# MySQL查询优化浅析

网易杭研-何登成

# 个人简介

- 姓名：何登成
- 工作：
  - 就职于网易杭州研究院，进行自主研发的TNT存储引擎的架构设计/研发工作
- 联系方式
  - 邮箱：he.dengcheng@gmail.com
  - 微博：何_登成
  - 主页：http://hedengcheng.com/

# 何为查询优化？

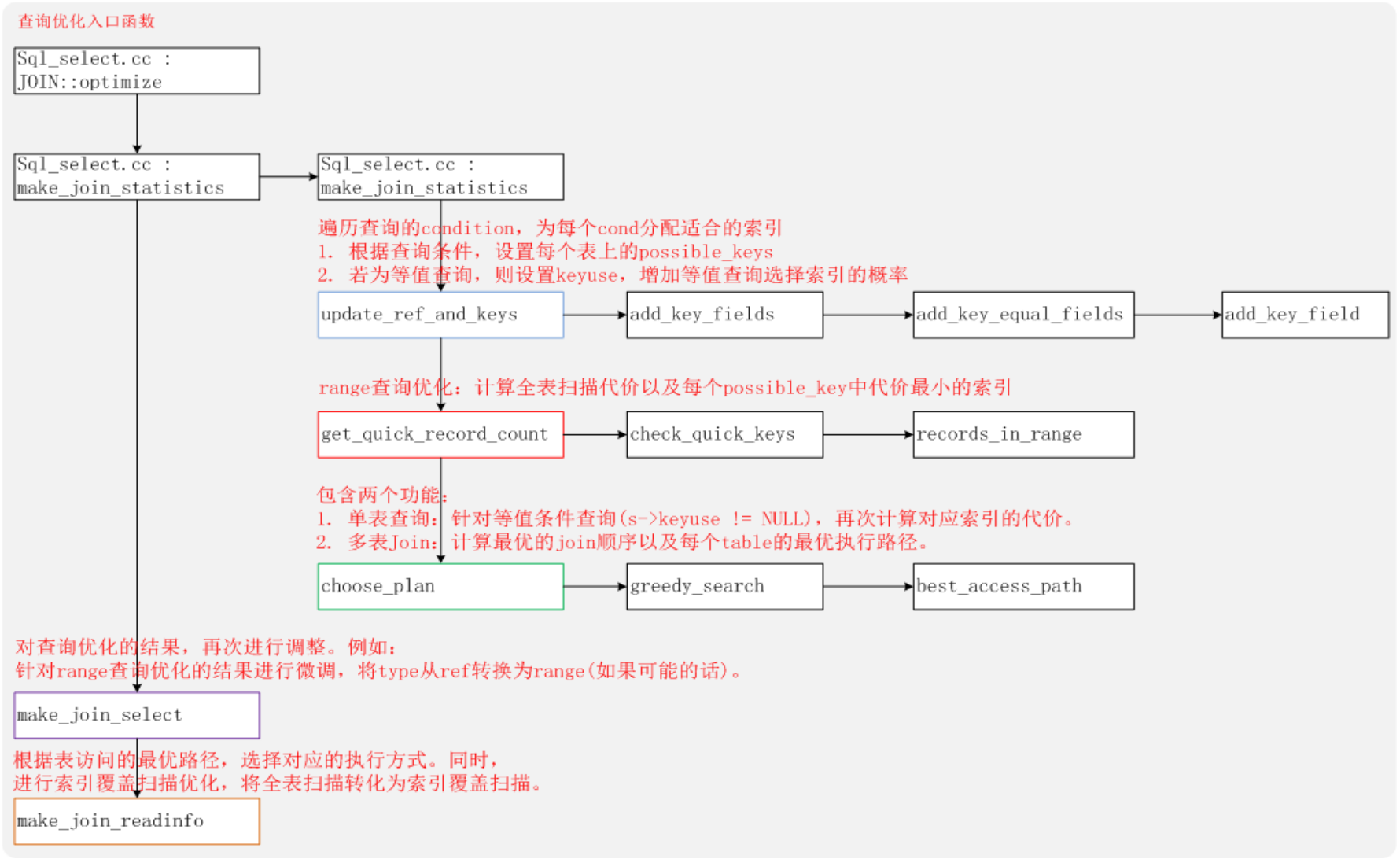
- **目标**
  - 给定一个SQL，查找SQL最优(局部最优)的执行路径，使得用户能够更快的得到SQL执行的结果

