

OB SQL调优



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 / OB 运维、 监控与异常处理

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

4.1 SQL 调优方法

4.1 OB 架构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

4.1 SQL 性能问题来源

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

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

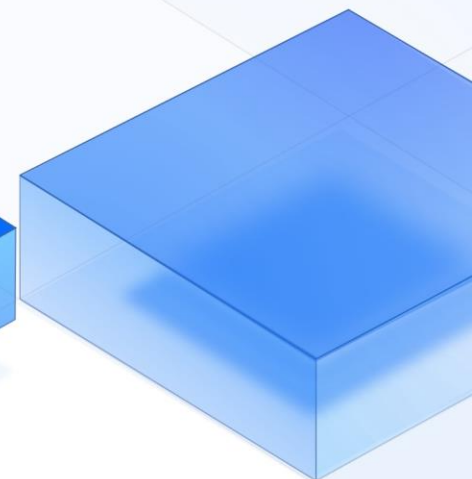
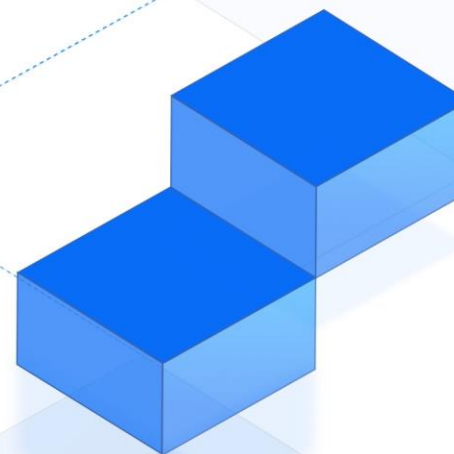
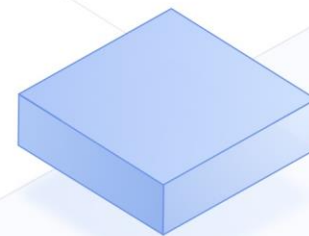
- 连接顺序、连接算法、分布式并行、查询改写

➤ 针对吞吐量的性能优化

- 优化慢 SQL

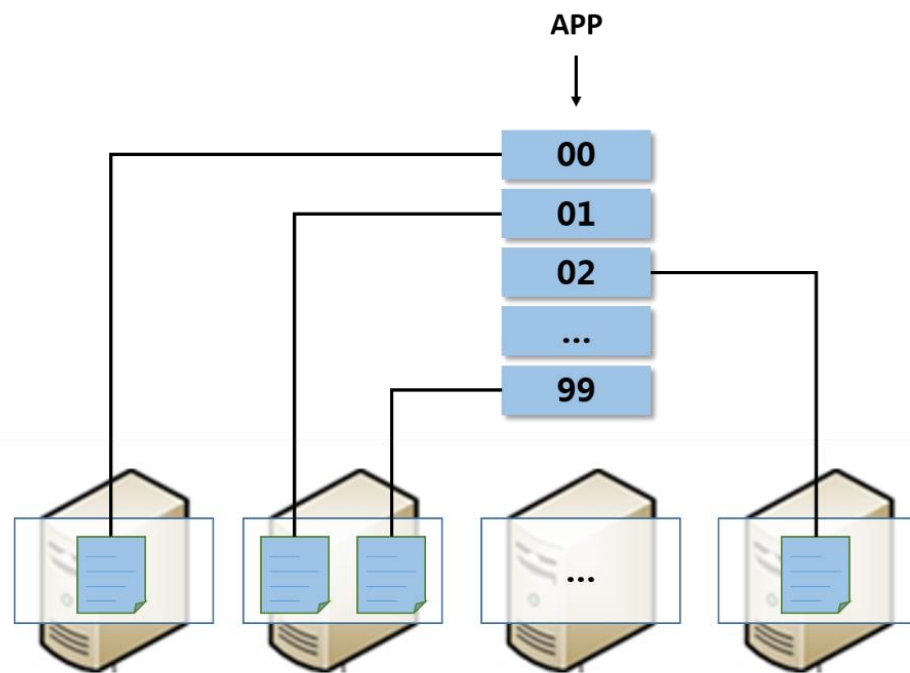
- 均衡 SQL 的流量资源

4.2 分区



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
 1. 可扩展性
 2. 可管理性
 3. 提高性能



4.2 OceanBase分区表特点

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

4.2 分区表

- 分为一级分区 和 二级分区
- OB 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

MySQL 模式

1. RANGE 分区
2. RANGE COLUMNS 分区
3. LIST 分区
4. LIST COLUMNS 分区
5. HASH 分区
6. KEY 分区
7. 组合分区

Oracle 模式

1. RANGE 分区
2. LIST 分区
3. HASH 分区
4. 组合分区

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)

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));
```

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')));
```


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));
```

4.2 生成列分区

一级分区

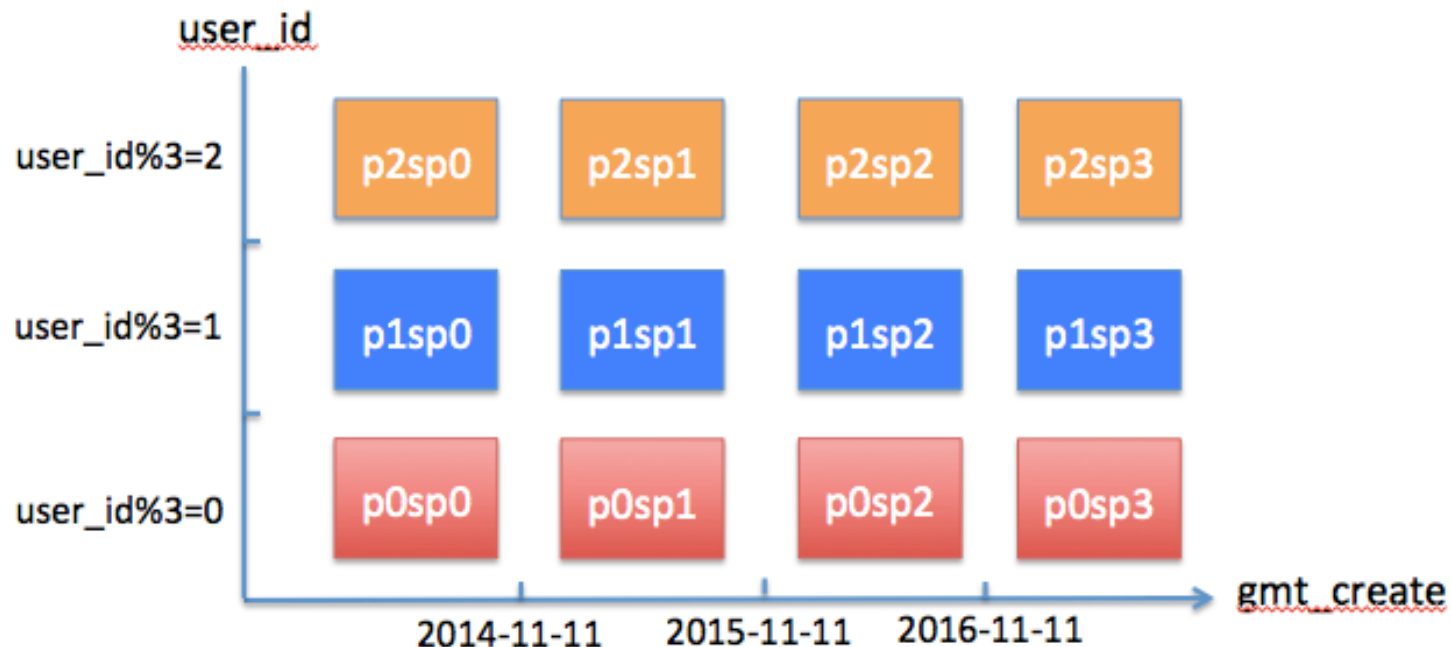
生成列是指这一列是由其他列计算而得

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;
```

