请确保AnalyticDB MySQL与需要连接的ECS创建于同一个VPC，否则它们无法通过内网互通，无法发挥最佳性能。 ○ 如果已创建符合您网络规划的VPC，直接选择该VPC。例如，如果您已创建ECS，且该ECS所在的VPC符合您的规划，那么选择该VPC。 ○ 如果您未创建符合您网络规划的VPC，您可以使用默认VPC和交换机。更多详情，请参见默认专有网络和交换机。 ○ 如果默认VPC和交换机无法满足您的要求，您可以自行创建VPC和交换机。创建方法，请参见创建和管理专有网络。
模式	选择AnalyticDB MySQL产品的模式，支持**弹性模式**和**预留模式**两种模式。关于两种模式的区别，请参见常见问题。
系列	选择AnalyticDB MySQL的产品系列，支持如下系列： ○ 若选择了**弹性模式**，则**系列**固定为**集群版（新版）**，无需选择。 **弹性模式集群版（新版）**通过存储分离架构，提供存储与计算资源独立、自由弹性伸缩、多用户隔离、存储冷热分离、异构化存储格式以及用户可编程的能力，可以通过较低的资源成本获得更高性能的灵活化实时分析体验。 ○ 若选择了**预留模式**，则**系列**固定为**集群版**，无需选择。 **预留模式集群版**采用了三副本高可用架构，提供金融级别的数据可靠性保证。支持节点和存储空间秒级弹性扩展，可承载更大吞吐数据实时写入和读取能力。 **说明** 不同地域支持的产品系列不同，请以实际界面为准。各个系列的详细介绍，请参见产品系列。
计算资源	选择AnalyticDB MySQL的计算资源。计算资源用于数据计算，增加计算资源可以提高数据查询速度。 **说明** 仅当**模式**为**弹性模式**时，支持该配置。
弹性IO资源	创建集群时，AnalyticDB MySQL将自动按照您购买的计算资源配备免费的存储IO资源。若需要单独扩容存储资源，选择需要增加的**弹性IO资源**数量即可。 **弹性IO资源**（Elastic IO Unit，简称EIU）是AnalyticDB MySQL弹性模式集群版（新版）衡量实例存储性能的元单位，可用于单独扩容存储资源。更多详情，请参见EIU详解。 **说明** 仅当**模式**为**弹性模式**，且**计算资源**为32核128 GB或以上规格时，支持该配置。

参数	说明
热数据存储空间说明 **冷数据存储空间说明**	均按实际数据存储量计费，创建集群时无需预付费购买存储空间。其中： ○ 热数据指访问频次较高的数据，采用ESSD云盘存储，IO性能好，能够满足高性能访问的需求。 ○ 冷数据指访问频次较低的数据，采用HDD介质存储，性价比高。 更多详情，请参见数据存储冷热分离。 **说明** 仅当**模式**为**弹性模式**时，支持该配置。
规格	选择产品规格。规格详情，请参见产品系列。 **说明** 仅当**模式**为**预留模式**时，支持该配置。
节点组数量	选择节点组的数量。每个节点组默认包含三个节点（副本）。 **说明** 仅当**模式**为**预留模式**时，支持该配置。
存储空间	选择单个节点组的存储空间大小。若购买了多个节点组，实际存储总空间=单个节点组的存储空间×节点组数量。 **说明** 仅当**模式**为**预留模式**时，支持该配置。
购买时长	选择集群的购买时长。 **说明** 仅当**商品类型**为**包年包月**时，支持该配置。

7. 根据您选择的**商品类型**，完成后续购买操作。
   - **包年包月**
     a. 在**确认订单**页面确认订单信息，阅读并选中服务协议。单击**去支付**。
     b. 在**支付**页面，确认未支付订单信息和支付方式，单击**订购**。
   - **按量付费**

     在**确认订单**页面确认订单信息，阅读并选中服务协议，单击**立即开通**即可。

> **说明** 支付成功后，需要约20分钟创建集群，之后您就可以在**数仓版（3.0）**页签中看到新创建的集群。

创建湖仓版（3.0）集群

1. 登录云原生数据仓库AnalyticDB MySQL控制台。
2. 在页面左上角，选择集群所在地域。

3. 在左侧导航栏，单击**集群列表**。
4. 在**湖仓版（3.0）**页签中，单击右上角**创建集群**。
5. 选择**商品类型**。
   - **按量付费**：属于后付费，即按小时扣费。适合短期需求，用完可以立即释放集群，节省费用。
   - **包年包月**：属于预付费，即在新建集群时需要支付费用。适合长期需求，价格比按量付费更实惠，且购买时长越长，折扣越多。
6. 设置参数后，单击右下角**立即购买**。

参数	说明
版本	选择**湖仓版（3.0）**。
地域	集群所在的地理位置，购买后无法更换地域。建议选择离业务最近的地域，以便于提升集群访问速度。
可用区	可用区是地域中的一个独立物理区域，不同可用区之间没有实质性区别。
网络类型	固定为**专有网络**，无需选择。 专有网络，也称为VPC（Virtual Private Cloud）。VPC是一种隔离的网络环境，安全性较高。
专有网络（VPC） **专有网络交换机**	请确保AnalyticDB MySQL与需要连接的ECS创建于同一个VPC，否则它们无法通过内网互通，无法发挥最佳性能。 ○ 如果已创建符合您网络规划的VPC，直接选择该VPC。例如，如果您已创建ECS，且该ECS所在的VPC符合您的规划，那么选择该VPC。 ○ 如果您未创建符合您网络规划的VPC，您可以使用默认VPC和交换机。更多详情，请参见默认专有网络和交换机。 ○ 如果默认VPC和交换机无法满足您的要求，您可以自行创建VPC和交换机。创建方法，请参见创建和管理专有网络。
计算预留资源	计算预留资源用于数据计算，可分配给Interactive型和Job型资源组。增加计算资源可以提高数据查询速度。 计算预留资源的取值范围为0 ACU~512 ACU，步长为16 ACU。 ○ 当计算预留资源为0时，无法创建存储预留资源，且后续仅能创建Job型资源组。 ○ 当计算预留资源大于0时，支持创建Interactive型和Job型资源组。
默认分配行为	计算预留资源是否全部分配给default资源组。
存储预留资源	存储预留资源用于数据存储，能带动的热数据空间为4 TB。 存储预留资源的取值范围为0 ACU~4800 ACU，步长为24 ACU。 **说明** 仅当**计算预留资源**不为0时，支持该配置。

7. 根据您选择的**商品类型**，完成后续购买操作。

- **包年包月**
  a. 在**确认订单**页面确认订单信息，阅读并选中服务协议。单击**去支付**。
  b. 在**支付**页面，确认未支付订单信息和支付方式，单击**订购**。
- **按量付费**

  在**确认订单**页面确认订单信息，阅读并选中服务协议，单击**立即开通**即可。

> **说明** 支付成功后，需要约20分钟创建集群，之后您就可以在**湖仓版（3.0）**页签中看到新创建的集群。

1.4. 创建数据库账号

本文介绍如何创建云原生数据仓库AnalyticDB MySQL版数据库账号以及高权限账号与普通账号的区别。

数据库账号类型

AnalyticDB MySQL版支持高权限账号和普通账号这两种数据库账号，两种账号的区别见下表。

数据库账号类型	说明
高权限账号	• 只能通过控制台创建和管理高权限账号。 • 一个集群中只能创建一个高权限账号，高权限账号可以管理所有普通账号和数据库。 • 使用高权限账号可以断开任意普通账号的连接。 • 开放了更多权限，可满足个性化和精细化的权限管理需求，例如可按用户分配不同表的查询权限等。 • AnalyticDB MySQL版中的高权限账号相当于MySQL中的root账号。
普通账号	• 只能通过SQL语句进行创建，创建方式，请参见CREATE USER。 • 一个集群最多可以创建256个普通账号。 • 需要手动为普通账号授予指定数据库的权限，详情请参见GRANT和权限模型。 • 普通账号不能断开其他普通账号的数据库连接。

创建高权限账号

1. 登录云原生数据仓库AnalyticDB MySQL控制台。
2. 在页面左上角，选择集群所在地域。
3. 在左侧导航栏，单击**集群列表**。
4. 根据您的集群类型，选择**数仓版（3.0）**。
5. 单击目标集群ID。
6. 在左侧导航栏单击**账号管理**。
7. 在**账号管理**页面右上角，单击**创建高权限账号**。
8. 在**创建高权限账号**面板，设置相关参数。

参数	说明
数据库账号	高权限账号的账号名称。名称需符合如下要求： ○ 长度为2~16个字符。 ○ 以小写字母开头，小写字母或数字结尾。 ○ 可包含小写字母、数字以及下划线（_）。
账号类型	固定为**高权限账号**，无需配置。
密码	高权限账号的密码，密码需符合如下要求： ○ 长度为8~32个字符。 ○ 至少包含大写字母、小写字母、数字或特殊字符中的任意三种。 ○ 特殊字符为：`!@#$%^&*()_+-=` 。
确认密码	再次输入高权限账号的密码。
备注说明	备注该账号的相关信息，便于后续账号管理。可选。

9. 单击**确定**即可。

创建普通账号

创建及授权普通账号，请参见CREATE USER、GRANT。

1.5. 设置白名单

创建云原生数据仓库AnalyticDB MySQL版集群后，您需要为集群设置白名单，以允许外部设备访问该集群。

背景信息

- 集群默认的白名单只包含IP地址127.0.0.1，表示任何设备均无法访问该集群。您可以通过设置白名单允许其他设备访问集群，例如填写IP段10.10.10.0/24，表示10.10.10.X的IP地址都可以访问该集群。若您需要添加多个IP地址或IP段，请用英文逗号（,）隔开（逗号前后都不能有空格），例如192.168.0.1,172.16.213.9。

  > **警告** 设置白名单时，禁止输入IP：0.0.0.0。

- 若您的公网IP经常变动，需要开放所有公网IP访问AnalyticDB MySQL集群，请提交工单联系技术支持。
- 白名单可以让AnalyticDB MySQL集群得到高级别的访问安全保护，建议您定期维护白名单。
- 设置白名单不会影响AnalyticDB MySQL集群的正常运行。设置白名单后，新的白名单将于1分钟后生效。

数仓版（3.0）设置白名单

1. 登录云原生数据仓库AnalyticDB MySQL控制台。
2. 在页面左上角，选择集群所在地域。
3. 在左侧导航栏，单击**集群列表**。
4. 在**数仓版（3.0）**页签中，单击目标**集群ID**。
5. 在左侧导航栏单击**数据安全**。

6. 在**白名单设置**页面，单击**default**白名单分组右侧的**修改**。

   > **说明** 您也可以单击**创建白名单分组**创建自定义分组。

7. 在**修改白名单分组**对话框中，删除默认IP 127.0.0.1，填写需要访问该集群的IP地址或IP段，然后单击**确定**。

1.6. 连接集群

云原生数据仓库AnalyticDB MySQL版支持通过DMS（Data Management Service）、MySQL客户端（Navicat for MySQL、DBeaver、DBVisualizer、SQL WorkBench/J）、BI可视化工具、或者MySQL命令行工具连接AnalyticDB MySQL版集群。您也可以在应用程序中通过配置集群连接地址、端口、数据库账号等信息连接AnalyticDB MySQL版集群。

背景信息

DMS是阿里云提供的图形化数据管理工具，可用于管理关系型数据库和NoSQL数据库，支持数据管理、SQL操作、数据方案（数据导入/导出、数据库克隆等）、性能与优化、安全审计等功能。

使用DMS连接云原生数据仓库AnalyticDB MySQL版

1. 登录云原生数据仓库AnalyticDB MySQL控制台。
2. 在页面左上角，选择集群所在地域。
3. 在左侧导航栏，单击**集群列表**。
4. 在**数仓版（3.0）**页签下，单击目标**集群ID**。
5. 在**集群信息**页面，单击右上角**登录数据库**。
6. 在弹出的对话框中，填写登录信息。

参数	说明
数据库类型	默认为**ADB3.0-MySQL**，无需选择。
实例地区	默认为当前实例所在地域，无需选择。 **说明** 若您需要登录其他地域下的AnalyticDB MySQL集群，从下拉列表中选择目标集群的所在地域即可。
实例ID	默认为当前集群的集群ID，无需选择。 **说明** 若您需要登录其他AnalyticDB MySQL集群，从下拉列表中选择目标集群ID即可。
数据库账号	集群的账号名称。

参数	说明
数据库密码	账号名对应的密码。 **说明** 您可以选中**记住密码**，方便之后再次登录当前AnalyticDB MySQL集群时，无需输入数据库账号和密码即可自动登录。

说明

- 首次通过DMS登录AnalyticDB MySQL集群时，管控模式默认为**自由操作**。登录成功后，您还可以通过编辑实例功能来修改管控模式。更多信息，请参见编辑实例和管控模式。
- 配置完登录参数后，您可以单击左下角**测试连接**。如果测试连接失败，请按照报错提示检查录入的集群信息，如账号或密码是否正确。
- 系统会自动尝试往云数据库的白名单中添加DMS的服务器访问地址，若自动添加失败请手动添加。详情信息，请参见设置白名单和DMS白名单列表。

7. 单击**登录**即可。

应用开发中通过代码连接到云原生数据仓库AnalyticDB MySQL版

- Java
- Druid连接池配置
- Python
- PHP
- C#（Mac）
- Golang

通过MySQL命令行工具连接到云原生数据仓库AnalyticDB MySQL版

MySQL命令行连接AnalyticDB for MySQL

通过客户端连接到云原生数据仓库AnalyticDB MySQL版

- DBeaver
- DBVisualizer
- Navicat
- SQL WorkBench/J

将云原生数据仓库AnalyticDB MySQL版连接到数据可视化工具

- FineBI
- Quick BI
- 永洪BI
- DataV
- Tableau
- QlikView

- FineReport
- Smartbi

1.7. 创建数据库

您可以使用DMS（Data Management Service）、客户端（Navicat for MySQL、DBeaver、DBVisualizer、SQL WorkBench/J）、业务系统中的程序代码或者MySQL命令行工具连接云原生数据仓库AnalyticDB MySQL版集群，然后通过CREATE DATABASE语句创建数据库。

背景信息

本文以DMS为例，介绍如何在云原生数据仓库AnalyticDB MySQL中创建数据库。

> **说明** 每个集群最多可以创建256个数据库。

操作步骤

1. 在**SQL INFORMATION_SCHEMA**页签下，在SQL Console中输入CREATE DATABASE语句创建数据库。

   - 语法：`CREATE DATABASE [IF NOT EXISTS] $db_name`
   - 参数说明：`db_name`：数据库名。以小写字符开头，可包含字母、数字以及下划线（_），但不能包含连续两个及以上的下划线（_），长度不超过64个字符。

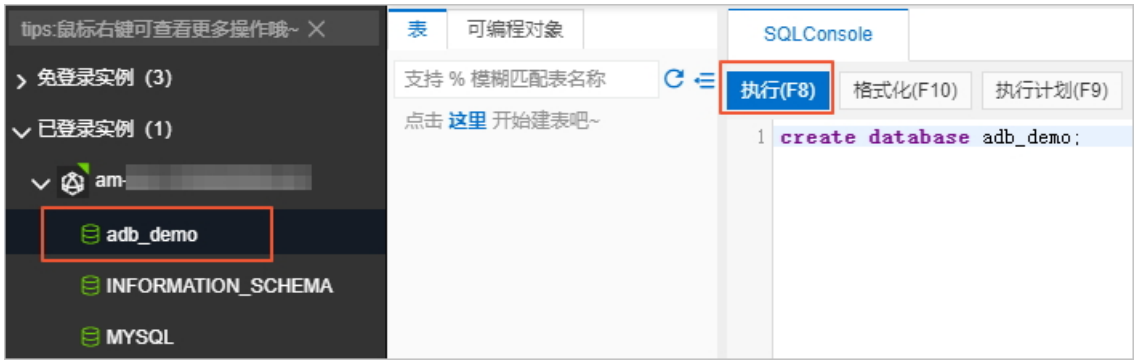
     > **说明** 数据库名不能是analyticdb，analyticdb是内置数据库。

   - 示例：

     ```
     create database adb_demo;
     ```

     ```
     create database if not exists adb_demo2;
     ```

2. 单击左上角的**执行**，数据库创建成功。



1.8. 导入数据并查询

AnalyticDB for MySQL提供多种数据同步方案，可满足不同场景下的数据同步需求。本文以数据文件存储在OSS中为例，介绍如何将OSS中的数据文件导入AnalyticDB for MySQL的 `adb_demo` 数据库中并进行查询。更多导入数据方式请参见支持的数据源。

前提条件

- 通过以下步骤在对象存储（Object Storage Service，简称OSS）中创建存储AnalyticDB for MySQL数据的目录。
  i. 开通OSS服务

  > 说明 OSS与AnalyticDB for MySQL所属Region相同。

  ii. 创建存储空间
  iii. 创建目录
  iv. 上传测试数据文件

  本示例将 `oss_import_test_data.txt` 文件上传至OSS中的 `bucket-name.oss-cn-hangzhou.aliyuncs.com/adb/` 目录，数据行分隔符为换行符，列分隔符为 `;` ，文件示例数据如下所示。

```
number;note
0001;hello_world_1
0002;hello_world_2
0003;hello_world_3
0004;hello_world_4
0005;hello_world_5
0006;hello_world_6
...
```

- 根据AnalyticDB MySQL入门指南，完成创建集群、设置白名单、创建账号和数据库等准备工作。

操作步骤

1. 通过**CREATE TABLE**，在 `adb_demo` 数据库中创建外表。创建CSV、Parquet或TEXT格式OSS外表的建表语法请参见创建OSS外表语法。
2. 查询OSS数据。

   查询外表映射表和查询AnalyticDB for MySQL内表语法没有区别，您可以方便地直接进行查询，如本步骤的示例代码所示。

   ```
   select uid, other from oss_parquet_external_table where uid < 100 limit 10
   ```

   - 对于数据量较大的CSV/TEXT数据文件，强烈建议您按照后续步骤导入AnalyticDB for MySQL后再做查询，否则查询性能可能会较差。
   - 对于Parquet格式数据文件，直接查询的性能一般也比较高，您可以根据需要决定是否进一步导入到AnalyticDB for MySQL后再做查询。
3. 通过CREATE TABLE，在 `adb_demo` 数据库中创建目标表 `adb_oss_import_test` 存储从OSS中导入的数据。

