ALIBABA CLOUD

Alibaba Cloud

云原生数据仓库AnalyticDB MySQL版

最佳实践

文档版本：20201111

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.结构设计

本文介绍云原生数据仓库AnalyticDB MySQL版（简称ADB，原分析型数据库MySQL版）中如何选择表类型、分布键、分区键、主键以及聚簇，使表的性能达到最优。

选择维度表或者普通表

- 维度表会在集群的每个节点存储一份数据，建议维度表的数据量不宜太大，每张维度表存储的数据不超过2万行。
- 普通表也叫作分区表，是为充分利用分布式系统的查询能力而设计的一类表。普通表可存储的数据量通常比较大，可以存储千万条甚至上亿条数据。

选择合适的分布键

ADB中创建普通表时，默认需要通过 `DISTRIBUTED BY HASH(column_name,...)` 指定分布键，按照 `column_name` 的HASH值进行分区。ADB支持将多个字段作为分布键。

分布键的选择依据：

- 尽可能选择参与JOIN的字段作为分布键，例如按照用户维度透视或者圈人，可以选择 `user_id` 作为分布键。
- 尽可能选择值分布均匀的字段作为分布键，例如交易ID、设备ID、用户ID或者自增列作为分布键。

选择合适的分区键

如果业务明确有增量数据导入需求，创建普通表时可以同时指定分布键和分区，分区可以实现数据的增量同步。

创建普通表时，通过 `PARTITION BY {VALUE(column_name) | VALUE(date_format(column_name, ?)}` 指定分区。

例如， `PARTITION BY VALUE(column_name)` 表示使用 `column_name` 的值做分区， `PARTITION BY VALUE(DATE_FORMAT(column_name, '%Y%m%d'))` 表示将 `column_name` 格式化为类似 `20190101` 的日期格式做分区。

```
# 直接用ds的值来做分区
PARTITION BY VALUE(ds)
# ds转换后的天做分区
PARTITION BY VALUE(DATE_FORMAT(ds, '%Y%m%d'))
# ds转换后的月做分区
PARTITION BY VALUE(DATE_FORMAT(ds, '%Y%m'))
# ds转换后的年做分区
PARTITION BY VALUE(DATE_FORMAT(ds, '%Y'))
```

分区的选择依据：

- 当数据量较大时，二级分区的选择至关重要，如果数据量大的表中没有二级分区或者二级分区切分不合理，将严重影响ADB的性能。
- 创建表时通过 `PARTITION BY Value(column_name | date_format)` 定义二级分区，数据将按照指定方式进行切分，目前切分粒度只支持年、月、日以及原始值。切分□粒度太大或太小都会影响查询性能和写入性

能，甚至影响ADB的稳定性。

- 二级分区粒度的选择原则为一个二级分区的数据量在3亿~10亿之间视为合理切分原则，如果小于3亿，表示切分粒度太小，可以增大切分粒度（例如将切分粒度由日改为月）；如果大于10亿，表示切分粒度太大，可以减小切分粒度（例如将切分粒度由月改为日）。
- 尽量使二级分区维持静态状态，不建议频繁更新二级分区，例如如果有每天频繁更新多个历史二级分区场景，应考虑使用的二级分区字段是否合理。

聚簇选择

聚簇中键值的逻辑顺序决定了表中相应行的物理顺序，每个表仅支持创建一个聚簇。

聚簇键的选择依据：

查询一定会携带的字段可以作为聚簇键。例如，电商卖家透视平台中每个卖家只访问自己的数据，卖家ID可以定义为聚集索引，保证数据的局部性，提升数据查询性能。

选择合适的主键

在表中定义主键可以去除重复数据，只有定义过主键的表支持数据更新操作（DELETE和UPDATE）。

主键的选择依据：

- 尽可能选择单数字类型字段作为主键，表的性能相对更好。

  ADB支持将字符串或者多字段组合作为主键。
- 主键中必须包含分布键和分区键。

2.流量控制最佳实践

云原生数据仓库MySQL版支持为内部系统查询、用户普通查询、用户ETL（Extract-Transform-Load）类查询三种查询队列设置最大可运行查询数以及最大排队查询数。

查询队列

为隔离内部系统查询、用户普通查询、用户ETL（Extract-Transform-Load）类查询（例如Insert into select）三种查询流量，云原生数据仓库MySQL版（简称ADB MySQL版）接入层将这三种查询的查询队列进行了隔离，而且ADB MySQL版支持为每种查询的查询队列设置最大可运行查询数以及最大排队查询数。

类型	查询队列	最大可运行查询数	最大排队查询数	配置项
系统查询	ROOT	400	500	• XIHE_ENV_QUERY_ROOT_MAX_CONCURRENT_QUERIES • XIHE_ENV_QUERY_ROOT_MAX_QUEUED_QUERIES
用户普通查询	NORMAL	20	200	• XIHE_ENV_QUERY_MAX_CONCURRENT_QUERIES • XIHE_ENV_QUERY_MAX_QUEUED_QUERIES
用户ETL查询	LOW	20	100	• XIHE_ENV_QUERY_LOW_PRIORITY_MAX_CONCURRENT_SIZE • XIHE_ENV_QUERY_LOW_PRIORITY_MAX_QUEUED_SIZE

