# 概述

参考：<https://zhuanlan.zhihu.com/p/617858587>

OceanBase SQL执行：<https://zhuanlan.zhihu.com/p/601989459>

当数据库接收到查询语句后，会按照如下的流程对其进行处理：

1. 语法和词法解析，生成查询语法树
2. 语义解析，对查询语法树进行语义分析，生成对应的查询对象
3. 依据关系代数，对查询对象依据规则进行等价改写
4. 为查询对象生成执行代价最优的逻辑执行计划
5. 依据逻辑执行计划，生成可执行的物理计划
6. 执行物理计划

按照上述的解析过程，分布式数据库的相关优化策略包括：

## 分区裁剪

参考：<https://cloud.tencent.com/developer/article/1654630>

### 静态裁剪

### 动态裁剪

## 子查询优化

### 查询下推

Oceanbase查询下推优化：<https://zhuanlan.zhihu.com/p/607051422>

### 聚合子查询提升

Oceanbase聚合子查询提升优化：<https://zhuanlan.zhihu.com/p/606747114>

### 子查询合并

Oceanbase子查询合并：<https://zhuanlan.zhihu.com/p/602296961>

### Where子查询提升

Oceanbase Where子查询提升：<https://zhuanlan.zhihu.com/p/604223928>

### 半连接/Semi-Join转内连接

Oceanbase半连接转内连接优化：<https://zhuanlan.zhihu.com/p/603851213>

### ANY/ALL子查询优化

Oceanbase ANY/ALL子查询优化：<https://zhuanlan.zhihu.com/p/602633311>

### winmagic优化

Winmagic原理：<https://zhuanlan.zhihu.com/p/384166270>

## JOIN优化

### Join reorder

Oceanbase JOIN Order生成优化：<https://zhuanlan.zhihu.com/p/610107681>

### 连接消除

Oceanbase连接消除优化：<https://zhuanlan.zhihu.com/p/605093512>

### 全外连接改写

Oceanbase全外连接改写：<https://zhuanlan.zhihu.com/p/604707982>

## 条件查询优化

### 谓词推导

Oceanbase谓词推导优化：<https://zhuanlan.zhihu.com/p/609408413>

### 谓词移动/谓词下推

Oceanbase谓词移动优化：<https://zhuanlan.zhihu.com/p/607768616>

## GROUP BY优化

### GROUP BY移动

Oceanbase group by移动：<https://zhuanlan.zhihu.com/p/607361694>

## HAVING优化

## 投影裁剪

Oceanbase投影裁剪优化：<https://zhuanlan.zhihu.com/p/608864723>

## 聚合函数优化

### Min-max聚合重写

Oceanbase Min-Max聚合改写：<https://zhuanlan.zhihu.com/p/606135975>

### 窗口函数提升

## DISTINCT优化

## ORDER BY优化

## LIMIT优化

### LIMIT消除

如果前面查询可以确定为空结果或者只有一行结果，此时的分页操作可以删除。

### 外连接LIMIT下推

OceanBase外连接limit下推优化：<https://zhuanlan.zhihu.com/p/606429327>

## 索引优化

可以采用智能索引、全局唯一索引、二级分区等方式优化查询。

## 热点优化

参考：

热点优化：<http://mysql.taobao.org/monthly/2022/02/03/>

## 并行查询

## 大数据查询优化器

参考：<https://www.51cto.com/article/648652.html>

好的查询优化器在架构设计上需要考虑以下的几个方面：

模块化。使用元数据和系统描述的高度可扩展的抽象，不再局限于特定的主机系统，如传统的优化器。相反，可以通过数据支持的插件将其快速移植到其他数据管理系统。

可扩展性。通过将查询及其优化的所有元素表示为具有同等地位的一等公民，避免多阶段优化的陷阱，在此阶段，某些优化将在事后进行处理。众所周知，多阶段优化器难以扩展，因为新的优化或查询构造通常与先前设置的阶段边界不匹配。

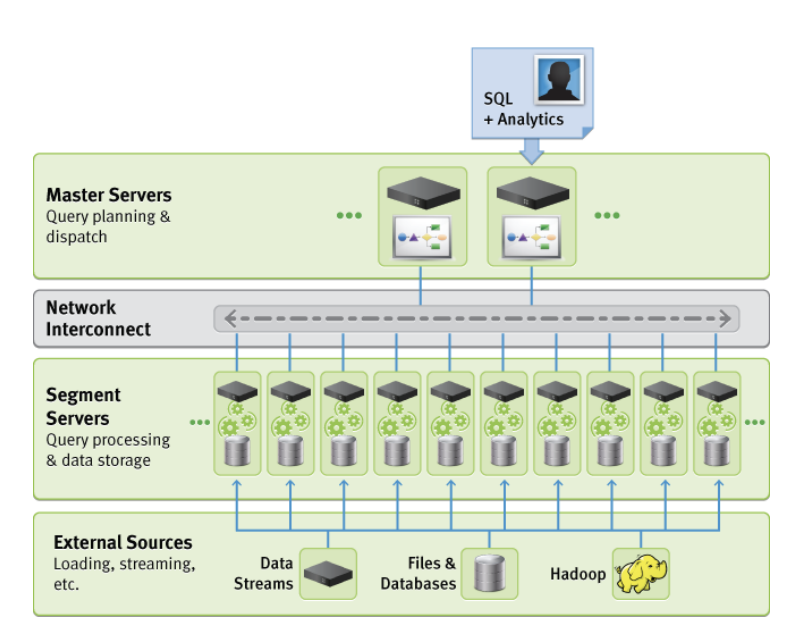
支持多核架构。系统需要部署一个高效的多核感知调度程序，该调度程序可在多个内核之间分配各个细粒度的优化子任务，以加快优化过程。

可验证性。有确定的特殊规定内置机制级别来保证正确性和性能。除了改善工程实践之外，这些工具还使人们能够高度自信地进行快速开发，并缩短了新功能和错误修复的周转时间。

性能。查询的性能是我们最希望等到的结果

### MPP架构

Orca是Pivatol开发的大数据模块化查询优化器，被用于Greemplum和Hawq之中。他就是按照上面的要求来设计的。



上图是Greemplum的体系架构，通过在多个服务器或主机之间分配负载以创建单个数据库阵列来处理大量数据的存储和处理，所有这些数据库共同工作以呈现单个数据库入口。主节点是GPDB的入口点，客户端可以在此连接并提交SQL语句。主节点与其他数据库实例(称为段)协同工作，以处理数据处理和存储。 当查询提交给主节点时，将对其进行优化并将其分解为较小的查询，这些较小的查询将被分配给各部分，以共同协作以交付最终结果。互连使用标准的千兆以太网交换结构，负责各段之间的进程间通信的联网层。

在查询执行期间，可以通过多种方式将数据分布到段，包括散列分布，其中元组根据某种哈希函数分布到段，复制分布，其中表的完整副本存储在每个段和单例分布，其中整个分布式表从多个段收集到单个主机(通常是主节点)。

### SQL on Hadoop 架构

在Hadoop上处理分析查询正变得越来越流行。最初，查询表示为MapReduce工作和Hadoop的吸引力在于其可扩展性和容错能力。编码，手动优化和维护MapReduce中的复杂查询非常困难，因此像类似SQL的声明性语言是在Hadoop之上开发的。HiveQL查询被编译到MapReduce作业中，并由Hadoop执行。 HiveQL加快了复杂查询的编码速度，但同时也很明显地表明，Hadoop生态系统，因为已编译的MapReduce作业显示了较差的性能。

Pivotal通过引入HAWQ 来应对挑战，它是一种在HDFS之上的大规模并行SQL兼容引擎。 HAWQ以Orca为核心来设计有效的查询计划，从而最大程度地降低访问Hadoop集群中数据的成本。 HAWQ的体系结构将创新的基于成本的最先进的优化器与Hadoop的可伸缩性和容错能力相结合，以实现PB级数据的交互式处理。

最近，包括Cloudera的Impala和Facebook的Presto在内的许多其他努力引入了新的优化器，以在Hadoop上应用SQL处理。当前，这些工作仅支持SQL标准功能的一部分，并且其优化仅限于基于规则的。相比之下，HAWQ具有完全符合标准的SQL接口和基于成本的优化器，这两者都是Hadoop查询引擎中前所未有的功能。

### Orca架构

# 基准测试优化

## TPCC

## TPCH

参考：

TPCH 22条SQL语句分析：<https://developer.aliyun.com/article/149715>

从TPC-H分析论文学习优化器的挑战与应对思路：

<https://zhuanlan.zhihu.com/p/608897702>

TPC-H分析详解及其调优：<https://zhuanlan.zhihu.com/p/566024607>

[PolarDB处理TPC-H查询的挑战和机遇](https://help.aliyun.com/zh/polardb/polardb-for-mysql/challenges-and-opportunities-for-polardb-to-process-tpc-h-queries)

## TP-CDS

# 优化器

## TDSQL

### Proxy/SQL优化(2.0)

#### 逻辑查询优化

##### 子查询优化

1、优先尝试优化不需要进行上下文推导的简单子查询（即不需要推导主子查询关系的这种情况），如果不可以优化，对于非关联子查询进行物化

2、针对可以优化的子查询：

1）移除多余子查询表达式：

IN/ALL/ANY/EXISTS子查询中仅有单行结果集的时候，可以直接将子查询提取到select list中，即：

SELECT \* FROM tb WHERE tb.a = (<single row subquery>);

SELECT a, (<single row subquery>) FROM tb;

2）为ON/WHERE/HAVING中的子查询构造上下文：

3）EXISTS子查询，如下场景可以优化：

子查询没有group by/having

非singlerow场景不应该带有聚合函数

4）ANY/ALL优化：

采用Max/Min重写ANY/ALL，即：

SELECT \* FROM tb1 WHERE b > ANY (SELECT a FROM tb2);

SELECT \* FROM tb1 WHERE b > (SELECT MIN(a) FROM tb2);