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'));
```

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

4.2 分区选择和分区裁剪

► 分区选择

```
mysql> create table t2(c1 int primary key, c2 int, c3
int) partition by hash(c1) partitions 5;
Query OK, 0 rows affected (0.58 sec)
```

```
OceanBase (root@oceanbase)> explain select * from t1
partition(p4);
```

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

Outputs & filters:

```
-----
0 - output([t1.c1], [t1.c2], [t1.c3]), filter(nil),
   access([t1.c1], [t1.c2], [t1.c3]), partitions(p4)
```

► 分区裁剪

```
mysql> explain select * from t2 where c1 = 5 or c1 = 4;
```

```
| =====
|ID|OPERATOR                                |NAME|EST. ROWS|COST|
-----
|0 |EXCHANGE IN DISTR                       |    |4        |224 |
|1 | EXCHANGE OUT DISTR                     |    |4        |123 |
|2 |  TABLE GET                            |t2  |4        |123 |
=====
```

Outputs & filters:

```
-----
0 - output([t2.c1], [t2.c2], [t2.c3]), filter(nil)
1 - output([t2.c1], [t2.c2], [t2.c3]), filter(nil)
2 - output([t2.c1], [t2.c2], [t2.c3]), filter(nil),
   access([t2.c1], [t2.c2], [t2.c3]), partitions(p0, p4)
```

4.2 分区表的使用建议

1. 业务形态（热点数据打散、历史数据维护便利性、业务SQL的条件形态（分区裁剪）
2. OB各种分区类型的设置要求
3. 分区键必须是主键的子集
4. Range分区，最后一个不能是maxvalue
5. 考虑分区裁剪、partition wise join优化
6. Leader binding\Tablegroup
7. 为了避免写入放大问题，选择表的自定义主键时，不要使用随机生成的值，要尽量有序，比如时序递增的。
8. 分区个数：单机分区上限、单机租户允许创建的最大分区数量上限、单表分区数上限

```
create table test1 (  
  a varchar(32),  
  b varchar(32),  
  c varchar(32),  
  primary key (a, b)  
) partition by hash (b) partitions 32  
;
```

```
mysql> explain select * from t2 where c1 = 5 or c1 = 4;  
|=====|  
|ID|OPERATOR                               |NAME|EST. ROWS|COST|  
-----|  
|0 |EXCHANGE IN DISTR                       |    |4         |224 |  
|1 | EXCHANGE OUT DISTR                     |    |4         |123 |  
|2 |  TABLE GET                            |t2  |4         |123 |  
=====
```

Outputs & filters:

```
-----  
0 - output([t2.c1], [t2.c2], [t2.c3]), filter(nil)  
1 - output([t2.c1], [t2.c2], [t2.c3]), filter(nil)  
2 - output([t2.c1], [t2.c2], [t2.c3]), filter(nil),  
   access([t2.c1], [t2.c2], [t2.c3]), partitions(p0, p4)
```

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

4.3 路径选择 (Access Path Selection)

► 何为路径

- ✓ 主键
- ✓ 二级索引

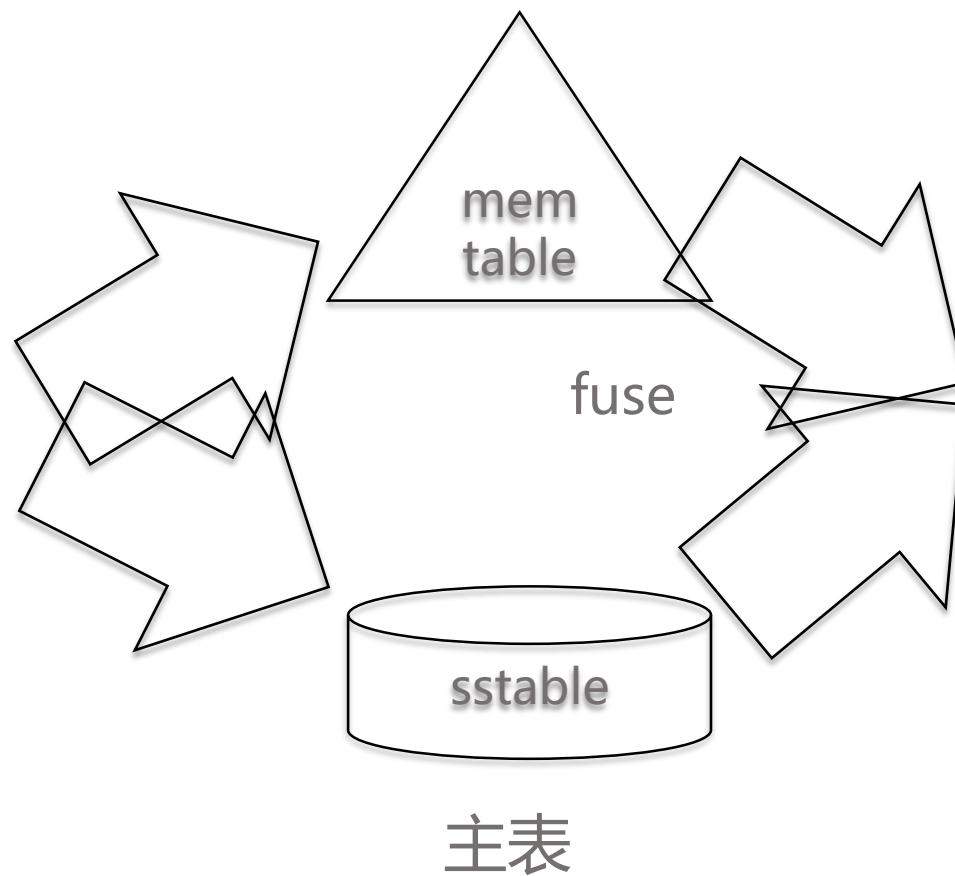
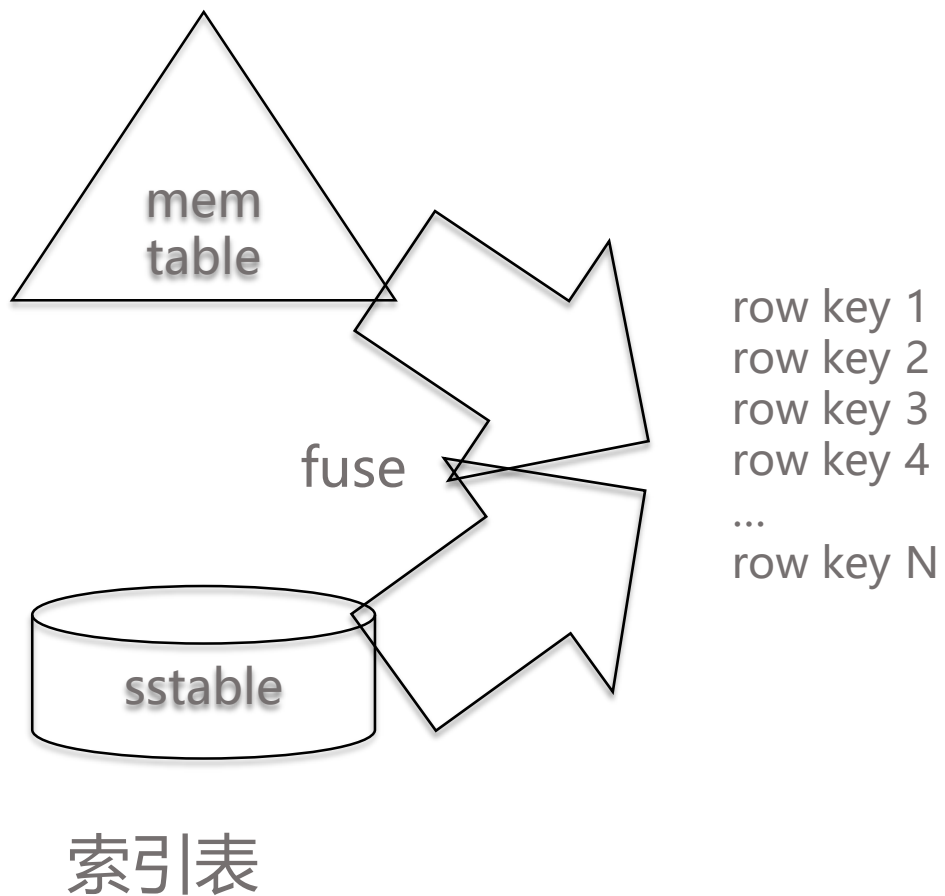
► 如何选择

- ✓ 规则模型
 - 前置规则 (正向)
 - 剪枝规则 (反向)
- ✓ 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

4.3 路径选择 – 索引回表



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

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?

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)
```