- 当查询队列中正在执行的查询数量大于或者等于最大可运行查询数时，新的查询将进入排队状态。
- 当查询队列中处于排队状态的查询数量大于或者等于最大排队查询数时，新的查询将直接被拒绝。
- 当正在执行的查询结束时，如果有排队状态的查询，ADB MySQL版将以FIFO（First Input First Output）方式从排队状态的查询中选取下一个查询进入执行状态。

> **说明** 队列大小为单个前端节点大小，整个集群的队列大小等于单个队列乘以前端节点个数，即 `集群的队列大小=单个队列*前端节点个数` 。

调整查询队列

```
-- 系统查询队列
set adb_config XIHE_ENV_QUERY_ROOT_MAX_CONCURRENT_QUERIES=400
set adb_config XIHE_ENV_QUERY_ROOT_MAX_QUEUED_QUERIES=500

--用户普通查询队列
set adb_config XIHE_ENV_QUERY_MAX_CONCURRENT_QUERIES=20
set adb_config XIHE_ENV_QUERY_MAX_QUEUED_QUERIES=200

--用户ETL查询
set adb_config XIHE_ENV_QUERY_LOW_PRIORITY_MAX_CONCURRENT_SIZE=20
set adb_config XIHE_ENV_QUERY_LOW_PRIORITY_MAX_QUEUED_SIZE=100
```

指定查询队列

```
-- 通过Hint指定查询使用LOW队列
/*+coordinator_query_queue=low_priority*/ select * from tbl limit 100;

-- 普通查询使用NORMAL队列
select * from tbl limit 100;

-- ETL查询默认使用LOW队列
insert into dst select * from tbl;
```

查询执行优先级

ADB MySQL版的计算模块在执行查询时，为隔离用户普通查询、用户ETL查询，将查询执行线程也进行了隔离。查询执行线程分为两组NORMAL和LOW：

- 内部系统查询、用户普通查询由NORMAL线程组执行。
- 用户ETL查询由LOW线程组执行。

NORMAL线程组相比LOW线程组具有更高的优先级，二者产生资源竞争时，NORMAL线程优先获得CPU资源。

查询类型	执行线程组	优先级
系统查询、普通用户查询	NORMAL	高
用户ETL查询	LOW	低

设置执行优先级

```
-- 通过Hint指定查询使用LOW线程组
/*+direct_low_priority_cpu_queue=true*/ select * from tbl limit 100;

-- 普通查询使用NORMAL线程组
select * from tbl limit 100;

-- ETL查询默认使用LOW线程组
insert into dst select * from tbl;
```

3.SQL设计优化

本文介绍AnalyticDB for MySQL中常用的SQL优化原则以及方法。

SQL优化原则

- **SQL尽量简单**

  大部分情况下SQL性能随SQL复杂度下降，例如，单表查询性能 （冗余设计）优于多表关联查询。

- **减少IO**

  使用索引扫描，尽可能少的进行列扫描，返回最小数据量，同时减少IO和内存开销。

- **本地计算**

  在大数据计算中，进行本地计算时应避免数据跨节点，充分利用分布式多计算资源的能力。

常用优化方法

- **去掉不必要的列**

  AnalyticDB for MySQL是行列混存数据库，返回列的数量直接影响SQL性能，建议在编写SQL时只写业务确认需要返回的列，不直接使用 *。

  典型的错误写法：

  ```
  select * from tab1 where c1>100 and c1<1000;
  ```

  正确写法中明确要返回的列：

  ```
  select col1, col2,... from tab1 where c1>100 and c1<1000;
  ```

- **尽早过滤数据**

  尽可能早的过滤数据，减少数据规模，有助于加速SQL执行。

  ```
  select count(*)
  from t1 join t2 on t1.id=t2.id
  where t1.time
  between '2017-12-10 00:00:00' and '2017-12-10 23:59:59' and t1.type= 100;
  ```

  上述SQL中，t2与t1表有相同的time和type过滤条件，建议将SQL修改为以下形式。

  ```
  select count(*)
  from t1 join t2 on t1.id=t2.id
  where t1.time between '2017-12-10 00:00:00' and '2017-12-10 23:59:59'
  and t1.type= 100
  and t2.time between '2017-12-10 00:00:00' and '2017-12-10 23:59:59'
  and t2.type=100
  ```

- **选择性使用索引**

  AnalyticDB for MySQL建表时默认创建全列索引，系统会尽量使用索引完成扫描工作。但在某些情况下，不使用索引效率更高，主要取决于数据选择率和计算能力。