###### 移除子查询冗余子句

###### Semi-Join

###### ANY/ALL等价改写

###### 子查询下推

###### 物化临时表

###### IN重写EXISTS

###### 虚表优化

1、不可以合并的场景：

1）存在group by/聚合函数

2）存在limit

3）存在distinct

2、排序字段下推，虚表子查询中的排序操作上提：

1）上层select不是union

2）上层select没有group by、聚合函数

3）上层select没有distinct

4）上层select没有order by

5）上层select只包含derived table一个表（不存在join）

##### 视图重写

TDSQL支持对视图的优化。优化的方法是把视图转换为对基表的查询，然后进行类似子查询的优化。TDSQL只支持重写简单的视图，复杂的视图不能重写。

函数：sql\_parse.cpp/ MYSQL\_PARSER::unfold\_view

##### 等价谓词重写

TDSQL支持等价谓词重写，会收集Join ON/WHERE中的等值链。

代码：join\_optimize/CJoinOptimizer::build

##### 条件化简

TDSQL主持的条件化简与MySQL一致，主要包括去除表达式中冗余的括号，常量传递，消除死码，表达式计算等。

代码：optimize\_cond.cpp/ my\_optimize\_cond（MySQL原生optimize\_cond）

##### JOIN优化

###### 外连接消除

TDSQL支持外连接转换为内连接，转换的条件是内表符合“空值拒绝”。

###### 子查询下推

###### 物化子查询

###### 最大资源限制

TDSQL为了避免Join大结果的时候导致计算节点卡死的现场，采用配置文件配置参与join的sql函数，单表可占用空间大小，后台参与join最大线程数。

##### UNION ALL优化

##### 语义优化

##### 针对非SPJ优化

###### 聚合函数优化

1、聚合函数下推

1）如果所有聚合函数都带有distinct，则将distinct下推

2）针对AVG聚合函数，为减少proxy的内存占用，采用拆分sum和count的方式优化：avg=sum/count

3）下推聚合函数，且存在group by，则加上group by

如果SQL中的聚合函数都能够下推DB执行，则将语法树中的聚合函数用获取的结果集替换，然后在此基础上proxy执行计算。

2、聚合函数不能下推

利用mysql的函数，按照新获取的列更新聚合函数

###### GROUPBY优化

1、存在group by时，判断group by的列是否是select list的子集，则将结果写入到临时表进行去重

2、不存在group by且无聚合函数时，将select list字段作为排序和分区的字段

###### ORDERBY优化

###### DISTINCT优化

###### LIMIT优化

TDSQL支持LIMIT的优化，包括尽可能将LIMIT算子下推，对于LIMIT 0直接返回空结果等。

##### UNION ALL优化

采用可配参数设置proxy收包的速率（默认4000条SQL/次）。

##### DML优化

#### 物理查询优化

##### 全局唯一索引

##### 多表关联顺序

TDSQL对多表关联的顺序进行优化，采用贪婪算法计算一个最优的执行计划。

##### 嵌入式数据库

### SQLEngine优化(3.0)

#### 子查询优化

##### 查询下推

Oceanbase查询下推优化：<https://zhuanlan.zhihu.com/p/607051422>

##### 聚合子查询提升

Oceanbase聚合子查询提升优化：<https://zhuanlan.zhihu.com/p/606747114>

##### 子查询合并

参考：

Oceanbase子查询合并：<https://zhuanlan.zhihu.com/p/602296961>

##### Where子查询提升

Oceanbase Where子查询提升：<https://zhuanlan.zhihu.com/p/604223928>

##### 半连接/Semi-Join转内连接

Oceanbase半连接转内连接优化：<https://zhuanlan.zhihu.com/p/603851213>

##### ANY/ALL子查询优化

Oceanbase ANY/ALL子查询优化：<https://zhuanlan.zhihu.com/p/602633311>

##### winmagic优化

#### JOIN优化

##### Join order生成/join reorder

Oceanbase JOIN Order生成优化：<https://zhuanlan.zhihu.com/p/610107681>

##### 连接消除

Oceanbase连接消除优化：<https://zhuanlan.zhihu.com/p/605093512>

##### 全外连接改写

Oceanbase全外连接改写：<https://zhuanlan.zhihu.com/p/604707982>

#### 聚合函数优化

##### Min/max聚合改写

Oceanbase Min-Max聚合改写：<https://zhuanlan.zhihu.com/p/606135975>

##### 窗口函数提升

Oceanbase窗口函数提升优化：<https://zhuanlan.zhihu.com/p/605363681>

#### 视图合并

Oceanbase视图合并优化：<https://zhuanlan.zhihu.com/p/603398204>

#### OR展开

OceanBase OR展开优化：<https://zhuanlan.zhihu.com/p/605834735>

#### 集合查询优化

Oceanbase集合语句优化：<https://zhuanlan.zhihu.com/p/602939413>

### 并行查询

## TiDB

### 系统调优

#### 性能分析工具

##### 60秒分析法

此分析法由《性能之巅》的作者Brendan Gregg及其所在的Netflix性能工程团队公布，所用到的工具均可从发行版的官方源获取，通过分析以下清单中的输出，可定位大部分常见的性能问题。

uptime

dmesg | tail

vmstat 1

mpstat -P ALL 1

pidstat 1

iostat -xz 1

free -m

sar -n DEV 1

sar -n TCP,ETCP 1

top

##### perf

Perf是Linux内核提供的一个重要的性能分析工具，它涵盖硬件级别（CPU/PMU和性能监视单元）功能和软件功能（软件计数器和跟踪点）。

##### BCC/bpftrace

CentOS从7.6版本起，内核已实现对bpf的支持，因此可根据上述清单的结果，选取适当的工具进行深入分析。相比perf/trace，bpf提供了可编程能力和更小的性能开销。相比kprobe，bpf提供了更高的安全性，更适合在生产环境上使用。

#### 性能调优

##### 处理器

###### 动态节能技术

cpufreq是一个动态调整CPU频率的模块，可支持五种模式。为保证服务性能应选用performance模式，将CPU频率固定工作在其支持的最高运行频率上，不进行动态调节，操作指令为cpupower frequency-set --governor performance。

###### 中断亲和性

自动平衡：可通过irqbalance服务实现。

手动平衡：

1. 确定需要平衡中断的设备，从CentOS7.5开始，系统会自动为某些设备及其驱动程序配置最佳的中断关联性。不能再手动配置其亲和性。目前已知的有使用be2iscsi驱动的设备，以及NVMe设置；
2. 对于其他设备，可查询其芯片手册，是否支持分发中断，若不支持，则该设备的所有中断会路由到同一个CPU上，无法对其进行修改。若支持，则计算smp\_affinity掩码并设置对应的配置文件。

##### NUMA绑核

为尽可能的避免跨NUMA访问内存，可以设置线程的CPU亲和性来实现NUMA绑核。对于普通程序，可使用numactl命令来绑定。

##### 内存

###### 透明大页

对于数据库应用，不推荐使用THP，因为数据库往往具有稀疏而不是连续的内存访问模式，且当高阶内存碎片化比较严重时，分配THP页面会出现较大的延迟。若开启针对THP的直接内存规整功能，也会出现系统CPU使用率激增的现象，因此建议关闭THP。

echo never /sys/kernel/mm/transparent\_hugepage/enabled

echo never /sys/kernel/mm/transparent\_hugepage/defrag

###### 虚拟内存参数

1. dirty\_ratio百分比值。当脏的page cache总量达到系统内存总量的这一百分比后，系统将开始使用pdflush操作将脏的page cache写入磁盘。默认值为20%，通常不需要调整。对于高性能SSD，比如NVMe设备来说，降低其值有利于提高内存回收时的效率。
2. dirty\_background\_ratio百分比值。当脏的page cache总量达到系统内存总量的这一百分比后，系统开始在后台将脏的page cache写入磁盘。默认值为19%，通常不需要调整。对于高性能SSD，比如NVMe设备来说，设置较低的值有利于提高内存回收时的效率。

##### 网络

网络子系统由具有敏感连接的许多不同部分组成。因此，CentOS7网络子系统旨在为大多数工作负载提供最佳性能，并自动优化其性能。因此，通常无需手动调整网络性能。

网络问题通常是由硬件或相关设施出现问题导致的，因此在调优协议栈前，请先排除硬件问题。

尽管网络堆栈在很大程度上是自我优化的。但是在网络数据包处理过程中，以下方面可能会成为瓶颈并降低性能：

1. 网卡硬件缓存：正确观察硬件层面的丢包方法是使用ethtool -S ${NIC\_DEV\_NAME}命令观察drops字段。当出现丢包现场时，主要考虑是硬/软中断的处理速度跟不上网卡接收速度。若接收缓存小于最大限制时，也可尝试增加RX缓存来防止丢包。查询命令为：ethtool -g ${NIC\_DEV\_NAME}，修改命令为ethtool -G ${NIC\_DEV\_NAME}。
2. 硬中断：若网卡支持Receive-Side Scaling（RSS也称为多网卡接收）功能，则观察/proc/interruputs网卡中断，如果出现了中断不均衡的情况，参考处理器调优。若不支持RSS或RSS数量小于物理CPU核数，则可配置Receive Packet Steering（RPS，可以看作RSS的软件实现），及RPS的扩展Receive Flow Steering（RFS）。
3. 软中断：观察/proc/net/softnet\\_stat监控。如果除第三列的其他列的数值在增长，则应适度调大net.core.netdev\\_budge或net.core.dev.\\_weight值，使softirq可以获得更多的CPU时间。除此之外，也需要检查CPU使用情况，确定哪些任务在频繁使用CPU，能够优化。
4. 应用的套接字接收队列：监控ss -nmp的Rece-q列，若队列已满，则应考虑增大应用程序套接字的缓存大小伙使用自动调整缓存的方式。除此之外，也要考虑能够优化应用层的架构，降低读取套接字的间隔。
5. 以太网流控：若网卡和交换机支持流控功能，可通过使能此功能，给内核一些时间来处理网卡队列中的数据，来规避网卡缓存溢出的问题。对于网卡测，可通过ethtool -a ${NIC\_DEV\_NAME}命令检查是否支持/使能，并通过ethtool -A ${NIC\_DEV\_NAME}命令开启。对于交换机，请查询其手册。
6. 中断合并：过于频繁的硬件中断会降低系统性能，而过晚的硬件中断会导致丢包。对于较新的网卡支持中断合并功能，并允许驱动自动调节硬件中断数。可通过ethtool -c ${NIC\_DEV\_NAME} 命令检查，ethtool -C ${NIC\_DEV\_NAME} 命令开启。自适应模式使网卡可以自动调节中断合并。在自适应模式下，驱动程序将检查流量模式和内核接收模式，并实时评估合并设置，以防止数据包丢失。不同品牌的网卡具有不同的功能和默认配置，具体请参考网卡手册。
7. 适配器队列：在协议栈处理之前，内核利用此队列缓存网卡接收的数据，每个CPU都有各自的backlog队列。此队列可缓存的最大packets数量为netdev\\_max\\_backlog。观察/proc/net/softnet\\_stat第二列，当某行的第二列持续增加，则意味着CPU [行 -1]队列已满，数据包被丢失，可通过持续加倍net.core.netdev\\_max\\_backlog值来解决。
8. 发送队列：发送队列长度值确定在发送之前可以排队的数据包数量。默认值是1000，对于10 Gbps足够。但若从ip -s link的输出中观察到TX errors值时，可尝试加倍该数据包数量：ip link set dev ${NIC\_DEV\_NAME} txqueuelen 2000。
9. 驱动：网卡驱动通常也会提供调优参数，请查询设备硬件手册及其驱动文档。

##### 存储及文件系统

内核I/O栈链路较长，包含了文件系统层、块设备层和驱动层。

###### I/O调度器

I/O调度程序确定I/O操作何时在存储设备上运行以及持续多长时间。也称为I/O升降机。对于SSD设备，宜设置为noop。

echo noop > /sys/block/$(SSD\_DEV\_NAME)/queue/scheduler

###### 格式化参数—块大小

块是文件系统的工作单元。块大小决定了单个块中可以存储多少数据，因此决定了一次写入或读取的最小数据量。

默认块大小适用于大多数使用情况。但是，如果块大小（或多个块大小）与通常一次读取或写入的数据量相同或稍大，则文件系统将性能更好，数据存储效率更高。小文件仍将使用整个块。文件可以分布在多个块中，但这会增加运行时开销。

使用mkfs命令格式化设备时，将块大小指定为文件系统选项的一部分。指定块大小的参数随文件系统的不同而不同。

###### 挂载参数

noatime读取文件时，将禁用对元数据的更新。它还启用了nodiratime行为，该行为会在读取目录时禁用对元数据的更新。

### 软件调优

#### 配置

#### 下推计算结果缓存

### SQL性能调优

SQL是一种声明性语言。一条SQL语句描述的是最终结果应该如何，而非按顺序执行的步骤。TiDB会优化SQL语句的执行，语义上允许以任何顺序执行查询的各部分，前提是能正确返回语句所描述的最终结果。  
 SQL性能优化的过程，可以理解为GPS导航的过程。你提供地址后， GPS软件利用各种统计信息（例如以前的行程、速度限制等元数据，以及实时交通信息）规划出一条最省时的路线。这与TiDB中的SQL性能优化过程相对应。

#### 执行计划

#### SQL优化流程

##### 子查询相关的优化

##### 列裁剪

列裁剪的基本思想在于：对于算子中实际用不上的列，优化器在优化的过程中没有必要保留它们。对这些列的删除会减少I/O资源占用，并为后续的优化带来便利。下面给出一个列重复的例子：