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)
```

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:
```

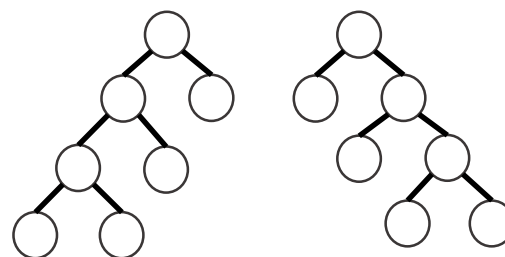
```
-----  
t1:optimization_method=rule_based, heuristic_rule=unique_index_with_indexback
```

```
Optimization Info:
```

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

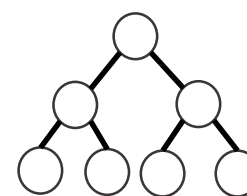
4.3 连接顺序

- 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



左深树

右深树



多枝树

- 优势
 - 搜索空间小
 - 更利于流水线
 - 内存空间 (foot-print) 小
 - 劣势
 - 无法利用并行执行
 - 可能错失最佳的执行计划
-
- 优势
 - 充分系统并行能力
 - 可能生成更好计划
 - 劣势
 - 搜索空间巨大
 - 执行消耗资源多

4.3 创建高效索引

1. 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
2. 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
3. 等值条件永远放在最前面
4. 过滤与排序数据量大的放前面

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,...)
```

4.3 创建索引

- 等值查询

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

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

- 范围查询

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

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，后续的字不段不参与索引过滤（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

- 等值和范围扫描

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

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）

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

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）。

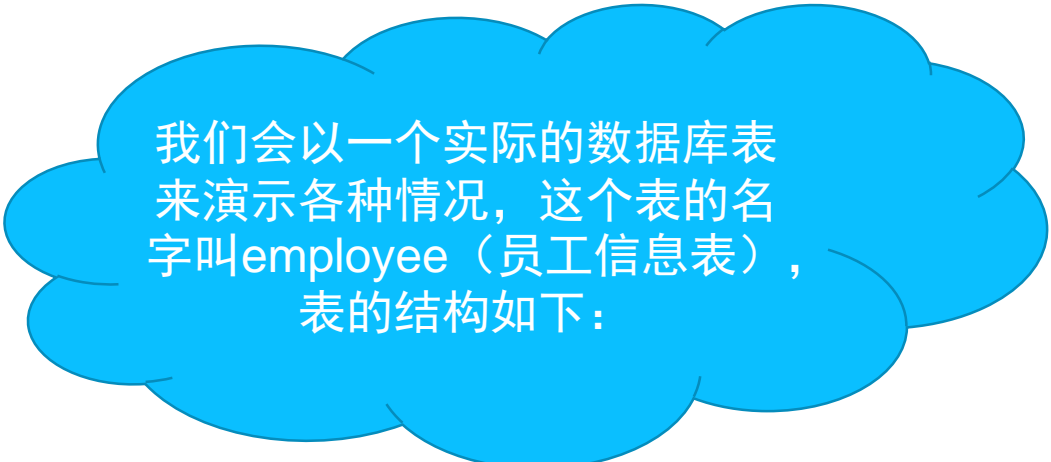
□ 主键

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 */
  ...
}
```



