

OCEANBASE

OBCP 认证培训



目录

第一章/ OB 分布式架构高级技术

第二章 / OB 存储引擎高级技术

第三章 / OB SQL 引擎高级技术

第四章/ OB SQL调优

第五章 / OB 分布式事务高级技术

第六章/ OBProxy 路由与使用运维

第七章 / OB 备份与恢复

第八章 / OceanBase 监控与故障排查

OCEANBASE

目录

第四章 / OB SQL调优

- 4.1 SQL 调优方法
- 4.2 分区
- 4.3 索引
- 4.4 局部索引与全局索引
- 4.5 Hint
- 4.6 SQL 执行性能监控

OCEANBASE

4.1 SQL 调优方法

OCEANBASE

4.1 OB 调优与传统数据库的差异

➤ LSM-tree存储引擎

- 数据分为静态数据 (SSTable) 和动态数据 (MemTable) 两部分
 - 当数据合并后, SQL的执行效率往往都有明显的提升
- buffer表是指那些被用户当做业务过程中的临时存储的数据表
 - 内存中“标记删除”(比如快速“写入-修改-删除”的数据)
 - 当有大量删除操作后立即访问被删除的数据范围, 仍然有可能遇到由于访问标记删除节点而导致的执行变慢的问题

➤ 分布式架构

- 传统的share-disk架构: 执行计划并不区分数据所在的物理节点, 所有的数据访问都可以认为是“本地”的
- 分布式share-nothing架构: 不同的数据被存储在不同的节点上
 - 连接两张表时, 如果两个表的数据分布在不同的物理节点上, 执行计划也变为分布式执行计划
 - 数据的切主, 可能出现之前的本地执行计划(所访问的数据在本机)变为远程执行计划或者分布式执行计划的问题

OCEANBASE

OceanBase 作为关系型数据库, 其调优的方法与思路与传统数据库有很多相似的地方, 但由于其自身特点主要有以下不同之处:

• LSM-tree 存储引擎

OceanBase 的存储引擎采用了两层 LSM-tree 的架构, 数据分为静态数据 (SSTable) 和动态数据 (MemTable) 两部分。针对写请求, 用户的修改按照 BTree 的方式写入内存中的 MemTable, 并定期通过合并过程合入到存储在磁盘上的 SSTable 中; 针对读请求, 存储引擎需要读取 MemTable、SSTable 两部分数据, 合成最终的行。这一特点决定了 OceanBase 在基表访问路径的代价模型上与传统数据库有较大的差异。例如, MemTable 为空 (全部数据在 SSTable 中) 时, 存储引擎的执行效率比 MemTable 中有数据的场景的执行效率要高很多。因此, 当数据合并后, SQL 的执行效率往往都有明显的提升。

“buffer”表是指那些被用户当做业务过程中的临时存储的数据表, 这些表的访问模型往往是: 写入-修改-删除, 且整个过程周期很短, 往往在几分钟或几个小时之内。由于 OceanBase 所有的 DML 操作都是“逻辑”的, 数据被删除时, 并不做原地修改, 而只是在内存中做“标记”删除。因此, 即使用户的行被删除了, 访问对应的“删除”标记也需要花费一定的时间。对此, OceanBase 通过 row purge, 在 BTree 中标记一个范围段的删除标记以加速数据的访问。row purge 的过程是异步的, 因此, 当有大量删除操作后立即访问被删除的数据范围, 仍然有可能遇到由于访问标记删除节点而导致的执行变

慢的问题。

- 分布式 share-nothing 架构

在传统的 share-disk 数据库中，执行计划并不区分数据所在的物理节点，所有的数据访问都可以认为是“本地”的。但在 OceanBase 的分布式 share-nothing 架构中，不同的数据被存储在不同的节点上，SQL 执行计划的生成必须考虑到数据的实际物理分布，并可能因此表现出不同的性能特征。

例如，当连接两张表时，如果两个表的数据分布在不同的物理节点上，则必然涉及节点之间的数据传输，执行计划也变为分布式执行计划，相比两表数据分布在同一个节点上的场景，执行代价必然有所增加。

另一个常见的场景是数据的切主，由于数据的主可以在分布在不同节点上的副本之间进行切换，有可能出现之前的本地执行计划（所访问的数据在本机）变为远程执行计划或者分布式执行计划的问题，也可能增加一定的执行时间。这一问题在用户开启轮转合并功能之后尤其明显（在数据重新选主后，客户端路由可能由于数据物理位置信息刷新不及时而不准确）。

4.1 SQL 性能问题来源

- 用户SQL写法 - 遵循开发规约
- 代价模型缺陷 - 绑定执行计划
- 统计信息不准确 - 仅支持本地存储，合并式更新
- 数据库物理设计 - 决定查询性能
- 系统负载 - 影响整体吞吐率，影响单sql rt
- 客户端路由 - 远程执行

OCEANBASE

4.1 SQL 调优方法

➤ 针对单条 SQL 执行的性能调优

- 单表访问场景
 - 索引、排序或聚合、分区、分布式并行
- 多表访问场景
 - 连接顺序、连接算法、分布式并行、查询改写

➤ 针对吞吐量的性能优化

- 优化慢 SQL
- 均衡 SQL 的流程资源
- 均衡子计划 RPC 的流量资源

OCEANBASE

针对单条 SQL 执行的性能调优

针对 SQL 执行时间的性能调优是最常见的性能调优，关注的问题是某一条或某一类 SQL 的执行时间或者执行资源的消耗（如内存、磁盘 IO 等）。单条 SQL 的性能调优往往与该 SQL 的执行计划相关，因此，执行计划的分析是该调优场景的最重要的手段。一般来说，应该首先通过静态分析 SQL 的执行计划来找到可能的调优点。

排除优化器自身的 bug 原因，为了使某些 SQL 的执行时间或资源消耗符合预期，一般需要用户对数据库的设置做相应的修改，常见的手段包括：

- 修改系统配置项或系统变量
- 对数据 schema 进行修改，包括创建数据分区、创建二级索引等
- 修改用户 SQL，包括对 SQL 做等价改写、增加 hint 等
- 调整并行查询的并行度

一条 SQL 从发送到数据开始执行，到返回结果给用户，会经历队列等待、plan cache 查询、计划优化（如果计划缓存不命中）、计划执行、返回结果等过程。当发现 SQL 的响应时间增加时，第一步应该明确具体是哪部分的耗时增加，此时采用的一般手段包括 SQL Trace、查询 SQL Audit 表、查看慢查询日志等。只有明确了具体耗时在哪里，才能有针对性的进一步分析问题的根源。

针对单条 SQL 的执行计划性能调优又可以分为单表访问和多表访问两种场景。

单表访问场景

对单表访问SQL来说，需要重点关注的问题包括：

- 访问路径是否开启索引扫描

访问路径的分析是单表查询的最重要的问题之一，对于使用主表扫描的访问路径来说，执行时间一般与需要扫描的数据量（范围）成正比。一般来说，可以使用 explain extended 命令，将表扫描的范围段展示出来。对于有合适索引的查询，使用索引可以大大减小数据的访问量，因此对于使用主表扫描的查询，要分析没有选中索引扫描的原因，例如是由于不存在可用的索引，还是索引扫描范围过大以至于代价过高。