假设表t里面有a b c d四列，执行如下语句：  
 select a from t where b > 5  
 在该查询的过程中，t表实际上只有a, b两列会被用到，而c, d的数据则显得多余。对应到该语句的查询计划，Selection算子会用到b列，下面接着的DataSource算子会用到a, b两列，而剩下c, d两列则都可以裁剪掉，DataSource算子在读数据时不需要将它们读进来。  
 出于上述考量，TiDB会在逻辑优化阶段进行自上而下的扫描，裁剪不需要的列，减少资源浪费。该扫描过程称作“列裁剪”，对应逻辑优化规则中的columnPruner。如果要关闭这个规则，可以在参照优化规则及表达式下推的黑名单中的关闭方法。

##### 关联子查询去关联

##### Max/Min消除

在SQL中包含了max/min函数时，查询优化器会尝试使用max/min消除优化规则来将max/min聚合函数转换为TopN算子，从而能够有效地利用索引进行查询。  
 根据select语句中max/min函数的个数，这一优化规则有以下两种表现形式：  
 • 只有一个max/min函数时的优化规则  
 • 存在多个max/min函数时的优化规则  
 只有一个max/min函数时的优化规则  
 当一个SQL满足以下条件时，就会应用这个规则：  
 • 只有一个聚合函数，且为max或者min函数。  
 • 聚合函数没有相应的group by语句。

##### 谓词下推

##### 分区裁剪

分区裁剪是只有当目标表为分区表时，才可以进行的一种优化方式。分区裁剪通过分析查询语句中的过滤条件，只选择可能满足条件的分区，不扫描匹配不上的分区，进而显著地减少计算的数据量。

##### TopN和Limit下推

SQL中的LIMIT子句在TiDB查询计划树中对应Limit算子节点，ORDER BY子句在查询计划树中对应Sort算子节点，此外，我们会将相邻的Limit和Sort算子组合成TopN算子节点，表示按某个排序规则提取记录的前N项。从另一方面来说，Limit节点等价于一个排序规则为空的TopN节点。  
 和谓词下推类似，TopN（及Limit，下同）下推将查询计划树中的TopN计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

##### Join Reorder

#### 控制执行计划

## PolarDB

## GoldenDB

和单机数据库相比，GoldenDB分布式数据库系统中多了一层全局概念模式到本地概念模式的映射。表数据也被水平切分到多个数据节点。分布式查询优化器的优劣直接影响着业务性能。GoldenDB分布式数据库查询优化器主要朝着如下三个方向努力：

1. 最大程度使数据操作本地化、局部化，减少网络通信的交互次数和交换的数据量，提升数据节点的并行计算是分布式数据库系统优化器致力的重要方向之一；
2. 如何降低数据全局一致性保证的开销，也是分布式数据库系统查询优化努力的方向；
3. 在分布式数据库系统中使用单机数据库的优化手段或者其变形，从而使存储节点承担更多的优化工作，在全局层面仅作少量的启发式优化。

### 影响因素

优化器的优化工作主要体现在计划树的生成上，GoldenDB的查询优化器设计实现主要考虑以下两个方面：

1. 代价模型的选择。

GoldenDB采用分布式系统代价估算模型，考虑节点间传输数据的代价，以减少数据传输的次数和数据量作为查询优化的目标，提高数据节点之间计算的并行度、减少计算节点的计算量。这主要考虑在分布式数据库系统环境中，表结构被水平或垂直拆分到多个数据节点，因此需要考虑语句如何分拆、分片之间数据如何移动、结果如何计算与合并的问题，网络通信开销不可忽视。

1. 考虑数据一致性开销。

在分布式数据库系统中，数据全局一致性机制相较于单机数据库需要更为复杂的控制。因此，如何降低数据全局一致性保证的开销，也是GoldenDB查询优化器的设计要求。

总体来讲，GOldenDB的分布式查询优化器遵循了上述的设计原则，以基于规则的优化为主，基于成本的优化为辅，在提升系统的灵活性的同时控制系统实现的复杂性。优化器内部内置大量的优化规则，通过查询重写的方式进行经验性优化。在优化规则的选择上，重点分析分片剪枝、并行执行、合并下压、条件下推、条件繁殖、排序消除、去重消除、排序下推等。

内置大量的优化规则，对上百个场景进行优化，复杂SQL语句兼容性和处理性能好，同时支持prepare预处理、执行计划缓存、数据集透传等功能，保证数据一致性条件下实现高性能SQL处理。

支持的典型优化包括：

1. 分片剪枝
2. 合并下压
3. 并行执行
4. 条件下推及条件繁殖
5. 排序下推、limit下推等
6. 聚合函数优化

得益于完善的优化器设计，使得GoldenDB对单节点、跨节点的复杂SQL的兼容支持程度很高，包括跨节点SUM、COUNT、AVG等汇聚类操作，跨节点WHERE、FROM等子查询，跨节点JOIN，跨节点GROUP BY、ORDER BY、LIMIT等。这是GoldenDB将Proxy命名为计算节点而非中间件的原因之一，也是其和很多分布式数据库产品中间件的主要区别。

### Proxy优化

#### LVS

#### 池化技术

##### 线程池

##### 连接池

Proxy对前端采用TCP长链接，客户端一次性将语句下达至proxy；

Proxy对后端采用连接池的方式处理，连接池的数量及用户密码可配置，和端口是一一对应的关系；

Proxy根据应用访问的端口号来选择对应的集群以及具体的连接；

Proxy连接池耗尽后是否可以动态申请可以配置。

#### 缓存

GoldenDB通过构建执行计划缓存、SQL缓存，提高SQL解析效率、提高数据读取效率，在高频度读写系统中可以减少磁盘I/O负担，提升整体系统效率。

##### 元数据缓存

Proxy本地会缓存元数据，当MDS发生元数据变更的时候会推送到proxy。

缓存信息包括表结构信息、SQL语句缓存、结果集缓存。

##### 执行计划缓存

##### 结果集缓存

#### 并发控制机制

GoldenDB在事务处理上通过采用成熟的主流技术来实现高效的事务管理，这些技术包括：以行级为主表级锁为辅的锁技术、多版本并发控制技术、全乐观锁+自动补偿机制。这些技术在保证事务ACID特征的前提下大大提高了事务的并发处理能力。

#### 流控

##### 限流

根据消息积压数进行计算，设置流量限定阈值。

Proxy对上做局部流控，如果当前连接数超过配置的最大连接数，要等某连接上的在线事务全部处理结束后再关闭该连接。

TDSQL对于热点数据处理是在计算节点采用一个hash表预先判断更新数据的分布，GoldenDB采用限流的方案（根据消息积压数计算），如果检测到分发到某个节点的写语句过多，则执行限流，这个粒度（针对group）相对比较大一些。

##### 黑名单

对于某些耗时比较久的SQL，会加入黑名单中。

##### 分包/分布式批处理

对于大结果的INSERT或者SELECT会根据具体的SQL数量分包下发，提高效率。

对于大结果集进行分包处理（结果集在内存中缓存），防止一次性处理卡顿。

分布式批处理。提供分布式架构下批处理功能，满足金融、政企、运营商等行业日终大数据批处理的要求，通过分布式FetchSize和存储过程功能对数据进行批处理，减少客户端与DB的交互次数，批量返回数据集并进行批量处理。

#### 热点数据

使用重分布解决热点数据问题。

#### 结果集透传

#### 读写分离

读写分离是指利用数据节点集群安全组多副本，将部分读请求发往备节点，提升系统的读能力。

在启动读写分离时，GoldenDB的计算节点在受到应用SQL请求时，根据当前的语句类型和负载策略选择SQL下发的数据节点，将写操作发往主节点，将读操作发往备节点。

注意：GoldenDB的读写分离是事务间的读写分离，如果一个事务内同时存在读写请求，该事务的所有SQL都会发往主节点。

在多个应用接入一个数据节点集群时，为了满足不同应用的需求，GoldenDB支持对同一集群不同的连接实例，设置不同的读写分离模式，包括以下三种：

1. 读主节点。读操作默认发往主节点，当应用强制指定将读请求发往备节点时，读请求即在备机间做负载均衡；
2. 读备节点。读操作仅在备机间根据配置的权重做读负载均衡。

除了上述连接实例级别的读写分离模式设置，GoldenDB还支持SQL级别的读写分离模式指定，应用可以在SQL语句后面添加hint信息强制发往主节点或备节点，SQL级别的优先级高于连接实例级别。常见的使用场景如下：

1. 由于数据在安全组内部的节点间同步存在时延，因此对实时性要求比较高的SQL请求，应用希望将其发往主节点；
2. 对于一些SQL如分析聚合类SQL，应用希望将其发往备节点，减少对主节点的影响。

结合上述连接实例和SQL级别的读写分离模式设置，应用可以根据自己的希望设计合理的读写分离策略。

#### group优先级

一般先读同城，再读异地。

#### 负载均衡

可以设置不同group的优先级，同城机房高于异地灾备机房。

### 分区裁剪/合并下压优化

这个类似TDSQL的分区裁剪。

GoldenDB的分布式优化，很重要的一个努力方向就是尽量利用数据节点的计算能力进行计算，避免不必要的从数据节点向计算节点的数据移动，并减少和数据节点交互的次数。计算节点分析语句后，尽量把能够一起执行的语句下发到数据节点。

注：这里涉及到表层次合并、主子查询合并以及JOIN等合并下发规则的判断。

SQL示例：

假定T1和T2分发属性相同：range表、分发键为col、分布在g1,g2,g3节点上，将“条件繁殖”部分的示例语句改写如下：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col=200 UR;

见“where条件下推”部分，根据where子句可以将查询数据定位到某个group上，则可以直接将语句下发到group节点上执行。

优化后执行计划：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col=200;

#### 表层次

##### 复制表+复制表

如果dup\_1所在的db分组与dup\_2所在的db分组有公共的交集，则可以合并下发，否则不能合并下发该语句；无需考虑等值链和值链对合并结果的影响。

例如：

dup\_t1分布在g1,g2,g3，dup\_t2分布在g1,g3

结论：合并下发

dup\_t1分布在g1,g3，dup\_t2分布在g2,g4

结论：不可以合并下发

##### 复制表+非复制表

1. 无where条件或者where条件为非分发键

如果复制表所在的db分组要完全覆盖非复制表所在的分组，则可以合并下发，否则不能合并下发该语句。

例如：

dup\_t分布在g1,g2,g3，非复制表分布在g1,g2,g3

结论：可以合并

dup\_t分布在g1,g2，非复制表分布在g1,g3

结论：不可以合并

1. where条件为分发键

如果非复制表经过where值的过滤后能落在单个db上，则判断依据参考复制表+复制表合并依据，否则参考复制表+非复制表且无where条件或where条件为非分发键合并依据。

例如：

dup\_t分布在g1,g3，非复制表分布在g1,g2

where条件：range\_t.a>20and range\_t.a<50

结论：可以合并

dup\_t分布在g1,g3，非复制表分布在g1,g2

where条件：range\_t.a>100 and range\_t.a<250

结论：不可以合并

##### hash表+hash表

1. 无where条件或where条件为非分发键

如果两个hash表所在的db分组一致且关联字段都是两个表的分发键时，则可以合并下发，否则不可合并下发。

1. where条件为分发键

如果通过值链和等值链过滤后有一个表被过滤落在单个db，则判断依据参考复制表+非复制表且where为分发键的合并依据。

如果通过值链和等值链过滤后两个表都落在单个db上，则判断依据参考复制表+复制表合并依据；否则参考hash表+hash表且无where条件或where为非分发键合并依据。

##### range表+range表

1. 无where条件或者where条件非分发键

如果两个range表同时满足如下条件，并且join on关联条件为每个表的分发键时，则可以合并下发，否则不可合并下发：

1. 两表的db分布完全一致
2. 满足条件1后，range1表在每个db上的分布范围也必须和range2在每个db上的分布范围完全一致

例如：

range1分布在g1范围为[-∞,-100)，g2范围为[100,200)

range2分布在g1范围为[-∞,-100)，g2范围为[100,200)