   ```
   CREATE TABLE IF NOT EXISTS adb_oss_import_test
   (
       uid string,
       other string
   )
   DISTRIBUTED BY HASH(uid)
   ```

4. 执行INSERT语句将OSS数据导入AnalyticDB for MySQL。
   - 执行 `INSERT INTO` 导入数据。

     ```
     insert into adb_oss_import_test
     select * from oss_import_test_external_table
     ```

   - 执行 `INSERT OVERWRITE INTO` 导入数据。

     ```
     insert overwrite into adb_oss_import_test
     select * from oss_import_test_external_table
     ```

   - 异步执行 `INSERT OVERWRITE INTO` 导入数据。

     ```
     submit job insert overwrite into adb_oss_import_test
     select * from oss_import_test_external_table ;
     +---------------------------------------+
     | job_id                                |
     +---------------------------------------+
     | 2020112122202917203100908203303000715 |
     ```

     关于异步提交任务详情请参见异步提交导入导出任务。

创建OSS外表语法

创建OSS CSV格式外表

示例的 `oss_import_test_data.txt` 文件为CSV格式，本节介绍CSV格式的OSS外表创建语法。

```
CREATE TABLE IF NOT EXISTS oss_import_test_external_table
(
    uid string,
    other string
)
ENGINE='OSS'
TABLE_PROPERTIES='{
    "endpoint":"oss-cn-hangzhou-internal.aliyuncs.com",
    "url":"oss://$bucketname/adb/oss_import_test_data.txt",
    "accessid":"LTAIF****5FsE",
    "accesskey":"Ccw****iWjv",
    "delimiter":";",
    "skip_header_line_count":1
}'
```

参数	说明
ENGINE='OSS'	表示该表是外部表，使用的存储引擎是OSS。
TABLE_PROPERTIES	用于告知AnalyticDB for MySQL如何访问OSS中的数据。
endpoint	OSS的 `EndPoint（域名节点）` 。 **说明** 目前仅支持AnalyticDB for MySQL通过OSS中ECS的VPC网络（内网）访问OSS。 登录OSS控制台，单击目标Bucket，在Bucket**概览**页面查看 `endpoint` 。
url	OSS中目标数据文件夹或文件的绝对地址。 **示例：** oss://$Bucket名称/adb/oss_import_test_data.txt
accessid	您在访问OSS中的文件时所持有的AccessKey ID。 如何获取您的accessid和accesskey，请参见获取账号的AK信息。
accesskey	您在访问OSS中的文件时所持有的Access Key Secret。
delimiter	定义CSV数据文件的列分隔符。

参数	说明

ossnull（可选）	标识 `NULL` 值，包含以下四种取值。
	• 默认值为1：EMPTY_SEPARATORS
	`a,"",,c --> "a","",NULL,"c"`
	• EMPTY_QUOTES
	`a,"",,c --> "a",NULL,"","c"`
	• BOTH
	`a,"",,c --> "a",NULL,NULL,"c"`
	• NEITHER
	`a,"",,c --> "a","","","c"`

skip_header_line_count（可选）	CSV文件第一行为表头，导入数据时设置为1可自动跳过第一行。或设置为其它值表示开头跳过的行数。默认为0，也就是不跳过。

创建OSS Parquet格式外表

```
CREATE TABLE IF NOT EXISTS oss_parquet_external_table
(
    uid string,
    other string
)
ENGINE='OSS'
TABLE_PROPERTIES='{
    "endpoint":"oss-cn-hangzhou-internal.aliyuncs.com",
    "url":"oss://****",
    "accessid":"LTAIF****5FsE",
    "accesskey":"Ccw****iWjv",
    "format":"parquet"
}'
```

参数	说明
ENGINE='OSS'	表示该表是外部表，使用的存储引擎是OSS。
TABLE_PROPERTIES	用于告知AnalyticDB for MySQL如何访问OSS中的数据。
endpoint	OSS的 `EndPoint（域名节点）` 。 **说明** 目前仅支持AnalyticDB for MySQL通过OSS中ECS的VPC网络（内网）访问OSS。 登录OSS控制台，单击目标Bucket，在Bucket**概览**页面查看 `endpoint` 。

参数	说明
url	OSS中目标数据文件或文件夹的绝对地址。 **示例：** oss://$Bucket名称/adb/oss_import_test_data.txt
accessid	您在访问OSS中的文件时所持有的AccessKey ID。 如何获取您的accessid和accesskey，请参见获取账号的AK信息。
accesskey	您在访问OSS中的文件时所持有的Access Key Secret。
format	数据文件的格式，创建Parquet格式文件的外表时必须将其设置为parquet。

创建Parquet格式文件的外表时，需要注意数据类型的对应关系，具体规则如下：

Parquet基本类型	Parquet的logicalType类型	可对应ADB类型
BOOLEAN	无	boolean
INT32	INT_8	tinyint
INT32	INT_16	smallint
INT32	无	int或integer
INT64	无	bigint
FLOAT	无	float
DOUBLE	无	double
• FIXED_LEN_BYTE_ARRAY • BINARY • INT64 • INT32	DECIMAL	decimal
BINARY	UTF-8	• varchar • string • json（如果已知Parquet该列内容为json格式）
INT32	DATE	date
INT64	TIMESTAMP_MILLIS	timestamp或datetime
INT96	无	timestamp或datetime

> **注意**
>
> - 外表定义中column的名称应与Parquet文件的中该column的名称必须完全对应（可忽略大小写），而顺序可以是随意的，但建议也保持同样顺序。
> - 外表定义中的column可以只选择Parquet文件的部分列，未被外表定义的Parquet文件的列将被忽略；反之如果定义了Parquet文件中未包含的列，该列的查询将均为NULL。

创建OSS TEXT格式外表

```
CREATE TABLE IF NOT EXISTS oss_text_external_table
(
    uid string,
    other string
)
engine='oss' TABLE_PROPERTIES='{
    "endpoint":"oss-cn-hangzhou-internal.aliyuncs.com",
    "accessid":"LTAIF****5FsE",
    "accesskey":"Ccw****iWjv",
    "format":"text",
    "row_delimiter":"\n",
    "field_delimiter":"\n",
    "URL":"oss://****"
}';
```

参数	说明
ENGINE='OSS'	表示该表是外部表，使用的存储引擎是OSS。
TABLE_PROPERTIES	用于告知AnalyticDB for MySQL如何访问OSS中的数据。
endpoint	OSS的 `EndPoint（域名节点）` 。 **说明** 目前仅支持AnalyticDB for MySQL通过OSS中ECS的VPC网络（内网）访问OSS。 登录OSS控制台，单击目标Bucket，在Bucket**概览**页面查看 `endpoint` 。
url	OSS中目标数据文件夹或文件的绝对地址。 **示例：** oss://$Bucket名称/adb/oss_import_test_data.txt
accessid	您在访问OSS中的文件时所持有的AccessKey ID。 如何获取您的accessid和accesskey，请参见获取账号的AK信息。
accesskey	您在访问OSS中的文件时所持有的Access Key Secret。
format	数据文件的格式，创建TEXT格式文件的外表时必须将其设置为text。

参数	说明
row_delimiter	定义TEXT文件的行分割符，目前仅支持一种：`\n`
field_delimiter	定义TEXT文件的列分隔符，只能为一个字符。设置为`\n`表示整行为一个字段。

针对带有分区的Parquet/CSV数据文件创建OSS外表

有的OSS数据源是包含分区的，会在OSS上形成一个分层目录，类似如下内容：

```
parquet_partition_classic/
├── p1=2020-01-01
│   ├── p2=4
│   │   ├── p3=SHANGHAI
│   │   │   ├── 000000_0
│   │   │   └── 000000_1
│   │   └── p3=SHENZHEN
│   │       └── 000000_0
│   └── p2=6
│       └── p3=SHENZHEN
│           └── 000000_0
├── p1=2020-01-02
│   └── p2=8
│       ├── p3=SHANGHAI
│       │   └── 000000_0
│       └── p3=SHENZHEN
│           └── 000000_0
└── p1=2020-01-03
    └── p2=6
        ├── p2=HANGZHOU
        └── p3=SHENZHEN
            └── 000000_0
```

上述数据中p1为第1级分区，p2为第2级分区，p3为第3级分区。对应这种数据源，一般都希望以分区的模式进行查询，那么就需要在创建OSS外表时额外指明分区列。具体的建表语法示例如下（本例为Parquet格式，分区也支持CSV格式）：

```
CREATE TABLE IF NOT EXISTS oss_parquet_partition_table
(
  uid varchar,
  other varchar,
  p1 date,
  p2 int,
  p3 varchar
)
ENGINE='OSS'
TABLE_PROPERTIES='{
  "endpoint":"oss-xxxx.aliyuncs.com",
  "url":"oss://****/****/oss_parquet_data_dir",
  "accessid":"****",
  "accesskey":"****",
  "format":"parquet",
  "partition_column":"p1, p2, p3"
}'
```

说明

- 如上例所示，除了在table的列定义中声明p1、p2、p3及其类型，还需要在 `TABLE_PROPERTIES` 部分中的**partition_column**属性里声明它们为分区列。且**partition_column**属性里必须按“第1级, 第2级, 第3级.....”的严格顺序声明（例中p1为第1级分区，p2为第2级分区，p3为第3级分区），在列定义中也需保持相同顺序，并将分区列置于列定义列表的末尾。
- 可以作为分区列的数据类型有：boolean、tinyint、smallint、int/integer、bigint、float、double、decimal、varchar/string、date、timestamp。
- 查询时分区列和其它数据列的表现和用法没有区别。

2.性能调优

2.1. 分析查询

2.1.1. 概述

本章节从AnalyticDB MySQL版的查询处理流程入手，介绍了执行计划的要素和影响查询性能的主要因素，如何定位慢查询或资源消耗较大的查询，以及如何通过SQL诊断功能、慢查询表等工具查看分析AnalyticDB MySQL版的查询执行计划。

2.1.2. 查询流程和执行计划

本文介绍AnalyticDB MySQL版的查询处理流程和执行计划中的相关概念。

查询流程

AnalyticDB MySQL版使用SQL语言完成用户和系统内部存储数据之间的交互。SQL查询的执行计划生成流程如下：

1. 客户端将SQL语句提交到AnalyticDB MySQL版的前端接入节点（即Controller节点）。更多关于接入节点的详情，请参见技术架构。
2. Controller节点中解析器（Parser）会对SQL语句进行解析并生成语法树，然后生成初步逻辑执行计划。
3. Controller节点中优化器（Optimizer）会评估是否需要对初步逻辑执行计划进行优化重写，并生成最终的逻辑执行计划（Plan），并根据数据是否需要在网络间传输来决定是否需要对计划进行切分。逻辑执行计划中会规定特定的执行处理方式，例如Join类型、Join顺序、聚合方式以及数据重分布方式等。
4. 执行计划任务的节点（即Executor节点）会接收最终的逻辑执行计划并将其转化成物理执行计划。物理执行计划由Stage和算子（Operator）组成，物理执行计划中的算子（Operator）会对数据按照一定规则进行处理。
5. Executor节点将数据处理的最终结果返回到客户端，或者写入AnalyticDB MySQL版集群的内部表以及其它外部存储系统（如OSS）中。

执行计划相关概念

了解以下概念能帮助您更好地分析AnalyticDB MySQL版的执行计划（即物理执行计划）：

- **Stage**

  在执行阶段，AnalyticDB MySQL版中的查询会首先被切分为多个Stage来执行，一个Stage就是执行计划中某一部分的物理实体。Stage的数据来源可以是底层存储系统中的数据或者网络中传输的数据，一个Stage由分布在不同Executor节点上相同类型的Task组成，多个Task会并行处理数据。

  > **说明** AnalyticDB MySQL版SQL诊断功能支持对Stage级别进行结果诊断。更多详情，请参见Stage级别诊断结果。

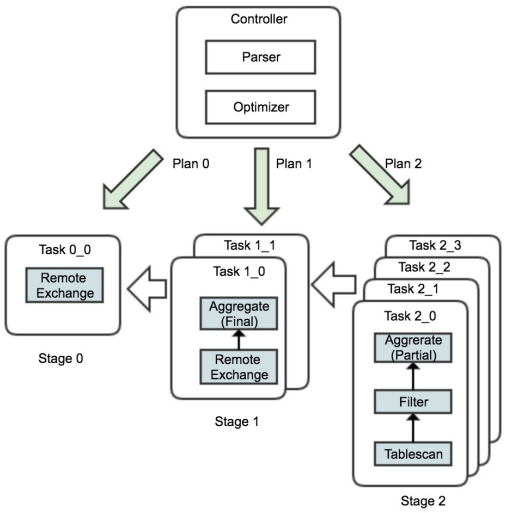
- **Task**

  Task是一个Stage在某个Executor节点上的执行实体，多个同类型的Task组成一个Stage，在集群内部并行处理数据。

- **算子（Operator）**

算子是AnalyticDB MySQL版的基本数据处理单元。AnalyticDB MySQL版会根据算子所表达的语义或算子间的依赖关系，决定使用并行还是串行执行来处理数据。

> **说明** AnalyticDB MySQL版SQL诊断功能支持对算子级别进行结果诊断。更多详情，请参见算子级别诊断结果。



上图是一个典型分组聚合查询的处理流程，AnalyticDB MySQL版的Controller节点会把查询的逻辑执行计划（Plan）分片下发到执行计划任务的各个节点上，其中：

- Stage 2由4个Task组成，并行执行数据的扫描、过滤以及局部聚合等操作。
- Stage 1由2个Task执行，并行执行最终的聚合操作。
- Stage 0由1个Task执行，负责汇总Stage 1的2个Task生成的最终聚合结果。

2.1.3. 算子

AnalyticDB中的一个算子负责完成一个基本的数据处理逻辑，一组算子按照执行计划完成数据的一组处理规则。AnalyticDB是一个分布式系统，大多数算子可以在多个节点上并行的完成计算任务，提高数据处理效率。

Sort

负责执行SQL语句中的Order By子句，执行对Order By字段的排序。

Aggregate

负责实现对数据的聚合操作或者分组聚合操作，例如sum、count、avg等。AnalyticDB是一个分布式数据库，可以多节点并行的完成聚合操作，ADB中典型的聚合流程一般可以包含Partial和Final两种，Partial阶段完成各个计算节点内部的局部聚合，局部聚合的结果会根据分组字段进行网络间的重分布，重分布后的数据需要执行Final阶段的最终聚合。

Tablescan

负责从数据源读取数据，VisualPlan可以看到扫描数据的Database名称、Table名称以及使用了Predicate Pushdown优化的过滤条件。该算子展示的数据过滤过程由底层数据源使用索引高效完成。有关VisualPlan的详细信息，可参见VisualPlan。

Filter

使用Filter算子完成在非数据源的数据过滤操作。ADB默认对所有字段创建了索引，但是有些过滤条件无法使用索引完成数据的过滤，例如过滤条件中包含对字段的function操作，或者用户手动删除了过滤条

件字段的索引，都会导致过滤操作无法通过索引完成，此时会使用Filter算子完成数据的过滤。

Window
执行开窗运算的算子。

Limit
执行limit操作的算子。

TopnRowNumber
对应开窗操作中的order by limit m,n查询。

Distinct limit
对应SQL语句中的group by distinct limit。

TopN
对应SQL语句中的order by limit m,n查询。

Tablewriter
ETL类型的SQL语句，例如insert into 或者replace into，在执行完对应的数据查询后，会使用Tablewriter算子完成目标表的写入操作。

Join
对应SQL语句中的join操作。AnalyticDB支持多种不同的Join类型，包括Inner Join、Left Outer Join、Right Outer Join、Full Outer Join、Semi/Anti Join。AnalyticDB在创建分布式表时需要制定分布字段(Distributed By)，join key是否为分布字段涉及到数据的重分布类型。有关数据的重分布类型的详细解释，可参见本文中的“RemoteExchange“。AnalyticDB会根据表结构、数据特征的不同使用不同的Join算法，目前主要有两类，一类是hash，该算法会把小表缓存到内存中，使用hash表查找的方式完成join操作；另外一类是index，该算法会充分利用AnalyticDB全索引的特点，使用join key的索引完成join操作。Join条件在VisualPlan中的join算子处使用Criteria标识。

Union
对应SQL语句中union操作。

RemoteExchange
该算子在SQL语句中没有特定的对应关系，用来表示当前Stage的数据来源于其他Stage。该算子负责执行从上游Stage拉取数据到当前Stage，主要分为repartition、replicated和gather三种方式。

- repartition表示当前Stage的某个计算节点只接收上游Stage的计算节点的特定分区的数据，在数据量较大时往往采用这种方式把相同的key值进行汇聚；
- replicated表示下游Stage每个计算节点的数据进行了广播，复制到了所有下游Stage的节点。在数据量较小时，往往采用这正方式，防止在join时对大表数据进行网络间的传输，或者通过复制小表的方式防止join时的数据倾斜发生；
- gather表示下游Stage的每个计算节点的数据没有复制和广播，而是汇聚到了当前Stage的一个节点上。

MarkDistinct
完成Distinct操作。

2.1.4. 影响查询性能的因素

本文介绍影响AnalyticDB MySQL版查询性能的因素。

背景信息

- **集群规格**

  AnalyticDB MySQL版集群支持多种规格（更多详情，请参见产品规格），不同集群规格的CPU核数、内存大小和数据存储介质等属性不同，处理子任务的能力也就不同，因此您需要结合业务查询特征来选择集群规格。例如，以Join或分组聚合为主的业务查询会消耗较多的CPU和内存资源；而以扫描数据和简单分组聚合操作为主的查询则会消耗较多的磁盘I/O资源。资源类型消耗量的不同会导致不同规格的集群存在不同的性能瓶颈，最终影响整体的查询效果。

- **节点数量**

  AnalyticDB MySQL版使用了分布式数据处理架构，一条查询会被分解成多个Stage在不同的节点上并行执行。所以如果集群中的节点数量越多，AnalyticDB MySQL版处理查询的能力也会越强。您可以根据实际的业务需求来决定集群节点的购买数量，更多详情，请参见创建集群。