- 是否存在排序或聚合操作

排序或聚合往往都是比较耗时的操作。优化器为了尽可能的降低执行时间，在有合适索引可用时，考虑直接使用索引的顺序以避免额外的排序操作，同理，用户也可以根据经常需要排序的列，创建合适的索引，以避免不必要的排序操作。

- 分区裁剪是否正确

分区裁剪是分区表优化的重要手段，一般来说，只要用户提供了合适的分区条件，优化器会自动跳过无需访问的分区。

- 是否需要调整查询的并行度

提高查询的并行度可以使用更多资源的代价获取单条 SQL 查询的性能提升，当查询牵涉到的数据量较大，分区数目较多时，可以通过提高并行度的方式加快执行时间。

多表访问场景

针对多表访问的 SQL，不仅要关注单表的 SQL 调优的重点问题，还需要关注多表间的连接问题，需要分析的点包括：

- 连接顺序

- 连接算法

- 跨机或并行连接的数据再分布方式

- 查询改写

针对吞吐量的性能调优

针对吞吐量的性能调优主要是考虑在一定资源(CPU, IO, 网络等)情况下，能够将数据库系统处理请求量最大化，我们在新业务上线以及各种大促活动前往往需要进行吞吐量评估及吞吐量的性能调优。

吞吐量性能调优可考虑以下几个方面：

- 优化慢 SQL

大量慢 SQL 请求会消耗大量的资源，导致整体吞吐量上不去，可按如下步骤处理：

1. 通过 OCP 的 TOP SQL 功能或利用 Plan Cache 视图查询耗时为 TOP N 的 SQL。
2. 找到具体的慢 SQL，可根据前面说的针对单条 SQL 进行性能调优。

- 均衡 SQL 的请求流量资源

在多机环境下，我们需要尽量将所有机器资源都能使用到，因此需要考虑流量是否均衡，可以通过 sql_audit 查看 SQL 请求是否均衡，影响均衡的因素主要有：

- ob_read_consistency 如何设置
- primary zone 如何设置
- proxy 或 java 客户端路由策略相关设置
- 业务热点查询分区是否均衡

•均衡子计划的 RPC 请求流量资源

大量分布式计划时一般都设置了弱读，资源消耗主要在子计划的 RPC 请求上，在 SQL 请求均衡的情况下，通过sql_audit 可查看子计划 RPC 请求是否均衡，影响这些子计划请求是否均匀的主要因素有：

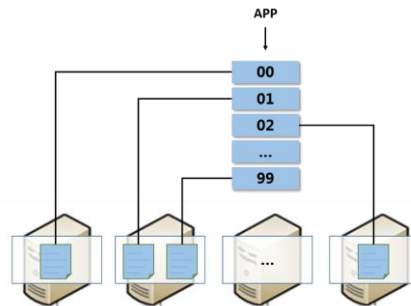
- observer 内部路由策略相关设置
- 业务热点查询的分区是否均衡

4.2 分区

OCEANBASE

4.2 分区表概述

- 分区是物理数据库设计技术，它的操作对象是表。实现分区的表，我们称之为分区表。表分布在多个分区上
- 创建分区的目的是为了在特定的SQL操作中减少数据读写的总量以减少响应时间



4.2 OceanBase分区表特点

- 可多机扩展
- 自动负载均衡、自动容灾
- 对业务透明，可以取代“分库分表”方案
- 支持分区间并行
- 单表分区个数最大8192
- 单机partition支持上限：8万（推荐不超过3万）

OCEANBASE

4.2 分区表

- 分为一级分区 和 二级分区
- OB MySQL模式现在支持的一级分区类型有：HASH， KEY， LIST， RANGE， RANGE COLOMNS， 生成列分区
- 二级分区相当于在一级分区的基础上， 又从第二个维度进行了拆分

OCEANBASE

4.2 HASH 分区

- 一级分区
- HASH分区需要指定分区键和分区个数。通过HASH的分区表达式计算得到一个int类型的结果，这个结果再跟分区个数取模得到具体这行数据属于那个分区。通常用于给定分区键的点查询，例如按照用户id来分区。HASH分区通常能消除热点查询。

```
create table t1 (c1 int, c2 int) partition by hash(c1 + 1) partitions 5
```

其中partition by hash(c1+1)指定了分区键c1和分区表达式c1 + 1； partitions 5指定了分区数

OB MySQL模式的Hash分区限制和要求：

- 分区表达式的结果必须是int类型。
- 不能写向量，例如partition by hash(c1, c2)

OCEANBASE

。

4.2 KEY 分区

一级分区

KEY分区与HASH分区类似，不同在于：

- 使用系统提供的HASH函数(murmurhash)对涉及的分区键进行计算后再分区，不允许使用用户自定义的表达式
- 用户通常没有办法自己通过简单的计算来得知某一行属于哪个分区
- 测试发现KEY分区所用到的HASH函数不太均匀

```
create table t1 (c1 varchar(16), c2 int) partition by key(c1) partitions 5
```


4.2 KEY 分区

- KEY分区不要求是int类型，可以是任意类型
- KEY分区不能写表达式（与HASH分区区别）
- KEY分区支持向量
- KEY分区有一个特殊的语法

```
create table t1 (c1 int primary key, c2 int) partition by key() partitions 5
```

KEY分区分区键不写任何column，表示key分区的列是主键

4.2 LIST 分区

一级分区

LIST分区是根据枚举类型的值来划分分区的，主要用于枚举类型

LIST分区的限制和要求

- 分区表达式的结果必须是int类型。
- 不能写向量，例如partition by list(c1, c2)

List columns 和 list 的区别是：

- list columns分区不要求是int类型，可是任意类型
- list columns分区不能写表达式
- list columns分区支持向量

```
create table t1 (c1 int, c2 int) partition by list(c1)
(partition p0 values in (1,2,3),
partition p1 values in (5, 6),
partition p2 values in (default));
```

OCEANBASE

4.2 RANGE 分区

一级分区

RANGE分区是按用户指定的表达式范围将每一条记录划分到不同分区

常用场景：按时间字段进行分区

目前提供对range分区的分区操作功能，能add/drop分区

add分区现在只能加在最后，所以最后不能是maxvalue的分区

```
CREATE TABLE `info_t`(id INT, gmt_create TIMESTAMP, info VARCHAR(20), PRIMARY KEY (gmt_create))
PARTITION BY RANGE(UNIX_TIMESTAMP(gmt_create))
(PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2015-01-01 00:00:00')),
PARTITION p1 VALUES LESS THAN (UNIX_TIMESTAMP('2016-01-01 00:00:00')),
PARTITION p2 VALUES LESS THAN (UNIX_TIMESTAMP('2017-01-01 00:00:00')));
```