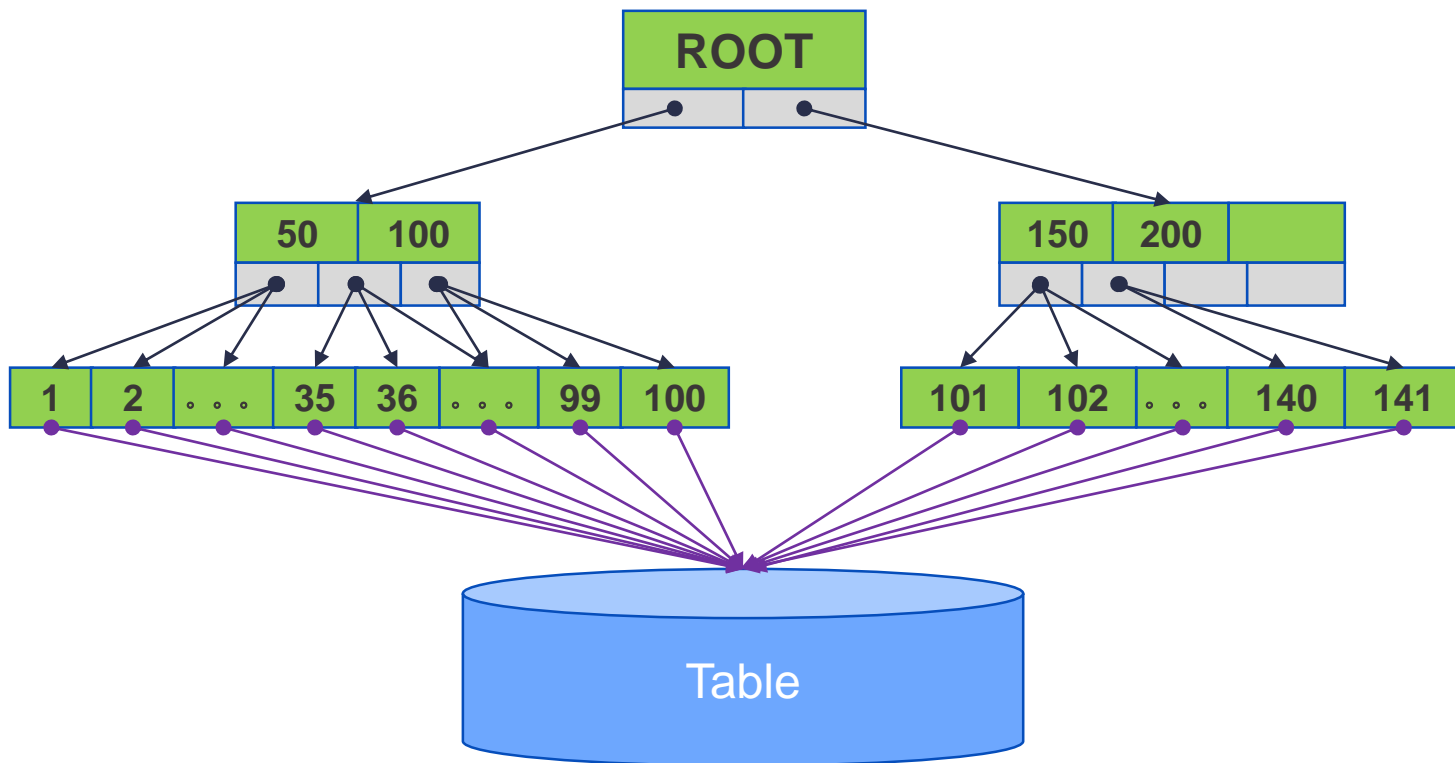
我们会以一个实际的数据库表来演示各种情况，这个表的名字叫employee（员工信息表），表的结构如下：

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_创建的索引：

idx_emp_id on employee (emp_id)



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

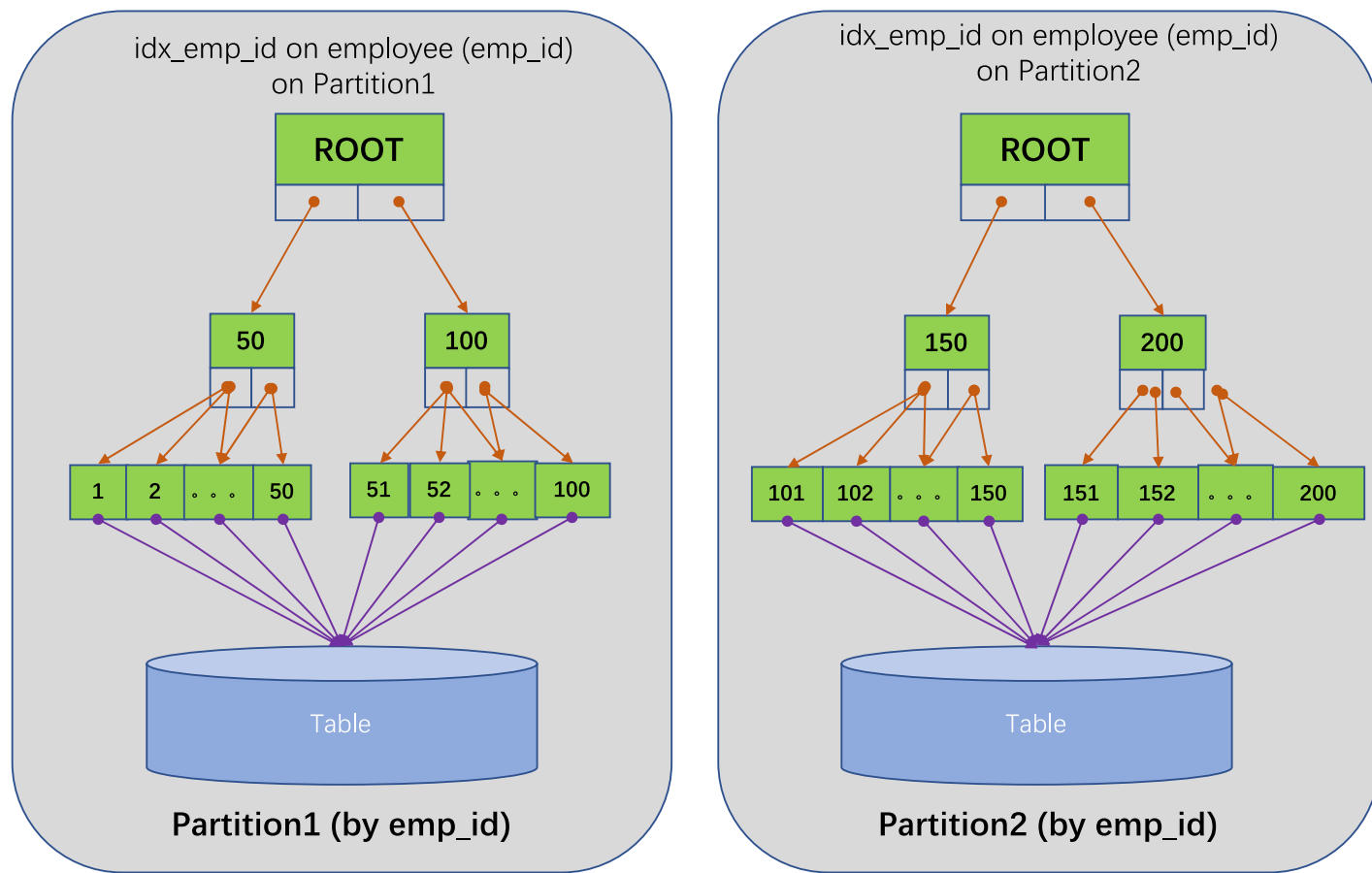
局部索引又名分区索引，创建索引的分区关键字是LOCAL，分区键等同于表的分区键，分区数等同于表的分区数，总之，局部索引的分区机制和表的分区机制一样。