- **数据分布特征**

  由于使用了分布式数据处理架构，AnalyticDB MySQL版具备将一条查询分解到多个节点上并行执行的能力。但AnalyticDB MySQL版能否充分利用多节点来并行处理查询，还取决于数据在存储节点上的分布特征。如果数据能够均匀分布在存储节点上，那么AnalyticDB MySQL版中的多个子任务在处理数据时，就能几乎同时结束任务，实现理想的查询处理；如果数据分布不均匀，那么子任务在处理数据时会存在时间上的长尾，从而影响最终的查询效果。

- **数据量大小**

  AnalyticDB MySQL版在处理查询时，通常不会将处理过程中的临时结果暂时写到磁盘里，而是尽量在内存中将所有数据处理掉。如果查询需要处理的数据量较大，就可能会长时间占用大量的资源，导致整体查询效率降低，进而影响最终的查询效果。

  此外，如果AnalyticDB MySQL版中表存储的数据量较大，那么在执行索引过滤、明细数据读取等操作时也会出现相互争抢磁盘I/O资源的情况，导致查询变慢。

- **查询并发度**

  由于集群规格和规模的限制，AnalyticDB MySQL版能同时处理的查询数量也会存在上限。如果查询的并发度过高，集群节点资源已到达瓶颈，那么后台的查询就会出现较长时间的排队，影响整体查询效果。

- **查询复杂度**

  查询的复杂度不同，对AnalyticDB MySQL版造成的压力也不同。例如，如果查询中过滤条件过于复杂，会在数据过滤时对存储节点造成一定压力；如果查询中Join算子过多，数据可能需要在不同节点间进行多次的网络传输，造成网络阻塞；如果查询中分组字段过多，也会占用较多的内存资源。

2.1.5. 典型慢查询

本文介绍AnalyticDB MySQL版中几种典型的慢查询以及导致慢查询的原因。

消耗内存的慢查询

查询的峰值内存（Peak Memory）可以帮助您评估内存的消耗情况。通常来说，查询峰值内存越大，内存消耗越大。

您可以通过SQL诊断功能来检索某个时间段内执行耗时较长的查询，然后在**查询属性**或不同级别的诊断结果中查看目标查询的**峰值内存**。更多详情，请参见查看查询属性与诊断结果。

导致查询内存消耗较大的原因通常有如下几种：

- Stage中有GROUP BY操作。

  AnalyticDB MySQL会将GROUP BY字段的值缓存在内存中。如果GROUP BY的字段唯一值较多，就会占用较多内存。

- Stage中有JOIN操作。

  在使用Hash方式进行Join时，AnalyticDB MySQL会把某张表的数据缓存到内存中。如果被缓存的表较大，就会占用较多内存。

- Stage中有SORT操作。

  在执行数据排序时，AnalyticDB MySQL会把数据缓存到内存中。如果需要排序的数据量较大，就会占用较多内存。

- Stage中有窗口函数操作。

  在执行窗口函数时，AnalyticDB MySQL会把数据缓存在内存中。如果需要执行窗口函数的数据量较大，就会占用较多内存。关于窗口函数的详情，请参见窗口函数。

消耗CPU的慢查询

查询的执行耗时（Time Consumed）可以帮助您评估CPU的消耗情况。通常来说，查询执行耗时越长，CPU消耗越大。

您可以通过SQL诊断功能来检索某个时间段内执行耗时较长的查询，然后在**查询属性**中查看目标查询的**耗时**。更多详情，请参见查看查询属性。

导致查询CPU消耗较大的原因通常有如下几种：

- 过滤条件没有下推到存储层。AnalyticDB MySQL在创建表时默认为所有字段创建了索引，使用索引进行过滤可以极大减少CPU资源的消耗，但是在某些场景下过滤条件没有下推。更多详情，请参见过滤条件没有下推。
- Join条件中带有过滤操作。如果Join中带有过滤条件，AnalyticDB MySQL会先对两表执行Join，再对Join后的数据执行过滤，此时的过滤无法使用索引。如果Join后产生的数据量较大，过滤操作就会消耗较大的CPU资源。
- Join时没有指定Join条件。如果没有指定Join条件，AnalyticDB MySQL会对左右两表执行笛卡尔积运算，产生的数据量行数是左右两表数据行数的乘积，该类操作会导致消耗较大的CPU资源。

消耗磁盘I/O的慢查询

查询的扫描行数（Scanned Rows）和扫描量（Amount of Scanned Data）可以帮助您评估磁盘I/O资源的消耗情况。通常来说，扫描行数和扫描量越多，磁盘I/O消耗越大。

您可以通过SQL诊断功能来检索某个时间段内执行耗时较长的查询，然后在**查询属性**或不同级别的诊断结果中查看目标查询的**扫描行数**和**扫描量**。更多详情，请参见查看查询属性与诊断结果。

导致查询磁盘I/O消耗较大的原因通常有如下几种：

- 过滤条件的数据筛选率较低，导致索引的使用效率不高，需要读取的索引量较大。
- 过滤条件没有下推，导致对源表进行了全表扫描。
- 过滤条件下推，但是过滤条件设置的范围较大，仍然有大量数据被扫描。
- 需要扫描的分区较多。通常情况下，分区越多意味着需要扫描的数据量越大。

2.1.6. 查询性能分析

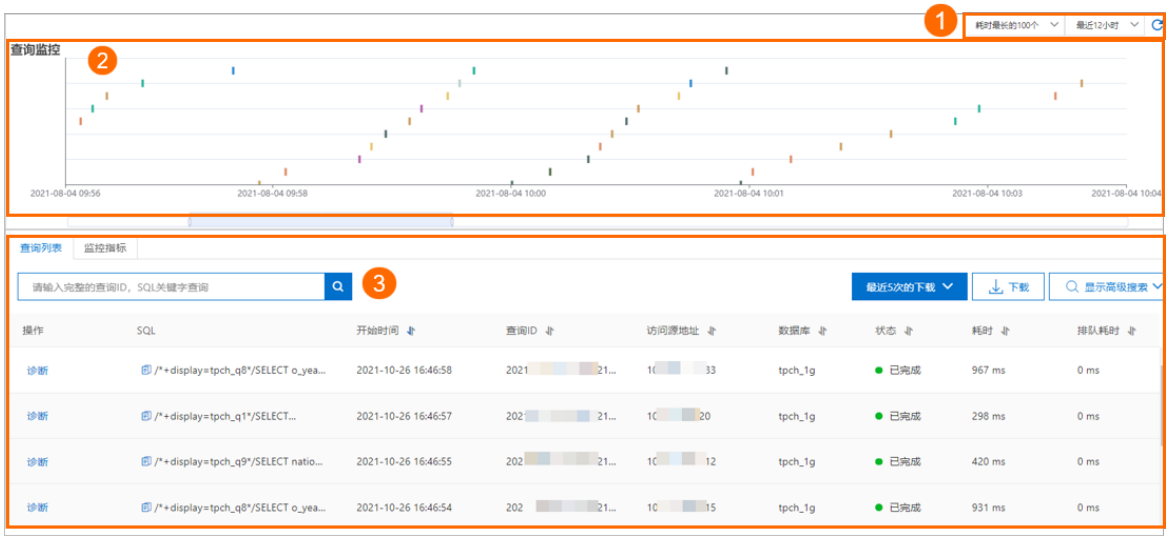
2.1.6.1. 通过SQL诊断功能分析执行计划

2.1.6.1.1. SQL诊断功能介绍

AnalyticDB MySQL版集群提供了SQL诊断功能，支持通过多种条件检索出符合条件的SQL查询（如慢查询），并将检索结果以图形化的方式展示，您还可以将检索结果下载保存到本地进行查看。本文介绍如何进入SQL诊断页面以及支持的检索方式。

进入SQL诊断页签

1. 登录云原生数据仓库AnalyticDB MySQL控制台。
2. 在页面左上角，选择集群所在地域。
3. 在左侧导航栏，单击**集群列表**。
4. 在**数仓版（3.0）**页签下，单击目标**集群ID**。
5. 在左侧导航栏，单击**诊断与优化**即可进入**SQL诊断**页签。



序号	介绍
①	SQL查询的检索条件。支持选择不同维度和时间范围来检索出符合条件的SQL查询信息。更多详情，请参见检索方式。
②	以图像化的方式展示SQL查询的检索结果。更多关于查询监控图的详情，请参见查询监控图。
③	以查询列表展示SQL查询的检索结果。更多关于查询列表的详情，请参见查询列表。

检索方式

AnalyticDB MySQL版集群支持通过整体检索和高级搜索两种方式来获取SQL查询详情：

- **整体检索**
  - **检索条件**：支持检索出在指定时间范围（如**最近5分钟**）内，不同SQL查询状态（例如**已完成的查询**）或查询耗时（例如**长耗时查询（>1min）**）的SQL查询详情。
  - **结果展示**：整体检索的结果会在**查询监控**和**SQL列表**中展示。更多**查询监控**和**SQL列表**的详情，请参见查询监控图和查询列表介绍。

> **说明**
>
> - 默认展示**最近5分钟**内**耗时最长的100个**SQL查询（不包含正在执行中的查询）。
> - AnalyticDB MySQL版支持自定义检索最近2周内的SQL查询详情。自定义检索的结束时间需晚于开始时间，且开始和结束时间间隔不能超过24小时。

- **高级搜索**
  - **搜索条件**：高级搜索功能可以根据内存、扫描量、用户名、数据库名、资源组等条件对整体检索的结果进行筛选。其中用户名、数据库名、资源组的可选取值是整体检索结果中已有的取值，而不是当前AnalyticDB MySQL版集群中的所有取值。

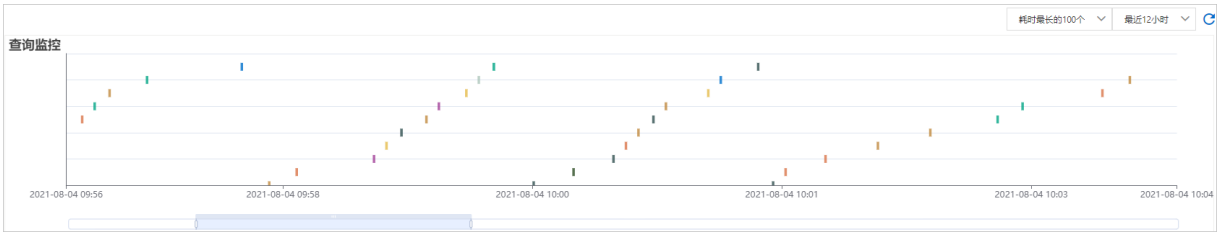
    例如，AnalyticDB MySQL版集群中有3个数据库：`db1`、`db2`、`db3`，而符合整体检索条件的查询仅涉及`db1`和`db2`。此时查询列表右上角高级搜索中的**数据库**选择范围仅为`db1`和`db2`。
  - **结果展示**：高级检索的筛选结果仅在**SQL列表**中展示，而不会影响**查询监控**中的结果。

2.1.6.1.2. 查询监控图和查询列表介绍

AnalyticDB MySQL版集群提供了SQL诊断功能，支持通过多种维度检索出符合条件的SQL查询（如慢查询），并将检索结果以图像化的方式展示，您还可以将检索结果下载保存到本地进行查看。本文介绍如何使用SQL诊断中的查询监控图和查询列表。

查询监控图

您可以在SQL诊断中的**查询监控**区域查看指定时间范围内，查询执行耗时的分布情况，方便快速定位执行耗时长的查询。



> **说明**
>
> - 图中每个色块即代表一次查询。
> - 矩形的颜色没有特殊含义，仅用来区分不同查询。色块长度越长，表示对应查询的执行耗时越长。
> - 将鼠标放在色块上即可查看对应查询的相关信息（例如，查询开始或结束时间、扫描量等），单击**详情**即可进入目标查询的详情页来查看**查询属性**、**查询语句**和**执行计划**等信息。更多详情，请参见使用执行计划分析查询。
> - **查询监控**中最多展示1万个查询，每个查询的具体信息会在监控图下方的**SQL列表**中展示。更多详情，请参见查询列表。
> - **查询监控**仅展示整体检索（如检索**最近5分钟**内**耗时最长的100个**查询）的结果，而**SQL列表**右上角的高级搜索不会影响**查询监控**的结果展示。更多详情，请参见检索方式。

查询列表

相较于查询监控图，查询列表能够展示SQL查询的更多信息，例如具体的SQL语句、访问源地址等。同时您还可以通过查询列表右上角的高级搜索功能对列表中的结果进行进一步筛选（这里的筛选结果并不会影响查询监控图的结果展示），帮助您更精准地定位到有问题的查询。

字段名	字段说明	使用说明
操作	单击**诊断**，查看执行详情，包括SQL语句、统计信息、执行计划以及诊断结果等。	进入**诊断**页面，单击**执行计划**分析查询。具体操作，请参见使用执行计划分析查询。
SQL	SQL语句。	用户可以单击复制完整SQL。
开始时间	查询提交的时间。	无
访问源地址	与客户端建立连接的IP地址。	在**诊断与优化**页面，单击**连接信息**，查看访问源地址。
数据库	客户端建立连接的数据库名。	用户可以使用高级搜索功能搜索出数据库变更的SQL。
状态	SQL语句的执行状态，包括： • 已完成 • 运行中 • 失败	无
总耗时	SQL语句从提交到最终执行完成的总耗时。	总耗时=排队耗时+执行计划耗时+执行耗时。
排队耗时	SQL语句从提交到AnalyticDB MySQL版接入节点后到开始执行前的排队时间。	AnalyticDB MySQL版在接入节点进行了流量控制，当并发较大时，超过流量控制队列大小的查询会进行排队，且排队时间较长。如果此时计算资源水位较低，可以尝试调大队列，具体操作，请参见查询流量控制；如果此时计算资源水位较高，则需要进行Bad SQL查找并定位原因。关于接入节点的详情，请参见整体架构。
执行计划耗时	生成执行计划的耗时。	SQl语句提交到接入节点后会生成执行计划，执行计划耗时较长的可能原因包括： • SQL语句比较复杂； • 接入节点压力较大。 如果SQL语句较复杂，可以尝试降低SQL复杂度，如减少Join的个数；如果接入节点压力较大，可以查看是否有较多数据返回到客户端或查询并发是否较大，具体详见Query级别诊断结果。

字段名	字段说明	使用说明
执行耗时	SQL语句的执行耗时。	可以根据执行耗时字段排序，排除排队耗时和执行计划耗时的影响，查找到消耗资源的SQL。
返回数据	Select语句返回到客户端的数据量。	返回到客户端的数据量不宜过大，数据量过大会导致查询占用前端队列资源，影响其他查询的提交和执行。用户可以根据返回数据大小进行排序，找到返回数据量较大的查询。
用户名	客户端建立连接时使用的用户名。	在**诊断与优化**页面，单击**连接信息**，查看用户名和用户连接数。
峰值内存	查询消耗的峰值内存。	AnalyticDB MySQL版在执行SQL的过程中，是分阶段（Stage）执行的，有依赖关系的阶段会串行执行，无依赖关系的阶段会并行执行，因此查询消耗的内存存在一个峰值，峰值内存可以体现查询对于内存资源的占用量。
扫描数据	查询从存储层返回到计算层的数据量。	扫描数据量的大小在一定程度上能体现查询对存储层的压力，读取的数据量越大，消耗的磁盘IO资源也越大，在计算层的处理也会消耗更多资源，影响查询速度。
总的Stage个数	查询生成的总的Stage个数。	Stage的个数多少，在一定程度上能体现一条查询语句的复杂度，Stage个数越多，AnalyticDB MySQL版在执行过程中需进行的网络交互就越多，对系统整体的压力也越大。如果某些查询的Stage个数较多，需要进行相应的调优。关于Stage的概念，请参见查询流程和执行计划。
ETL的写表行数	ETL相关SQL写出到目标表的行数。	无

您可以在**SQL列表**页签下执行如下操作：

支持的操作	说明
下载	单击**下载**即可将当前的查询结果保存为Excel表格，查询结果表格会自动保存在**最近5次的下载**的下拉列表中。 **说明** 当前单次最多支持下载10万条记录。
最近5次的下载	从**最近5次的下载**下拉列表中选择需要下载的SQL查询搜索结果即可完成下载。
高级搜索	单击**显示高级搜索**，根据业务需要选择查询维度，例如按照**资源组**或**访问源地址**维度对**查询监控**的搜索结果进行进一步的筛查。
诊断	单击**操作**栏中的**诊断**，即可进入目标查询的详情页来查看**查询属性**、**查询语句**和**执行计划**等信息。更多详情，请参见使用执行计划分析查询。

> **说明**
>
> - SQL列表右上角的高级搜索支持对整体检索的结果进行进一步筛选，筛选条件的可选取值是整体检索结果中已有的取值，而不是当前AnalyticDB MySQL版集群中的所有取值。更多详情，请参见检索方式。
> - 当整体检索条件选择**正在运行的查询**时，检索结果仅显示执行耗时超过10s的SQL查询详情，且在SQL列表中会展示**资源消耗排名**列，**资源消耗排名**的数值越小，说明该查询消耗的CPU、内存等资源越多。
> - SQL列表的SQL列，最长支持显示5120个字符，超出字符限制的SQL语句会被截断显示。您可以在下载的Excel文件中，或目标查询详情页的**查询语句**页签下查看完整的SQL。

2.1.6.1.3. 使用执行计划分析查询

AnalyticDB MySQL版的SQL诊断功能支持以树形图的形式展现SQL查询的执行计划。执行计划树分为两层：第一层是Stage层，第二层是算子（Operator）层。本文介绍如何使用Stage层和算子层执行计划树来分析查询。

Stage层执行计划树

Stage层执行计划树由多个Stage节点组成，数据流向自下而上，先由具有扫描算子的Stage进行数据扫描，再经过中间Stage节点的层层处理后，再由最上层的根节点将查询结果返回客户端。