OCEANBASE

4.2 RANGE COLUMNS 分区

一级分区

RANGE COLUMNS分区与RANGE分区类似，但不同点在于RANGE COLUMNS分区可以按一个或多个分区键向量进行行分区，并且每个分区键的类型除了INT类型还可以支持其他类型，比如VARCHAR、DATETIME等

RANGE COLUMNS和RANGE的区别是

- RANGE COLUMNS分区不要求是int类型，可以是任意类型
- RANGE COLUMNS分区不能写表达式
- RANGE COLUMNS分区支持向量

```
CREATE TABLE `info_t`(id INT, gmt_create DATETIME, info VARCHAR(20), PRIMARY KEY (gmt_create))
PARTITION BY RANGE COLUMNS(gmt_create)
(PARTITION p0 VALUES LESS THAN ('2015-01-01 00:00:00' ),
PARTITION p1 VALUES LESS THAN ('2016-01-01 00:00:00' ),
PARTITION p2 VALUES LESS THAN ('2017-01-01 00:00:00' ),
PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

OCEANBASE

4.2 生成列分区

一级分区

生成列是指这一列是由其他列计算而得

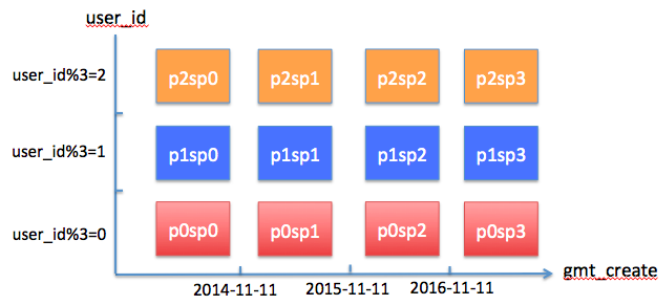
生成列分区是指将生成列作为分区键进行分区，该功能能够更好的满足期望将某些字段进行一定处理后作为分区键的需求（比如提取一个字段的一部分，作为分区键）

```
CREATE TABLE gc_part_t(t_key varchar(10) PRIMARY KEY, gc_user_id VARCHAR(4) GENERATED ALWAYS AS  
(SUBSTRING(t_key, 1, 4)) VIRTUAL, c3 INT)  
PARTITION BY KEY(gc_user_id)  
PARTITIONS 10;
```

OCEANBASE

4.2 二级分区

按两个维度划分数据



```
CREATE TABLE history_t (user_id INT, gmt_create DATETIME, info VARCHAR(20), PRIMARY KEY(user_id, gmt_create))
PARTITION BY RANGE COLUMNS (gmt_create)
SUBPARTITION BY HASH(user_id) SUBPARTITIONS 3
(PARTITION p0 VALUES LESS THAN ('2014-11-11'),
PARTITION p1 VALUES LESS THAN ('2015-11-11'),
PARTITION p2 VALUES LESS THAN ('2016-11-11'),
PARTITION p3 VALUES LESS THAN ('2017-11-11'));
```

OCEANBASE

4.2 分区管理

场景

- 对于按时间范围分区的表，有时需要作过期数据清理 -> DROP PARTITION
- 伴随数据量增长，range分区需要能够扩展 -> ADD PARTITION

限制

- 只有range分区，可以删除任意一个一级range分区
- 只能以append方式往后添加分区，也就是说，新加分区的range value总是最大的

4.2 分区管理 – 添加分区

为已经分区的表添加后续分区

语法: ALTER TABLE ... ADD PARTITION

```
CREATE TABLE members (  
  id INT,  
  fname VARCHAR(25),  
  lname VARCHAR(25),  
  dob DATE  
)  
PARTITION BY RANGE(YEAR(dob)) (  
  PARTITION p0 VALUES LESS THAN (1970),  
  PARTITION p1 VALUES LESS THAN (1980),  
  PARTITION p2 VALUES LESS THAN (1990)  
);
```



```
ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2000));
```

alter table t1 drop partition (p0); // 这里需要加括号

4.3 索引

1.3.1 主键和二级索引

1.3.2 本地索引和全局索引

OCEANBASE

4.3 路径选择 (Access Path Selection)

► 何为路径

- ✓ 主键
- ✓ 二级索引

► 如何选择

- ✓ 规则模型
 - 决定性规则
 - 剪枝规则
- ✓ 代价模型

► 考虑因素

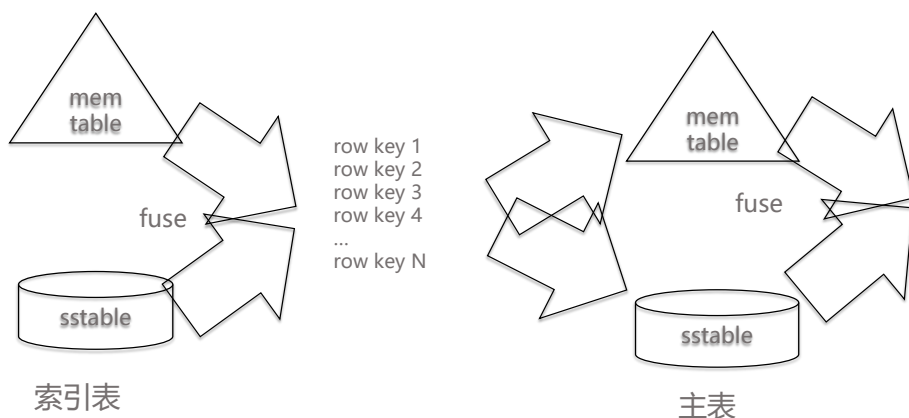
- ✓ 扫描范围
- ✓ 是否回表
- ✓ 路径宽度
- ✓ 过滤条件
- ✓ Interesting order

OCEANBASE

主索引是表的主键，二级索引可以根据你自己需要用到表的任何字段的组合来创建

系统自动选择，先是基于规则模型，如果规则不确定的话，使用代价模型。OceanBase的规则体系分为前置规则（正向规则）和Skyline剪枝规则(反向规则)。前置规则直接决定了一个查询选取什么样的路径，是一个强匹配的规则体系。Skyline剪枝规则会两两比较两个路径，如果一个路径在一些定义的维度上优于另外一个路径，那么这个路径会被剪掉，最后没有被剪掉的路径会进行代价比较，选出最优的路径。优化器会优先使用前置规则，如果前置规则不能得到一个确定最优的路径，那么优化器会进一步通过Skyline剪枝规则剪掉一些路径，最后代价模型会在没有被剪掉的索引中选择代价最低的路径。

4.3 路径选择 – 索引回表



OCEANBASE

比如这个insert、update、delete的例子，需要基线数据和内存数据fuse（合并在一起），索引表和物理表（主表）是同构的

索引表没有cover，就要回表

Row purge（回表）就表示这个delete的操作，内存中删除这个操作链

Row key表示根据主键去回表

比如oracle内部存储的索引表是堆表（根据row id），所以比OB的回表效率要高一些（row id可以拿到主表的偏移量），当然OB2.0会把索引这块做优化来提高效率

OB现在是拿到主键的row key的值，回到主表中，二分查找

4.3 路径选择

- 目前仅支持B+索引
- 两种访问
 - ✓ get: 索引键全部等值覆盖
 - ✓ scan: 返回有序数据
- 字符串条件: 'T%' ('%T%' , '%T' 无法利用索引)
- 扫描顺序由优化器智能决定

OCEANBASE

4.3 路径选择

覆盖索引
(主键补全)

```
OceanBase (root@oceanbase)> explain select c2 from t2;
=====
|ID|OPERATOR  |NAME      |EST. ROWS|COST|
=====
|0 |TABLE SCAN|t2(t2_c2)|1000     |1145|
=====
```

Outputs & filters:

```
0 - output([t2.c2]), filter(nil),
   access([t2.c2]), partitions(p0)
```

原因?