- **指标**
  - 代价模型；
  - SQL的每一种执行路径，均可计算一个对应的执行代价，代价越小，执行效率越高；反之则反之；

# 大纲

- MySQL Optimizer流程
- **MySQL Range Optimizer** (分享重点)
  - Cost模型
  - 统计信息
    - MySQL Server层统计信息
    - InnoDB层统计信息
    - 动态收集统计信息
    - 统计信息收集策略
  - Range Query Examples
- MySQL Join Optimizer
- MySQL Optimizer Enhancement

# 总流程


查询优化入口函数
Sql_select.cc :
JOIN::optimize
Sql_select.cc :
make_join_statistics
Sql_select.cc :
make_join_statistics
遍历查询的condition，为每个cond分配适合的索引
1. 根据查询条件，设置每个表上的possible_keys
2. 若为等值查询，则设置keyuse，增加等值查询选择索引的概率
update_ref_and_keys
add_key_fields
add_key_equal_fields
add_key_field
range查询优化：计算全表扫描代价以及每个possible_key中代价最小的索引
get_quick_record_count
check_quick_keys
records_in_range
包含两个功能：
1. 单表查询：针对等值条件查询(s->keyuse != NULL)，再次计算对应索引的代价。
2. 多表Join：计算最优的join顺序以及每个table的最优执行路径。
choose_plan
greedy_search
best_access_path
对查询优化的结果，再次进行调整。例如：
针对range查询优化的结果进行微调，将type从ref转换为range(如果可能的话)。
make_join_select
根据表访问的最优路径，选择对应的执行方式。同时，
进行索引覆盖扫描优化，将全表扫描转化为索引覆盖扫描。
make_join_readinfo

# MySQL Range Optimizer

- **Range Optimizer有哪些问题**
  - 全表扫描 or 索引扫描选择？
  - 全表扫描的代价如何计算？
  - 聚簇索引Range查询代价如何计算？
  - 二级索引Range查询代价如何计算？
  - 索引覆盖扫描 vs 索引非覆盖扫描？
  - 表级统计信息有哪些？
  - 统计信息在Range查询优化中何用？
  - 统计信息何时收集？收集算法？

# Range Query-代价模型

- 总代价模型
  - **COST = CPU Cost + IO Cost**
- CPU Cost
  - MySQL上层，处理返回记录所花开销
  - CPU Cost = records / TIME_FOR_COMPARE = records / 5
  - 每5条记录的处理时间，作为 1 Cost
- IO Cost
  - 存储引擎层面，读取页面的IO开销。
  - 以下InnoDB为例
    - 聚簇索引
    - 二级索引

# Range Query-聚簇索引

- 聚簇索引(IO Cost)
  - 全扫描
    - IO Cost = table->**stat_clustered_index_size**
    - 聚簇索引页面总数
    - 一个页面作为 1 Cost
  - 范围扫描
    - IO Cost = [(**ranges** + **rows**) / **total_rows** ] * **全扫描 IO Cost**
    - 聚簇索引范围扫描与返回的记录成比率。

# Range Query-二级索引

- 二级索引(IO Cost)
  - 索引覆盖扫描
    - 索引覆盖扫描，减少了返回聚簇索引的IO代价
      - **keys_per_block** =
        (stats_block_size / 2) / (key_info[keynr].key_length + ref_length + 1)
      - *stats_block_size / 2* → 索引页半满
    - IO Cost
      - (records + keys_per_block - 1) / keys_per_block
    - 计算range占用多少个二级索引页面，既为索引覆盖扫描的IO Cost

# Range Query-二级索引

- 二级索引(IO Cost 续)
  - 索引非覆盖扫描
    - 索引非覆盖扫描，需要回聚簇索引读取完整记录，增加IO代价
    - IO Cost = (ranges + rows)
    - **ranges：** 多少个范围。
      - 对于IN查询，就会转换为多个索引范围查询
    - **rows：** 为范围中一共有多少记录。
      - 由于每一条记录都需要返回聚簇索引，因此每一条记录都会产生 1 cost