Stage层执行计划树中主要包含如下信息：

- **基本信息**

  图中的每个矩形框代表一个Stage，框里会包含Stage ID、数据输出类型、耗时或内存（选择**按内存**排序时展示）等信息。

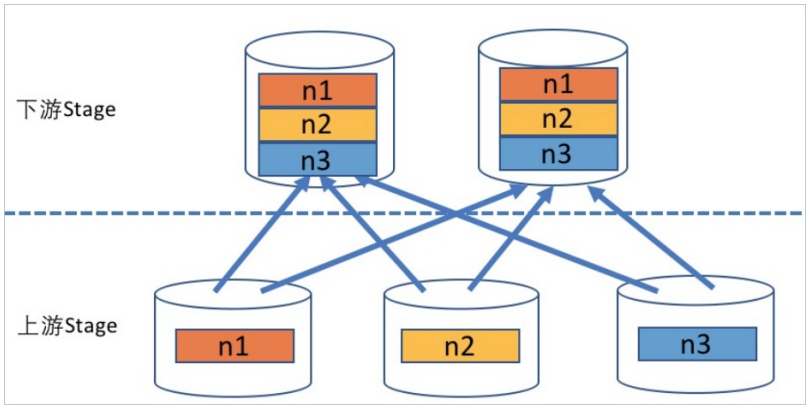
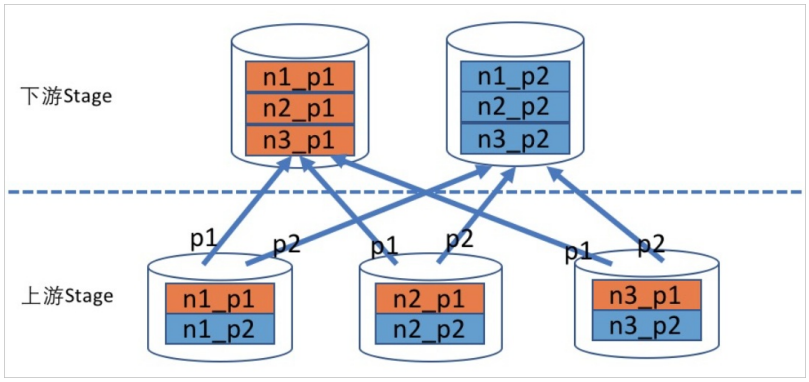
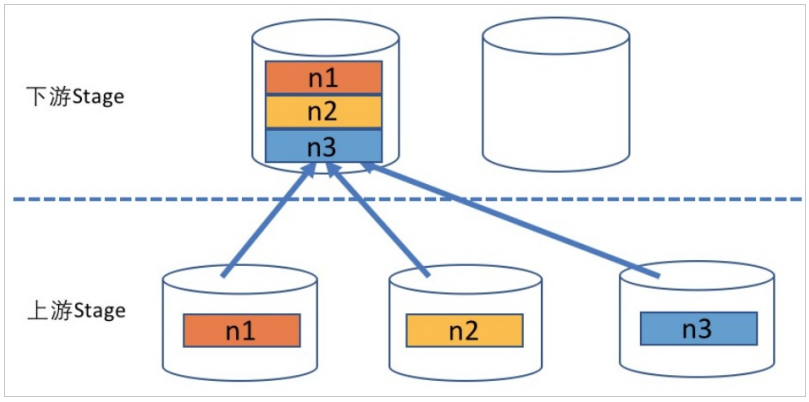
  > **说明** 当Stage层执行计划树上出现红色警示号，表示该Stage被诊断出存在可优化点。

- **数据输出行数**

  两个相邻Stage间连线上的数字表示上游Stage向下游Stage输出的数据行数。数据输出行数越多，Stage间的连线越粗。

- **数据输出方法**

  表示在两个相邻的Stage间，上游向下游Stage传输数据时所用的方法。AnalyticDB MySQL版支持以下数据输出方法。

数据输出方法	说明
Broadcast	表示上游Stage中每个计算节点的数据都会复制到所有下游Stage的计算节点。 
Repartition	表示上游Stage中每个节点的数据会按照固定的规则切分后，再分发到下游Stage的指定计算节点。 
Gather	表示上游Stage中每个节点的数据会集中到下游Stage中某一个特定的计算节点。 

- **查看内存使用率或执行耗时Top 10的Stage详情**

执行计划树右侧的**Top10耗时/内存倒序**页签下，会展示执行耗时占总查询耗时比例最大，或使用内存占总查询使用内存比例最大的前10个Stage的ID和对应的比例。

> 说明
> - 默认**按耗时**排序，您也可以选择执行计划树右上角的**按内存**排序。
> - **Top10耗时/内存倒序**页签下不会显示内存使用量或执行耗时占比小于1%的Stage数据。
> - 由于统计方式的差异，可能存在一个查询中的所有Stage耗时或内存的百分比之和不等于100%的情况。

- **诊断结果**

  单击执行计划树中某个Stage（如**Stage[1]**），即可在右侧查看对应Stage的**诊断结果**详情，包括如下两类诊断：

  - **Stage诊断**：这类诊断结果包含了对目标Stage诊断结果的详细说明，包括诊断出的问题（如存在较大的数据量被广播或数据倾斜）以及对应的调优方案。
  - **算子诊断**：这类诊断仅展示当前Stage中有问题的算子名称以及对应问题的概述，详细的说明和调优方案需要到算子执行计划树中才能看到。更多详情，请参见算子层计划执行树。

  关于Stage的诊断结果的说明，请参见Stage级别诊断结果。

- **统计信息**

  诊断结果下方展示了目标Stage中各项指标的**统计信息**。

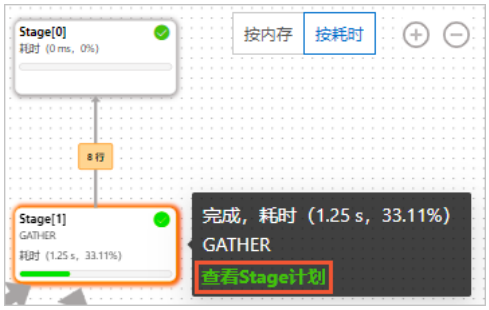
指标项	说明
峰值内存	目标Stage的峰值内存消耗量。系统会根据实际内存消耗量选择Bytes、KB、MB、GB或TB为单位。
累积耗时	目标Stage内存中所有算子多节点多线程执行时耗时的累加，系统会根据实际耗时选择ms、s、m或h为单位。 **说明** 该累积耗时不能和当前查询的总耗时进行比较。
输出行数	目标Stage输出的行数。
输出大小	目标Stage输出的数据量。系统会根据实际数据量选择Bytes、KB、MB、GB或TB为单位。
输入行数	目标Stage输入的行数。
输入大小	目标Stage输入的数据量。系统会根据实际数据量选择Bytes、KB、MB、GB或TB为单位。
扫描行数	目标Stage扫描的行数。 **说明** 仅当目标Stage中存在数据扫描算子时支持该参数。

指标项	说明
扫描量	目标Stage扫描的数据量。系统会根据实际数据量选择Bytes、KB、MB、GB或TB为单位。 **说明** 仅当目标Stage中存在数据扫描算子时支持该参数。

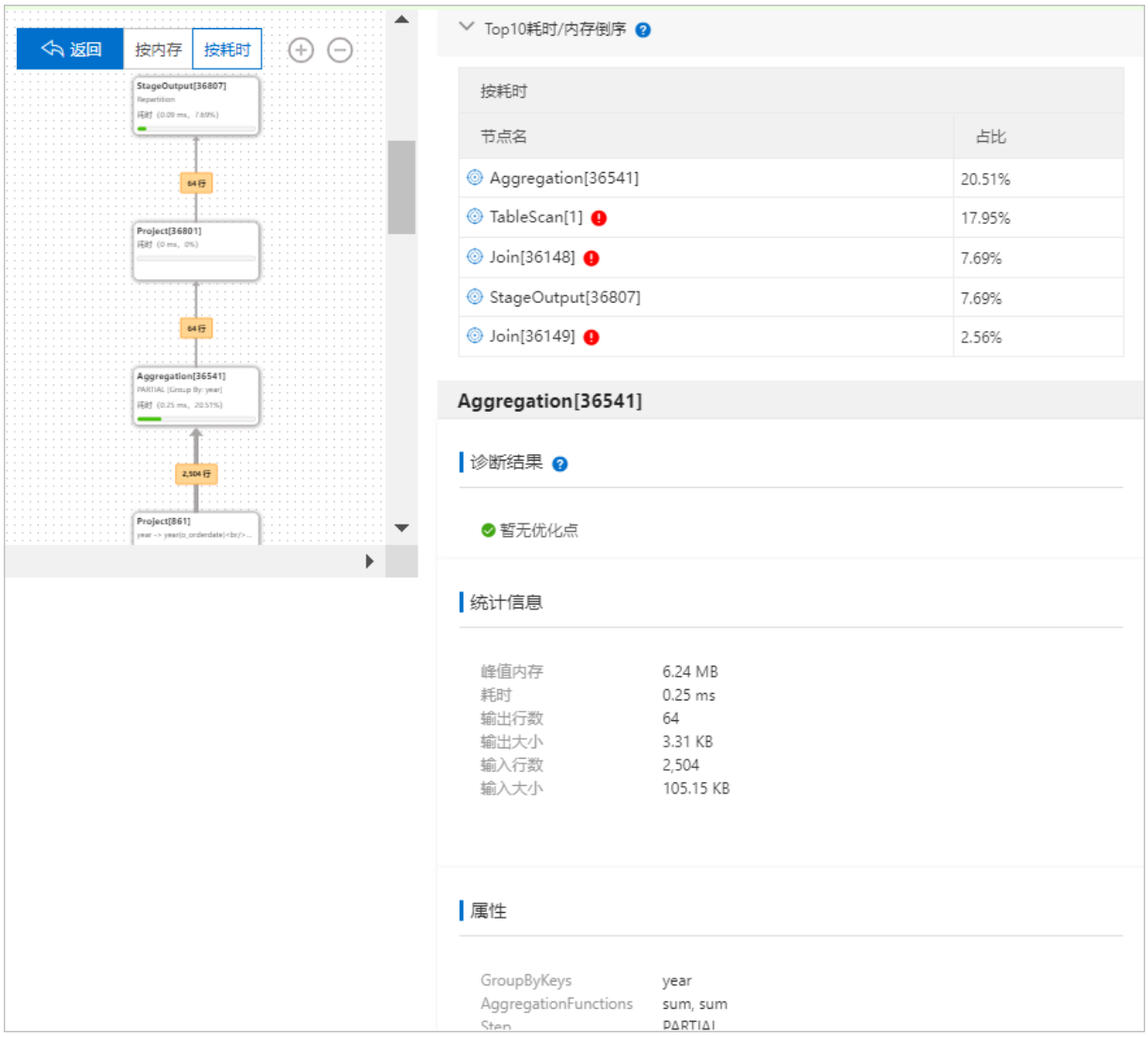
算子层计划执行树

算子层执行计划由多个算子组成，图中的每个矩形框代表一个算子，数据流向自下而上，扫描数据过程或接收网络数据由最上游的算子（TableScan和RemoteSource）完成，扫描到的数据和接收到的网络数据经过中间算子层层处理后，再由最上层的根节点算子（StageOutput或Output）将数据输出到下游Stage或将查询结果返回给客户端。

您可以将鼠标移动到目标Stage上，在弹出的信息框中单击**查看Stage计划**，即可进入对应Stage的计划详情页来查看算子层计划执行树。



算子层执行计划树中包含如下信息：



> **说明** 当算子层执行计划树上出现红色警示号，表示该算子被诊断出存在可优化点。

- **基本信息**

  图中的每个矩形框代表一个算子，框里会包含算子名称及ID、算子属性（如Join算子的Join条件、算法等）、耗时或内存（选择**按内存**排序时展示）等信息。

- **数据输出行数**

  两个相邻算子间连线上的数字表示上游算子向下游算子输出的数据行数。数据输出行数越多，算子间的连线越粗。

- **查看内存使用率或执行耗时Top 10的算子详情**

  执行计划树右侧的**Top10耗时/内存倒序**页签下，会展示执行耗时占总查询耗时比例最大，或使用内存占总查询使用内存比例最大的前10个算子的ID和对应的比例。

> 说明
> - 默认**按耗时**排序，您也可以选择执行计划树右上角的**按内存**排序。
> - **Top10耗时/内存倒序**页签下不会显示内存使用量或执行耗时占比小于1%的算子数据。
> - 由于统计方式的差异，可能存在一个Stage中的所有算子耗时或内存的百分比之和不等于100%的情况。

- **诊断结果**

  单击执行计划树中某个算子（如**Join[36148]**），即可在右侧查看对应算子的**诊断结果**详情，包括诊断出的问题（如存在数据膨胀或Join的右表过大）以及对应的调优方案。更多关于算子诊断的详情，请参见算子级别诊断结果。

- **统计信息**

  诊断结果下方展示了目标算子中各项指标的**统计信息**。

指标项	说明
峰值内存	目标算子的峰值内存消耗量。系统会根据实际内存消耗量选择Bytes、KB、MB、GB或TB为单位。
耗时	目标算子在一定并发度上的平均耗时，系统会根据实际耗时选择ms、s、m或h为单位。 **说明** 该耗时可以和当前查询的耗时进行比较。
输出行数	目标算子输出的行数。
输出大小	目标算子输出的数据量。系统会根据实际数据量选择Bytes、KB、MB、GB或TB为单位。
输入行数	目标算子输入的行数。

指标项	说明
输入大小	目标算子输入的数据量。系统会根据实际数据量选择Bytes、KB、MB、GB或TB为单位。
Builder统计信息	包括Builder的类型、峰值内存、耗时、输出输入行数和数据量等信息。主要有如下Builder类型： ○ HashBuiler：用于构建Hash表来完成Hash Join计算。 ○ SetBuilder：用于构造Set结构来完成Semi Join计算。 ○ NestLoopBuilder：用于完成Nest Loop Join计算。 **说明** 仅Join算子支持该统计指标项。
属性	不同算子的属性信息不同，例如Join算子的属性信息中包括了Join的类型、方式等。更多详情，请参见算子。

2.1.6.2. 通过EXPLAIN和EXPLAIN ANALYZE分析执行计划

本文介绍如何使用 `EXPLAIN` 和 `EXPLAIN ANALYZE` 命令来分析查询执行计划。

前提条件

AnalyticDB MySQL版集群需为3.1.3或以上版本。

说明 如何查看集群版本，请参见查看版本。如需升级版本，请提交工单联系技术支持。

EXPLAIN

您可以通过 `EXPLAIN` 命令来评估查询语句的执行方式，评估结果仅供参考，并不等于实际的执行结果。

- **语法**

```
EXPLAIN (format text) <SELECT statement>;
```

说明 如果查询SQL不复杂，您可以在 `EXPLAIN` 命令中加上 `(format text)`，来提高返回结果中计划树层次结构的易读性。

- **示例**

```
EXPLAIN (format text)
      SELECT count(*)
      FROM
      nation, region, customer
      WHERE
      c_nationkey = n_nationkey
      AND n_regionkey = r_regionkey
      AND r_name = 'ASIA';
```

返回结果如下：

```
Output[count(*)]
│   Outputs: [count:bigint]
│   Estimates: {rows: 1 (8B)}
│   count(*) := count
└─ Aggregate(FINAL)
   │   Outputs: [count:bigint]
   │   Estimates: {rows: 1 (8B)}
   │   count := count(`count_1`)
   └─ LocalExchange[SINGLE] ()
      │   Outputs: [count_0_1:bigint]
      │   Estimates: {rows: 1 (8B)}
      └─ RemoteExchange[GATHER]
         │   Outputs: [count_0_2:bigint]
         │   Estimates: {rows: 1 (8B)}
         └─ Aggregate(PARTIAL)
            │   Outputs: [count_0_4:bigint]
            │   Estimates: {rows: 1 (8B)}
            │   count_4 := count(*)
            └─ InnerJoin[(`c_nationkey` = `n_nationkey`)][$hashvalue, $hashvalue_0_6]
               │   Outputs: []
               │   Estimates: {rows: 302035 (4.61MB)}
               │   Distribution: REPLICATED
               ├─ Project[]
               │  │   Outputs: [c_nationkey:integer, $hashvalue:bigint]
               │  │   Estimates: {rows: 1500000 (5.72MB)}
               │  │   $hashvalue := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_c
ode`(`c_nationkey`), 0))
               │  └─ RuntimeFilter
               │     │   Outputs: [c_nationkey:integer]
               │     │   Estimates: {rows: 1500000 (5.72MB)}
               │     ├─ TableScan[adb:AdbTableHandle{schema=tpch, tableName=customer, par
titionColumnHandles=[c_custkey]}]
               │     │      Outputs: [c_nationkey:integer]
               │     │      Estimates: {rows: 1500000 (5.72MB)}
               │     │      c_nationkey := AdbColumnHandle{columnName=c_nationkey, type=4
, isIndexed=true}
               │     └─ RuntimeCollect
               │        │   Outputs: [n_nationkey:integer]
               │        │   Estimates: {rows: 5 (60B)}
               │        └─ LocalExchange[ROUND_ROBIN] ()
               │           │   Outputs: [n_nationkey:integer]
               │           │   Estimates: {rows: 5 (60B)}
               │           └─ RuntimeScan
               │                  Outputs: [n_nationkey:integer]
               │                  Estimates: {rows: 5 (60B)}
               └─ LocalExchange[HASH][$hashvalue_0_6] ("n_nationkey")
                  │   Outputs: [n_nationkey:integer, $hashvalue_0_6:bigint]
                  │   Estimates: {rows: 5 (60B)}
                  └─ Project[]
                     │   Outputs: [n_nationkey:integer, $hashvalue_0_10:bigint]
                     │   Estimates: {rows: 5 (60B)}
                     │   $hashvalue_10 := `combine_hash`(BIGINT '0', COALESCE(`$operator$
hash_code`(`n_nationkey`), 0))
                     └─ RemoteExchange[REPLICATE]
                        │   Outputs: [n_nationkey:integer]
```

```
                           │   Outputs: [n_nationkey:integer]
                           │   Estimates: {rows: 5 (60B)}
                           └─ InnerJoin[(`n_regionkey` = `r_regionkey`)][$hashvalue_0_7, $ha
shvalue_0_8]
                              │   Outputs: [n_nationkey:integer]
                              │   Estimates: {rows: 5 (60B)}
                              │   Distribution: REPLICATED
                              ├─ Project[]
                              │  │   Outputs: [n_nationkey:integer, n_regionkey:integer, $ha
shvalue_0_7:bigint]
                              │  │   Estimates: {rows: 25 (200B)}
                              │  │   $hashvalue_7 := `combine_hash`(BIGINT '0', COALESCE(`$o
perator$hash_code`(`n_regionkey`), 0))
                              │  └─ RuntimeFilter
                              │     │   Outputs: [n_nationkey:integer, n_regionkey:integer]
                              │     │   Estimates: {rows: 25 (200B)}
                              │     ├─ TableScan[adb:AdbTableHandle{schema=tpch, tableName=n
ation, partitionColumnHandles=[]}]
                              │     │      Outputs: [n_nationkey:integer, n_regionkey:intege
r]
                              │     │      Estimates: {rows: 25 (200B)}
                              │     │      n_nationkey := AdbColumnHandle{columnName=n_natio
nkey, type=4, isIndexed=true}
                              │     │      n_regionkey := AdbColumnHandle{columnName=n_regio
nkey, type=4, isIndexed=true}
                              │     └─ RuntimeCollect
                              │        │   Outputs: [r_regionkey:integer]
                              │        │   Estimates: {rows: 1 (4B)}
                              │        └─ LocalExchange[ROUND_ROBIN] ()
                              │           │   Outputs: [r_regionkey:integer]
                              │           │   Estimates: {rows: 1 (4B)}
                              │           └─ RuntimeScan
                              │                 Outputs: [r_regionkey:integer]
                              │                 Estimates: {rows: 1 (4B)}
                              └─ LocalExchange[HASH][$hashvalue_0_8] ("r_regionkey")
                                 │   Outputs: [r_regionkey:integer, $hashvalue_0_8:bigint]
                                 │   Estimates: {rows: 1 (4B)}
                                 └─ ScanProject[table = adb:AdbTableHandle{schema=tpch, tabl
eName=region, partitionColumnHandles=[]}]
                                       Outputs: [r_regionkey:integer, $hashvalue_0_9:bigint
]
                                       Estimates: {rows: 1 (4B)}/{rows: 1 (B)}
                                       $hashvalue_9 := `combine_hash`(BIGINT '0', COALESCE(
`$operator$hash_code`(`r_regionkey`), 0))
                                       r_regionkey := AdbColumnHandle{columnName=r_regionke
y, type=4, isIndexed=true}
```