以下场景中，您可以不使用索引，SQL性能将更好。

- 低筛选列

  当SQL中包含多个查询条件时，可优先选择高筛选条件的索引进行扫描，其他条件可以通过表扫描方式。例如以下SQL通过c1字段和time字段过滤数据，通过 `c1=3` 可以快速查询到少量记录（假设10000条），而单独通过 `time>'2010-01-01 00:00:00'` 时返回的记录数非常大。由此可见，通过c1的筛选条件选择率较高，通过time的筛选条件选择率较低。

  ```
  select c1,c2 from tab1 where c1=3 and time >='2010-01-01 00:00:00';
  ```

  以下SQL只使用c1列的索引，time条件通过内部扫描，例如 `time >='2010-01-01 00:00:00'` 通过表扫描。计算引擎首先检索c1列的索引，得出满足条件 `c1=3` 的行集合，然后读取每行所对应的time列数据，如果满足 `time >='2010-01-01 00:00:00'` ，则将该行数据加入返回结果。

  ```
  /*+ no_index_columns=[tab1.time]*/ select c1,c2 from
  tab1 where c1=3 and time >='2010-01-01 00:00:00';
  ```

- 不等值条件比较

  例如 `select c1,c2 from tab1 where c1=3 and c2<>100;` 中存在不等值条件 `c2<>100` ，无法有效过滤记录。此时可以不通过索引扫描，通过内部扫描效率更高。

  ```
  /*+ no_index_columns=[tab1.c2]*/ select c1,c2 from tab1 where c1=3 and c2<>100;
  ```

- 非前缀索引

  中缀或后缀查询中，建议不通过索引扫描，通过内部扫描。

  例如SQL为 `select c1,c2 from tab1 where c1=3 and c3 like '%abc'%;` ，可以不通过索引扫描，通过增加 `no-index hint` 进行内部扫描，对应的SQL为 `/*+ no_index_columns=[tab1.c3]*/ select c1,c2 from tab1 where c1=3 and c3 like '%abc%';` 。

- 索引与NULL

  AnalyticDB for MySQL不同于其他类型的数据库，AnalyticDB for MySQL中可以保存NULL值。因此，如果WHERE条件中使用了 `is null` 或 `is not null` ，则可以使用索引。

- **分区裁剪**

  AnalyticDB for MySQL是分布式数据库，数据分布在多个分片中，支持在分片的基础上进行二级分区。在表设计阶段，确定好分片和分区规则后，后期编写SQL语句时，应尽量增加对分片字段和分区字段的使用，快速过滤数据，缩小搜索范围。

```
CREATE TABLE t_fact_mail_status (
mail_id varchar COMMENT '',
scan_timestamp timestamp COMMENT '',
biz_date bigint COMMENT '',
org_code varchar COMMENT '',
org_name varchar COMMENT '',
dlv_person_name varchar COMMENT '',
receiver_name varchar COMMENT '',
receiver_phone varchar COMMENT '',
receiver_addr varchar COMMENT '',
product_no varchar COMMENT '',
mag_no varchar COMMENT '',
op_1_timestamp bigint COMMENT '',
op_2_timestamp bigint COMMENT '',
op_3_timestamp bigint COMMENT '',
op_4_timestamp bigint COMMENT '',
op_5_timestamp bigint COMMENT '',
PRIMARY KEY (mail_id,org_code,biz_date)
)
DISTRIBUTED BY (org_code)
PARTITIONED BY VALUE(biz_date)
COMMENT '';
```

针对以上表，可通过以下SQL实现分区裁剪。

```
select count(*) as cn from t_fact_mail_status t
where
t.org_code = '21111101'and t.biz_date = 20171128
```

- **数据重分布**

  多个普通表进行关联时，若访问的关联数据不在同一分片或分区，即涉及跨分片或分区的数据访问时，需要做数据重分布。

  数据重分布操作非常消耗资源，因此应尽量保证数据在同一分片或分区内完成关联操作。因此普通表关联时， ON条件尽量包含分区列且是等值查询，以减少因分区没有对齐而造成的数据重分布。

常见问题

哪些问题会导致索引失效?

- 函数（列）

  SQL中的函数转换可能造成索引失效，例如 `select c1,c2 from tab1 where substr(cast(time as varchar),1,10)= '2017-10-12';`

  。

- 模糊匹配

  SQL中含有模糊匹配条件可能导致索引失效。

  ```
  select c1,c2
  from tab1
  where time like '2017-10-12%';
  ```

  可将上述SQL改为以下形式，避免索引失效。

  ```
  select c1,c2
  from tab1
  where time>='2017-10-12 00:00:00' and time<='2017-10-12 23:59:59';
  ```

4.优化高级特性

为解决Hash Join多表关联时过多消耗资源问题，AnalyticDB for MySQL推出Nested Loop Join多表关联方式。本文以具体示例介绍Nested Loop Join的使用方法。