```
OceanBase (root@oceanbase)> explain select c1, c2 from t2;
=====
|ID|OPERATOR  |NAME      |EST. ROWS|COST|
=====
|0 |TABLE SCAN|t2(t2_c2)|1000     |1180|
=====
```

Outputs & filters:

```
0 - output([t2.c1], [t2.c2]), filter(nil),
   access([t2.c1], [t2.c2]), partitions(p0)
```

原因?

OCEANBASE

C1是主键，建立索引的时候只使用了c2，这时候，OB建立索引表的时候会自动主键补全（c1,c2都在索引表中）

4.3 路径选择 - Interesting Order

```
OceanBase (root@oceanbase)> explain select * from t1 order by c1;
```

```
|=====|
|ID|OPERATOR  |NAME|EST. ROWS|COST|
|-----|
|0 |TABLE SCAN|t1  |1000      |2327|
|=====|
```

无需排序

```
Outputs & filters:
```

```
-----
0 - output([t1.c1], [t1.c2], [t1.c3]), filter(nil),
   access([t1.c1], [t1.c2], [t1.c3]), partitions(p0)
```

OCEANBASE

Name里面t1代表主表查询，有括号的话代表索引查询

因为主键 / 索引已经排序了

索引的序就是order需要的序（主键 / 索引已经有序了）

如果order by c2就不是interesting order了，这个时候如果有需求的话，就再建一个索引表（基于c2）

4.3 路径选择 – 逆序索引扫描

```
OceanBase (root@oceanbase)> explain select * from t1 order by c1 desc;
| =====
| ID|OPERATOR  |NAME          |EST. ROWS|COST|
|-----|
| 0 |TABLE SCAN|t1 (Reverse) |1000      |2327|
|=====
```

Outputs & filters:

```
-----
0 - output([t1.c1], [t1.c2], [t1.c3]), filter(nil),
   access([t1.c1], [t1.c2], [t1.c3]), partitions(p0)
```

OCEANBASE

4.3 OB的索引选择

- OB的索引选择有大量的规则挡在代价模型之前

- 正向规则: 一旦命中规则直接选择该索引
 - 命中唯一性索引
- 逆向规则(skyline剪枝规则)
 - 通过比较两个索引, 剪掉一些比较“差”的索引 (Query range, 序, 是否需要回表)
 - 剩下的索引通过代价模型选出

```
create table t1(a int, b int, c int, unique key idx1(a, b), key idx2(b));
```

```
explain extended select * from t1 where a = 1 and b = 1;
```

```
explain extended select * from t1 where a = 1 order by b;
```

```
Optimization Info:
```

```
t1:optimization_method=rule_based, heuristic_rule=unique_index_with_indexback
```

```
Optimization Info:
```

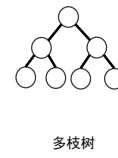
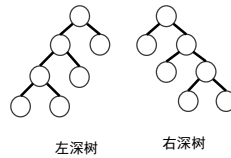
```
t1:optimization_method=cost_based, available_index_name[t1,idx1], pruned_index_name[idx2]
```

OCEANBASE

4.3 连接顺序

- 不同的连接顺序对执行效率影响极大
- 目前只考虑左深树（某些特定场景除外）
 - 搜索空间
 - 对内存占用更友好
- 连接顺序的选择是一个动态规划的过程
- 可通过hint指定连接顺序
- 存在显式连接条件的连接优先于笛卡尔积连接

OCEANBASE



- | | |
|--|--|
| <ul style="list-style-type: none">• 优势<ul style="list-style-type: none">• 搜索空间小• 更利于流水线• 内存空间 (foot-print) 小• 劣势<ul style="list-style-type: none">• 无法利用并行执行• 可能错失更佳执行计划 | <ul style="list-style-type: none">• 优势<ul style="list-style-type: none">• 充分系统并行能力• 可能生成更好计划• 劣势<ul style="list-style-type: none">• 搜索空间巨大• 执行消耗资源多 |
|--|--|

只支持左深树，举例可以用三表join，explain里面缩进体现

4.3 创建高效索引

- 索引表与普通数据表一样都是实体表，在数据表进行更新的时候会先更新索引表然后再更新数据表
- 索引要全部包含所查询的列：包含的列越全越好，这样可以尽可能的减少回表的行数
- 等值条件永远放在最前面
- 过滤与排序数据量大的放前面

OCEANBASE

4.3 创建索引

- 可以对一张表的单列或多列创建索引来提高表查询速度。创建合适的索引，能够减少对磁盘的读写
 - 建表的时候创建，立即生效
 - 建表后再创建索引，是同步生效，表中数据量大时需要等待一段时间

```
CREATE [UNIQUE] INDEX indexname ↓  
    ON tblname (index_col_name,...)
```

OCEANBASE

4.3 创建索引

- 等值索引

索引中的字段	命中索引的SQL	未命中索引的SQL
(A,B,C)	where A = ? and B = ? and C = ? where A = ? and B = ? where A = ? and C = ?	where B = ? and C = ? where C = ?

条件的先后顺序不影响索引能效，如where A = ? and B = ? 和 where B = ? and A = ? 效果相同，从索引能效来看：【Where A=? And B=? and C=?】 > 【Where A=? and B=?】 > 【Where A=? and C=?】

4.3 创建索引

- 范围索引

索引中的字段	命中索引的SQL	未命中索引的SQL
(A,B,C)	where A > ? and B > ? and C < ? where A > ? and B > ? where A > ? and C < ?	where B > ? and C < ? where C in (?,?)

常见的范围查询有：大于、小于、大于等于、小于等于、between...and 、 in(?,?)
遇到第一个范围查询字段后，后续的字段不参与索引过滤（不走索引）
如 【where A > ? and B > ? and C < ?】 只能走A字段的索引

4.3 创建索引

- 等值和范围索引

索引中的字段	命中索引的SQL	未命中索引的SQL
(A,B,C)	where A = ? and B > ? and C = ? where A = ? and B = ? and C > ? where A = ? and B > ? and C > ?	where B > ? and C < ? where C in (?,?) where C = ?

遇到第一个范围查询字段后，后续字段不参与索引过滤（不走索引）
从索引能效看， 【where A = ? and B = ? and C > ?】 > 【where A = ? and B > ? and C > ?】
【where A = ? and B > ? and C = ?】 = 【where A = ? and B > ? and C > ?】

4.4 局部索引与全局索引

OCEANBASE

4.4.1 主键和二级索引

□ 主表

指使用CREATE TABLE语句创建的表对象。也是索引对象所依赖的表（即CREATE INDEX语句中ON子句所指定的表）。

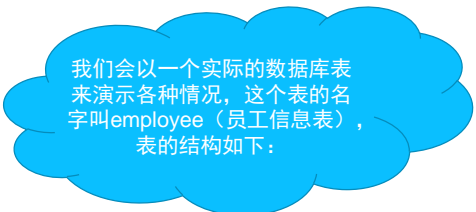
□ 主键

OceanBase 的每一张表都有主键，并在内部以主键为序组织数据。如果在创建用户表时不显式指定主键，系统会自动为表生成隐藏主键，隐藏主键不可被查询。

□ 索引（索引表）

指使用CREATE INDEX语句创建的索引对象。有时为了便于大家理解，也会把索引对象类比为一个表对象，即索引表。