返回结果中的主要参数说明见下表。

参数	说明
`Outputs: [symbol:type]`	每个算子的输出列及数据类型。

参数	说明
`Estimates: {rows: %s (%sB)}`	每个算子的估算行数及数据量。估算结果可用来决定优化器的Join Order和Data Shuffle。

EXPLAIN ANALYZE

您可以通过 `EXPLAIN ANALYZE` 命令查看查询的分布式执行计划以及实际执行代价，包括执行耗时、内存使用量，输入输出数据量等。

- **语法**

```
EXPLAIN ANALYZE <SELECT statement>;
```

- **示例**

```
EXPLAIN ANALYZE
SELECT count(*)
      FROM
      nation, region, customer
      WHERE
      c_nationkey = n_nationkey
      AND n_regionkey = r_regionkey
      AND r_name = 'ASIA';
```

返回结果如下：

```
Fragment 1 [SINGLE]
    Output: 1 row (9B), PeakMemory: 178KB, WallTime: 1.00ns, Input: 32 rows (288B); per t
ask: avg.: 32.00 std.dev.: 0.00
    Output layout: [count]
    Output partitioning: SINGLE []
    Aggregate(FINAL)
    │   Outputs: [count:bigint]
    │   Estimates: {rows: 1 (8B)}
    │   Output: 2 rows (18B), PeakMemory: 24B (0.00%), WallTime: 70.39us (0.03%)
    │   count := count(`count_1`)
    └─ LocalExchange[SINGLE] ()
       │   Outputs: [count1:bigint]
       │   Estimates: {rows: ? (?)}
       │   Output: 64 rows (576B), PeakMemory: 8KB (0.07%), WallTime: 238.69us (0.10%)
       └─ RemoteSource[2]
              Outputs: [count2:bigint]
              Estimates:
              Output: 32 rows (288B), PeakMemory: 32KB (0.27%), WallTime: 182.82us (0.08%
)
              Input avg.: 4.00 rows, Input std.dev.: 264.58%
Fragment 2 [adb:AdbPartitioningHandle{schema=tpch, tableName=customer, dimTable=false, sh
ards=32, tableEngineType=Cstore, partitionColumns=c_custkey, prunedBuckets= empty}]
    Output: 32 rows (288B), PeakMemory: 6MB, WallTime: 164.00ns, Input: 1500015 rows (20.
03MB); per task: avg.: 500005.00 std.dev.: 21941.36
    Output layout: [count4]
    Output partitioning: SINGLE []
    Aggregate(PARTIAL)
    │   Outputs: [count4:bigint]
```

```
    │   Estimates: {rows: 1 (8B)}
    │   Output: 64 rows (576B), PeakMemory: 336B (0.00%), WallTime: 1.01ms (0.42%)
    │   count_4 := count(*)
    └─ INNER Join[(`c_nationkey` = `n_nationkey`)][$hashvalue, $hashvalue6]
       │   Outputs: []
       │   Estimates: {rows: 302035 (4.61MB)}
       │   Output: 300285 rows (210B), PeakMemory: 641KB (5.29%), WallTime: 99.08ms (41.4
5%)
       │   Left (probe) Input avg.: 46875.00 rows, Input std.dev.: 311.24%
       │   Right (build) Input avg.: 0.63 rows, Input std.dev.: 264.58%
       │   Distribution: REPLICATED
       ├─ ScanProject[table = adb:AdbTableHandle{schema=tpch, tableName=customer, partiti
onColumnHandles=[c_custkey]}]
       │      Outputs: [c_nationkey:integer, $hashvalue:bigint]
       │      Estimates: {rows: 1500000 (5.72MB)}/{rows: 1500000 (5.72MB)}
       │      Output: 1500000 rows (20.03MB), PeakMemory: 5MB (44.38%), WallTime: 68.29ms
(28.57%)
       │      $hashvalue := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_code`(`c_
nationkey`), 0))
       │      c_nationkey := AdbColumnHandle{columnName=c_nationkey, type=4, isIndexed=tr
ue}
       │      Input: 1500000 rows (7.15MB), Filtered: 0.00%
       └─ LocalExchange[HASH][$hashvalue6] ("n_nationkey")
          │   Outputs: [n_nationkey:integer, $hashvalue6:bigint]
          │   Estimates: {rows: 5 (60B)}
          │   Output: 30 rows (420B), PeakMemory: 394KB (3.26%), WallTime: 455.03us (0.19
%)
          └─ Project[]
             │   Outputs: [n_nationkey:integer, $hashvalue10:bigint]
             │   Estimates: {rows: 5 (60B)}
             │   Output: 15 rows (210B), PeakMemory: 24KB (0.20%), WallTime: 83.61us (0.0
3%)
             │   Input avg.: 0.63 rows, Input std.dev.: 264.58%
             │   $hashvalue_10 := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_cod
e`(`n_nationkey`), 0))
             └─ RemoteSource[3]
                    Outputs: [n_nationkey:integer]
                    Estimates:
                    Output: 15 rows (75B), PeakMemory: 24KB (0.20%), WallTime: 45.97us (0
.02%)
                    Input avg.: 0.63 rows, Input std.dev.: 264.58%
Fragment 3 [adb:AdbPartitioningHandle{schema=tpch, tableName=nation, dimTable=true, shard
s=32, tableEngineType=Cstore, partitionColumns=, prunedBuckets= empty}]
    Output: 5 rows (25B), PeakMemory: 185KB, WallTime: 1.00ns, Input: 26 rows (489B); per
task: avg.: 26.00 std.dev.: 0.00
    Output layout: [n_nationkey]
    Output partitioning: BROADCAST []
    INNER Join[(`n_regionkey` = `r_regionkey`)][$hashvalue7, $hashvalue8]
    │   Outputs: [n_nationkey:integer]
    │   Estimates: {rows: 5 (60B)}
    │   Output: 11 rows (64B), PeakMemory: 152KB (1.26%), WallTime: 255.86us (0.11%)
    │   Left (probe) Input avg.: 25.00 rows, Input std.dev.: 0.00%
    │   Right (build) Input avg.: 0.13 rows, Input std.dev.: 264.58%
    │   Distribution: REPLICATED
```

```
    ├─ ScanProject[table = adb:AdbTableHandle{schema=tpch, tableName=nation, partitionCol
umnHandles=[]}]
    │      Outputs: [n_nationkey:integer, n_regionkey:integer, $hashvalue7:bigint]
    │      Estimates: {rows: 25 (200B)}/{rows: 25 (200B)}
    │      Output: 25 rows (475B), PeakMemory: 16KB (0.13%), WallTime: 178.81us (0.07%)
    │      $hashvalue_7 := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_code`(`n_r
egionkey`), 0))
    │      n_nationkey := AdbColumnHandle{columnName=n_nationkey, type=4, isIndexed=true}
    │      n_regionkey := AdbColumnHandle{columnName=n_regionkey, type=4, isIndexed=true}
    │      Input: 25 rows (250B), Filtered: 0.00%
    └─ LocalExchange[HASH][$hashvalue8] ("r_regionkey")
       │   Outputs: [r_regionkey:integer, $hashvalue8:bigint]
       │   Estimates: {rows: 1 (4B)}
       │   Output: 2 rows (28B), PeakMemory: 34KB (0.29%), WallTime: 57.41us (0.02%)
       └─ ScanProject[table = adb:AdbTableHandle{schema=tpch, tableName=region, partition
ColumnHandles=[]}]
             Outputs: [r_regionkey:integer, $hashvalue9:bigint]
             Estimates: {rows: 1 (4B)}/{rows: 1 (4B)}
             Output: 1 row (14B), PeakMemory: 8KB (0.07%), WallTime: 308.99us (0.13%)
             $hashvalue_9 := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_code`(`
r_regionkey`), 0))
             r_regionkey := AdbColumnHandle{columnName=r_regionkey, type=4, isIndexed=tr
ue}
             Input: 1 row (5B), Filtered: 0.00%
```

返回结果中的主要参数说明见下表。

参数	说明
`Outputs: [symbol:type]`	每个算子的输出列及数据类型。
`Estimates: {rows: %s (%sB)}`	每个算子的估算行数及数据量。估算结果可用来决定优化器的Join Order和Data Shuffle。
`PeakMemory: %s`	内存使用总和，用于分析内存使用的瓶颈点。
`WallTime: %s`	算子执行时间的累加总和，用于分析计算瓶颈点。 **说明** 由于存在并行计算，所以该时间并不是真实的执行时间。
`Input: %s rows (%sB)`	输入行数及数据量。
`per task: avg.: %s std.dev.: %s`	平均行数和其标准差，用于分析Stage内部的数据倾斜。
`Output: %s row (%sB)`	输出行数及数据量。

使用场景

您可以通过 `EXPLAIN ANALYZE` 分析一些常见的计划问题。

- **过滤器未下推**

在如下两个查询中，相较于SQL 1，SQL 2中由于存在不能下推的函数 `length(string_test)` ，需要扫描全量数据进行计算：

- SQL 1

```
SELECT count(*) FROM test WHERE string_test = 'a';
```

- SQL 2

```
SELECT count(*) FROM test WHERE length(string_test) = 1;
```

使用 `EXPLAIN ANALYZE` 分别分析上述两个查询的执行计划，对比计划中的Fragment 2，可以看出：

- SQL 1使用的算子是 `TableScan` ，且 `Input avg.` 为 `0.00 rows` ，说明过滤器下推成功，扫描数据量为0行。

```
Fragment 2 [adb:AdbPartitioningHandle{schema=test4dmp, tableName=test, dimTable=false,
shards=4, tableEngineType=Cstore, partitionColumns=id, prunedBuckets= empty}]
    Output: 4 rows (36B), PeakMemory: 0B, WallTime: 6.00ns, Input: 0 rows (0B); per tas
k: avg.: 0.00 std.dev.: 0.00
    Output layout: [count_0_1]
    Output partitioning: SINGLE []
    Aggregate(PARTIAL)
    │   Outputs: [count_0_1:bigint]
    │   Estimates: {rows: 1 (8B)}
    │   Output: 8 rows (72B), PeakMemory: 0B (0.00%), WallTime: 212.92us (3.99%)
    │   count_0_1 := count(*)
    └─ TableScan[adb:AdbTableHandle{schema=test4dmp, tableName=test, partitionColumnHan
dles=[id]}]
           Outputs: []
           Estimates: {rows: 4 (0B)}
           Output: 0 rows (0B), PeakMemory: 0B (0.00%), WallTime: 4.76ms (89.12%)
           Input avg.: 0.00 rows, Input std.dev.: ?%
```

- SQL 2使用的算子是 `ScanFilterProject`，且 `Input` 为 `9999 rows`，同时，`filterPredicate` 属性不为空（即 `filterPredicate = (`test4dmp`.`length`(`string_test`) = BIGINT '1')`）表明没有下推的过滤器，扫描数据量为9999行。

```
Fragment 2 [adb:AdbPartitioningHandle{schema=test4dmp, tableName=test, dimTable=false,
shards=4, tableEngineType=Cstore, partitionColumns=id, prunedBuckets= empty}]
    Output: 4 rows (36B), PeakMemory: 0B, WallTime: 102.00ns, Input: 0 rows (0B); per t
ask: avg.: 0.00 std.dev.: 0.00
    Output layout: [count_0_1]
    Output partitioning: SINGLE []
    Aggregate(PARTIAL)
    │   Outputs: [count_0_1:bigint]
    │   Estimates: {rows: 1 (8B)}
    │   Output: 8 rows (72B), PeakMemory: 0B (0.00%), WallTime: 252.23us (0.12%)
    │   count_0_1 := count(*)
    └─ ScanFilterProject[table = adb:AdbTableHandle{schema=test4dmp, tableName=test, pa
rtitionColumnHandles=[id]}, filterPredicate = (`test4dmp`.`length`(`string_test`) = BIG
INT '1')]
           Outputs: []
           Estimates: {rows: 9999 (312.47kB)}/{rows: 9999 (312.47kB)}/{rows: ? (?)}
           Output: 0 rows (0B), PeakMemory: 0B (0.00%), WallTime: 101.31ms (49.84%)
           string_test := AdbColumnHandle{columnName=string_test, type=13, isIndexed=tr
ue}
           Input: 9999 rows (110.32kB), Filtered: 100.00%
```

- **Bad SQL内存使用率**

  您可以直接查看每个Fragment的 `PeakMemory` 来定位资源消耗的问题。排除计划中Disaster Broadcast的情况，高 `PeakMemory` 通常是因为连接数据膨胀、连接的左表数据量过大、`TableScan` 算子扫描数据量过大等，需要从业务角度加条件限制数据量。另外，您也可以通过查看每个算子的 `PeakMemory` 百分比定位资源消耗最大的算子，再进一步分析。

2.1.6.3. 使用慢查询表

本文介绍如何通过慢查询表information_schema.kepler_slow_sql_merged来查找资源（如CPU、内存的等）消耗较多的SQL查询，便于初步定位慢查询原因。

使用说明

- 慢查询表及其详情信息保存在前端接入节点（即Controller节点）的本地磁盘文件中，该表存储的慢查询数量受AnalyticDB MySQL版集群规模和SQL复杂度的影响。集群规模越大或者SQL越复杂，执行一条查询所产生并保存在表里的信息也就越多，能保存的慢查询数量就越少。
- AnalyticDB MySQL版的慢查询表information_schema.kepler_slow_sql_merged记录了执行耗时大于1秒（默认时间）的SQL语句的部分统计信息。

> **说明** 如果要修改默认时间，请提交工单联系技术支持。

常用的字段

字段	说明
start_time	查询的执行开始时间。
time	查询的总耗时。单位：ms。
user	提交查询的用户名。
db	建立连接的数据库名称。
peak_mem	查询的峰值内存。单位为Byte。可用于判断目标查询SQL是否消耗了较多内存资源。
state	查询的状态，支持如下取值： • *SUCCESS*：执行成功。 • *FAILED*：执行失败。 • *RUNNING*：执行中。
scan_rows	从数据源表扫描的行数。可用于判断目标查询SQL是否扫描了大量数据。
scan_size	从数据源表扫描的数据量。单位：Byte。
scan_time	从底层存储扫描数据消耗的时间累加值。单位：ms。 该时间包含了使用索引过滤的时间，以及扫描符合检索条件的慢查询明细数据的时间，因此可用于初步判断过滤条件是否合适、过滤条件是否下推、索引是否生效等。
return_row_counts	查询输出的总行数。可用于判断一条查询最终总的输出量。
planning_time	查询在AnalyticDB MySQL版的优化器中使用最优执行计划计算的耗时。单位：ms。
wall_time	执行一条SQL时使用的物理算子消耗的CPU时间总和。可用来判断计算量较大的SQL。单位：ms。
sql	用户提交的SQL查询语句。
queued_time	查询的排队时间。单位：ms。
access_ip	查询提交使用的客户端IP地址。

常用的查询

- 查看慢查询表中保存的查询个数，语句如下：

```
SELECT count(*) FROM information_schema.kepler_slow_sql_merged;
```

- 查看慢查询表中已保存的查询SQL的最早执行开始时间，语句如下：

```
SELECT MIN(start_time) from information_schema.kepler_slow_sql_merged;
```

- 查看目标时间段内慢查询的相关属性，如峰值内存、扫描行数、扫描数据量、相关SQL、耗时以及执行开

始时间等。

SQL语句如下：

```
SELECT peak_mem, scan_rows, scan_size, sql, time, start_time
FROM information_schema.kepler_slow_sql_merged
WHERE start_time > ${timeStart}
  AND start_time < ${timeEnd};
```

查看执行开始时间在 `2021-08-20 01:41:33` 至 `2021-08-20 02:41:33` 范围内的慢查询的相关属性，语句示例如下：

```
SELECT peak_mem, scan_rows, scan_size, sql, time, start_time
FROM information_schema.kepler_slow_sql_merged
WHERE start_time > '2021-08-20 01:41:33'
  AND start_time < '2021-08-20 02:41:33';
```

- 查看符合模糊检索条件的慢查询的相关属性，例如峰值内存、扫描行数、扫描数据量、耗时以及执行开始时间等。

  SQL命令如下：

  ```
  SELECT peak_mem, scan_rows, scan_size, time, start_time
  FROM information_schema.kepler_slow_sql_merged
  WHERE sql LIKE '%date_test%';
  ```

- 查看执行耗时在目标范围内的慢查询相关属性，例如峰值内存、扫描行数、扫描数据量、耗时、执行开始时间，SQL语句详情等。

  SQL命令如下：

  ```
  SELECT peak_mem, scan_rows, scan_size, time, start_time, sql
  FROM information_schema.kepler_slow_sql_merged
  WHERE time > ${min_time}
    AND time < ${max_time};
  ```