背景信息

AnalyticDB for MySQL默认使用Hash Join进行表关联，Hash Join是一种通用的表关联方式，使用时会消耗更多的资源。对于一些数据规模较小、有较好关联选择率时，Hash Join并不是最优选择。AnalyticDB for MySQL中的Nested Loop Join能够很好解决这一问题，Nested Loop Join内部对应右表查询，使用索引访问方式，因此Nested Loop Join也称为Index Join。

适用场景

Nested Loop Join适用于左表过滤后只查询出少量数据，且右表在Join Column上有较好选择率的场景。

注意事项

- Nested Loop Join仅支持Inner Join和Left Join，不支持Right Loin和Full Outer Join。
- 使用Nested Loop Join时右表必须有索引，且右表不支持多分区列。
- 关联列两边数据类型完全一致。
  - 支持数据类型：Int、Boolean、Varchar、Time、Date、Timestamp、Datetime。
  - 不支持数据类型：Float、Double、Decimal、Json、Multivalue、Geoo2d、Bytes、Array、Blob、Binary、Null、Other。
- 事实表Join维度表时：
  - 维度表记录数不超过2千万条。
  - 事实表Join维度表时，不限制关联条件。
  - Left Join时左表不能是维度表。
- 事实表Join事实表时，关联条件必须包含分布键。

使用方法

可以通过Hint开启Nested Loop Join。

- `/*+nested_loop_join=true*/`

  不做表数量判断，所有符合条件的表都使用Nested Loop Join。
- `/*+nlj_index_join_small_table_max_row, nlj_index_join_large_table_max_row*/`

  进行左右表数量判断，小表或大表超过阈值时将使用Hash Join。

示例

```
/*+nested_loop_join=true */
SELECT  t1.c1
FROM  T1 INNER JOIN T2
ON T1.C1=T2.C1;
```

上述SQL执行对比计划如下所示。

5.查询性能调优

5.1. 查询的处理流程

本文介绍AnalyticDB如何在内部完成查询的处理。您可以结合慢查询表和查询的统计信息来定位查询性能问题。

AnalyticDB使用SQL语言完成用户和系统内部存储的数据之间的交互。SQL语句提交到AnalyticDB的前端Controller节点后，AnalyticDB会对SQL语句进行语句解析生成语法树，然后生成逻辑执行计划，并对逻辑执行计划进行优化，生成最终的逻辑执行计划。Executor节点会把逻辑执行计划进一步翻译成物理执行计划，物理执行计划由物理算子组成，使用物理算子对数据按照一定规则进行处理。数据处理的最终结果会返回到用户的客户端，写入AnalyticDB内部表或者外部存储系统中。

5.2. 查询的执行计划

5.2.1. 执行计划的生成

查询的执行计划生成主要包含以下几个步骤。

- Controller节点接收查询请求、解析SQL语句；
- 优化器对解析后的SQL语句进行分析，评估是否需要对查询树进行重写来提高查询性能；
- 优化器生成最优的逻辑执行计划，从而规定特定的执行处理方式，例如join类型、join顺序、aggregation方式以及数据重分布方式等；
- Executor节点将接收到的逻辑执行计划转换成物理执行计划。

5.2.2. 执行计划的要素

查询计划也即物理执行计划，包括Stage、Task和Operator三个要素。

Stage

AnalyticDB中的查询在执行阶段会首先被切分为多个Stage执行，一个Stage是执行计划中某一部分的物理实体。Stage的数据可以来源于底层数据源或者网络上的数据重分布（上游Stage），一个Stage由分布在多个Executor节点上的Task并行执行来完成数据处理。关于执行计划的生成，请参见执行计划的生成。

Task

Task是一个Stage在某个Executor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。

Operator

Operator是AnalyticDB基本处理单元。多个Operator在一个Task内部组成上下游关系，或并行或串行处理数据。是否并行或者串行取决于多个Operator所表达的语义或者Operator之间的依赖关系。

如下图所示，Controller节点把Plan分片下发到Executor节点，其中的Stage 2由4个Executor上的Task执行，Stage 1由2个Executor上的Task执行，Stage 0由一个Executor上的Task执行，执行的结果由Stage 0所在的Task返回到Controller节点，然后返回到客户端。

5.2.3. 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。

5.3. 查询的统计信息和关键指标

5.3.1. 查询详情页面

查询详情页面通过数字和表格的形式展示Query级别、Stage级别和Task级别的实际执行过程中生成的统计信息，包括耗时、CPU Time、内存以及扫描或者从网络接收的数据行数和数据量等信息。有关查询性能问题定位的更多信息，可参见如何诊断慢查询。