```
employee
{
  emp_id, /* 员工ID */
  emp_name /* 员工名字 */
  dpet_id, /* 部门ID */
  ...
}
```



我们会以一个实际的数据库表来演示各种情况，这个表的名字叫employee（员工信息表），表的结构如下：

OCEANBASE

OceanBase 的每一张表都有主键，并在内部以主键为序组织数据。如果在创建用户表时不显式指定主键，系统会自动为表生成隐藏主键，隐藏主键不可被查询。

主键可以是单一字段或多个字段的组合，组合主键的字段数不能超过 64。主键一旦指定，不可更改。当用作分区表时，主键必须覆盖所有的分区列。

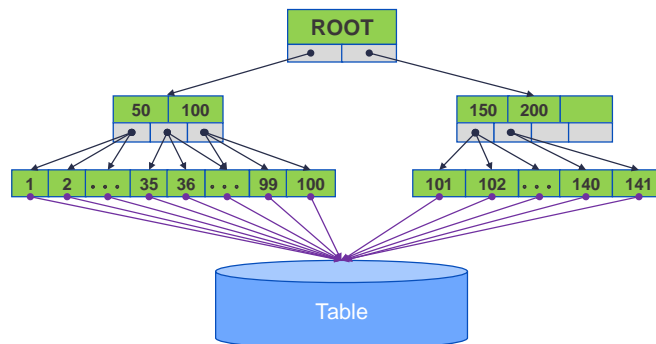
像传统的关系数据库一样，OceanBase 支持外键功能，这是 OceanBase 相对于其他大多数分布式数据库系统而言，在完整性约束方面的重大优势。

4.4.1 传统“非分区表”中，主表和索引的关系

传统的“非”分区表中，主表和索引的对应关系：

主表的所有数据都保存在一个完整的数据结构中，主表上的每一个索引也对应一个完整的数据结构（比如最常见的B+ Tree），主表的数据结构和索引的数据结构之间是一一对应的关系，如下图所示，在 employee 表中，以 emp_id 创建的索引：

idx_emp_id on employee (emp_id)



OCEANBASE

4.4.2 局部索引与全局索引

当分区表出现之后，情况发生了变化：主表的数据按照分区键（Partitioning Key）的值被分成了多个分区，每个分区都是独立的数据结构，分区之间的数据没有交集。这样一来，索引所依赖的单一数据结构不复存在，那索引需要如何应对呢？这就引入了“局部索引”和“全局索引”两个概念。

□ 局部索引

局部索引又名分区索引，创建索引的分区关键字是LOCAL，分区键等同于表的分区键，分区数等同于表的分区数，总之，局部索引的分区机制和表的分区机制一样。

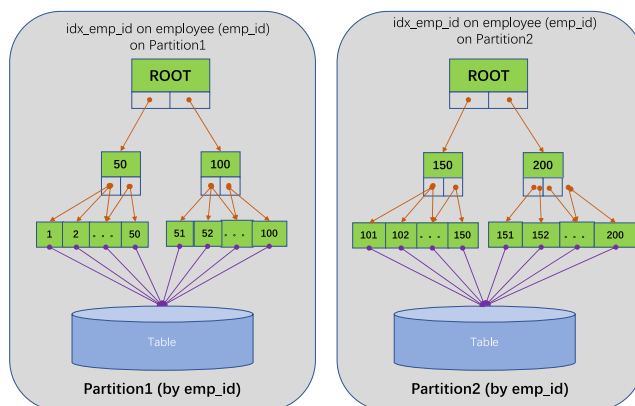
□ 全局索引

全局索引的创建规则是在索引属性中指定GLOBAL关键字，与局部索引相比，全局索引最大的特点是全局索引的分区规则跟表分区是相互独立的，全局索引允许指定自己的分区规则和分区个数，不一定需要跟表分区规则保持一致。

OCEANBASE

4.4.2 局部索引

分区表的局部索引和非分区表的索引类似，索引的数据结构还是和主表的数据结构保持一对一的关系，但由于主表已经做了分区，主表的“每一个分区”都会有自己单独的索引数据结构。局部索引的结构如下图所示：



OCEANBASE

对每一个局部索引数据结构来说，里面的键（Key）只映射到自己分区中的主表数据，不会映射到其它分区中的主表，因此这种索引被称为局部索引。从另一个角度来看，这种模式下索引的数据结构也做了分区处理，因此有时也被称为本地分区索引（Local Partitioned Index）。

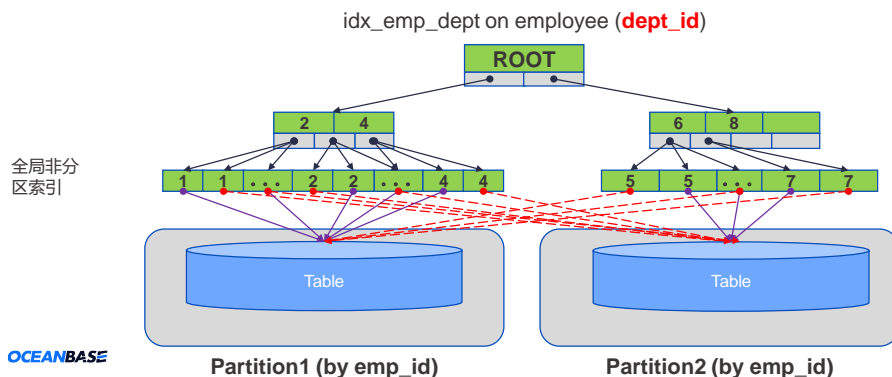
4.4.2 全局索引

分区表的全局索引不再和主表的分区保持一对一的关系，而是将所有主表分区的数据合成一个整体来建立全局索引。

更进一步，全局索引可以定义自己独立的数据分布模式，既可以选择非分区模式也可以选择分区模式。

❑ 全局非分区索引 (Global Non-Partitioned Index)

❑ 全局分区索引 (Global Partitioned Index)



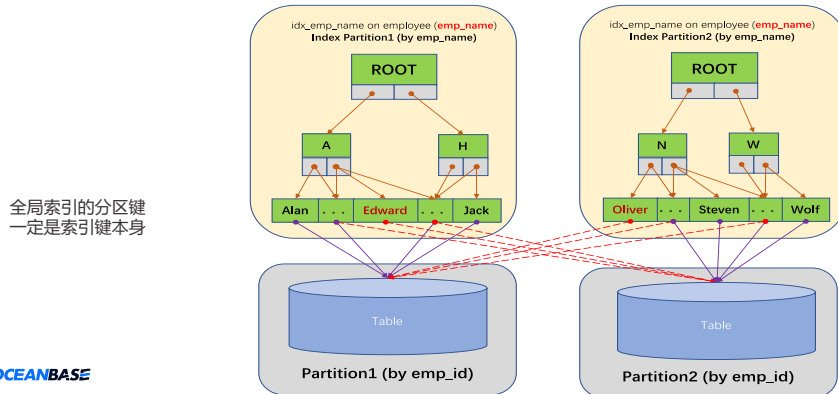
全局索引中的一个键可能会映射到多个主表分区中的数据（当索引键有重复值时）。在上图中，虽然employee表按照emp_id做了分区，但是创建在dept_id上的全局索引并没有分区，因此同一个键值可能会映射到多个分区中。

4.4.2 全局索引

分区表的全局索引不再和主表的分区保持一对一的关系，而是将所有主表分区的数据合成一个整体来建立全局索引。更进一步，全局索引可以定义自己独立的数据分布模式，既可以选择非分区模式也可以选择分区模式：

❑ 全局非分区索引（Global Non-Partitioned Index）

❑ 全局分区索引（Global Partitioned Index）



索引数据按照指定的方式做分区处理，比如做哈希（Hash）分区或者范围（Range）分区，将索引数据分散到不同的分区中。但索引的分区模式是完全独立的，和主表的分区没有任何关系，因此对于每个索引分区来说，里面的某一个键都可能映射到不同的主表分区（当索引键有重复值时），索引分区和主表分区之间是“多对多”的对应关系。全局分区索引的结构如上图所示：

employee表按照emp_id做了范围分区，同时在emp_name上做了全局分区索引。同一个索引分区里的键，会指向不同的主表分区。

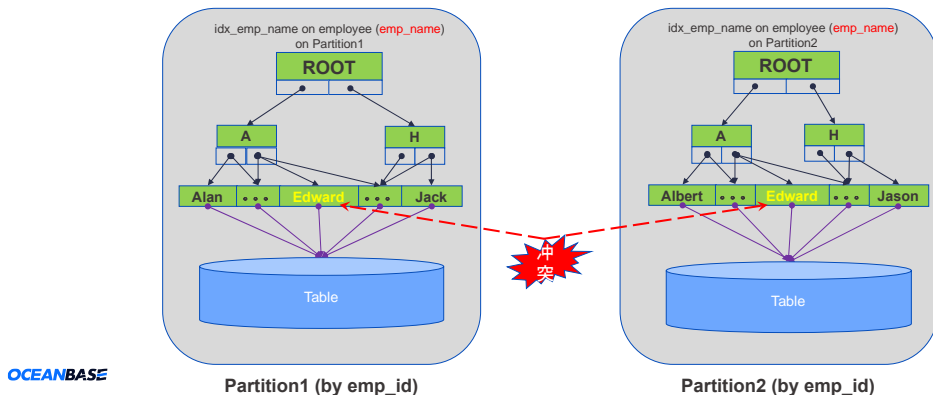
全局索引的分区键一定是索引键本身，在查询到索引键值后可以利用索引表中存储的主键信息计算出主表的分区位置，进而对主表也能进行快速的分区定位，避免扫描主表的所有分区。

由于全局索引的分区模式和主表的分区模式完全没有关系，看上去全局索引更像是另一张独立的表，因此也会将全局索引叫做索引表，理解起来会更容易一些（和主表相对应）。

这里特别说一点：“非”分区表也可以创建全局分区索引。但如果主表没有分区的必要，通常来说索引也就没有必要分区了。

4.4.2 局部索引与全局索引的比较

局部索引在“索引键没有包含主表所有的分区键字段”的情况下，此时索引键值对应的索引数据在所有分区中都可能存在。如下图，employee按照emp_id做了分区，但同时想利用局部索引建立关于emp_name的唯一约束是无法实现的。由于某索引键值在所有分区的局部索引上都可能存在，索引扫描必须在所有的分区上都做一遍，以免造成数据遗漏。这会导致索引扫描效率低下，并且会在全局范围内造成CPU和IO资源的浪费