  > **说明** 慢查询表默认仅保留执行时长超过1秒的SQL，因此 `min_time` 的取值需大于等于1秒。

  示例如下：

  ```
  SELECT peak_mem, scan_rows, scan_size, time, start_time, sql
  FROM information_schema.kepler_slow_sql_merged
  WHERE time > 20*1000
    AND time < 30*1000;
  ```

- 查看峰值内存在目标范围内的慢查询相关属性，例如峰值内存、扫描行数、扫描数据量、耗时、执行开始时间，SQL语句详情等。

  SQL命令如下：

  ```
  SELECT peak_mem, scan_rows, scan_size, time, start_time, sql
  FROM information_schema.kepler_slow_sql_merged
  WHERE peak_mem > ${min_memory}
      AND peak_mem < ${max_memory};
  ```

  语句示例如下：

```
SELECT peak_mem, scan_rows, scan_size, time, start_time, sql
FROM information_schema.kepler_slow_sql_merged
WHERE peak_mem > 1 * 1024 * 1024 * 1024
  AND peak_mem < 10 * 1024 * 1024 * 1024;
```

- 根据客户端IP进行聚合分析。

  根据 `access_ip` 字段进行分组聚合，然后根据平均峰值内存最多的 `access_ip` 进行降序排列，SQL命令如下：

```
SELECT
  max(peak_mem),
  avg(peak_mem) avg_peak_mem,
  count(*),
  max(scan_rows),
  avg(time),
  access_ip
FROM
  information_schema.kepler_slow_sql_merged
group by
  access_ip
order by
  avg_peak_mem desc;
```

2.2. 调优查询

2.2.1. 概述

您可以在分析查询的基础上，对检索出的慢查询或资源消耗大的查询进行调优。性能调优需要结合目标查询的执行计划数、统计指标以及诊断结果来完成，其中SQL自诊断功能可以对SQL查询的Query、Stage和算子（Operator）级别的信息分别进行统计，再在统计信息的基础上进行诊断并提供调优建议。

2.2.2. SQL自诊断

2.2.2.1. 查看查询属性与诊断结果

AnalyticDB MySQL版的SQL诊断功能可以对SQL查询的Query、Stage和算子（Operator）级别的信息分别进行统计，再在统计信息的基础上进行诊断并提供调优建议。本文介绍如何在AnalyticDB MySQL版控制台上查看目标查询的属性与各级别的诊断结果。

查看查询属性

1. 进入**SQL诊断**页签。详细操作步骤，请参见进入SQL诊断页签。
2. 在**SQL诊断**页签的右上角，根据业务需要选择整体检索条件，例如检索**最近5分钟**内**耗时最长的100个**SQL查询。
3. 在检索结果的图表中，您可以选择如下任意一种方式进入查询详情页面：
   - **方式一**

     在**查询监控**区域的趋势图中，每个色块代表一次SQL查询，将鼠标放在色块上可以查看目标查询的相关信息（例如查询开始或结束时间），单击**详情**即可进入目标查询的详情页面。

- **方式二**

  在**SQL列表**页签下，单击目标查询左侧**操作**栏中的**诊断**，即可进入目标查询的详情页面。

4. 在查询详情页面的**查询属性**区域，可以查看目标查询语句的执行情况，例如**扫描数据**、**返回数据**、**峰值内存**等。

   单击页面右上角**返回**即可回到查询趋势图所在页，查看其它查询的相关信息。

查看诊断结果

1. 根据查看查询属性中的步骤进入查询详情页面。
2. 在查询详情页面，单击**执行计划**页签来查看目标查询的执行计划树图。您可以通过执行计划树来查看Query、Stage和Operator级别的诊断结果：

   - **查看Query级别的诊断结果**

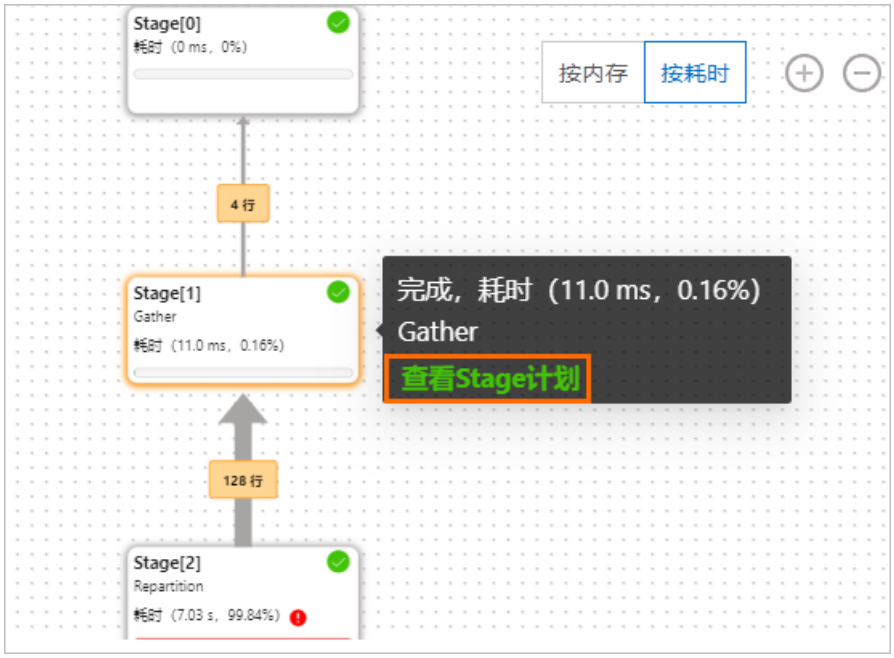
     在执行计划树右侧，单击**查询级别诊断结果**，即可直接查看Query级别的诊断结果详情。更多关于Query级别诊断结果的详情，请参见Query级别诊断结果。

- **查看Stage级别的诊断结果**

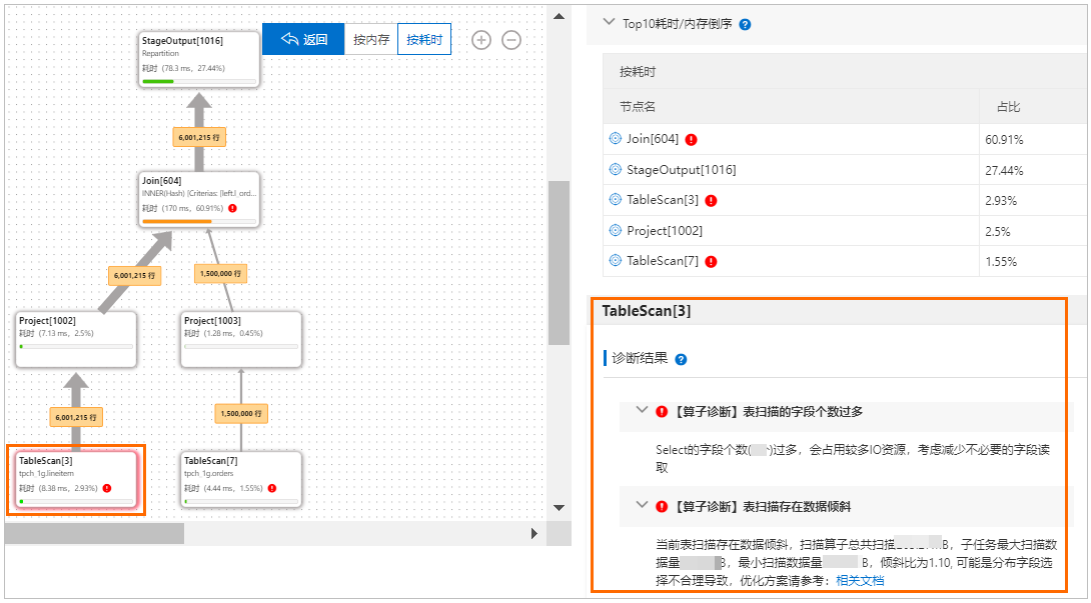
  单击执行计划树中某个Stage（如**Stage[2]**），即可在右侧查看对应Stage的**诊断结果**详情。更多关于Stage级别诊断结果的详情，请参见Stage级别诊断结果。

- **查看算子（Operator）级别的诊断结果**

  a. 将鼠标移动到目标Stage上，在弹出的信息框中单击**查看Stage计划**，进入对应Stage的计划详情页。



b. 在Stage计划详情页，单击执行计划树中某个算子（如TableScan[3]），即可在右侧查看对应算子的**诊断结果**详情。更多详情，请参见算子级别诊断结果。



2.2.2.2. Query级别诊断结果

AnalyticDB MySQL版的SQL诊断功能可以对SQL查询的Query、Stage和算子（Operator）级别的信息分别进行统计，再在统计信息的基础上进行诊断并提供调优建议。本文介绍如何查看和分析Query级别诊断结果。

诊断结果类型

> **说明** 查看Query级别诊断结果的方法，请参见查看诊断结果。

- 查询返回客户端的数据量较大
- 查询消耗的内存资源较大
- 查询生成的Stage个数较多
- 查询读取的数据量较大

查询返回客户端的数据量较大

- **问题**

  大量数据返回到客户端会导致慢查询，还会占用部分网络前端资源。

  > **说明** 您可以在查询详情页面的**查询属性**区域查看**返回数据**的信息。查看方法，请参见查看查询属性。

- **建议**
  - 减少直接返回到客户端的数据量。例如您可以在查询时增加LIMIT关键字或增加查询的过滤条件。
  - 通过外表将数据导出至其他系统（例如OSS。详细操作步骤，请参见通过外表导出AnalyticDB MySQL数据至OSS）来减少返回到客户端的数据量。

查询消耗的内存资源较大

- **问题**

  查询消耗了大量内存，可能导致其他查询无法执行、执行变慢甚至影响到AnalyticDB MySQL版集群整体的稳定性。

  > **说明** 您可以在查询详情页面的**查询属性**区域查看**峰值内存**信息。查看方法，请参见查看查询属性。

- **建议**

  先定位查询消耗较大内存的原因，再通过AnalyticDB MySQL版SQL诊断中的执行计划找到消耗内存较多的Stage或者算子，更多详情，请参见消耗内存的慢查询和使用执行计划分析查询。

查询生成的Stage个数较多

- **问题**

  查询生成的Stage个数较多，会占用较多的网络资源，并且提升系统处理的复杂度，对集群整体稳定性有一定风险。

  > **说明** 更多关于Stage的信息，请参见影响查询性能的因素。

- **建议**
  - 在数据写入AnalyticDB MySQL版集群前，通过提前联表来减少集群中的表个数。
  - SQL中的Join数量越多，AnalyticDB MySQL版为查询生成的Stage数量越多。因此，您可以通过优化SQL减少Join数量，来减少因查询生成的Stage个数。
  - 使用物化视图进行查询时，AnalyticDB MySQL版能通过复用Stage来减少因查询生成的Stage数量。更多关于物化视图的详情，请参见物化视图。

查询读取的数据量较大

- **问题**

  查询读取的数据量较大，会占用较多的磁盘IO资源，影响其他查询或者数据的写入过程。

  > **说明** 您可以在查询详情页面的**查询属性**区域查看**扫描数据**的信息。查看方法，请参见查看查询属性。

- **建议**

  先找到读取数据量较大的Stage以及相关的表扫描算子（TableScan）。您可以在AnalyticDB MySQL版SQL诊断的Stage层或算子层执行计划中的**统计信息**区域查看对应Stage**扫描行数**、**扫描量**，或TableScan算子的**输入行数**和**输入大小**来判断Stage和TableScan算子的扫描数据量。更多详情，请参见Stage统计信息和算子统计信息。

  找到扫描数据量较大的表扫描算子后，您可以考虑如下方式进行调优：
  - 在查询中增加AND过滤条件。
  - 调整已有的过滤条件，减少过滤后的数据量。
  - 检查是否存在没有下推的过滤条件。若存在，请参见过滤条件没有下推中的建议进行优化。

2.2.2.3. Stage级别诊断结果

AnalyticDB MySQL版的SQL诊断功能可以对SQL查询进行Query、Stage和算子（Operator）级别的信息统计，再在统计信息的基础上进行诊断并提供调优建议。本文介绍如何查看和分析Stage级别诊断结果。

诊断结果类型

> **说明** 查看Stage级别诊断结果的方法，请参见查看诊断结果。

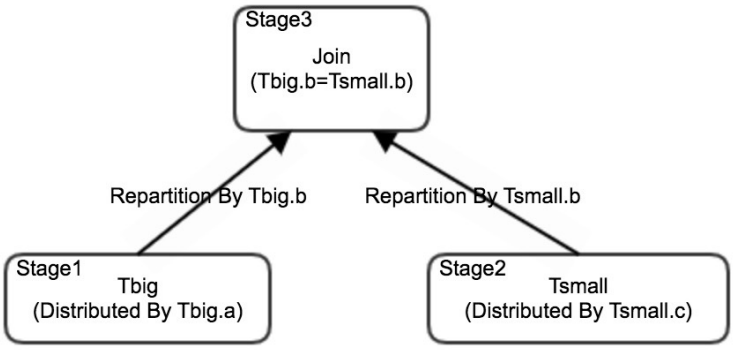
- 较大的数据量被广播
- Stage输入数据倾斜
- Stage输出数据倾斜

较大的数据量被广播

- **问题**

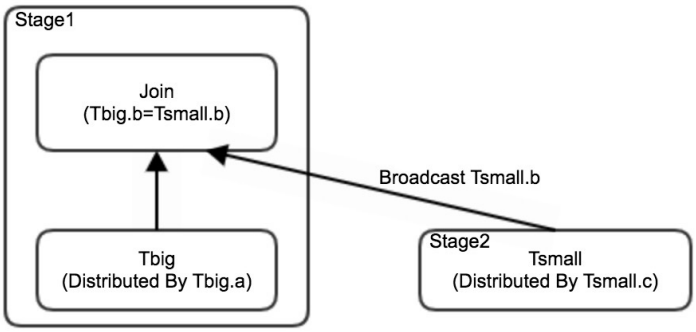
  广播（Broadcast）是在两个相邻的Stage间，上游向下游Stage传输数据时所用的一种方法（更多详情，请参见数据输出类型）。如果某个Stage广播了较多数据，可能会导致查询占用较大的峰值内存。

  某个Stage中的数据被广播，一般是因为这些广播后的数据会作为Join操作的右表（Builder端）来在内存中构建哈希表，所以右表越小越好。在高并发查询的场景下，这样有利于减少节点间的网络连接，提升系统整体稳定性。对于Join条件存在数据倾斜的场景，如果不广播小表，那么会出现如下图的执行流程：



假设上图中的表`Tsmall`在`b`字段上存在严重数据倾斜，那么当表`Tbig`以`a`字段均匀地分布在AnalyticDB MySQL版的存储节点上时，对`Tbig`表的重分布会存在处理时间长尾，而且在下游Stage执行Join时也会存在长尾。

如果`Tbig`表不做重分布，而只广播`Tsmall`表，会有如下的执行流程：



如上图所示，只广播`Tsmall`这张小表，可以缓解数据倾斜带来的处理长尾问题。在ADB中，默认只有当对右表的估计行数少于1万行才会有可能选择broadcast join，如下

- **建议**
  - 首先需要判断当前的广播操作是否合理。
  - 在某些场景下，例如统计信息过期，会导致预估的表大小有偏差，从而导致广播了大量数据，此时可以考虑使用Hint `JOIN_DISTRIBUTION_TYPE=repartitioned` 来关闭数据的广播功能。

Stage输入数据倾斜

- **问题**

  导致Stage输入数据倾斜的可能原因如下：
  - 建表时选择的分布字段不合理，导致Stage中的某个数据扫描算子在扫描数据时存在倾斜。
  - 上游Stage的数据通过网络传输到当前Stage时存在倾斜。
- **建议**
  - 建表时选择合适的分布字段。更多详情，请参见分布字段合理性诊断。
  - 查看上游Stage是否存在Stage输出数据倾斜问题。更多详情，请参见Stage输出数据倾斜。

Stage输出数据倾斜

- **问题**

  Stage输出数据倾斜会导致当前Stage处理耗时不均匀，存在长尾；如果下游Stage处理过程复杂，也会导致下游Stage在处理数据时存在长尾，最终都会影响查询整体性能。
- **建议**

  通过诊断结果中提示的字段名来判断是否是这些字段存在数据倾斜（如出现大量空值）。

2.2.2.4. 算子级别诊断结果

AnalyticDB MySQL版的SQL诊断功能可以对SQL查询进行Query、Stage和算子（Operator）级别的信息统计，再在统计信息的基础上进行诊断并提供调优建议。本文介绍如何查看和分析算子级别诊断结果。

诊断结果类型

> **说明** 查看算子级别诊断结果的方法，请参见查看诊断结果。

- 聚合算子聚合度低
- 过滤条件没有下推
- Join存在数据膨胀
- Join的右表过大
- 存在Cross Join
- 扫描算子读取字段个数较多
- 表扫描数据量倾斜
- 索引不高效

聚合算子聚合度低

- **问题**

聚合算子的聚合度一般指GROUP BY分组聚合操作中的输入数据量和输出数据量的比值（即InputDataSize和OutputDataSize的比值），该值越小表明聚合度越低，聚合效果越差。在AnalyticDB MySQL版中，分组聚合操作一般分为两步：初步聚合（PARTIAL）和最终聚合（FINAL）。如果聚合算子分组的个数较多，会导致聚合度低，那么在第一步的初步聚合操作中，不但不能减少网络数据传输，反而会消耗大量的计算资源。

- **建议**

  可以考虑跳过初步聚合操作，直接在各个节点间对数据进行重分布，之后直接进行一次最终聚合。更多详情，请参见分组聚合查询优化。

过滤条件没有下推

- **问题**

  AnalyticDB MySQL版在存储数据时默认对表的全部字段创建了索引，您可以在查询时使用这些索引来加速数据的过滤。但在如下场景中AnalyticDB MySQL版不会将过滤条件下推：

  - 查询语句中使用了 `no_index_columns` 或 `filter_not_pushdown_columns` Hint，或集群使用了adb_config filter_not_pushdown_columns配置，导致过滤条件下推功能被关闭。
  - 过滤条件中使用了函数（包括 `cast` 操作符）。
  - 过滤条件中的相关字段没有索引（例如建表时指定了 `no_index` 关键字，或建表后执行 `DROP INDEX` 删除了索引）。