> **说明** 从3.1.2版本开始支持该功能。有关版本信息的查看方法，可参见如何查看实例版本信息。

Query级别统计信息

查询基本信息

可以看到查询的最终状态、执行用户的用户名、客户端IP地址、连接所使用的数据库名以及总的Stage和Task个数等基本信息，如下图所示：

查询时间信息

可以看到查询相关的时间信息，例如查询的提交时间、查询从提交到结果返回的总时间、查询的在正式开始执行前的排队时间以及查询生成执行计划的耗时等，如下图所示：

资源使用信息

可以看到查询使用的资源信息，包括CPU耗时和峰值内存，如下图所示：

SQL语句

Stage级别统计信息

State

State表示一个Stage的最终执行状态：FINISHED（成功）、CANCELED（被取消）、FAILED（失败）。

Operator Cost

一个Stage所有Task的所有算子的CPU耗时的累加值，和该Query内其他Stage的Operator Cost比较可以用来判断计算量较大的Stage。

Peak Memory

一个Stage在执行过程中使用的峰值内存大小，和该Query内其他Stage的Peak Memory比较可以用来判断消耗内存较大的Stage。

CPU Time

Stage分布在计算节点的Task的算子在计算时消耗的CPU时间的最大值、最小值和平均值，可用来初步判断该Stage在处理数据时是否存在长尾，出现长尾的可能原因是每个Task处理的数据量存在差异或者某些计算节点压力增大导致的。

Input Size

Stage的所有Task的输入数据量的最大值、最小值和平均值，可以用来初步判断该Stage要处理的数据量是否存在倾斜，数据量的倾斜会导致CPU Time的长尾，数据倾斜的可能原因包括表的分布键选择不合理或者数据重分布时的分布键存在倾斜。

Scan Size

Stage如果存在数据源算子(tablescan)，那么该字段表示各个Task在扫描数据量上的的最大值、最小值和平均值，如果这三个值差异较大，则说明某些表在数据源存在分布键选择不合理的可能。但是注意一个Stage可能存在多个tablescan算子，会从不同的表中读取数据。

Scan Time

Stage如果存在数据源算子(tablescan)，那么该字段表示各个Task在从数据源读取数据耗时的最大值、最小值和平均值，如果这三个值存在较大差异，可能是由于分布键选择不合理导致数据倾斜，也可能是执行节点压力不均导致的。

Task级别统计信息

点击Stage上每个表格的某行的任意位置，可以在Task详情表中查看该Stage的所有Task的详细指标信息，这些指标信息包括：

State

State表示一个Task的最终执行状态：FINISHED(成功)；CANCELED(被取消)；FAILED(失败)。

ElapsedTime

AnalyticDB的Task内部是多线程并行处理数据分片。ElapsedTime表示的是Task从启动到最后一个线程处理完数据分片的时间。如下图所示，共4个线程参与某个Task的数据处理任务，ElapsedTime就是T2 - T1得到的结果：

Peak Memory

表示某个Task在处理数据时使用内存的峰值。

Operator Cost

表示某个Task在处理数据时使用的CPU Time的总和，是个累加值。如上图所示的Task的Operator Cost即为：

```
OperatorCost = 100ms + 110ms + 70ms + 90ms
```

Input Rows/ Input Size
表示每个Task的输入数据的总行数和总数据量。

Output Rows/Output Size
表示每个Task的输出数据的总行数和总数据量。

Scan Rows/Scan Size/ScanCost
如果某个Task包含从源表读取数据的算子，那么这三个指标分别表示扫描的数据行数/扫描的数据量/扫描的总耗时。

5.3.2. VisualPlan

VisualPlan通过图形化的形式展示查询实际执行过程中使用的执行计划。有关查询性能问题定位的更多信息，可参见如何诊断慢查询。

VisualPlan按照Stage进行划分。最下边表示最上游的Stage，最上游Stage一般需要从源表读取和过滤数据。箭头表示数据的流向。在Stage内部，数据处理逻辑使用算子组合表示，箭头方向表示数据在算子间的流向。

StageSummary
StageSummary表示当前stage所有task重要指标的累加值，具体包括：

- Output：当前Stage的输出数据量大小和输出行数。
- Stage：当前Stage的StageId，和详情页的StageId对应。
- State：当前Stage的最终状态。
- Operator Cost：当前Stage在执行过程中的消耗的CPU Time总和，是多机多线程的累加值。
- Peak Memory：当前Stage在执行过程中使用的峰值内存大小。
- Input：当前Stage的输出数据量大小和输入行数。

估算信息

Est. I/O：表示某个算子估算的输入和输出行数。如果是Join算子，则只有估算的输出行数，没有估算的输入行数。估算信息是在优化器优化阶段，使用特定方法对输出行数不确定的算子进行的数据量估计，从而在执行计划的优化阶段选择较优的执行计划。如果估算值和实际值相差较大，可能会导致使用较差的执行计划，最终导致查询消耗较大的服务器资源或者导致慢查询的产生。

算子
有关算子的更多信息，可参见算子。

5.4. 查询性能问题定位

5.4.1. 慢查询表

本文介绍如何在慢查询表**information_schema.kepler_slow_sql_merged**上执行查询语句，从而对慢SQL进行初步的原因定位，或者查找消耗资源（例如CPU或内存）较大的bad SQL。