结论：可以合并

range1分布在g1范围为[-∞,-50)，g2范围为[50,100)

range2分布在g1范围为[-∞,-60)，g2范围为[60,100)

结论：不可以合并

1. where条件为分发键

如果两个range表同时满足如下条件，并且join on关联条件为每个表的分发键时，则可以合并下发，否则不能合并下发：

1. 两表的db分布完全一致
2. 满足1条件后，经过where过滤后的range1表的g1和g2范围分布落在range2表原始大范围内，同时经过where过滤后的range2表的g1,g2小范围落在range1的g1,g2原始大范围内

例如：

range1分布在g1范围为[-∞,-50)，g2范围为[50,100)

range2分布在g1范围为[-∞,-60)，g2范围为[60,90)

where条件为：

range1.c>30 and range1.c<40 and range2.b>30 and range2.b<30

结论：可以合并

range1分布在g1范围为[-∞,-50)，g2范围为[50,100)

range2分布在g1范围为[-∞,-60)，g2范围为[60,150)

where条件为：

range1.c>30 and range1.c<40 and range2.b>110 and range2.b<140

结论：不可以合并

##### list表+list表

1. 无where条件或者where条件为非分发键

如果两个list表同时满足如下条件，并且join on关联条件为每个表的分发键时，则可以合并下发，否则不可合并下发：

1. 两表的db分布完全一致
2. 满足条件1后，list1表在某个db上的分布列表值，不能出现在list2表除了g1之外的任何db上

例如：

list1分布在g1范围为(1,6,7)，g2范围为(9,11,12)

list2分布在g1范围为(1,6,8)，g2范围为(9,13)

结论：合并下发

list1分布在g1范围为(1,6,7)，g2范围为(8,11,12)

list2分布在g1范围为(1,6,8)，g2范围为(9,13)

结论：不能合并下发

list1分布在g1范围为(1,6,7)，g2范围为(9,11,12)

list2分布在g1范围为(3,5)，g2范围为(9,13)

结论：合并下发

1. where条件为分发键

如果两个list表同时满足如下条件，并且join on关联条件为每个表的分发键时，则可以合并下发，否则不可合并下发：

1. 两表的db分布完全一致
2. 满足条件1后，经过where过滤后的list1表的g1和g2小范围分别落在list2表的g1,g2原始大范围内，同时经过where过滤后的list2表的g1,g2小范围分别落在list1表的g1,g2原始大范围内

例如：

list1分布在g1范围为(1,6,7)，g2范围为(9,11,12)

list2分布在g1范围为(1,6,8)，g2范围为(9,13)

where条件为：(list1.c=1 and list2.c=6) or (list1.c=11 or list2.b=9)

list1过滤后范围：g1范围(1)，g2范围(11)

list2过滤后范围：g1范围(6)，g2范围(9)

结论：合并下发

list1分布在g1范围为(1,6,7)，g2范围为(8,11,12)

list2分布在g1范围为(1,6,8)，g2范围为(9,11)

where条件为：(list1.c=1 and list2.b=6)

list1过滤后范围：g1范围(1)

list2过滤后范围：g1范围(6)

结论：合并下发

list1分布在g1范围为(1,6,7)，g2范围为(8,11,12)

list2分布在g1范围为(1,6,8)，g2范围为(9,11)

where条件为：(list1.c=1 and list2.b=8)

list1过滤后范围：g1范围(1)

list2过滤后范围：g1范围(8)

结论：不可以合并下发

#### 主子查询

##### FROM子查询

不能合并的场景：

1. 如果FROM子查询本身不是SQLNode或MSQLNode

代表子查询的JoinNode或UnionNode，不能合并入主查询，因为本身就不可合并

1. 同时满足：多db分布（handle\_on=3）&（有汇聚函数/distinct）&（非复制表）&（分发键不在group中）

如果非复制表多db分布且分发键不在group by中，则汇聚函数必须在proxy层做聚合和distinct计算

1. 如果FROM子查询中有不能合并的子查询时
2. 同时满足：FROM子查询多db分布且有limit

limit需要proxy计算

##### WHERE子查询

不能合并的场景：

1. where子查询为JoinNode、UnionNode
2. CR场景，where子查询不能与主查询合并

CR场景涉及gtid列，则where子查询中的gtid无法处理，比如：where t1.a in (select t2.a,t2.gtid ...)

1. Where子查询为非复制表多db分布，且有聚合函数
2. 主查询中标为复制属性，子查询表为非复制属性，分布于多个db且查询字段为非分发键时，主子查询不能合并下发（不管查询字段是否为分发键，都不能合并）

###### IN/NOT IN子查询

1. SPJ查询语句

合并下发前提：

1. 涉及的所有表都是hash分布
2. 主子查询能直接使用分发键关联
3. 多表的分发类型需要一致
4. 分发键需要NOT NULL限制

合并下发原理：

在分发键类型一致的情况下，通过唯一的hash算法可以将相同值统一的发布到相同的数据节点上。这样可以保证not in子查询数据分布的一致性，因此可以考虑合并下发。

1. 非SPJ查询语句
2. 子查询有order by，有group by+分发键，子查询投影中为分发键且有distinct，having子句

上述对合并下发没有影响。

1. 子查询group by中没有分发键：不能合并

汇聚函数计算需要在所有db数据上进行，如果没有汇聚函数情况下，group by也会隐藏部分分发键值。

1. 子查询有limit：不能合并

limit需要在所有db数据上进行操作。

1. 子查询投影中有单纯的汇聚函数且group by中包含子查询表的分发键：不能合并

在子查询投影中有汇聚函数，即使有group by+分发键也不能合并。

###### EXISTS/NOT EXISTS子查询

1. 非关联子查询：不能合并

exist/not exists存在和不存在判断是对于子查询表的所有数据生效。

1. SPJ关联子查询

合并下发前提：

1. 涉及的所有表都是hash分布
2. 主子查询能够直接使用分发键等值关联
3. 多表的分发键类型需要一致
4. 分发键需要NOT NULL限制

合并下发原理：在分发键类型一致的情况下，通过唯一的hash算法可以将等值统一的发布到相同的数据节点上。这样可以保证关联的exists子查询数据分布的一致性。因此可以考虑合并下发。

1. 非SPJ关联子查询

1）子查询有order by，有group by+分发键，having子句

上述对合并下发没有影响。

2）子查询group by中没有分发键：不能合并

汇聚函数计算需要在所有db数据上进行，如果没有汇聚函数情况下，group by也会隐藏部分分发键值。

3）子查询有limit：不能合并

limit需要在所有db数据上进行操作。

###### IN/NOT IN item\_row子查询

1、SPJ查询语句

合并下发前提：

1）涉及的所有表都是hash分布

2）主子查询能直接使用分发键关联

3）多表的分发类型需要一致

4）分发键需要NOT NULL限制

合并下发原理：

在分发键类型一致的情况下，通过唯一的hash算法可以将相同值统一的发布到相同的数据节点上。这样可以保证not in子查询数据分布的一致性，因此可以考虑合并下发。

2、非SPJ查询语句

1）子查询有order by，有group by+分发键，子查询投影中为分发键且有distinct，having子句

上述对合并下发没有影响。

2）子查询group by中没有分发键：不能合并

汇聚函数计算需要在所有db数据上进行，如果没有汇聚函数情况下，group by也会隐藏部分分发键值。

3）子查询有limit：不能合并

limit需要在所有db数据上进行操作。

4）子查询的group by子句中有分发键：与SPJ查询合并前提一致，同时需要子查询的投影与分发键相关联，只有这样才能保证最终结果没有重复数据

在子查询投影中有汇聚函数，即使有group by+分发键也不能合并。

#### UNION

1. 单DB场景

CR：

select1 UNION select2时，如果左右表的个数不一致时不能合并下发，原因是每张表都要添加gtid。

select1 UNION select2时，如果左右表的个数一致时可以进一步判断是否合并下发。

UR：

不论左右select中标的个数是否一致，都需要进一步判断是否可以合并下发。

1. 多DB场景

CR：

如果左右select中表的个数不一致，则不能合并下发。

如果左右select中表的个数一致，需要进一步判断是否可以合并下发。

UR：

select1 UNION select2同时满足如下条件才可以考虑合并下发：

1. 左右select中没有group by、没有聚合函数同时没有distinct
2. 没有使用union distinct

#### Hint

采用samedb、storagedb的hint信息优化下发db。

Samedb表示自此SQL以后的所有SQL都是采用相同分片，不需要计算分片（适用于跑批业务），storagedb适用于业务侧已知分片信息的情况。

### SQL引擎优化

日程表

描述已自动生成

#### 条件繁殖

条件繁殖是指优化器对已知条件进行推断，从而衍生出其他条件进行改造，缩小数据检索的范围。繁殖后的条件，或推入基表、或下压到数据节点执行。

SQL示例1：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col>100 UR;

优化后执行计划示例：

优化器会识别出本例中的条件传递从而推断出T2.col>100，并将该条件推入基表。语句被重写：

select T1.col,T1.col1 from {T1 where col>100} join {T2 where col>100} on T1.col=T2.col UR;

注：在等价语句改造的时候，也可以利用ON条件等做等值的传递。

SQL示例2：

select T1.col, T1.col1 from T1 join T2 on T1.col=T2.col where T1.col1 in (100,200) and T2.col1=T1.col1 UR;

优化器执行计划示例：

优化器会识别出本例中的条件传递从而推出T2.col1 in (100,200)，并将该条件推入基表。语句被重写为：

select T1.col, T1.col1 from {T1 where col1 in (100,200)} join {T2 where col1 in (100,200)} von T1.col=T2.col UR;

#### OR索引失效优化

对于cond1 OR cond2这种会导致索引失效，可以采用如下的方法：

cond1 UNION cond2。

#### 并行执行

并行执行是指执行计划在各个分区间进行并行执行，从而提升执行效率。

当SQL查询在分区剪裁后，仍然涉及多个分区时，会生成一个分布式执行计划，该分布式计划会被调度到分区所在不同机器上进行执行。GoldenDB在判断语句需要下发到多个分区时，会将语句拆分成多个同时下发到对应的节点并行执行。

SQL示例：

假定T1、T2在g1,g2,g3三个分区上：

select T1.col,T1.col1 from T1 join T2 on T1.col=T2.col where T1.col1=200 UR;

优化后执行计划，语句被拆分如下：

select T2.col from T2 order by T2.col ASC;

select T1.col,T1.col1 from T1 where T1.col1=200 order by T1.col ASC;

同时下发到对应的三个节点上并行执行后，将结果汇总到proxy层做join。

#### AVG优化

在GoldenDB中，AVG被自动重写成SUM和COUNT两个计算，在每个数据节点上，只返回本数据节点的SUM、COUNT；在计算节点层面，再对各数据节点返回的SUM、COUNT进行累计，然后再用SUM/COUNT得到AVG的最终结果。

SQL示例：

select avg(col) from T where col>100 UR;

优化后执行计划示例：

select sum(col),count(col) from T where col>100;

#### JOIN优化

采用MULTI\_STEP\_QUERY强制使用小表驱动大表。

#### 下推优化

##### WHERE条件下推

在分布式数据库系统实现中，为了尽量减少数据节点向计算节点移动的数据量，系统被设计为尽可能将where条件下推到数据节点。

SQL示例：

假定T为单分发键表，且分发键为col：

select col,col1 from T where col=100 and col1>10 UR;

优化后执行计划示例：

根据where子句可以将查询数据定位到某（几）个GROUP，执行时将语句直接下发到对应的group节点上执行：

select col,col1 from T where col=100 and col1>10;

##### order by下推

GoldenDB对于排序处理通常会优先考虑推入数据节点完成。利用数据节点的计算能力并行完成排序操作；涉及结果合并的，计算节点再对有序数据集进行合并排序。

SQL示例：

select col,col1 from T where col1>100 order by col UR;