- **建议**
  - 如果是查询语句使用了Hint或集群使用了配置，导致的过滤条件下推功能关闭，请检查使用该Hint或配置的原因并判断是否可以取消该Hint或配置。更多详情，请参见过滤条件不下推。
  - 如果是使用了函数，可以考虑是否直接使用该函数写入数据，而在查询时去掉函数。
  - 如果是过滤条件中的相关字段没有索引导致过滤条件没有下推，那么需要检查没有索引的原因。

Join存在数据膨胀

- **问题**

  Join的数据膨胀率是Join的输出行数与Join的输入行数的比值，其中Join的输入行数为左表行数与右表行数之和。对于合理Join条件，一般Join的输出行数会小于输入行数，如果Join的输出行数大于输入行数，那么会存在Join数据膨胀的问题，Join数据膨胀会导致较多的计算资源和内存资源被占用，导致查询较慢。

- **建议**
  - 如果是数据本身特征导致的Join数据膨胀，例如左右表都大量存在的相同的值，可以考虑在表过滤阶段将这些相同值都过滤掉不参与Join。
  - 如果是不优的Join顺序导致的数据膨胀，可以考虑手动调整Join顺序。调整方法，请参见手动调整Join顺序。

Join的右表过大

- **问题**

  AnalyticDB MySQL版中的Join右表一般指Builder表，用于构建内存中的Hash结构或者Set结构。右表过大，可能会占用较多的内存资源，影响集群整体稳定性。导致Join右表过大的可能原因如下：

  - SQL中有Left Join。由于Left Join的右表在执行过程中必须作为Builder表，所以如果Left Join的右表过大，必然占用较多内存资源。
  - AnalyticDB MySQL版在预估左右表数据量时，由于统计信息过期等原因导致估计错误。

- **建议**
  - 建议将Left Join优化改写成Right Join。改写方法，请参见Left Join优化改写为Right Join。
  - 如果是统计信息过期导致的，可以提交工单联系技术支持。

存在Cross Join

- **问题**

  Cross Join，即没有Join条件的Join操作，输出的行数是左右两表行数的乘积。如果左右表都较大，会极大地影响AnalyticDB MySQL版集群的稳定性。

- **建议**

  考虑增加Join条件，消除Cross Join。

扫描算子读取字段个数较多

- **问题**

  扫描算子会在AnalyticDB MySQL版的存储层进行数据的过滤和明细数据的读取，如果SELECT的字段个数较多，需要读取的明细数据也较多，那么就会占用较大的磁盘I/O资源，影响AnalyticDB MySQL版集群整体稳定性。

- **建议**

  可以优化SQL语句，减少SELECT语句中不必要的字段。

表扫描数据量倾斜

- **问题**

  AnalyticDB MySQL版是分布式执行架构，大表的数据一般需要指定分布字段，数据写入时根据分布字段分散到不同的存储节点上。如果分布字段的值分布不均匀，那么数据存储在各个节点上时也会不均匀，最终导致数据读取时，各个节点在读取数据时存在时间上的长尾，影响最终的查询效果。

- **建议**

  通过选择合适的分布字段来减少表扫描数据量的倾斜。优化方法，请参见分布字段合理性诊断。

索引不高效

- **问题**

  AnalyticDB MySQL版在使用索引进行数据过滤时，若需要过滤的字段过滤度（即过滤算子的输出数据量和输入数据量的比值）较低时，使用索引不一定能获得预期的过滤效果。

- **建议**

  可以考虑不下推过滤条件，而在计算节点中直接执行过滤操作。更多详情，请参见过滤条件不下推。

2.2.3. Join调优

2.2.3.1. Left to right

Left join是实践中常用的一种表关联方式，由于Hash Join实现会以右表做build，且left Join不会做左右表的重新排序，在右表数据量很大时会造成执行慢、消耗过多内存资源等多个问题。本文以具体示例介绍什么时候应该用right join替代left join。

背景信息

AnalyticDB MySQL版默认使用Hash Join进行表关联。Hash Join在实现时会用右表构建哈希表，这是一个比较消耗资源的计算过程，由于outer join（包括left join，right join）不同于inner join，从语义上不能交换左右表顺序，因此在右表数据量大的场景下，会出现执行慢、内存资源消耗大的情况，在极端场景下（右表数据量很大）还会影响集群的性能，或执行时直接报错Out of Memory Pool size pre cal。此时，可以使用本章节提供的优化方法来减少资源消耗。

使用场景

通过修改SQL语句或者加hint的方式，可以将left join调整为right join，原left join的左表会变为右表来构建哈希表。这时如果右表过大也会对性能有影响，因此，建议在left join左表较小，右表较大的场景下进行优化。

较小、很大的概念是相对的，和关联列、集群资源等都有关系。在实践中，我们可以通过Explain analyze查看执行计划的相关参数，通过关注PeakMemory、WallTime等参数的变化来判断是否应该使用right join。

使用方法

通常有以下两种方法可以把left join调整为right join：

- 直接修改SQL，例如将a left join b on a.col1 = b.col2 改为b right join a on a.col1 = b.col2。
- 通过加hint指定优化器根据cost把left join转为right join。这种用法中，优化器会根据左右表的估算大小来决定是否把left join转为right，例如/*+LEFT_TO_RIGHT_ENABLED=true,CASCADES_OPTIMIZER_ENABLED=false*/。

示例

如下示例中，nation是一个25行的小表，customer是一个15000000行的大表，通过explain analyze查看一条包含left join的SQL的执行计划。

```
explain analyze
select
  count(*)
from
  nation t1
  left join customer t2 on t1.n_nationkey = t2.c_nationkey
```

可以看到，进行join计算的stage2的计划如下。其中，Left Join这个算子中包含如下信息：

- PeakMemory: 515MB (93.68%), WallTime: 4.34s (43.05%)：PeakMemory的占比高达93.68%，可以判断left join为整个SQL的性能瓶颈。
- Left (probe) Input avg.: 0.52 rows；Right (build) Input avg.: 312500.00 rows：即右表为大表，左表为小表。

这种场景下，我们可以将left join转为right，来优化这条SQL。

```
Fragment 2 [HASH]
    Output: 48 rows (432B), PeakMemory: 516MB, WallTime: 6.52us, Input: 15000025 rows (200.
27MB); per task: avg.: 2500004.17 std.dev.: 2410891.74
    Output layout: [count_0_2]
    Output partitioning: SINGLE []
    Aggregate(PARTIAL)
    │   Outputs: [count_0_2:bigint]
    │   Estimates: {rows: ? (?)}
    │   Output: 96 rows (864B), PeakMemory: 96B (0.00%), WallTime: 88.21ms (0.88%)
    │   count_2 := count(*)
    └─ LEFT Join[(`n_nationkey` = `c_nationkey`)][$hashvalue, $hashvalue_0_4]
       │   Outputs: []
       │   Estimates: {rows: 15000000 (0B)}
       │   Output: 30000000 rows (200.27MB), PeakMemory: 515MB (93.68%), WallTime: 4.34s (4
3.05%)
       │   Left (probe) Input avg.: 0.52 rows, Input std.dev.: 379.96%
       │   Right (build) Input avg.: 312500.00 rows, Input std.dev.: 380.00%
       │   Distribution: PARTITIONED
       ├─ RemoteSource[3]
       │      Outputs: [n_nationkey:integer, $hashvalue:bigint]
       │      Estimates:
       │      Output: 25 rows (350B), PeakMemory: 64KB (0.01%), WallTime: 63.63us (0.00%)
       │      Input avg.: 0.52 rows, Input std.dev.: 379.96%
       └─ LocalExchange[HASH][$hashvalue_0_4] ("c_nationkey")
          │   Outputs: [c_nationkey:integer, $hashvalue_0_4:bigint]
          │   Estimates: {rows: 15000000 (57.22MB)}
          │   Output: 30000000 rows (400.54MB), PeakMemory: 10MB (1.84%), WallTime: 1.81s (
17.93%)
          └─ RemoteSource[4]
                 Outputs: [c_nationkey:integer, $hashvalue_0_5:bigint]
                 Estimates:
                 Output: 15000000 rows (200.27MB), PeakMemory: 3MB (0.67%), WallTime: 191.3
2ms (1.90%)
                 Input avg.: 312500.00 rows, Input std.dev.: 380.00%
```

通过修改SQL的方式实现left to right：

```
select
  count(*)
from
  customer t2
  right join nation t1 on t1.n_nationkey = t2.c_nationkey
```

通过加hint的方式实现left join to right：

```
/*+LEFT_TO_RIGHT_ENABLED=true,CASCADES_OPTIMIZER_ENABLED=false*/
select
  count(*)
from
  nation t1
  left join customer t2 on t1.n_nationkey = t2.c_nationkey
```

上述任意一种SQL，执行explain analyze后可以看到，在执行计划中，left Join变为了right Join，可以判断hint是生效的。并且调整后PeakMemory: 889KB (3.31%)，PeakMemory从515MB下降到889KB，已经不是计算热点。

```
Fragment 2 [HASH]
    Output: 96 rows (864B), PeakMemory: 12MB, WallTime: 4.27us, Input: 15000025 rows (200.2
7MB); per task: avg.: 2500004.17 std.dev.: 2410891.74
    Output layout: [count_0_2]
    Output partitioning: SINGLE []
    Aggregate(PARTIAL)
    │   Outputs: [count_0_2:bigint]
    │   Estimates: {rows: ? (?)}
    │   Output: 192 rows (1.69kB), PeakMemory: 456B (0.00%), WallTime: 5.31ms (0.08%)
    │   count_2 := count(*)
    └─ RIGHT Join[(`c_nationkey` = `n_nationkey`)][$hashvalue, $hashvalue_0_4]
       │   Outputs: []
       │   Estimates: {rows: 15000000 (0B)}
       │   Output: 15000025 rows (350B), PeakMemory: 889KB (3.31%), WallTime: 3.15s (48.66%
)
       │   Left (probe) Input avg.: 312500.00 rows, Input std.dev.: 380.00%
       │   Right (build) Input avg.: 0.52 rows, Input std.dev.: 379.96%
       │   Distribution: PARTITIONED
       ├─ RemoteSource[3]
       │      Outputs: [c_nationkey:integer, $hashvalue:bigint]
       │      Estimates:
       │      Output: 15000000 rows (200.27MB), PeakMemory: 3MB (15.07%), WallTime: 634.81m
s (9.81%)
       │      Input avg.: 312500.00 rows, Input std.dev.: 380.00%
       └─ LocalExchange[HASH][$hashvalue_0_4] ("n_nationkey")
          │   Outputs: [n_nationkey:integer, $hashvalue_0_4:bigint]
          │   Estimates: {rows: 25 (100B)}
          │   Output: 50 rows (700B), PeakMemory: 461KB (1.71%), WallTime: 942.37us (0.01%)
          └─ RemoteSource[4]
                 Outputs: [n_nationkey:integer, $hashvalue_0_5:bigint]
                 Estimates:
                 Output: 25 rows (350B), PeakMemory: 64KB (0.24%), WallTime: 76.34us (0.00%
)
                 Input avg.: 0.52 rows, Input std.dev.: 379.96%
```

2.2.3.2. STRAIGHT_JOIN

本章节内容适用于版本号为3.1.3及以上的AnalyticDB for MySQL实例。STRAIGHT_JOIN与JOIN类似，只是不会调整执行计划中的左右表顺序。可用于联接优化器以次优顺序处理表的情况。

语法结构

```
join_table:
    table_reference STRAIGHT_JOIN table_factor [join_condition]
table_reference:
    table_factor
  | join_table
table_factor:
    tbl_name [alias]
  | table_subquery alias
  | ( table_references )
join_condition:
    ON expression
```

> **说明**
>
> - STRAIGHT_JOIN执行结果与INNER JOIN执行结果相同。
> - a STRAIGHT_JOIN b 语法执行时a表会做连接的左表，b表会做连接的右表，且a和b直接连接，优化器不会再做表连接顺序的优化。

使用场景

本语法用于指定INNER JOIN执行时的左右表，可用于表大小明确，或发现AnalyticDB for MySQL执行计划选择的INNER JOIN左右表不合理时的业务场景。

在默认hash join场景下，我们选择大表在左小表在右，在AnalyticDB for MySQL场景下会达到较好性能。如果指定nested loop join，则应选择小表在左大表在右。

如下第一句SQL，如果已知最佳连接顺序为 region、nation、customer，则可以将第一句SQL改写为第二句SQL，指定连接顺序及左右表，提高查询效率。

```
SELECT count(*)
       FROM    customer, nation, region
       WHERE c_nationkey = n_nationkey
       AND n_regionkey = r_regionkey
       AND r_name = 'ASIA';
```

```
SELECT count(*)
       FROM    region STRAIGHT_JOIN nation on n_regionkey = r_regionkey
       STRAIGHT_JOIN customer ON c_nationkey = n_nationkey
       WHERE r_name = 'ASIA';
```

示例

本示例中，STRAIGHT_JOIN会生成与INNER JOIN语法相同的结果，接着优化器会决定这个连接生成的结果表和customer表连接时的左右表。

```
SELECT count(*)
       FROM    region STRAIGHT_JOIN nation on n_regionkey = r_regionkey
       INNER JOIN customer ON c_nationkey = n_nationkey
       WHERE r_name = 'ASIA';
```

2.2.3.3. 手动调整Join顺序

本文介绍如何通过Hint实现手动调整Join顺序。

功能介绍

AnalyticDB MySQL版支持复杂的联表（Join）查询且默认提供了自动调整Join顺序的功能。但查询语句和表的过滤条件随时可能发生变化，而且如果数据特征复杂，自动调整Join顺序功能不一定在所有场景下都能很好地预估查询特征，并选择出最优的Join顺序。而不优的Join顺序可能造成中间结果集膨胀中间结果集数据量较大、内存消耗大等问题，进而影响查询性能。

为解决上述问题，AnalyticDB MySQL版支持通过Hint `/*reorder_joins*/` 来开启或关闭调整Join顺序功能，其中：

- `/*reorder_joins=true*/`：开启自动调整Join顺序功能。开启后，系统会自动调整Join顺序。AnalyticDB MySQL版默认开启调整Join顺序功能，因此在执行SQL查询时无需使用该Hint也会自动调整Join顺序。
- `/*reorder_joins=false*/`：关闭自动调整Join顺序功能。关闭后，您可以根据查询的数据特征手动调整Join顺序，让查询直接根据SQL书写方式中的Join顺序来执行。

> **说明** `/*reorder_joins*/` 为会话级别的Hint，仅对目标SQL查询语句生效。

调整方法

- **调整前**
  - **查询语句**

    原始的Query10语句如下：

    > **说明**
    > - 本文以TPC-H中的Query10为例介绍手动调整Join顺序的方法及查询效果。更多关于TPC-H的详细说明，请参见TPC-H。
    > - 由于AnalyticDB MySQL版的自动调整Join顺序功能默认开启，在执行如下查询语句时使用了Hint `/*reorder_joins=false*/` 来模拟Join顺序不优的场景。

```
SELECT   c_custkey,
         c_name,
         Sum(l_extendedprice * (1 - l_discount)) AS revenue,
         c_acctbal,
         n_name,
         c_address,
         c_phone,
         c_comment
FROM     customer c,
         orders o,
         lineitem l,
         nation n
WHERE    c_custkey = o_custkey
AND      l_orderkey = o_orderkey
AND      o_orderdate >= date '1993-10-01'
AND      o_orderdate <  date '1993-10-01' + INTERVAL '3' month
AND      l_returnflag = 'R'
AND      c_nationkey = n_nationkey
GROUP BY c_custkey,
         c_name,
         c_acctbal,
         c_phone,
         n_name,
         c_address,
         c_comment
ORDER BY revenue DESC
LIMIT    20;
```

- **Join顺序**

  如果按照如上SQL的书写方式，Join的顺序应该是：

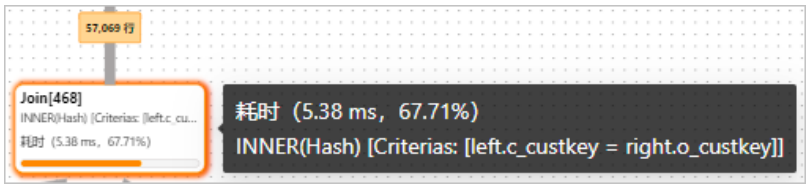
```
customer JOIN orders JOIN lineitem JOIN nation;
```

- **查询结果**

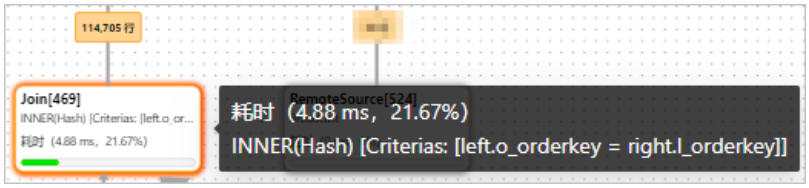
  此时，执行计划中各个JOIN的临时结果如下：

  > **说明** 查看执行计划的步骤，请参见使用执行计划分析查询。

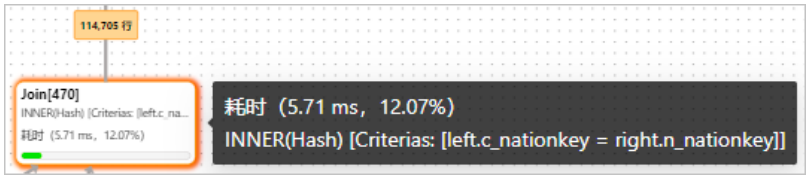
  a. 表 `customer` Join表 `orders` ，输出57069行数据至临时结果集 `tmp1` 。

  b. 临时结果集 `tmp1` Join表 `lineitem` ，输出114705行数据至另一个临时结果集 `tmp2` 。

  c. 临时结果集 `tmp2` Join表 `nation` ，输出114705行数据作为最终结果。

  3次Join累积输出结果行数为：57069 + 114705 + 114705 = 286479。

- **调整后**

- **查询语句**

  在SQL语句中添加 `/*reorder_joins=false*/` 的Hint来关闭AnalyticDB MySQL版的自动调整Join顺序功能，并手动调整Join的顺序，调整顺序后的SQL语句如下：

```
/*reorder_joins=false*/
SELECT   c_custkey,
         c_name,
         Sum(l_extendedprice * (1 - l_discount)) AS revenue,
         c_acctbal,
         n_name,
         c_address,
         c_phone,
         c_comment
FROM     customer c,
         orders o,
         nation n,
         lineitem l
WHERE    c_custkey = o_custkey
AND      c_nationkey = n_nationkey
AND      l_orderkey = o_orderkey
AND      o_orderdate >= date '1993-10-01'
AND      o_orderdate <  date '1993-10-01' + INTERVAL '3' month
AND      l_returnflag = 'R'
GROUP BY c_custkey,
         c_name,
         c_acctbal,
         c_phone,
         n_name,
         c_address,
         c_comment
ORDER BY revenue DESC
LIMIT    20;
```