简介

慢查询表**information_schema.kepler_slow_sql_merged**记录了在AnalyticDB (ADB) 中执行的SQL语句耗时大于1秒（默认时间）时的部分统计信息。如果要修改默认时间，请提交工单处理。

慢查询表及其详情信息保存在前端节点的本地磁盘文件中，该表存储的慢SQL个数根据用户的集群规模或者SQL复杂度而不同。集群规模越大或者SQL越复杂，需要保存的信息也越多，相应地保存的慢SQL个数也越少。您可以对该表执行**count**查看慢SQL个数或者执行min(start_time)来确定保存的慢SQL的最早时间。

常用字段说明

字段	说明
start_time	查询的提交时间。
time	查询的总耗时。单位为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。可用于判断消耗内存资源较多的SQL。
state	查询状态。其值为SUCCESS或者FAILED。
scan_rows	从数据源表扫描的行数。可用于判断某条SQL是否扫描了大量数据。
scan_size	从数据源表扫描的数据量。单位为byte。
scan_time	从底层存储扫描数据消耗的时间累加值。单位为ms。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ADB的优化器中进行最优执行计划计算的耗时。单位为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。单位为ms。
sql	用户提交的SQL语句。
queued_time	查询的排队时间。单位为ms。
access_ip	查询提交使用的客户端IP地址。

常用的查询

- 查看某个时段的内存、扫描行数、扫描数据量、相关SQL、耗时以及提交时间

  SQL模版

```
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}
```

SQL示例

查找提交时间在 `2020-09-16 01:41:33` 和 `2020-09-16 02:41:33` 这一小时之内的SQL:

```
SELECT peak_mem, scan_rows, scan_size, sql, time, start_time
FROM information_schema.kepler_slow_sql_merged
WHERE start_time > '2020-09-16 01:41:33'
 AND start_time < '2020-09-16 02:41:33'
```

- 根据SQL条件模糊查询内存、扫描行数、扫描数据量、耗时以及提交时间

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

- 根据耗时条件查询某个时段的SQL情况

  SQL模板

```
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}
```

  SQL示例

```
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
```

- 根据 `peak_mem` 字段查询内存消耗在某个范围内的SQL情况

  SQL模板

```
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}
```

  SQL示例

```
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` 进行分组聚合，然后根据平均消耗内存最多的 `access_ip` 进行降序排列