针对这个问题，有些数据库在分区表中会增加限制，要求主键和唯一索引的定义中必须包含主表所有的分区键字段。有了这个限制，索引中的某一个键所对应的索引数据只可能存在于一个分区中，因此只要在每一个分区内保证唯一性约束，即可在全表范围内保证唯一性约束。这个限制虽然解决了数据库的难题，却大大增加了开发人员的烦恼，因为它会导致主键和唯一索引的可选范围大大缩小（只能是主表分区键的“超集”），很多业务需求因此无法满足。

4.4.2 局部索引与全局索引的比较

全局索引的分区键一定是索引键的前缀，所以：

□ 全局非分区索引：

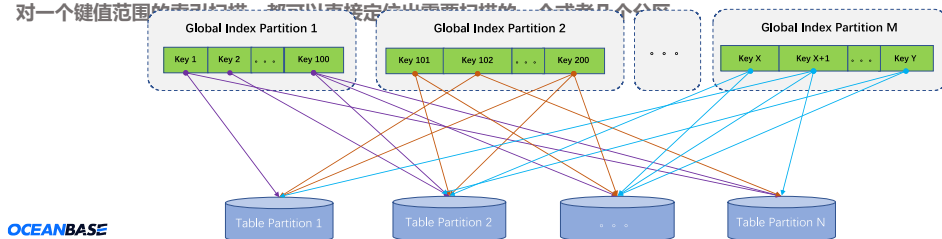
此时索引的结构和“非分区”表没有区别，只有一个完整的索引树，自然保证唯一性。

并且只有一个完整的索引树，自然没有多分区扫描的问题

□ 全局分区索引：

数据只可能落在一个固定的索引分区中，因此每一个索引分区内保证唯一性约束，就能在全表范围内保证唯一性约束。

全局索引能保证某一个索引键的数据只落在一个固定的索引分区中，所以无论是针对固定键值的索引扫描，还是针对一个键值范围的索引扫描，都可以直接定位到需要扫描的一个或者几个分区。



从使用者的角度来看，分区表的全局索引和非分区表的索引已经没有太大区别，除了一点：全局索引需要定义索引的分区模式。这里虽然也可以使用全局“非分区”索引，但是引进全局索引的目的就是针对数据量较大的分区表，对应的索引数据量往往也是非常大的，因此还是推荐使用全局分区索引，不但可以突破非分区索引面临的单点容量限制，在并发较大的情况下也能获得更好的性能。当然，全局索引也面临一些问题，主要是架构复杂，实现难度大，以及由此引发的一些相关问题，比如当索引数据和对应的主表数据位于不同的机器时，在事务内会面临数据一致性和性能方面的挑战。但考虑到全局索引给数据库使用者带来的巨大便利，付出一点代价也是值得的。

4.4.2 局部索引与全局索引的比较

综合二者的特点来看，局部索引和全局索引的适用场景是不一样的：

- 如果索引键里“包含完整的分区键”，使用本地索引是最高效的。
- 如果需要“不包含完整分区键”的唯一约束，只能用全局索引。
- 其它情况，需要case by case来看：通常来说，全局索引能为高频且精准命中的查询（比如单记录查询）提速并减少IO消耗，对范围查询则不一定哪种索引效果更好。
- 不能忽视全局索引在DML语句中引入的额外开销：数据更新时带来的跨机分布式事务，事务的数据量越大则分布式事务越复杂。

OCEANBASE

关于全局索引的实现原理，以及全局索引和本地索引的比较，前面都已经有过详细的描述，这是业界实现全局索引的基本思路，也是OceanBase中实现全局索引的基本思路。但是，前面我们更多的是讨论索引数据结构和主表数据结构的映射关系（一对一、一对多、多对多等），而忽略了另外一个很重要的因素要，那就是数据的**物理分布**。对于传统的**单点数据库**来说，每个数据结构之下都是“本地可访问”的存储层，比如一个本地数据库表空间（里面包含若干本地文件或设备），无论有多少个主表分区或者索引分区，数据库总是可以通过本地机器访问到，即Share-Everything的架构。但是对于OceanBase这样的**分布式数据库**来说，每一个主表分区和索引分区都可能分布在不同的机器上，比如“N个主表分区+M个全局索引分区”这种多对多的复杂情况，可能是（N+M）台物理机器上形成的（N*M）种索引键到主表分区的映射关系，这会带来诸多挑战：如果一个事务中要修改的N条记录分别位于N台不同的机器上，而它们对应的全局索引数据又位于另外M台不同的机器上，如何在这（N+M）台机器间保证主表数据和索引数据的同步更新？

还是上面所说的主表数据和索引数据分布在不同机器的场景，在保证数据一致性的前提下，如何进一步缩短跨机器访问的时间，进一步提高查询效率？

如何保证全局索引数据在全局范围内的读写一致性？一台机器上更新的索引数据，如何确保一定能被其它机器上后续的访问者读到？

可以说，如果不能解决上述问题，全局索引在分布式数据库中就失去了存在的

基础。因此，虽然业内有很多分布式数据库，但全局索引功能却并不是一个标配，很多大家所熟悉的分布式数据库产品（如MongoDB，Cassandra，Hbase等）并不提供全局索引功能，只有少数产品才能提供，比如Google F1/Spanner和CouchBase，这也从一个侧面印证了在分布式数据库中实现全局索引的难度。下面我就简单给大家介绍一下，OceanBase数据库是如何跨越上述的一个个难关，在2.0版本中实现了全局索引的功能。

首先来看第一个问题，如何保证主表数据和索引数据的跨机器同步更新。这个问题的本质，其实是分布式数据库中“分布式事务一致性”问题，主表数据和索引数据是同一个事务中的两部分数据，当它们分布在不同的机器上时，分布式事务需要保证事务的原子性（Atomicity）：两部分数据要么全部更新成功，要么全部失败，不会因事务异常而导致主表数据和索引数据的不同步。OceanBase数据库在几年前的1.0版本中就提供了分布式事务功能，利用Paxos协议和经过改良的“两阶段提交（Two-phase Commit）”方法，能在跨机器的分布式事务内保证ACID，即使事务的参与者发生异常（如机器宕机），也能确保分布式事务的完整性，避免了传统两阶段提交方法会导致的“事务部分未决（In-doubt Transaction）”问题，因此OceanBase完全可以保证跨机器事务中主表数据和索引数据的同步。

解决了分布式事务的一致性问题，我们来关注一下第二个问题，就是索引数据和主表数据分跨机器访问时的效率问题。这里面临的最大的挑战就是分布式事务的网络延迟和多次日志落盘，而这种物理开销是没有办法完全消除的，为此Google在F1的论文中明确说明不推荐太多的全局索引，并且应尽量避免对有全局索引的表做大事务访问。那OceanBase是如何应对这个问题的呢？前面提到过，OceanBase使用了经过改良的两阶段提交方法，这种改良不仅体现在保证数据的一致性上，也体现在性能上：和传统的两阶段提交相比，OceanBase的两阶段提交过程中会有更少的网络交互次数以及更少的写日志次数，而这些恰恰都是分布式事务中最耗时的操作。有了这样的优化，在很大程度上缩短了分布式事务的处理时间，提高了全局索引的处理效率。