□ 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。

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：



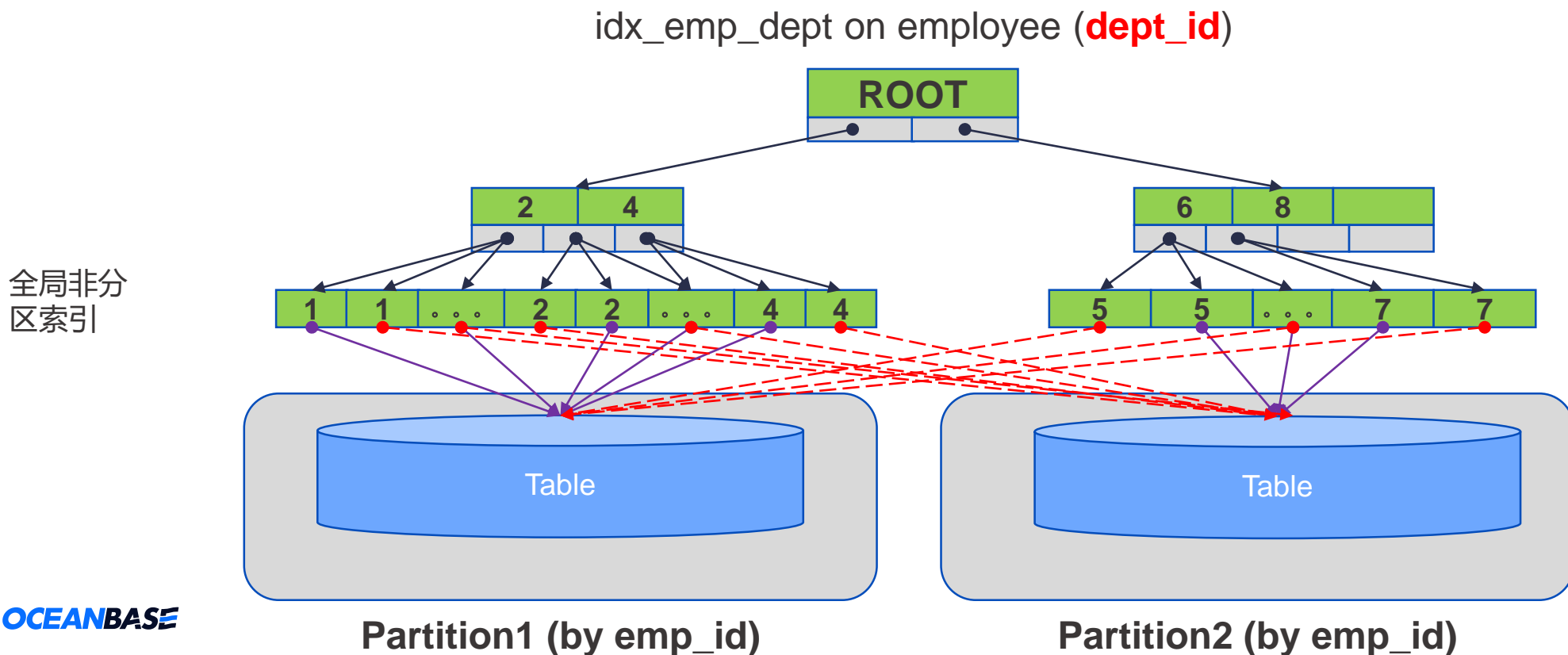
4.4.2 全局索引

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)



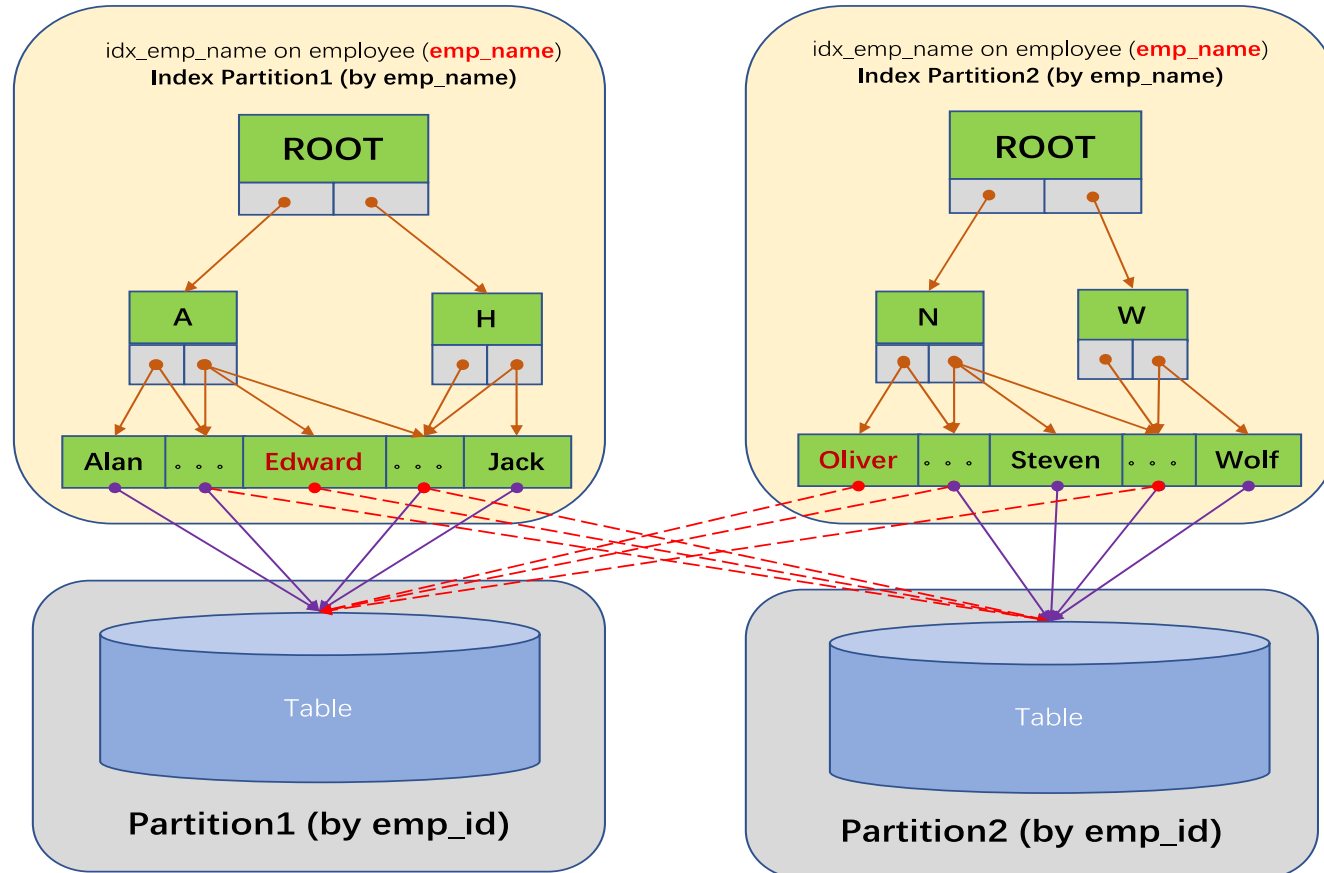
4.4.2 全局索引

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)