优化后执行计划示例：

将语句下发到各个节点上并行执行，并在proxy层汇总结果，如果下发为多节点，则需要执行sort merge排序操作。

select col,col1 from T where col1>100 order by col;

##### distinct下推

GoldenDB中，遇到不能合并下发的SQL语句，如果其中含有distinct，则计算节点在拆分语句时，会考虑将distinct下推入数据节点执行。以减少从数据节点提取到计算节点的数据量。

UR场景下，如果查询数据分布在同一个节点上或者select list为分发键的情况下，将distinct下推入节点执行，汇总结果不需要在proxy层再做distinct。

SQL示例：

假定T为多分发键表，且分发键为col1,col1：

select distinct col,col1 from T where col>100 and col1=20 UR;

优化后执行计划示例：

select distinct col,col1 from T where col>100 and col1=20;

##### limit下推

在GoldenDB中，limit下推的主要目的是在需要计算节点进一步计算的场景下，尽量减少从数据节点提取到计算节点的数据量。

优化原则：

1. SQL语句能下发到一个db group执行的，limit子句不用调整
2. SQL语句下发到多个db group执行的，需要在proxy层汇总数据，做limit操作：
3. SQL语句能下发，但是需要下发到多个db group执行的，limit子句需要调整，调整格式如下：limit x,y --> limit 0,x+y
4. SQL语句不能下发，需要把数据拉到db proxy层计算的，limit子句不变

SQL示例：

查询数据分布在多个节点上，且查询语句能下发：

select col,col1 from T where col1>100 limit 2,2 UR;

优化后执行计划：

select col,col1 from T where col1>100 limit 0,4;

#### 常数折叠

在GoldenDB中，为了减少对确定值的反复计算而先进行计算的优化方法。此过程一般发生在S（Select）F（From）W（Where）中SW阶段。

SQL示例：

假定T为单分发键表，且分发键为col：

select col,col1 from T where col=50+50 and col1>10 UR;

优化后执行计划示例：

在此查询中，50+50会被先计算成100。查询重写后，条件变为where col=100 and col1>10。根据重写后的where条件可以将查询语句定位到某（几）个group上执行，执行时下发原始条件。

select col,col1 from T where col=50+50 and col>10;

#### 非逻辑优化

在GoldenDB中，会针对NOT运算进行处理，通常是将其下推，将表达式整体取反变为表达式分量补集的运算。

|  |  |
| --- | --- |
| 处理前 | 处理后 |
| NOT (col!=5) | col=5 |
| NOT(col1<=4 OR col2>0) | col1>4 AND col2<=0 |
| NOT(col1<=4 AND col2>0) | col1>4 OR col2<=0 |

经过变换，可以减少一次逻辑运算并在一定条件下使范围扫描可用。

注：对于WHERE cond1 OR cond2这种，cond条件中的字段会索引失效，可以采用UNION ALL替换的方式，即WHERE cond1 UNION ALL WHERE cond2。

#### 死代码消除

GoldenDB分布式优化器中的处理逻辑通过判断出为恒指或者逻辑冗余的条件，然后在运行时减少不必要的逻辑判断，从而提升执行效率。

SQL示例：

假定T为range表，col为单分发键：

select col,col1 from T where col>0 and col > 200 UR;

优化后执行计划：

此例子中，根据where条件确定下发group（col>200所在group）：

select col,col1 from T where col>0 and col>200;

SQL示例：

select col,sol1 from T where col<0 and col>200 UR;

优化后执行计划：

此例中，根据where条件获取下发group为0，选择一个group下发执行：

select col,col1 from T where col<0 and col>200;

#### 全局(唯一)索引

#### 锁

采用乐观锁（重试）和悲观锁（select for update）两种控制策略，针对不同场景设置不同锁类型。

对于聚合函数的场景，采用共享锁，而不是排它锁，减少加锁范范围（但是也带来了一致性问题）。

#### force index

为了避免update、delete中where条件索引失效，造成全表锁（对于悲观锁，proxy会先下发select for update where锁住对应数据，然后执行更新），采用在where条件中增加force\_index。

#### Hint

##### MULTI\_STEP

存在两表或多表关联操作时，可以使用MULTI\_STEP\_QUERY表示语句被拆分为多步骤执行，目的是提高查询性能。

SQL查询语句中表不需要一定出现在MULTI\_STEP\_QUERY中，如果MULTI\_STEP\_QUERY中定义了表，表的先后顺序表示在SQL语句的执行计划中的执行顺序。

示例：

SELECT

s.s\_name,c.c\_id, concat(c.c\_last, c.c\_first), o.o\_all\_local, o.o\_ol\_cnt

FROM

customer c inner join oorder o

on c.c\_id = o.o\_c\_id

left join warehaouse w

on o.o\_w\_id = w.w\_id

where

w.w\_id = 193 and c.c\_id >= 435 and c.c\_id <= 445

order by w.w\_id, c.c\_id, o.o\_id;

数据量：

customer表：1500万

oorder表：1500万

warehouse表：500万

首先使用原始语句执行，单条语句在16C32G虚拟机上执行报错：

ERROR 10435(HY000):ERR Write IO\_CACHE Fail!

内存监控发现内存写满了，使用count查看结果集，显示为565000条。

**优化方案一：**

修改hash分发方式，保证语句群发，join操作在DB层面执行，proxy层制作排序：

Customer表修改为DISTRIBUTED BY HASH(c\_id)(g1,g2,g3,g4);

初始的分发方式为DISTRIBUTED BY HASH(c\_w\_id)(g1,g2,g3,g4);

Ooder表修改为DISTRIBUTED BY HASH(o\_c\_id)(g1,g2,g3,g4);

初始的分发方式为DISTRIBUTED BY HASH(o\_w\_id)(g1,g2,g3,g4);

warehouse表修改为DISTRIBUTED BY DUPLICATE(g1,g2,g3,g4);

初始的分发方式为DISTRIBUTED BY HASH(w\_id)(g1,g2,g3,g4);

但语句耗时8.89秒

注：该方法不可行，业务不能为了一条语句的提升而导致整体业务性能下降。

**优化方案二：**

分析3张表的数据情况，发现可以使用分布式数据库特有的功能MULTI\_STEP\_QUERY强制先用小表进行JOIN：

SELECT

s.s\_name,c.c\_id, concat(c.c\_last, c.c\_first), o.o\_all\_local, o.o\_ol\_cnt

FROM

customer c inner join oorder o

on c.c\_id = o.o\_c\_id

left join warehaouse w

on o.o\_w\_id = w.w\_id

where

w.w\_id = 193 and c.c\_id >= 435 and c.c\_id <= 445

order by w.w\_id, c.c\_id, o.o\_id MULTI\_STEP\_QUERY(w,o,c);

##### NOGTID

##### READMASTER

##### READSLAVE

##### READBALANCE

##### storagedb

不需要计算分片，直接到对应的group。

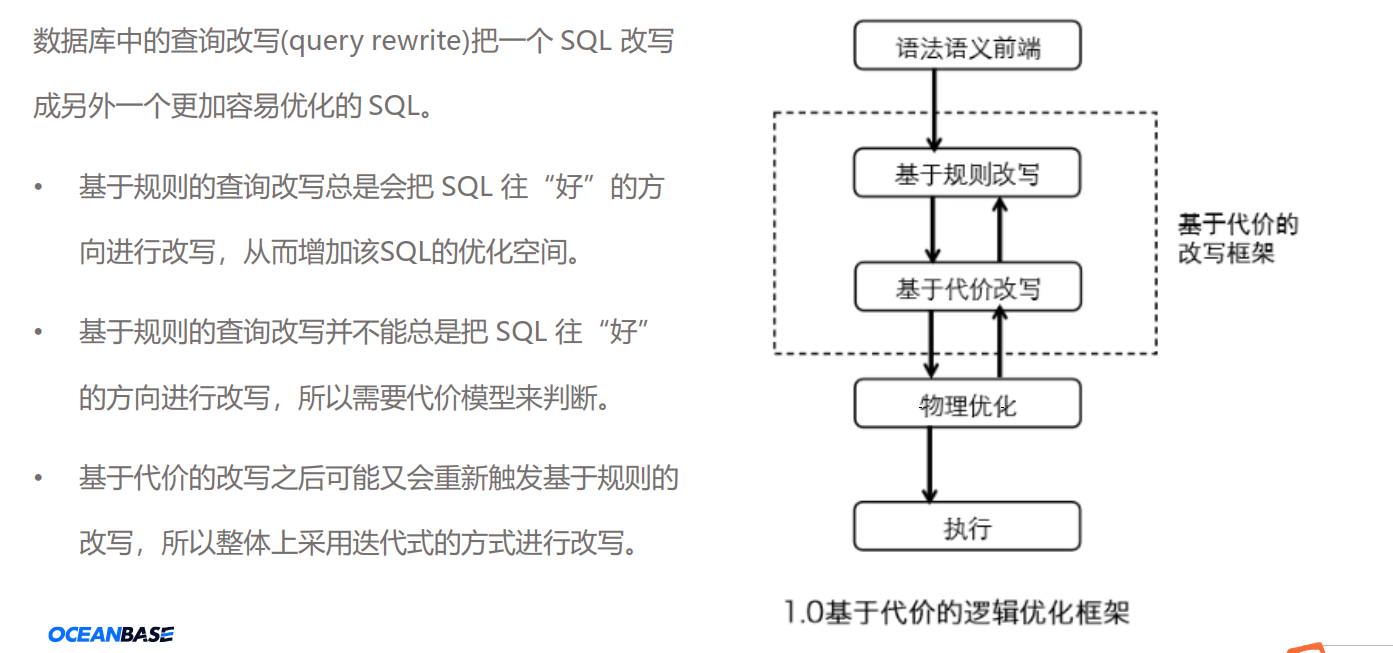
##### samedb

如果可以确定某一个操作全部是对一个group的操作，前面第一个计算分片信息后，后面的全部添加samedb的hint信息，这样就可以直接用前面缓存的group信息了。