最后一个问题，则是分布式数据库中全局（跨多台机器）的读写一致性问题。对OceanBase这样实现了快照隔离级别（Snapshot Isolation）和多版本并发控制（MVCC）的分布式数据库来说，如何在机器间有时钟差异的情况下，仍能维持时间戳（即版本号）在全局范围内的前后一致性，这是一个很重要的问题。在OceanBase数据库的2.0版本中，我们提供了“全局一致性快照”功能，并在此基础上实现了全局范围内的快照隔离级别和多版本并发控制，这样就能保证全局范围内前后事务的读写一致性，满足了全局索引的要求。

4.5 Hint

OCEANBASE

4.5 Hint

- 基于代价的优化器，与Oracle的Hint类似
- 如果使用MySQL的客户端执行带Hint的SQL语句，需要使用-c选项登陆，否则MySQL客户端会将Hint作为注释从用户SQL中去除，导致系统无法收到用户Hint
- 如果server端不认识你SQL语句中的Hint，直接忽略而不报错
- Hint只影响数据库优化器生成计划的逻辑，而不影响SQL语句本身的语义

4.5 Hint

➤ OceanBase支持的hint有以下几个特点：

- 不带参数的，如/*+ FUNC */
- 带参数的，如/*+ FUNC(param) */
- 多个hint可以写到同一个注释中，用逗号分隔，如/*+ FUNC1, FUNC2(param) */
- SELECT语句的hint必须紧接在关键字SELECT之后，其他词之前。如：SELECT /*+ FUNC */ ...
- UPDATE, DELETE语句的hint必须紧接在关键字UPDATE, DELETE之后

4.5 Hint 举例

- `/*+READ_CONSISTENCY(STRONG)*/`,
- `/*+READ_CONSISTENCY(WEAK)*/`
- `/*+query_timeout(100000000)*/` 单位微秒
- `/*+USE_MERGE(表名1 表名2)*/`
- `/*+INDEX(表名 索引名) */`
 - `(SELECT /*+ INDEX(t1 i1) , INDEX(t2 i2)*/ from t1, t2 WHERE t1.c1=t2.c1;)`

OCEANBASE

1. 表明是强一致性读还是弱一致性读，如果SQL语句中不指定，默认值根据系统变量`ob_read_consistency`的值决定。FROZEN表示读最近一次冻结点的数据，冻结版本号由系统自动选择。
2. 设定Server端执行语句的超时时间，超时以后Server端中断执行并返回超时错误码。
3. Merge Join 是先将关联表的关联列各自做排序，然后从各自的排序表中抽取数据，到另一个排序表中做匹配，因为merge join需要做更多的排序，所以消耗的资源更多。
4. 和Oracle类似，设定对于指定表的查询强制走索引名，如果索引不存在或者不可用，也不报错。例如

4.5 Hint 举例

- `/*+ PARALLEL(N)*/`
 - 指定语句级别的并发度。当该hint指定时，会忽略系统变量`ob_stmt_parallel_degree`的设置
- `/*+ leading(table_name_list)*/`
 - 指定表的连接顺序
 - 如果发现hint指定的`table_name`不存在，`leading` hint失效；如果发现hint中存在重复`table`，`leading` hint失效
- 更多hint，参考官网OB 文档

OCEANBASE

1. 指定并行数目， 否则是串行， 只有满足以下条件之一的SELECT语句， 才会并发
 1. 包含聚合函数
 2. 包含group by
 3. 包含order by
 4. 不包含limit例如select `/*+ parallel(5) */ count(*) from t1;`
2. 在一个多表关联的查询中， 该Hint指定由哪个表作为驱动表， 告诉优化器首先要访问哪个表上的数据

4.5 Hint 的行为理念

Hint是为了告诉优化器考虑hint中的方式，其它数据库的行为更像贪心算法，不会考虑全部可能的路径最优，hint的指定的方式就是为了告诉数据库加入到它的考虑范围。

OB优化器更像是动态规划，已经考虑了所有可能，因此hint告诉数据库加入到考虑范围就没有什么意义。**基于这种情况，OB的hint更多是告诉优化器按照指定行为做。**

OCEANBASE

4.5 OB Hint与MySQL, Oracle不一致的地方

除了之前提到的hint行为理念不一致外，其他不一致的细节有：

- MySQL 5.6版本 index hint如果Index不存在会报错；OB中如果Hint中index不存在，则Hint不生效，但是SQL语句也不会报错；这一点MySQL会在后面也改成不报错的方式
- Oracle leading hint，出现不存在的表时候hint是否生效会做推算，导致行为不确定，部分情况有效，部分情况全部无效。OB中如果Hint中表名不存在，则Hint不生效，但是SQL语句也不会报错。
- 健壮性更好，减少因Hint不正确而报错的情况。

OCEANBASE

4.5 OB当前支持的Hint

语句级别的hint

- FROZEN_VERSION
- QUERY_TIMEOUT
- READ_CONSISTENCY
- LOG_LEVEL
- QB_NAME
- ACTIVATE_BURIED_POINT
- TRACE_LOG
- MAX_CONCURRENT

计划相关的hint

- FULL
- INDEX
- LEADING
- USE_MERGE
- USE_HASH
- USE_NL
- ORDERED
- NO_REWRITE

注：更多hint，参考官网<https://www.oceanbase.com/docs/oceanbase/V2.2.50/hint>

OCEANBASE

4.6 SQL 执行性能监控

OCEANBASE

4.6 (g)v\$sql_audit

(g)v\$sql_audit 是全局 SQL 审计表，可以用来查看每次请求客户端来源，执行 server 信息，执行状态信息，等待事件及执行各阶段耗时等。

sql_audit 相关设置

- 设置 sql_audit 使用开关。