全局索引的分区键
一定是索引键本身

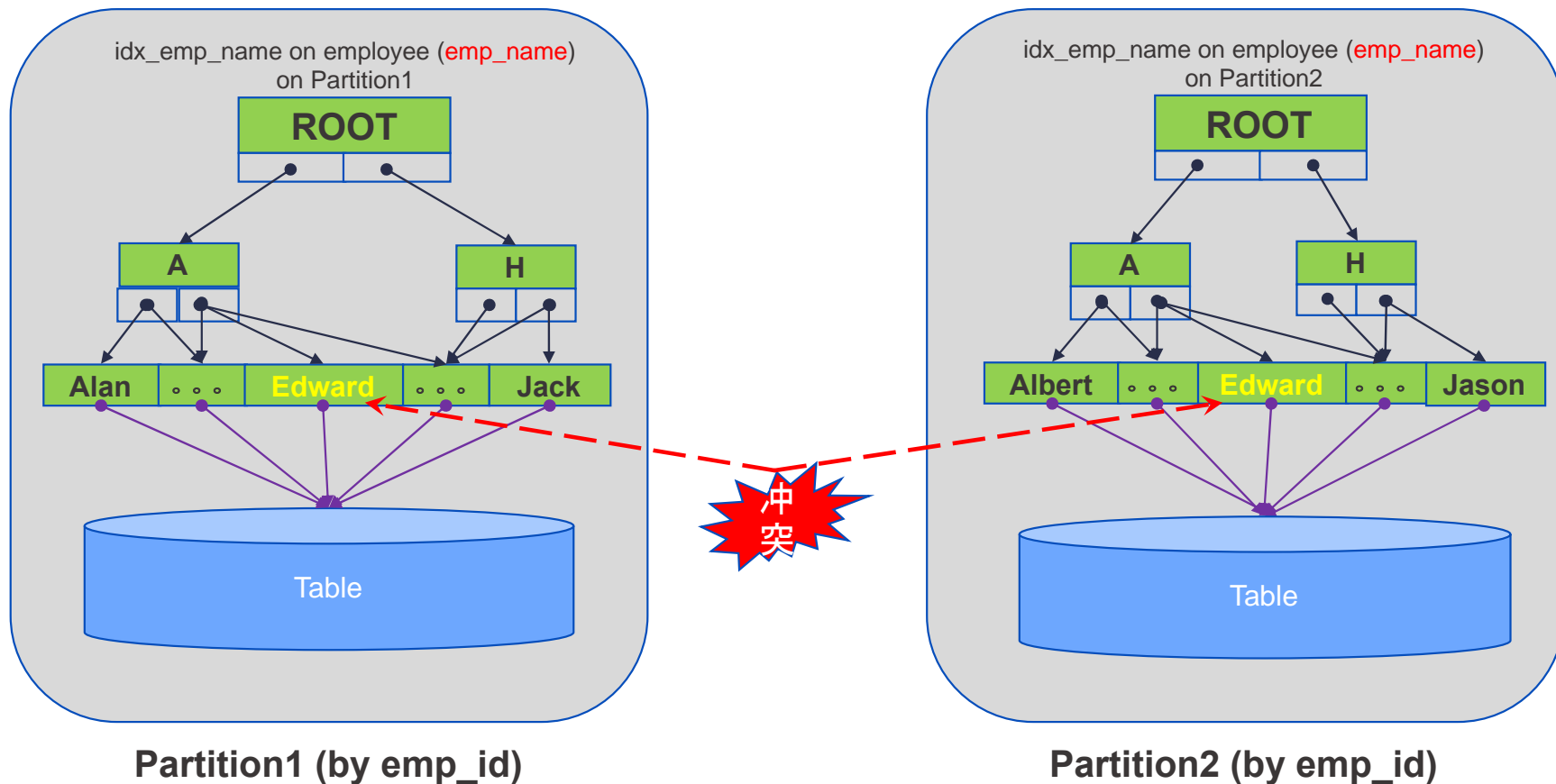


4.4.2 功能需求：在表的'分区键无关'的字段上建唯一索引

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



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：

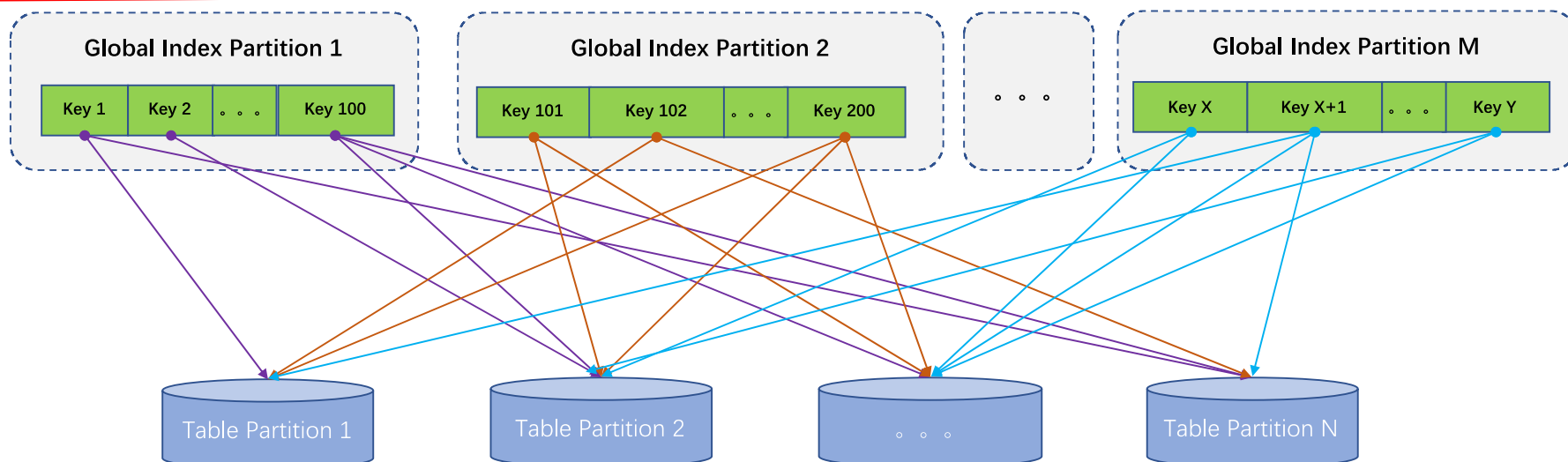
此时索引的结构和“非分区”表没有区别，只有一个完整的索引树，自然保证唯一性。

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，都可以直接定位出需要扫描的一个或者几个分区。



4.4.2 局部索引与全局索引的执行计划的比较

```
create index idx_t_p_hash_c1 on t_p_hash (c1) local;  
create index idx_t_p_hash_c3 on t_p_hash (c3) local ;  
explain extended select c1,c2 from t_p_hash where c3='100' ;
```