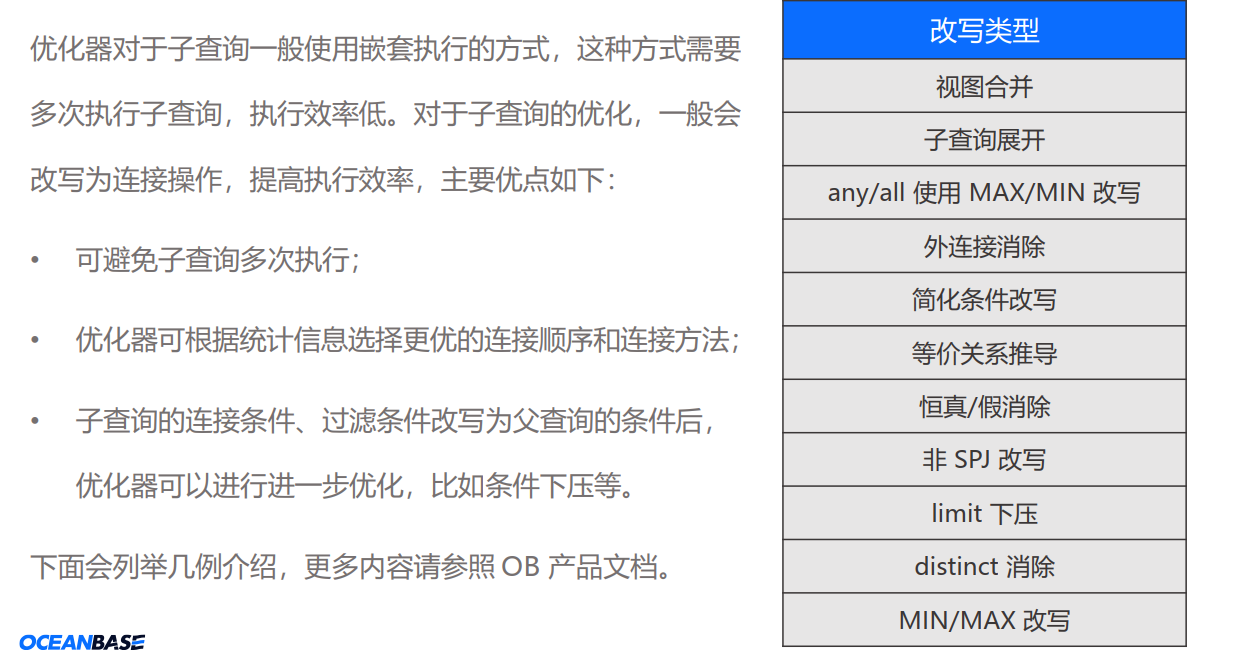
### 执行器

采用MPP大数据组件presto执行OLAP的查询。

## OceanBase



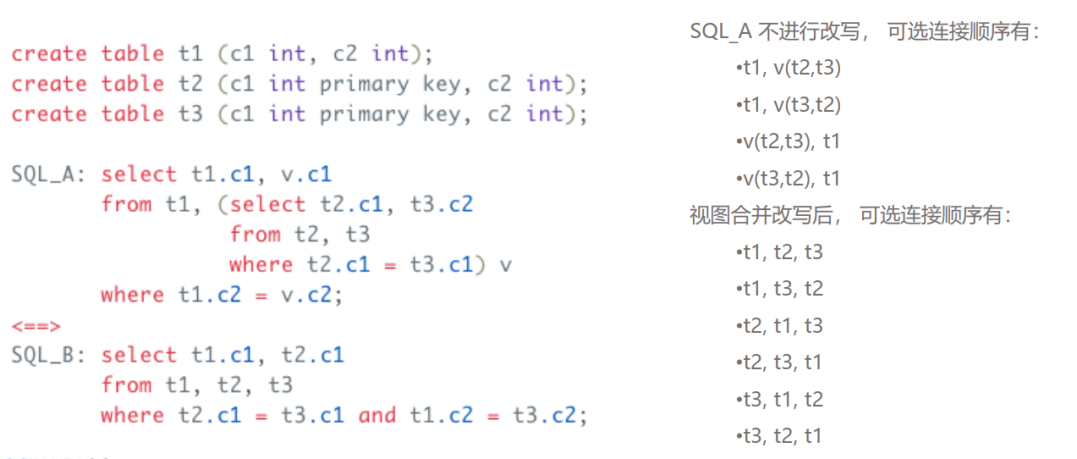
### 基于规则查询改写



#### 子查询相关改写

##### 视图合并

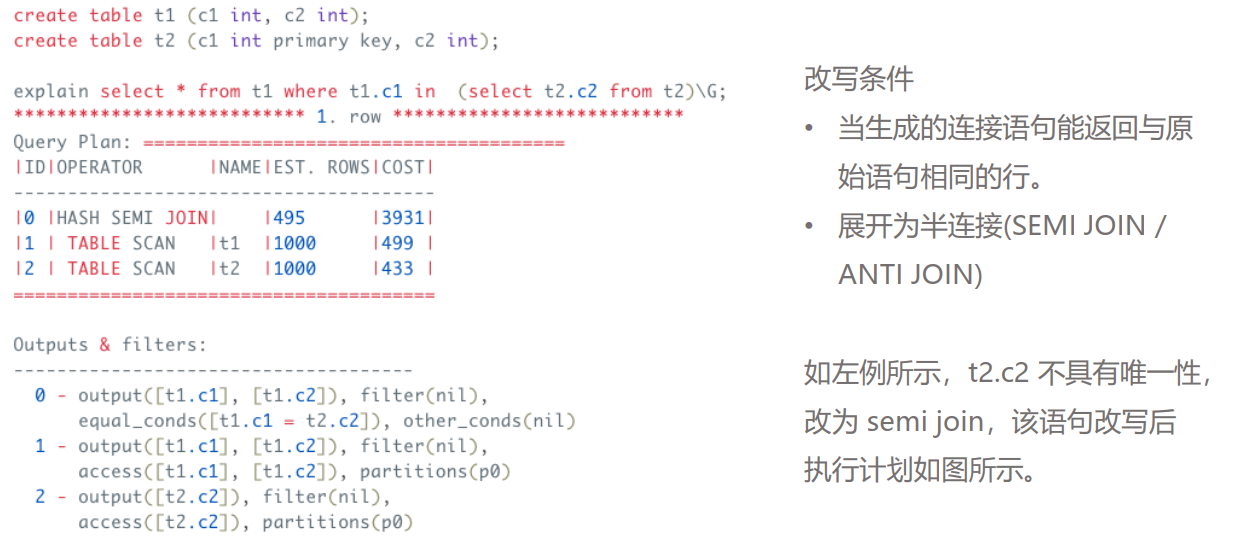
视图合并是指将代表一个视图的子查询合并到包含该视图的查询中，视图合并后，有助于优化器增加连接顺序的选择、访问路径的选择以及进一步做其他改写操作，从而选择更优的执行计划。



##### 子查询展开

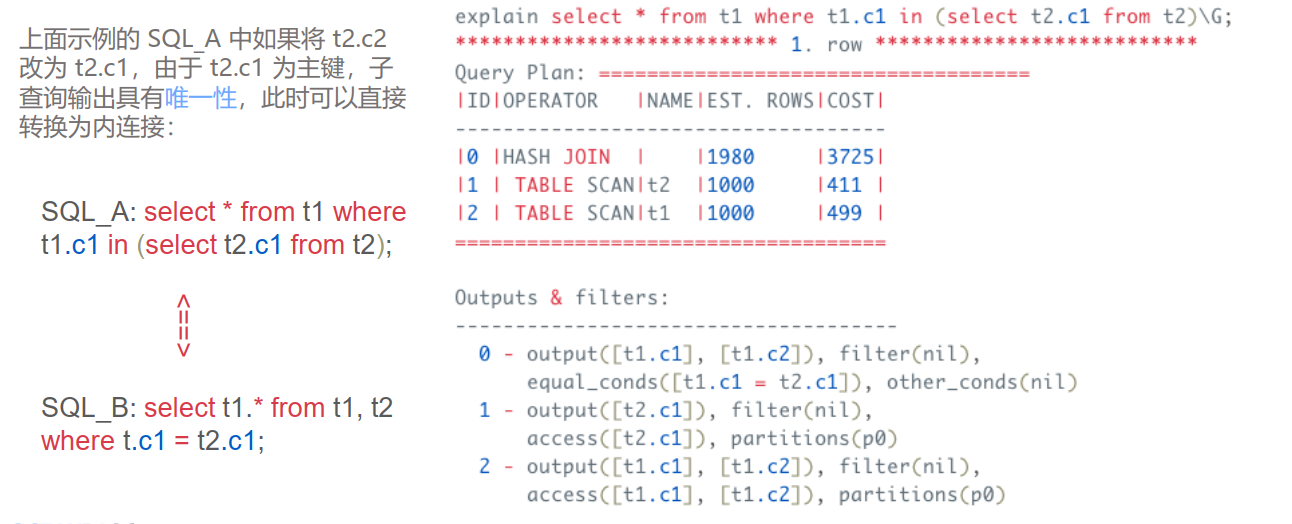
###### 子查询展开为semi-join/anti-join

子查询展开是指将where条件中子查询提升到父查询中，并作为连接条件与父查询并列进行展开。一般涉及的子查询表达式有not in、in、not exist、exist、any、all



###### 子查询展开为内连接

子查询展开是指将 where 条件中子查询提升到父查询中，并作为连接条件与父查询并列进行展开。一般涉及的子查询表达式有not in、in、not exist、exist、any、all。



#### ANY/ALL使用MAX/MIN改写

对于ANY/ALL的子查询，如果子查询中没有GROUP BY子句、聚集函数和HAVING条件时，以下表达式可以使用聚集函数MIN/MAX进行等价转换，其中 col\_item 为单独列且有非 NULL 属性：

val > ALL(SELECT col\_item ...) <==> val > (SELECT MAX(col\_item) ...);

val >= ALL(SELECT col\_item ...) <==> val >= (SELECT MAX(col\_item) ...);

val < ALL(SELECT col\_item ...) <==> val < (SELECT MIN(col\_item) ...);

val <= ALL(SELECT col\_item ...) <==> val <= (SELECT MIN(col\_item) ...);

val > ANY(SELECT col\_item ...) <==> val > (SELECT MIN(col\_item) ...);

val >= ANY(SELECT col\_item ...) <==> val >= (SELECT MIN(col\_item) ...);

val < ANY(SELECT col\_item ...) <==> val < (SELECT MAX(col\_item) ...);

val <= ANY(SELECT col\_item ...) <==> val <= (SELECT MAX(col\_item) ...);

将子查询更改为含有 MAX/MIN 的子查询后，再结合使用 MAX/MIN 进行改写，可减少改写前对内表的多次扫描，如下例所示：

obclient> SELECT c1 FROM t1 WHERE c1 > ANY(SELECT c1 FROM t2);

<==>

obclient> SELECT c1 FROM t1 WHERE c1 > (SELECT MIN(c1) FROM t2);

结合 MAX/MIN 进行改写后，可利用 t2.c1 的主键序将 LIMIT 1 直接下压到 TABLE SCAN，将 MIN 值输出，执行计划如下所示：

obclient> EXPLAIN SELECT c1 FROM t1 WHERE c1 > ANY(SELECT c1 FROM t2);

Query Plan:

===================================================

|ID|OPERATOR |NAME |EST. ROWS|COST|

---------------------------------------------------

|0 |SUBPLAN FILTER | |1 |73 |

|1 | TABLE SCAN |t1 |1 |37 |

|2 | SCALAR GROUP BY| |1 |37 |

|3 | SUBPLAN SCAN |subquery\_table|1 |37 |

|4 | TABLE SCAN |t2 |1 |36 |

===================================================

Outputs & filters:

-------------------------------------

0 - output([t1.c1]), filter([t1.c1 > ANY(subquery(1))]),

exec\_params\_(nil), onetime\_exprs\_(nil), init\_plan\_idxs\_([1])

1 - output([t1.c1]), filter(nil),

access([t1.c1]), partitions(p0)

2 - output([T\_FUN\_MIN(subquery\_table.c1)]), filter(nil),

group(nil), agg\_func([T\_FUN\_MIN(subquery\_table.c1)])

3 - output([subquery\_table.c1]), filter(nil),

access([subquery\_table.c1])

4 - output([t2.c1]), filter(nil),

access([t2.c1]), partitions(p0),

limit(1), offset(nil)

#### 外连接消除

外连接消除是指将外连接转换成内连接，从而可以提供更多可选择的连接路径，供优化器考虑。外连接消除需要存在“空值拒绝条件”，即where条件中，存在当内表生成的值为null时，使得输出为false的条件。  
 示例：



这是一个外连接，在其输出行中t2.c2可能为null。 如果加上一个条件t2.c2 > 5，则通过该条件过滤后，t2.c1输出不可能为NULL，从而可以将外连接转换为内连接：



#### 简化条件改写

##### HAVING条件消除

如果查询中没有聚集操作和GROUP BY，则 HAVING可以合并到WHERE条件中，并将HAVING条件删除，从而可以将HAVING条件在 WHERE条件中统一管理，并做进一步优化。

obclient>SELECT \* FROM t1, t2 WHERE t1.c1 = t2.c1 HAVING t1.c2 > 1;

<==>

obclient>SELECT \* FROM t1, t2 WHERE t1.c1 = t2.c1 AND t1.c2 > 1;

改写后计划如下例所示, t1.c2 > 1 条件被下压到了 TABLE SCAN 层。

obclient> EXPLAIN SELECT \* FROM t1, t2 WHERE t1.c1 = t2.c1 HAVING t1.c2 > 1;