```
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
```

5.4.2. 慢查询的分类

本文介绍几种典型的慢查询相关的资源消耗问题。您可以通过查询详情页和VisualPlan定位慢查询带来的资源消耗，更多信息可参见如何诊断慢查询，查询详情页面和VisualPlan。

消耗内存的慢查询

首先可以通过慢查询表来确定某个时间段内耗时较大的查询，然后打开查询详情页面，在详情页的Stage级别统计信息处，根据Peak Memory字段排序，找到Peak Memory较大的Stage，然后打开VisualPlan，在VisualPlan找到对应的Stage，和SQL语句结合判断内存消耗较大的算子。

一般消耗内存较大的情况包括：

- Stage中有 `group by` 操作，并且 `group by` 的字段唯一值较多，会导致在计算过程中使用较大内存进行 `group by` 字段值的缓存。
- Stage中有 `join` 操作。在使用hash的方式进行 `join` 时会把某张表的数据缓存到内存中，所以如果被缓存在内存中的表较大，会占用较多内存。
- Stage中有 `sort` 操作。在执行数据的排序时，会把数据缓存到节点中，所以如果需要排序的数据量较大，会占用较多内存。
- Stage中有 `window` 操作。在执行开窗操作时，会把数据缓存在内存中，如果需要执行开窗的数据量较大，会占用较多内存。

消耗CPU的慢查询

首先可以通过慢查询表来确定某个时间段内CPU消耗较大的查询，然后打开查询详情页面，在详情页的Stage级别统计信息处，根据Operator Cost字段排序，找到Operator Cost较大的Stage，然后打开VisualPlan，在VisualPlan找到对应的Stage，和SQL语句结合进行CPU消耗较大的算子的判断。

一般消耗CPU较大的情况包括：

- 过滤条件没有下推到存储层。ADB在创建表时默认为所有字段创建了索引，如果可以使用索引进行过滤，可以极大的减少CPU的消耗，但是在某些情况下，例如在过滤条件中使用了function，会导致查询条件无法使用索引，ADB需要对每条扫描上来的数据执行过滤，此时会导致占用较多CPU资源。
- join中带有过滤条件。如果join中带有过滤条件，ADB会先对两表执行join，然后对join后的数据执行过滤，此时无法使用索引，如果join后产生的数据量较大，那么过滤操作会消耗较大的CPU资源。
- join时没有指定join的criterion。没有指定criterion的join，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，该类操作会带来较大的CPU资源消耗。

消耗磁盘IO的慢查询

首先可以通过慢查询表来确定某个时间段内Scan Time较大的查询，然后打开查询详情页面，在详情页的Stage级别统计信息处，根据Scan Time字段排序，找到Scan Time较大的stage，然后打开VisualPlan，在VisualPlan中找到对应的Stage，根据Tablescan算子的输入行数判断某个表是否存在大量的数据扫描。

一般消耗磁盘IO较大的问题包括：

- 过滤条件的筛选率较低，会导致使用索引的效率不高。
- 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。
- 扫描的二级分区较多。

5.4.3. 如何诊断慢查询

背景信息

您可以通过查询详情页面和VisualPlan查看执行计划的详细信息，从而定位可能导致慢查询的性能问题。

操作步骤

1. 连接目标集群。
2. 在左侧导航栏单击**诊断与优化**。
3. 单击右侧页签**历史查询**。
4. 选择想要定位的慢查询发生的时间段（最近7天以内），单击**查询**。

   

   您还可以从列表中选择**慢SQL趋势**查看慢SQL趋势图。单击趋势图上面的节点可以查看特点时间点的慢SQL详情。

   

5. 在**操作**列中单击**查询执行计划**查看慢查询的查询详情页面。有关查询详情页面的更多信息，可参见查询详情页面。
6. 在查询详情页面单击**可视化查看SQL计划**查看VisualPlan。有关VisualPlan的更多信息，可参见VisualPlan。

6.执行计划

6.1. 执行计划的分类

AnalyticDB 中的执行计划包括：Join Order、Join Type、Join Method、Access Method、Data Shuffle、Aggregation 等算子。

Join Order

根据查询的负载特征，提供多种关联顺序（Join Reordering）策略，主要包括Left Deep Tree （LDT）与Bushy Tree （BT）两种。其中，LDT一般适用于连结表分区对齐、优化开销相对较小、查询相对简单的场景。BT一般适用较复杂的查询场景，需要相对更长的优化时间。

> **说明** Join的Outer Table和Inner Table分别称为左表和右表。

Join Type

Join Type包括Inner Join、Left Outer Join、Right Outer Join、Full Outer Join、Semi/Anti Join。对应在EXPLAIN中的计划输出分别为：InnerJoin、LeftJoin、RightJoin、FullJoin、SemiJoin。

> **说明** Anti Join显示为Semi Join加Filter。

Access Method

单表访问默认为Index Access（通过Filter下推），充分利用AnalyticDB for MySQL全索引设计的优势，因此在查询计划中不做特别区分，一般标注为TableScan。某些特定场景可以通过指定No Index Hint规避使用索引。

Data Shuffle

在AnalyticDB for MySQL的MPP Share Nothing执行架构下，根据查询以及数据分布特性，会出现对数据进行重分布（Data Shuffling）的需求，包括对基表数据以及查询中间结果的数据重分布。Data Shuffling包括Broadcast和Repartition两种数据重分布策略，对应的EXPLAIN的计划输出分别为RemoteExchange[REPLICATED]和RemoteExchange[REPARTITION]，RemoteExchange[GATHER]算子对应数据汇集。

Aggregation

无论是Data Warehouse的Batch Query还是Interactive Query，都会涉及到多种Aggregation（聚合）计算。这些聚合算子在EXPLAIN的计划中输出为Aggregation，并会标注具体的聚合算子类型。此外，对于分布式聚合计算，AnalyticDB for MySQL会根据数据特性来决定是否拆分为多步分布式聚合计算。

6.2. 通过EXPLAIN查询执行计划

在AnalyticDB for MySQL中支持通过两种方式查看生成的查询计划：通过EXPLAIN命令返回文本格式的查询计划或者使用DMS集成的图形化计划输出功能。本文介绍如何通过EXPLAIN命令返回文本格式的查询计划，EXPLAIN是大多数商业数据库的标准命令接口，AnalyticDB for MySQL也在不断扩展和优化EXPLAIN功能。

输出对应语句的执行计划

- **语法**

```
Explain sql_statement;
```

- **示例**

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

上述EXPLAIN语句的返回结果如下：

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

在上例输出的执行计划中，标注了18个mini-plan，每一个对应一个执行算子，EXPLAIN结果中Plan的缩进代表了算子间相互的逻辑执行顺序。

上述示例中，最顶层是由Aggregation计算Count，Aggregation的结果来自于Join（Step 6），Join Type是Inner Join，Join Method是Hash Join，Customer是Hash Join的左表，而Nation与Region表Join的结果是Hash Join的右表。Nation表和Region表的Join (Step 12) Type是Inner Join，Join Method是HASH Join。在Inner Join算子下，先计算左表（Nation表），Step 15指Build Hash Table，Step 16是Broadcast，后出现的是右表（Region表）。

输出对应语句的执行计划（包含明细）

- **语法**

```
EXPLAIN(FORMAT DETAIL) sql_statement;
```

可以使用EXPLAIN(FORMAT DETAIL)命令的Option来输出详细的计划细节。

- **示例**

AnalyticDB for MySQL对EXPLAIN的输出格式做了进一步优化处理，例如可以只看查询计划的总结，不查看计划细节。也可以通过mini-plan的编号将查询计划细节和总结对应起来，EXPLAIN的结果将被分成两部分输出，如下所示。

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