ID	OPERATOR	NAME	EST. ROWS	COST
0	PX COORDINATOR		0	10
1	EXCHANGE OUT DISTR	:EX10000	0	10
2	PX PARTITION ITERATOR		0	10
3	TABLE SCAN	t_p_hash(idx_t_p_hash_c3)	0	10

Outputs & filters:

```
0 - output([t_p_hash.c1(0x7f337ac4a590)], [t_p_hash.c2(0x7f337ac49210)]), filter(nil)  
1 - output([t_p_hash.c2(0x7f337ac49210)], [t_p_hash.c1(0x7f337ac4a590)]), filter(nil), dop=1  
2 - output([t_p_hash.c2(0x7f337ac49210)], [t_p_hash.c1(0x7f337ac4a590)]), filter(nil)  
3 - output([t_p_hash.c2(0x7f337ac49210)], [t_p_hash.c1(0x7f337ac4a590)]), filter(nil),  
   access([t_p_hash.c2(0x7f337ac49210)], [t_p_hash.c1(0x7f337ac4a590)]), partitions(p[0-2]),  
   is_index_back=true,  
   range_key([t_p_hash.c3(0x7f337ac4a0b0)], [t_p_hash.c2(0x7f337ac49210)], [t_p_hash.__pk_increment(0x7f337ac5a440)]), range(100,MIN,MIN ; 100,MAX,MAX),  
   range_cond([t_p_hash.c3(0x7f337ac4a0b0) = '100'(0x7f337ac49a70)])
```

4.4.2 局部索引与全局索引的执行计划的比较

create table t_p_key (c1 varchar(20),c2 int,c3 varchar(20)) partition by key (c2) partitions 3;
create unique index idx_t_p_key_c3_g on t_p_key (c3) global partition by key (c3) partitions 3;
explain extended select c1,c2 from t_p_key where c3='66';

ID	OPERATOR	NAME	EST. ROWS	COST
0	EXCHANGE IN REMOTE		1	91
1	EXCHANGE OUT REMOTE		1	91
2	TABLE LOOKUP	t_p_key	1	91
3	TABLE SCAN	t_p_key(idx_t_p_key_c3_g)	1	36

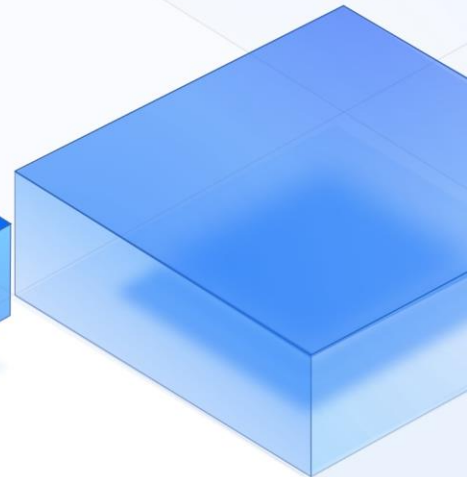
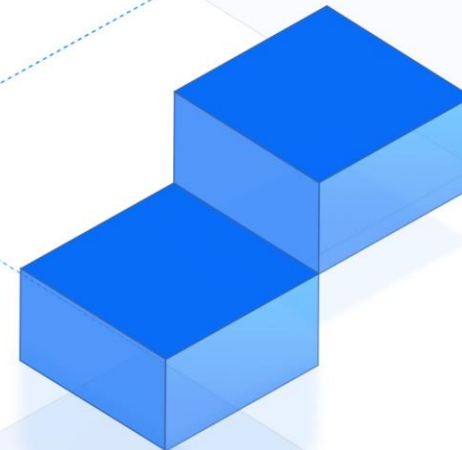
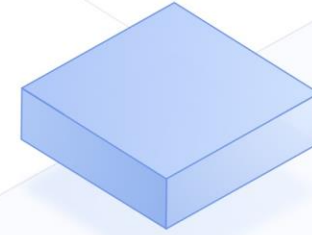
Outputs & filters:

0 - output([t_p_key.c1(0x7f337ac4b750)], [t_p_key.c2(0x7f337ac49270)]), filter(nil)
1 - output([t_p_key.c2(0x7f337ac49270)], [t_p_key.c1(0x7f337ac4b750)]), filter(nil)
2 - output([t_p_key.c2(0x7f337ac49270)], [t_p_key.c1(0x7f337ac4b750)]), filter(nil), partitions(p[0-2])
3 - output([t_p_key.c2(0x7f337ac49270)], [t_p_key.__pk_increment(0x7f337ac5b600)]), filter(nil), access([t_p_key.c2(0x7f337ac49270)], [t_p_key.__pk_increment(0x7f337ac5b600)]), partitions(p2), is_index_back=false, range_key([t_p_key.c3(0x7f337ac4a7b0)], [t_p_key.shadow_pk_0(0x7f337ac5c0b0)], [t_p_key.shadow_pk_1(0x7f337ac5c320)]), range(66,MIN,MIN ; 66,MAX,MAX), range_cond([t_p_key.c3(0x7f337ac4a7b0) = '66'(0x7f337ac4aea0)])

4.4.2 局部索引与全局索引的取舍

1. 如果查询条件里“包含完整的分区键”，使用本地索引是最高效的。
2. 如果需要“不包含完整分区键”的唯一约束，
 1. 用全局索引
 2. 或者本地索引，且需要索引列上必须带上表的分区键
3. 其它情况，case by case：
 1. 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；对范围查询则不一定哪种索引效果更好。
 2. 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。
4. 如果数据量较大，或者容易出现索引热点，可考虑创建全局分区索引

4.5 Hint



4.5 Hint

1. 基于代价的优化器，与Oracle的Hint类似
2. 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
3. 如果server端不认识你SQL语句中的Hint，直接忽略而不报错
4. 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:

1. 不带参数的, 如/*+ FUNC */

2. 带参数的, 如/*+ FUNC(param) */

3. 多个hint可以写到同一个注释中, 用逗号分隔, 如/*+ FUNC1, FUNC2(param) */

4. SELECT语句的hint必须近接在关键字SELECT之后, 其他词之前。如: SELECT /*+ FUNC */ ...

5. UPDATE, DELETE语句的hint必须紧接在关键字UPDATE, DELETE之后

4.5 Hint 举例

1. `/*+READ_CONSISTENCY(STRONG)*/,`
`/*+READ_CONSISTENCY(WEAK)*/`
2. `/*+query_timeout(1000000000)*/` 单位微秒
3. `/*+USE_MERGE(表名1 表名2)*/`
4. `/*+INDEX(表名 索引名)*/`