Query Plan:

=========================================

|ID|OPERATOR |NAME|EST. ROWS|COST|

-----------------------------------------

|0 |NESTED-LOOP JOIN| |1 |59 |

|1 | TABLE SCAN |t1 |1 |37 |

|2 | TABLE GET |t2 |1 |36 |

=========================================

Outputs & filters:

-------------------------------------

0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil),

conds(nil), nl\_params\_([t1.c1])

1 - output([t1.c1], [t1.c2]), filter([t1.c2 > 1]),

access([t1.c1], [t1.c2]), partitions(p0)

2 - output([t2.c1], [t2.c2]), filter(nil),

access([t2.c1], [t2.c2]), partitions(p0)

##### 等价关系推导

等价关系推导是指利用比较操作符的传递性，推倒出新的条件表达式，从而减少需要处理的行数或者选择到更有效的索引。

OceanBase 数据库可对等值联接进行推导，例如，一张表中有 a 列和 b 列，通过 a ＝ b AND a > 1 可以推导出 a ＝ b AND a > 1 AND b > 1， 如果 b 列上有索引，且 b > 1 在该索引选择率很低，则可以大大提升访问 b 列所在表的性能。

如下例所示，条件 t1.c1 = t2.c2 AND t1.c1 > 2，等价推导后为 t1.c1 = t2.c2 AND t1.c1 > 2 AND t2.c2 > 2，从计划中可以看到 t2.c2 已下压到 TABLE SCAN，并且使用了 t2.c2 对应的索引。

obclient> CREATE TABLE t1(c1 INT PRIMARY KEY, c2 INT);

Query OK, 0 rows affected

obclient> CREATE TABLE t2(c1 INT PRIMARY KEY, c2 INT, c3 INT, KEY IDX\_c2(c2));

Query OK, 0 rows affected

/\*此命令需运行于 MySQL 模式下\*/

obclient> EXPLAIN EXTENDED\_NOADDR SELECT t1.c1, t2.c2 FROM t1, t2 WHERE t1.c1 = t2.c2 AND t1.c1 > 2;

Query Plan:

==========================================

|ID|OPERATOR |NAME |EST. ROWS|COST|

------------------------------------------

|0 |MERGE JOIN | |5 |78 |

|1 | TABLE SCAN|t2(IDX\_c2)|5 |37 |

|2 | TABLE SCAN|t1 |3 |37 |

==========================================

Outputs & filters:

-------------------------------------

0 - output([t1.c1], [t2.c2]), filter(nil),

equal\_conds([t1.c1 = t2.c2]), other\_conds(nil)

1 - output([t2.c2]), filter(nil),

access([t2.c2]), partitions(p0),

is\_index\_back=false,

range\_key([t2.c2], [t2.c1]), range(2,MAX ; MAX,MAX),

range\_cond([t2.c2 > 2])

2 - output([t1.c1]), filter(nil),

access([t1.c1]), partitions(p0),

is\_index\_back=false,

range\_key([t1.c1]), range(2 ; MAX),

range\_cond([t1.c1 > 2])

##### 恒真假消除

对于如下恒真恒假条件可以进行消除：

false and expr ＝ 恒 False

true or expr = 恒 True

如下例所示，对于 WHERE 0 > 1 AND c1 ＝ 3，由于 0 > 1 使得 AND 恒假， 所以该 SQL 不用执行，可直接返回，从而加快查询的执行。

obclient> EXPLAIN EXTENDED\_NOADDR SELECT \* FROM t1 WHERE 0 > 1 AND c1 = 3;

Query Plan:

===================================

|ID|OPERATOR |NAME|EST. ROWS|COST|

-----------------------------------

|0 |TABLE SCAN|t1 |0 |38 |

===================================

Outputs & filters:

-------------------------------------

0 - output([t1.c1], [t1.c2]), filter([0], [t1.c1 = 3]), startup\_filter([0]),

access([t1.c1], [t1.c2]), partitions(p0),

is\_index\_back=false, filter\_before\_indexback[false,false],

range\_key([t1.\_\_pk\_increment], [t1.\_\_pk\_cluster\_id], [t1.\_\_pk\_partition\_id]),

range(MAX,MAX,MAX ; MIN,MIN,MIN)always false

#### 非SPJ的改写

##### 冗余排序消除

冗余排序消除是指删除 Order Item 中不需要的项，减少排序开销。以下三种情况可进行排序消除：

* ORDER BY 表达式列表中有重复列，可进行去重后排序。

obclient> SELECT \* FROM t1 WHERE c2 = 5 ORDER BY c1, c1, c2, c3 ;

<==>

obclient> SELECT \* FROM t1 WHERE c2 = 5 ORDER BY c1, c2, c3;

* ORDER BY 列中存在 WHERE 中有单值条件的列，该列排序可删除。

obclient> SELECT \* FROM t1 WHERE c2 = 5 ORDER BY c1, c2, c3;

<==>

obclient> SELECT \* FROM t1 WHERE c2 = 5 ORDER BY c1, c3;

* 如果本层查询有 ORDER BY 但是没有 LIMIT，且本层查询位于父查询的集合操作中，则 ORDER BY 可消除。因为对两个有序的集合做 UNION 操作，其结果是乱序的。但是如果 ORDER BY 中有 LIMIT，则语义是取最大/最小的 N 个，此时不能消除 ORDER BY，否则有语义错误。

obclient> (SELECT c1,c2 FROM t1 ORDER BY c1) UNION (SELECT c3,c4 FROM t2 ORDER BY c3);

<==>

obclient> (SELECT c1,c2 FROM t1) UNION (SELECT c3,c4 FROM t2);

##### LIMIT 下压子查询

LIMIT 下压改写是指将 LIMIT 下降到子查询中，OceanBase 数据库现在支持在不改变语义的情况下，将 LIMIT 下压到视图（示例 1）或 UNION 对应的子查询（示例 2）中。

示例 1：将 LIMIT 下压到视图。

obclient> SELECT \* FROM (SELECT \* FROM t1 ORDER BY c1) a LIMIT 1;

<==>

obclient> SELECT \* FROM (SELECT \* FROM t1 ORDER BY c1 LIMIT 1) a LIMIT 1;

示例 2：将 LIMIT 下压到 UNION 对应的子查询中。

obclient> (SELECT c1,c2 FROM t1) UNION ALL (SELECT c3,c4 FROM t2) LIMIT 5;

<==>

obclient> (SELECT c1,c2 FROM t1 LIMIT 5) UNION ALL (SELECT c3,c4 FROM t2 limit 5) LIMIT 5;

##### LIMIT 下压外连接或交叉连接

对于外连接和交叉连接，如果 SQL 语句中没有 WINDOW FUNCTION、DISTINCT、GROUP BY 或 HAVING，且 WHERE 条件或者 ORDER BY 仅和连接的一侧表有关或者不含 WHERE条件和 ORDER BY 时，可将 LIMIT 语句下压到连接的表的一侧（外连接）或多侧（多表交叉连接），这种改写方式称为 LIMIT 下压外连接或交叉连接。通过 LIMIT 下压，可有效减少连接的行数，从而降低执行查询的开销。

对于外连接，在下压 LIMIT 时，会在被下压的表上封装一层视图，如下以左外连接的 Q1 查询为例：

Q1:

SELECT \* FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1 LIMIT 1;

==>

Q2:

SELECT \* FROM V LEFT JOIN t2 ON t1.c1 = t2.c1 LIMIT 1;

V: (SELECT \* FROM t1 LIMIT 1);

Q1 查询不含上面提到的限制，且没有 WHERE条件和 ORDER BY 语句，因此可在左侧表 t1 上封装一层视图V，并将 LIMIT 1 封装到视图 V 中，从而改写为 Q2 查询。需要注意的是 LIMIT 下压后，原先的 LIMIT 也必须保留。改写后的计划如下：

=============================================

|ID|OPERATOR |NAME |EST. ROWS|COST|

---------------------------------------------

|0 |LIMIT | |1 |4 |

|1 | MERGE OUTER JOIN| |1 |4 |

|2 | SUBPLAN SCAN |VIEW1 |1 |2 |

|3 | TABLE SCAN |t1(idx)|1 |2 |

|4 | SORT | |1 |2 |

|5 | TABLE SCAN |t2 |1 |2 |

=============================================

Outputs & filters:

-------------------------------------

0 - output([1]), filter(nil), rowset=256, limit(1), offset(nil)

1 - output(nil), filter(nil), rowset=256,

equal\_conds([VIEW1.t1.c1 = t2.c1]), other\_conds(nil)

2 - output([VIEW1.t1.c1]), filter(nil), rowset=256,

access([VIEW1.t1.c1])

3 - output([t1.c1]), filter(nil), rowset=256,

access([t1.c1]), partitions(p0),

limit(1), offset(nil)

4 - output([t2.c1]), filter(nil), rowset=256, sort\_keys([t2.c1, ASC])

5 - output([t2.c1]), filter(nil), rowset=256,

access([t2.c1]), partitions(p0)

同理，右外连接可在满足上述条件的情况下，将 LIMIT 下压到右表中。

对于交叉连接，则可将 LIMIT 下压到交叉连接的两侧，示例如下：

Q3:

SELECT 1 FROM t1, t2 WHERE t1.c1 > 0 ORDER BY t1.c1 LIMIT 1;

==>

Q4:

SELECT 1 FROM V1, V2 LIMIT 1;

V1: SELECT 1 FROM t1 WHERE t1.c1 > 0 ORDER BY t1.c1 LIMIT 1;

V2: SELECT 1 FROM t2 LIMIT 1;

Q3 查询中不含上述 LIMIT 下压外连接时提到的限制，且仅含 t1 表上的 WHERE条件和 ORDER BY，则可分别在 t1 和 t2 表上创建视图 V1 和 V2，然后将 LIMIT 下压到视图中，从而改写为查询 Q4。改写后的计划如下：

=====================================================

|ID|OPERATOR |NAME |EST. ROWS|COST|

-----------------------------------------------------

|0 |LIMIT | |1 |5 |

|1 | NESTED-LOOP JOIN CARTESIAN| |1 |5 |

|2 | SUBPLAN SCAN |VIEW1|1 |3 |

|3 | TOP-N SORT | |1 |3 |

|4 | TABLE SCAN |t1 |3 |3 |

|5 | MATERIAL | |1 |2 |

|6 | SUBPLAN SCAN |VIEW2|1 |2 |

|7 | TABLE SCAN |t2 |1 |2 |

=====================================================

Outputs & filters:

-------------------------------------

0 - output([1]), filter(nil), rowset=256, limit(1), offset(nil)

1 - output(nil), filter(nil), rowset=256,

conds(nil), nl\_params\_(nil)

2 - output(nil), filter(nil), rowset=256,

access(nil)

3 - output([t1.c1]), filter(nil), rowset=256, sort\_keys([t1.c1, ASC]), topn(1)

4 - output([t1.c1]), filter([t1.c1 > 0]), rowset=256,

access([t1.c1]), partitions(p0)

5 - output(nil), filter(nil), rowset=256

6 - output(nil), filter(nil), rowset=256,

access(nil)

7 - output([1]), filter([t2.c1 > 0]), rowset=256,

access([t2.c1]), partitions(p0),

limit(1), offset(nil)

对于多表连接且满足上述条件的 SQL 查询，可以分别甚至多次应用外连接 LIMIT 下压和交叉连接 LIMIT 下压，从而扩展改写的空间，达到更优的改写效果。

##### DISTINCT 消除

* 如果 Select Item 中只包含常量，则可以消除 DISTINCT，并加上 LIMIT 1。

obclient> SELECT DISTINCT 1,2 FROM t1 ;

<==>

obclient> SELECT 1,2 FROM t1 LIMIT 1;

obclient> CREATE TABLE t1 (c1 INT PRIMARY KEY, c2 INT);