```
+---------------+
| Plan Details |
+---------------+
1- Output[count(*)] => [count:bigint]
count(*) := count
2  - Aggregate(FINAL) => [count:bigint]
count := `count`(`count_0_5`)
3   - LocalExchange[SINGLE] () => count_0_5:bigint
4    - RemoteExchange[GATHER] => count_0_5:bigint
5     - Aggregate(PARTIAL) => [count_0_5:bigint]
count_0_5 := `count`(*)
6      - InnerJoin[(`c_nationkey` = `n_nationkey`)][$hashvalue, $hashvalue_0_6] => []
```

```
7 - ScanProject[table = adb:com.alibaba.cloud.analyticdb.connector.AdbTableHandle@354b905b, originalConstraint = (SELECT `c_nationkey` FROM tpch_100g.customer)] => [c_nationkey:integer, $hashvalue:bigint]
$hashvalue := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_code`(`c_nationkey`), 0))
LAYOUT: com.alibaba.cloud.analyticdb.connector.AdbTableLayoutHandle@582531c1
c_nationkey := com.alibaba.cloud.analyticdb.connector.AdbColumnHandle@11d903d7
8 - LocalExchange[HASH][$hashvalue_0_6] ("n_nationkey") => n_nationkey:integer, $hashvalue_0_6:bigint
9 - RemoteExchange[REPLICATE] => n_nationkey:integer, $hashvalue_0_7:bigint
10 - Project[] => [n_nationkey:integer, $hashvalue_0_12:bigint]
$hashvalue_0_12 := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_code`(`n_nationkey`), 0))
11 - InnerJoin[(`r_regionkey` = `n_regionkey`)][$hashvalue_0_8, $hashvalue_0_9] => [n_nationkey:integer]
12 - ScanProject[table = adb:com.alibaba.cloud.analyticdb.connector.AdbTableHandle@201cb3f2, originalConstraint = (SELECT `r_regionkey`, `r_name` FROM tpch_100g.region WHERE ('ASIA' = `r_name`))] => [r_regionkey:integer, $hashvalue_0_8:bigint]
$hashvalue_0_8 := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_code`(`r_regionkey`), 0))
LAYOUT: com.alibaba.cloud.analyticdb.connector.AdbTableLayoutHandle@7c6d54a
r_regionkey := com.alibaba.cloud.analyticdb.connector.AdbColumnHandle@20e861e
13 - LocalExchange[HASH][$hashvalue_0_9] ("n_regionkey") => n_nationkey:integer, n_regionkey:integer, $hashvalue_0_9:bigint
14 - RemoteExchange[REPLICATE] => n_nationkey:integer, n_regionkey:integer, $hashvalue_0_10:bigint
15 - ScanProject[table = adb:com.alibaba.cloud.analyticdb.connector.AdbTableHandle@6ad3e560, originalConstraint = (SELECT `n_nationkey`, `n_regionkey` FROM tpch_100g.nation )] => [n_nationkey:integer, n_regionkey:integer, $hashvalue_0_11:bigint]
$hashvalue_0_11 := `combine_hash`(BIGINT '0', COALESCE(`$operator$hash_code`(`n_regionkey`), 0))
LAYOUT: com.alibaba.cloud.analyticdb.connector.AdbTableLayoutHandle@1fd4d6d
n_nationkey := com.alibaba.cloud.analyticdb.connector.AdbColumnHandle@3b12e11a
n_regionkey := com.alibaba.cloud.analyticdb.connector.AdbColumnHandle@29806c3
```

在Plan Summary中，屏蔽了大量一般用户不感兴趣的细节，只留下查询计划中最重要的计划信息，例如Join Method、Join Type、Join Order、Data Shuffling、Relation Name等相关信息。一般情况下，用户只需要查看Plan Summary中的计划信息就可以大致了解查询的执行计划。

6.3. 通过DMS查询执行计划

您可以登录DMS，通过图形化界面查看执行计划。

使用DMS连接ADB数据库，通过SQL窗口中的执行计划功能查看执行计划。

从下往上，按箭头方向显示计划的执行顺序。整体查询计划被分成若干个子计划显示，即图中标注Plan SubStage的方框。跨子计划节点之间需要进行数据交换（Data Shuffle)。每个Plan SubStage包含对应计划执行时的算子，且会标注该子计划的输入和输出的行数以及数据大小统计。

DMS展示的查询执行计划主要包括Join Order的选择以及Join Type、Join Method、Access Method、Data Shuffle、Aggregation等主要相关算子。

6.4. 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/Outer表

  根据Join Method选择Inner和Outer表， 一般情况下AnalyticDB for MySQL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 for MySQL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 for MySQL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 for MySQL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 for MySQL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 for MySQL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 for MySQL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 for MySQL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 for MySQL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 for MySQL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 for MySQL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。