```
alter system set enable_sql_audit = true/false;
```

- 设置 sql_audit 内存上限。默认内存上限为 3G，可设置范围为 [64M, +∞]。

```
alter system set sql_audit_memory_limit = '3G';
```

4.6 (g)v\$sql_audit淘汰机制

后台任务每隔 1s 会检测是否需要淘汰。

触发淘汰的标准：

- 1、当内存或记录数达到淘汰上限时触发淘汰；

1.1 sql_audit 内存最大可使用上限： $\text{avail_mem_limit} = \min(\text{OBServer 可使用内存} * 10\%, \text{sql_audit_memory_limit})$;

淘汰内存上限：

当 avail_mem_limit 在 [64M, 100M] 时, 内存使用达到 $\text{avail_mem_limit} - 20\text{M}$ 时触发淘汰;

当 avail_mem_limit 在 [100M, 5G] 时, 内存使用达到 $\text{availmem_limit} * 0.8$ 时触发淘汰;

当 avail_mem_limit 在 [5G, $+\infty$] 时, 内存使用达到 $\text{availmem_limit} - 1\text{G}$ 时触发淘汰;

1.2 淘汰记录数上限：

当 sql_audit 记录数超过 900w 条记录时，触发淘汰。

停止淘汰的标准：

2.1 如果是达到内存上限触发淘汰则：

当 avail_mem_limit 在 [64M, 100M] 时, 内存使用淘汰到 $\text{avail_mem_limit} - 40\text{M}$ 时停止淘汰;

当 avail_mem_limit 在 [100M, 5G] 时, 内存使用淘汰到 $\text{availmem_limit} * 0.6$ 时停止淘汰;

当 avail_mem_limit 在 [5G, $+\infty$] 时, 内存使用淘汰到 $\text{availmem_limit} - 2\text{G}$ 时停止淘汰;

OCEANBASE 2.2 如果是达到记录数上限触发的淘汰则淘汰到 800w 行记录时停止淘汰。

4.6 SQL Trace

SQL Trace 能够交互式的提供上一次执行的 SQL 请求执行过程信息及各阶段的耗时。

SQL Trace 开关

SQL Trace 功能默认时关闭的，可通过 session 变量来控制其关闭和打开。

```
set ob_enable_trace_log = 0/1;
```

Show Trace

当 SQL Trace 功能打开后，执行需要诊断的 SQL，然后通过 show trace 能够查看该 SQL 执行的信息。

这些执行信息以表格方式输出，每列说明如下：

列名	说明
Title	记录执行过程某一个阶段点
KeyValue	记录某一个阶段点产生的一些执行信息
Time	记录上一个阶段点到这次阶段点执行耗时

4.6 Plan Cache 视图

•(g)v\$plan_cache_stat : 记录每个计划缓存的状态, 每个计划缓存在该视图中有一条记录;

•(g)v\$plan_cache_plan_stat :记录计划缓存中所有 plan 的具体信息及每个计划总的执行统计信息, 每个 plan 在该视图中一条记录;

•(g)v\$plan_cache_plan_explain : 记录某条 SQL 在计划缓存中的执行计划。

感谢学习

OCEANBASE