- **Join顺序**

  如果按照如上SQL的书写方式，Join的顺序应该是：

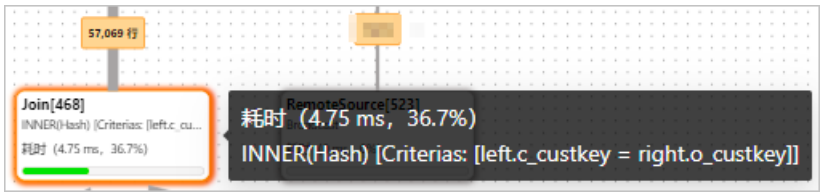
```
customer JOIN orders JOIN nation JOIN lineitem
```

- **查询结果**

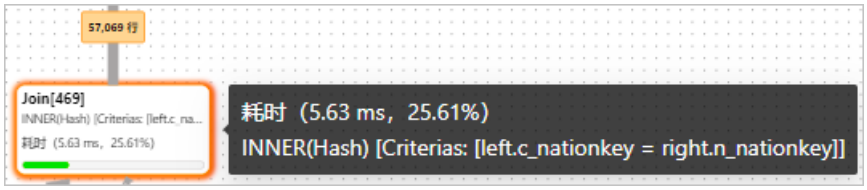
  此时，执行计划中各个JOIN的临时结果如下：

  > **说明** 查看执行计划的步骤，请参见使用执行计划分析查询。

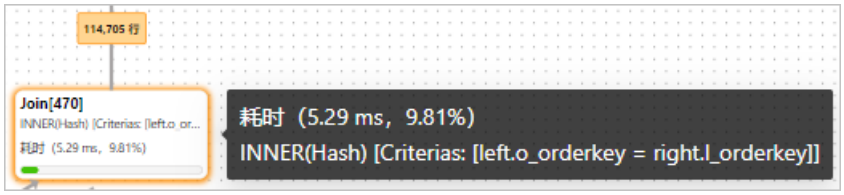
  a. 表 `customer` Join表 `orders` ，输出57069行数据至临时结果集 `tmp1` 。

  b. 临时结果集 `tmp1` Join表 `nation` ，输出57069行数据至临时结果集 `tmp2` 。

  c. 临时结果集 `tmp2` Join表 `lineitem` ，输出114705行数据作为最终结果。

  3次Join累积输出结果行数为：57069 + 57069 + 114705 = 228843。

  相较于Join顺序调整前（输出286479行数据），减少了20%的输出行数。从上述对比结果中可以看出，不同的Join顺序会影响中间临时结果集的大小，因此若发现AnalyticDB MySQL版的Join顺序不优，您可以手动调整Join顺序来提升SQL查询性能。

2.2.4. 过滤条件不下推

本文介绍过滤条件不下推的使用场景与方法。

功能介绍

云原生数据仓库AnalyticDB MySQL版在创建表时，默认为所有的字段创建了索引，使数据过滤的效率更高。然而在某些场景中，使用索引过滤数据不一定能得到较好的性能，甚至会影响整体性能。此时不建议继续使用索引进行数据过滤。虽然用户可以手动删除某些字段的索引，但这种做法可能导致需要使用索引时却没有索引可用的问题。云原生数据仓库AnalyticDB MySQL版过滤条件不下推功能，可以在查询级别或实例级别暂时蔽掉某些字段的过滤条件下推能力，带来更好整体查询收益。

以下场景不建议使用索引过滤数据：

- 数据唯一值少。数据唯一值较少，意味着数据经过过滤后返回的数据仍然很多，那么使用索引进行数据过滤的效果可能不一定好。
- 磁盘IO压力大。如果用户业务的查询特征是占用较多的IO资源，或者数据写入较多导致占用了较多IO资源，那么使用索引进行数据过滤时，存在磁盘IO资源的争抢，过滤效果也可能较差。
- 同时有多个条件下推，且下推的条件中有LIKE、字符串比较等比较复杂的操作时，会对存储节点相关资源

消耗很大，影响整体的性能。

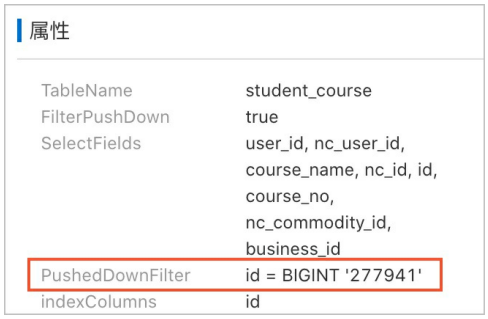
查看过滤条件是否下推

您可以通过执行页面查看过滤条件是否下推。

1. 在查询的**执行计划**页签，单击包含TableScan算子的Stage。

   > **说明** 进入**执行计划**页签的步骤，请参见查看诊断结果。

2. 单击**查看Stage计划**。
3. 在Stage计划页面，单击TableScan算子。
4. 在右侧**属性**中，查看是否显示**PushedDownFilter**属性。如果显示，表示该过滤条件已下推；否则表示没有下推的过滤条件。



> **说明** 通过执行计划，也可以确认过滤条件是否下推。
>
> - 弹性模式的集群，您可以查看下游Stage的执行计划中是否显示Filter算子。如果显示则表示该算子相关的过滤条件没有下推。
> - 预留模式的集群，您可以查看当前Stage计划中是否显示Filter算子。如果显示则表示该算子相关的过滤条件没有下推。

查询级别关闭特定字段的过滤条件下推能力

针对某个查询，使用Hint关闭某些字段的过滤条件下推。只对使用了Hint的查询生效，其他查询不受影响。

语法：

内核版本为3.1.4及以上，请使用下面的Hint：

```
/*+ filter_not_pushdown_columns=[${database}.${tableName}:${col1Name}|${col2Name}] */
```

内核版本为3.1.4以下，请使用下面的Hint：

```
/*+ no_index_columns=[${tableName}.${col1Name};${col2Name},${tableName1}.${col1Name}] */
```

> **说明** 查询内核版本的方法，请参见如何查看实例版本信息。

示例1：

以3.1.4及以上的内核版本为例，当前查询，包含字段`id`、`product`（均属于表`table01`）的过滤条件不会下推。

```
/*+ filter_not_pushdown_columns=[test01.table01:id|product] */
```

示例2：

以3.1.4以下的内核版本为例，当前查询，包含字段 `id` （属于表 `table02` ）、字段 `product` （属于表 `table02` ）、字段 `key` （属于表 `table03` ）的过滤条件不会下推。

```
/*+ no_index_columns=[table02.id;product,table03.key] */
```

集群级别关闭特定字段的过滤条件下推能力

您可以执行以下命令，对当前集群的所有查询，关闭特定字段的过滤条件下推能力。

语法：

内核版本为3.1.4及以上版本，请使用下方语句：

```
set adb_config filter_not_pushdown_columns=[${database}.${tableName}:${col1Name}|${col2Name
}]
```

内核版本为3.1.4以下，请使用下面的语句：

```
set adb_config no_index_columns=[${tableName}.${col1Name};${col2Name},${tableName1}.${col1N
ame}]
```

> **说明** 查询内核版本的方法，请参见如何查看实例版本信息。

示例：

以3.1.4及以上的内核版本为例，当前实例的所有查询中，只要过滤条件包含字段 `id` （属于数据库 `test02` 的表 `table02` ），该过滤条件就不下推。

```
set adb_config filter_not_pushdown_columns=[test02.table02:id]
```

2.2.5. 分组聚合查询优化

本文介绍如何在AnalyticDB MySQL版中对分组聚合查询进行优化。

分组聚合流程

AnalyticDB MySQL版是分布式数据仓库，其分组聚合查询默认分为两步：

1. 完成数据的局部（PARTIAL）聚合。

   局部聚合节点只需要占用少量内存，聚合过程为流式过程，数据不会堆积在局部聚合节点上。

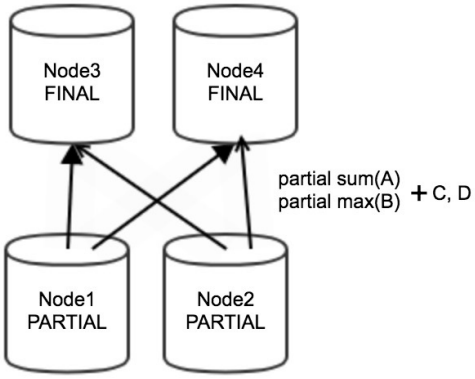
2. 局部聚合完成后，数据根据分组字段进行节点间的数据重分布，执行最终（FINAL）聚合。

   局部聚合后的结果会通过网络传输到下游Stage的节点（更多关于Stage的信息，请参见影响查询性能的因素）。因为数据已经经过了局部聚合，所以需要网络传输的数据较少，网络压力较小。数据重分布完成后，执行最终聚合，在最终聚合节点，需要把一个分组的值及其聚合状态维护在内存中，直到所有数据处理完成，以确保某个特定的分组值没有新的数据需要处理，所以最终聚合节点可能会占用较大的内存空间。

例如执行以下的SQL分组聚合语句。

```
SELECT sum(A), max(B) FROM tb1 GROUP BY C,D;
```

上述语句在进行分组聚合时，数据会首先在上游Stage的Node1和Node2节点中执行局部聚合，局部聚合的结果是partial sum(A)、partial sum(B)、C、D。局部聚合的结果会通过网络传输到下游Stage的Node3和Node4节点中进行最终聚合。流程图如下。



通过Hint优化分组聚合

- **适用场景**

  在大多数场景下，两步聚合可以在内存和网络资源之间实现较好的平衡，但在分组聚合的分组数较多（即GROUP BY字段的唯一值较多）等特殊场景下，两步聚合不一定是最好的处理方法。

  例如，在需要使用手机号码或用户ID进行分组的场景下，如果依旧使用典型的两步聚合方式，那么在局部聚合阶段，可以被聚合的数据较少，但是局部聚合流程依旧会执行（例如计算分组的HASH值、去重以及执行聚合函数）。由于分组数多，局部聚合阶段并没有减少网络传输的数据量，却消耗了很多计算资源。

- **使用方法**

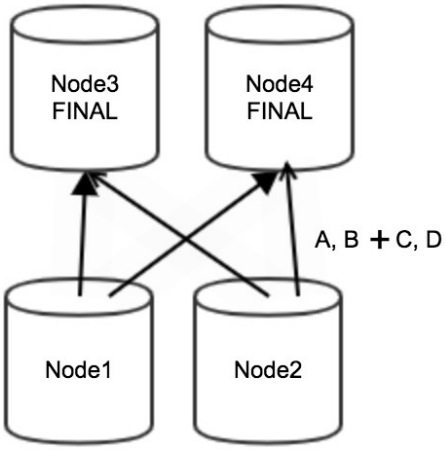
  为解决上述聚合度较低的分组聚合查询问题，您可以在执行查询时添加Hint `/*aggregation_path_type=single_agg*/` 来跳过局部聚合，直接进行最终聚合，来减少不必要的计算开销。

  > **说明** 如果在查询SQL中使用了 `/*aggregation_path_type=single_agg*/` Hint，那么该SQL中所有的分组聚合查询都会使用这个特定的优化流程，因此最佳的方式是先分析原始执行计划中的聚合算子的特点，并评估该Hint带来的收益，最后决定是否使用该优化方案。

- **优化说明**

  聚合度较低的情况下，上游Stage的Node1和Node2节点进行局部聚合并没有减少网络传输的数据量，却消耗了很多计算资源。

  优化后Node1和Node2节点不需要进行局部聚合，全部数据（即A、B、C、D）直接由下游Stage的Node3和Node4节点进行最终聚合，从而减少了计算量，流程图如下。



说明 该优化不一定能起到优化内存使用的目的，因为在聚合度较低的情况下，数据还是会大量的积攒在内存中进行去重和聚合以确保某个分组值的数据全部处理完成。

2.3. 常见问题以及改进措施

由于数据分布和查询复杂度等因素，可能出现查询性能不符合预期的情况，检查查询的执行计划是重要的问题排查方式之一。

常见计划问题

- Join Method以及Inner和Outer表

  根据Join Method选择Inner和Outer表， 一般情况下AnalyticDB MySQL版自动选择Join的左右表，您也可以自行检查左右表的选择是否合理。

  Hash Join中，右表创建Hash，左表去右表中查找符合条件的数据，一般右表要尽量小于左表，以减少创建Hash表的开销以及Hash表的大小。您可以通过检查Join表过滤后的大小来查看对应的左右表选择是否合理。但由于还有多表Join的中间结果，以及Join Type等因素影响，Join的左右表的选择也不能单纯依赖表过滤后的大小来选择。

- Join Order

  Join Order的优化是优化器的核心挑战之一，也是经典的NP-hard问题。AnalyticDB MySQL版优化器对于不同的负载提供两种Join Reordering策略。Left Deep Tree（LDT）一般适用于较简单查询场景，Bushy Tree（BT）适用较复杂的负载查询。Bushy Tree一般会将过滤效果更好的表放在前面，对于复杂查询能够生成更优的Join Order。

  对于复杂查询，人为确认最佳Join Order非常困难。Join表数量较大时，需要资深的专家进行调优。建议将过滤效果更好的表放在Join Sequence之前，可以更早更快的过滤掉不需要的数据。

- 分布式Aggregation

  AnalyticDB MySQL版提供分布式聚合计算能力，可以根据计算数据量分步做聚合计算。一般情况下，AnalyticDB MySQL版的优化器可以选择最佳聚合计算计划，但在数据倾斜比较严重等场景下，优化器对于聚合数据分布估算的误差会比较大，从而造成聚合计算性能问题。例如，一般AnalyticDB MySQL版会选择两阶段聚合计算，在各个计算节点本地做一次部分聚合（Partial Aggregation），减少聚合计算数据量，然后再根据聚合列Reshuffle做一次Final Aggregation。一般情况下，部分聚合可以显著减少聚合计算的数据量，但如果遇到数据倾斜严重，或者部分聚合列比较Unique从而不能减少聚合计算数量的情况下，部分聚合反而会带来额外的性能开销而非收益。

- 其它问题

  本文只列举了几类常见的查询计划问题，例如全表扫描时出现无过滤条件的大表、可以下推的过滤条件没有下沉到存储等复杂计划和性能问题等，需要AnalyticDB MySQL版专家服务小组来协助定位排查。

改进执行计划

- 收集统计信息

  AnalyticDB MySQL版的查询优化器根据统计信息估算不同计划的开销，因此，准确的统计信息对于生成查询计划至关重要。如果有查询计划问题，最佳处理办法是更新统计信息，确保查询优化器可以拿到最新、准确的统计信息。

  如果未执行过Analyze收集统计信息，建议执行 `analyze table table_name;` 收集某张表的统计信息。如果一个数据库中有多张表，可以使用 `use db_name;analyze table all;` 一次收集所有表的统计信息。收集完统计信息后，优化器可以对绝大多数查询生成最优计划。但在数据极端分布等场景，优化器也可能无法生成最优计划，需要一些特殊的调优手段，其中以调整Join Order最为常见。

- 调整Join Order

  通常AnalyticDB for MySQL可以选择最佳的Join Order，由于数据分布特性以及查询自身的复杂度等因素，在某些场景下可能存在无法选择最优Join Order，查询性能较低，您可以通过在Hint中引入 `reorder_joins` 参数设置是否手动调整Join Order。

  - `/*+reorder_joins=false*/;` ，打开手动调整Join Order开关，然后通过修改查询中各个表出现的顺序来控制Join Order。
  - `/*+reorder_joins=true*/;` ，关闭手动调整Join Order开关，系统会自动选择Join Order，绝大多数场景下可以获得最佳Join Order。

  例如上述示例中的nation、region、customer表，AnalyticDB MySQL版给出的Join Order是region > nation > customer。如果根据实际查询性能得出更好的Join Order：customer > nation > region，可以如下所示在查询前使用Hint改变查询中表的顺序。

```
/*+ reorder_joins=false */
       EXPLAIN SELECT count(*)
       FROM    customer, nation, region
       WHERE c_nationkey = n_nationkey
       AND n_regionkey = r_regionkey
       AND r_name = 'ASIA';
```

```
| Plan Summary  |
+---------------+
1- Output[ Query plan ]
2  -> Aggregate (FINAL)
3    -> LocalExchange[SINGLE]
4      -> Exchange[GATHER]
5        -> Aggregate (PARTIAL)
6          -> InnerJoin[Hash Join]
7            - Project
8              -> InnerJoin[Hash Join]
9                - ScanProject {table: customer}
10                 -> TableScan {table: customer}
11               -> LocalExchange[HASH]
12                 -> Exchange[REPLICATE]
13                   - ScanProject {table: nation}
14                     -> TableScan {table: nation}
15           -> LocalExchange[HASH]
16             -> Exchange[REPLICATE]
17               - ScanProject {table: region}
18                 -> TableScan {table: region}
```

- Plan Hint For Join Order

  除了上述通过 `reorder_joins` 参数修改查询中表出现的顺序来调整Join Order外，AnalyticDB MySQL版还提供Plan Hint For Join Order帮助您调整Join Order。通过在查询头部嵌入 `leading_join_order` 来调整AnalyticDB MySQL版的Join Order。

  > **说明** 为了了解同一个表在查询中被多次引用的情况，使用Join Order Plan Hint时需要为查询中的表指定别名。

  对于上述示例，在查询前加入Plan Hint For Join Order也可以得到相同的Join Order。

```
/*+ leading_join_order=((c n) r) */
    EXPLAIN SELECT count(*)
    FROM    nation n , region r, customer c
    WHERE c_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'ASIA';
```

查询计划调优是一个非常广泛的领域，本文简要讨论AnalyticDB MySQL版中的查询计划调优基础方法，后续将不断更新更多的调优最佳实践。