Query OK, 0 rows affected

obclient> EXPLAIN EXTENDED\_NOADDR SELECT DISTINCT 1,2 FROM t1;

Query Plan:

===================================

|ID|OPERATOR |NAME|EST. ROWS|COST|

-----------------------------------

|0 |TABLE SCAN|t1 |1 |36 |

===================================

Outputs & filters:

-------------------------------------

0 - output([1], [2]), filter(nil),

access([t1.c1]), partitions(p0),

limit(1), offset(nil),

is\_index\_back=false,

range\_key([t1.c1]), range(MIN ; MAX)always true

* 如果 Select Item 中包含确保唯一性约束的列，则 DISTINCT 能够被消除。如下示例中 (c1, c2) 为主键，可确保 c1、c2 和 c3 唯一性， 从而 DISTINCT 可消除。

obclient> CREATE TABLE t2(c1 INT, c2 INT, c3 INT, PRIMARY KEY(c1, c2));

Query OK, 0 rows affected

obclient> SELECT DISTINCT c1, c2, c3 FROM t2;

<==>

obclient> SELECT c1, c2 c3 FROM t2;

obclient> EXPLAIN SELECT DISTINCT c1, c2, c3 FROM t2;

Query Plan:

===================================

|ID|OPERATOR |NAME|EST. ROWS|COST|

-----------------------------------

|0 |TABLE SCAN|t2 |1000 |455 |

===================================

Outputs & filters:

-------------------------------------

0 - output([t2.c1], [t2.c2], [t2.c3]), filter(nil),

access([t2.c1], [t2.c2], [t2.c3]), partitions(p0)

##### MIN/MAX 改写

* 当 MIN/MAX 函数中的参数为索引前缀列，且不含 GROUP BY 时，可将该 scalar aggregate 转换为走索引扫描 1 行的情况，如下例所示：

obclient> CREATE TABLE t1 (c1 INT PRIMARY KEY, c2 INT, c3 INT, KEY IDX\_c2\_c3(c2,c3));

Query OK, 0 rows affected

obclient> SELECT MIN(c2) FROM t1;

<==>

obclient> SELECT MIN(c2) FROM (SELECT c2 FROM t2 ORDER BY c2 LIMIT 1) AS t;

obclient> EXPLAIN SELECT MIN(c2) FROM t1;

Query Plan:

==================================================

|ID|OPERATOR |NAME |EST. ROWS|COST|

--------------------------------------------------

|0 |SCALAR GROUP BY| |1 |37 |

|1 | SUBPLAN SCAN |subquery\_table|1 |37 |

|2 | TABLE SCAN |t1(idx\_c2\_c3) |1 |36 |

==================================================

Outputs & filters:

-------------------------------------

0 - output([T\_FUN\_MIN(subquery\_table.c2)]), filter(nil),

group(nil), agg\_func([T\_FUN\_MIN(subquery\_table.c2)])

1 - output([subquery\_table.c2]), filter(nil),

access([subquery\_table.c2])

2 - output([t1.c2]), filter([(T\_OP\_IS\_NOT, t1.c2, NULL, 0)]),

access([t1.c2]), partitions(p0),

limit(1), offset(nil)

* 如果 SELECT MIN/SELECT MAX 的参数为常量，而且包含 GROUP BY，可以将 MIN/MAX 改为常量，从而减少 MIN/MAX 的计算开销。

obclient> SELECT MAX(1) FROM t1 GROUP BY c1;

<==>

obclient> SELECT 1 FROM t1 GROUP BY c1;

obclient> EXPLAIN EXTENDED\_NOADDR SELECT MAX(1) FROM t1 GROUP BY c1;

Query Plan:

===================================

|ID|OPERATOR |NAME|EST. ROWS|COST|

-----------------------------------

|0 |TABLE SCAN|t1 |1000 |411 |

===================================

Outputs & filters:

-------------------------------------

0 - output([1]), filter(nil),

access([t1.c1]), partitions(p0),

is\_index\_back=false,

range\_key([t1.c1]), range(MIN ; MAX)always true

* 如果 SELECT MIN/SELECT MAX 的参数为常量，而且不含 GROUP BY，可以按照如下示例进行改写，从而走索引只需扫描 1 行。

obclient> SELECT MAX(1) FROM t1;

<==>

obclient> SELECT MAX(t.a) FROM (SELECT 1 AS a FROM t1 LIMIT 1) t;

obclient> EXPLAIN EXTENDED\_NOADDR SELECT MAX(1) FROM t1;

Query Plan:

==================================================

|ID|OPERATOR |NAME |EST. ROWS|COST|

--------------------------------------------------

|0 |SCALAR GROUP BY| |1 |37 |

|1 | SUBPLAN SCAN |subquery\_table|1 |37 |

|2 | TABLE SCAN |t1 |1 |36 |

==================================================

Outputs & filters:

-------------------------------------

0 - output([T\_FUN\_MAX(subquery\_table.subquery\_col\_alias)]), filter(nil),

group(nil), agg\_func([T\_FUN\_MAX(subquery\_table.subquery\_col\_alias)])

1 - output([subquery\_table.subquery\_col\_alias]), filter(nil),

access([subquery\_table.subquery\_col\_alias])

2 - output([1]), filter(nil),

access([t1.c1]), partitions(p0),

limit(1), offset(nil),

is\_index\_back=false,

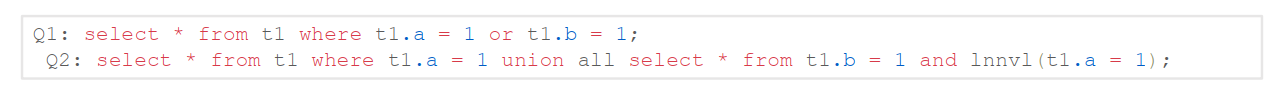
range\_key([t1.c1]), range(MIN ; MAX)always true

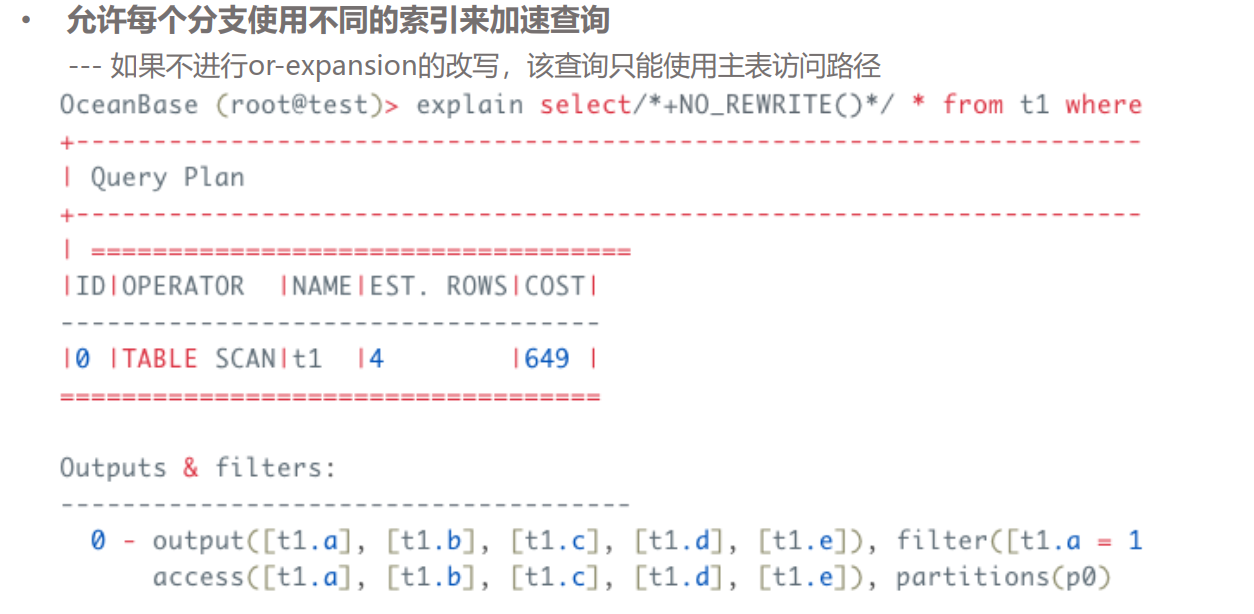
### 基于代价的查询改写

OceanBase 目前只支持基于代价的查询改写—或展开（Or-Expansion） 。  
或展开（Or-Expansion）：把一个查询改写成若干个用union组成的子查询，这个改写可能会给每个子查询提供更优的优化空间，但是也会导致多个子查询的执行，所以这个改写需要基于代价去判断。  
 通常来说，Or-Expansion的改写主要有如下三个作用:  
 • 允许每个分支使用不同的索引来加速查询。  
 • 允许每个分支使用不同的连接算法来加速查询，避免使用笛卡尔连接。  
 • 允许每个分支分别消除排序，更加快速的获取top-k结果。

#### 分支使用不同索引

• 允许每个分支使用不同的索引来加速查询  
 Q1: select \* from t1 where t1.a = 1 or t1.b = 1;  
 Q2: select \* from t1 where t1.a = 1 union all select \* from t1.b = 1 and lnnvl(t1.a = 1);  
 示例：Q1会被改写成Q2的形式，其中Q2中的谓词lnnvl(t1.a = 1)保证了这两个子查询不会生成重复的结果。  
 如果不进行改写，Q1一般来说会选择主表作为访问路径，对于Q2来说，如果t1上存在索引（a）和索引（b）， 那么该改写可能会让Q2中的每一个子查询选择索引作为访问路径：







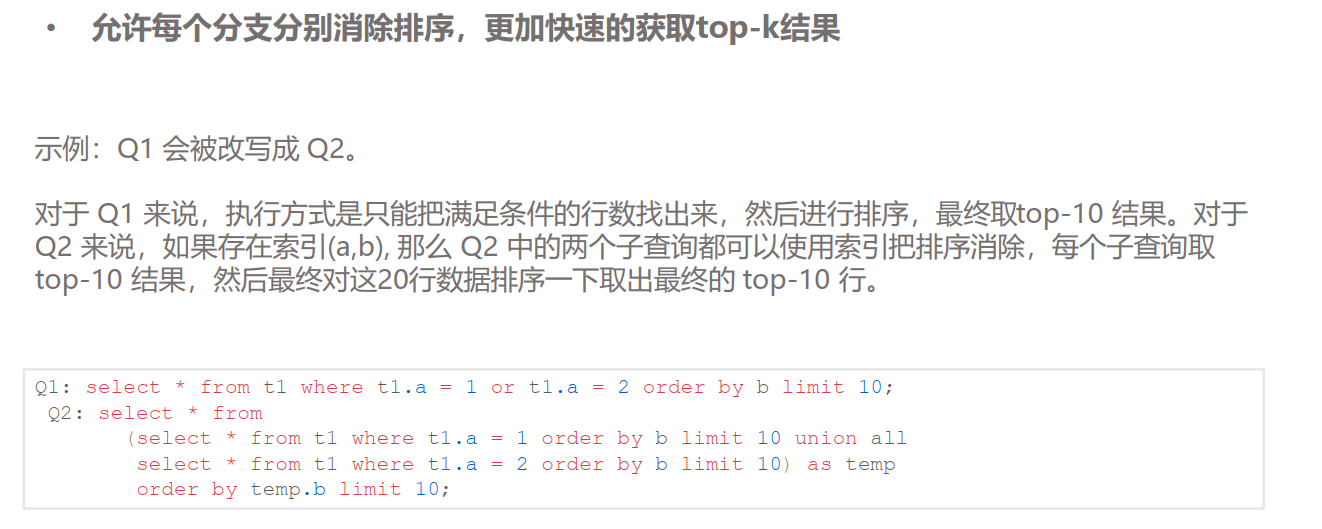
#### 分支使用不同连接算法







#### 分支分别消除排序







### 查询优化

#### 访问路径

#### 连接算法