4.5 Hint 举例

1. `/*+ PARALLEL(N)*/`指定语句级别的并发度。

① 当该hint指定时，会忽略系统变`ob_stmt_parallel_degree`的设置

2. `/*+ leading(table_name_list)*/`

① 指定表的连接顺序

② 如果发现hint指定的`table_name`不存在，leading hint失效；

③ 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

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。**

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>

4.5 查询限流的例子

- ▶ Hint中使用max_concurrent
- ▶ ? 表示需要参数化的参数

```
OceanBase (root@oceanbase)> create table t1(a int primary key, b int, c int);
Query OK, 0 rows affected (0.15 sec)
OceanBase (root@oceanbase)> create outline ol_1 on select/*+max_concurrent(0)*/ * from t1 where b =1 and c = 1;
Query OK, 0 rows affected (0.06 sec)
OceanBase (root@oceanbase)> select * from t1 where b =1 and c = 1;
ERROR 5268 (HY000): SQL reach max concurrent num 0
OceanBase (root@oceanbase)> select * from t1 where b =1 and c = 2;
Empty set (0.01 sec)
OceanBase (root@oceanbase)> create outline ol_2 on select/*+max_concurrent(0)*/ * from t1 where b =1 and c = ?;
Query OK, 0 rows affected (0.05 sec)
OceanBase (root@oceanbase)> select * from t1 where b =1 and c = 1;
ERROR 5268 (HY000): SQL reach max concurrent num 0
OceanBase (root@oceanbase)> select * from t1 where b =1 and c = 2;
ERROR 5268 (HY000): SQL reach max concurrent num 0
```

4.6 SQL 执行性能监控

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看什么

1. retry 次数是否很多(RETRY_CNT字段), 如果次数很多, 则可能有锁冲突或切主等情况
2. queue time 的值是否过大(QUEUE_TIME 字段), 很高表明CPU资源不够用
3. 获取执行计划时间(GET_PLAN_TIME), 如果时间很长, 一般会伴随 IS_HIT_PLAN = 0, 表示没有命中 plan cache
4. 查看 EXECUTE_TIME 值, 如果值过大, 则:
 1. 查看是否有很长等待事件耗时
 2. 分析逻辑读次数是否异常多(突然有大账户时可能会出现)

```
***** 1. row *****
      ELAPSED_TIME: 1145
      EXECUTE_TIME: 1098
    USER_IO_WAIT_TIME: 0
          QUEUE_TIME: 10
          RETRY_CNT: 0
MEMSTORE_READ_ROW_COUNT: 62
  SSSTORE_READ_ROW_COUNT: 186
1 row in set (3.23 sec)
```

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： $\text{avail_mem_limit} = \min(\text{OBServer可用内存} \times 10\%, \text{sql_audit_memory_limit})$;

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 $\text{availmem_limit} \times 0.8$ 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 $\text{availmem_limit} \times 0.6$ 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;

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

Title	KeyValue	Time
query start	trace_id: "[Y3B6C6451982C-3E9627]";	0
parse start	stmt: "select count(*) from __all_table";	99
pc get plan start		16
pc get plan end		50
resolve start		62
resolve end		355
transform start		105
transform end		107
optimizer start		3
optimizer end		623
CG start		1
CG end		156
execution start		87
execution end		28364
query end		166

列
Title
KeyValue
Time

息
耗时

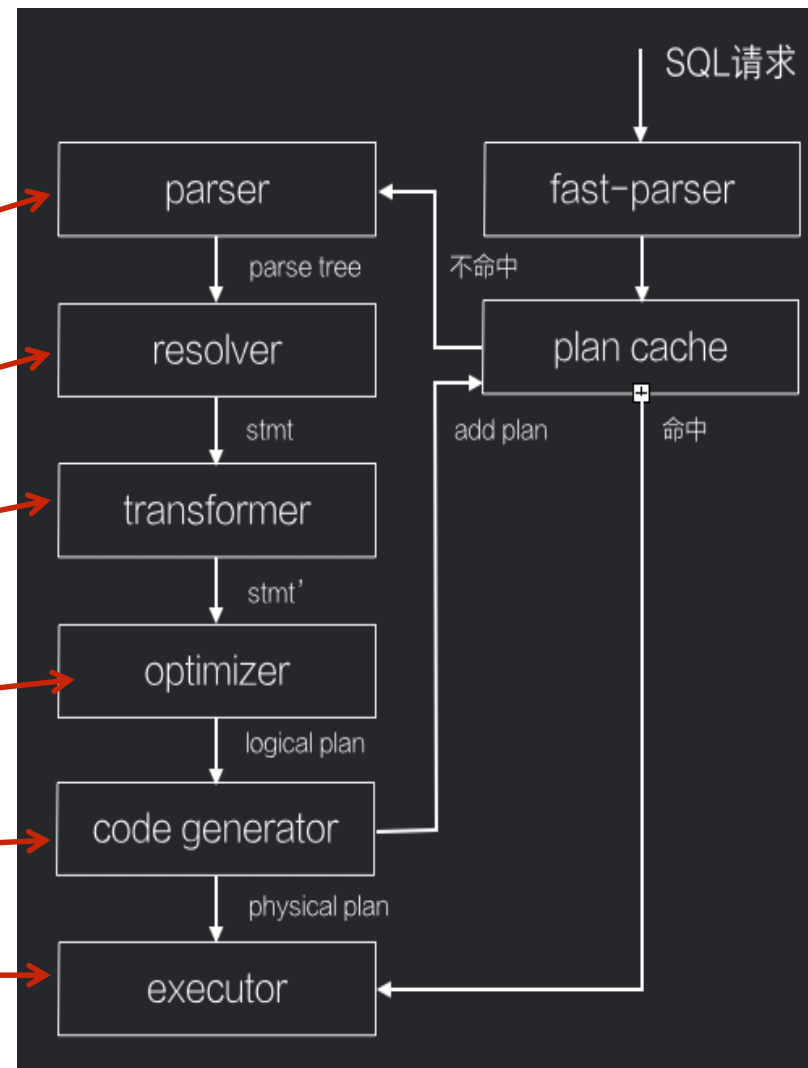
4.6 查看各阶段耗时

```
oceanbase> set ob_enable_trace_log = 1
Query OK, 0 rows affected (0.01 sec)
OceanBase (root@oceanbase)> select count(*) from __all_table;
+-----+
| count(*) |
+-----+
|      168 |
+-----+
1 row in set (0.03 sec)
```

```
OceanBase (root@oceanbase)> show trace;
```

Title	KeyValue	Time
query start	trace_id: "[Y3B6C6451982C-3E9627]";	0
parse start	stmt: "select count(*) from __all_table";	99
pc get plan start		16
pc get plan end		50
resolve start		62
resolve end		355
transform start		105
transform end		107
optimizer start		3
optimizer end		623
CG start		1
CG end		156
execution start		87
execution end		28364
query end		166

```
15 rows in set (0.01 sec)
```



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