# Cost模型分析

- 聚簇索引扫描代价为索引页面总数量
- 二级索引覆盖扫描代价较小
- 二级索引非覆盖扫描，代价巨大
  - 未考虑类似于Oracle中的聚簇因子(Cluster factor)影响？
- Cost模型的计算，需要统计信息的支持
  - **stat_clustered_index_size**
  - **ranges**
  - **records/rows**
  - **stats_block_size**
  - **key_info[keynr].key_length**
  - **rec_per_key**
  - **…**

# 统计信息

- **MySQL Server层的统计信息**
  - ha_statistics
    - 引擎负责设置
  - CONST
  - VARIABLE
- **InnoDB层的统计信息**
  - dict_table_struct
- **语句级统计信息**
  - 每个查询语句，指定不同的Range
    - 不同的Range，包含的records数量不同
    - 同一Range，不同的索引，包含的records数量不同
  - **records_in_range**

# MySQL Server层统计信息

- **CONST统计信息**
  - 此类统计信息，在表创建之后，就基本维持不变，类似于常量(非完全不变)

- 种类
  - max_data_file_length、data_file_name、**block_size**... 不变
  - **block_size**
    - 计算索引覆盖扫描Cost所需，页面大小
  - **rec_per_key**... 会变化
    - 标识一个**索引键(包括前缀键值)**相同相同取值的**平均个数**
    - 算法：*rec_per_key = total_rows / key_distinct_count*
    - 此参数，是MySQL进行Join Optimize的基础

- 收集策略
  - 表第一次open
  - analyze命令
  - 由InnoDB收集，并返回MySQL Server

# MySQL Server层统计信息

- **VARIABLE统计信息**
  - 此类统计信息，随着记录的U/D/I操作，会发生显著的变化

- 种类
  - **records**：　　记录数量
    - 直接从InnoDB的统计信息中复制，**不重新收集**

      n_rows　　= ib_table->stat_n_rows;

      stats.records　= (ha_rows)n_rows;

    - **计算全表扫描CPU代价**；
  - data_file_length：　　聚簇索引总大小(非叶 + 叶)
  - index_file_length：　　所有二级索引总大小
  - ...

- 收集策略
  - 表第一次open
  - analyze命令
  - 语句执行时

# InnoDB层统计信息

- **InnoDB层统计信息**
  - 除了设置MySQL Server层统计信息外，还在本层维护了自身的统计信息
  - 根据此统计信息，计算全表扫描/索引扫描代价

- 主要统计信息
  - **stat_n_rows**
    - 表记录数量；I/U/D操作时，实时修改；
    - 用于设置MySQL Server层的records信息
  - **stat_clustered_index_size**
    - 聚簇索引页面总数量
    - 计算MySQL Server层，data_file_length信息
    - **计算全表扫描IO代价**
  - stat_sum_of_other_index_size
  - **stat_modified_counter**
    - I/U/D，此值++

- 收集策略
  - 第一次open
  - **stat_modified_counter**取值：( > 2 000 000 000) or > (stat_n_rows/16)

# InnoDB层统计信息

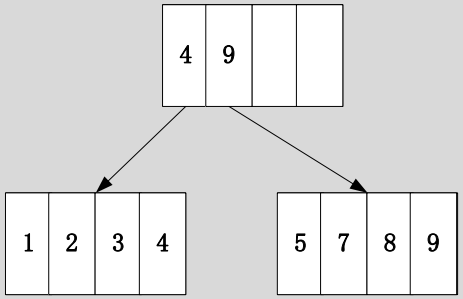
- **收集算法**
  - 统计索引中叶页面数量
    - index->stat_n_leaf_pages
  - 随机定位索引中的8个叶页面
    - srv_stats_sample_pages = 8;
  - **统计页面中，前缀索引列组合的Distinct数量**
    - 例如：Index idx (a, b, c)，包含3列
    - Distinct[a] = ? ; Distinct[a, b] = ? ; Distinct[a,b,c] = ?
  - 根据以上信息，计算
    - 表数据量
    - **每个索引前缀组合的Distinct数量**
      - 用于计算MySQL Server层的**rec_per_key**信息
      - 是Join Optimizer最重要的统计信息
- 优化
  - 统计信息持久化：MySQL 5.6.2
  - 统计信息更准确：增加Sample Rate
    - srv_stats_persistent_sample_pages = 20;

# InnoDB层统计信息

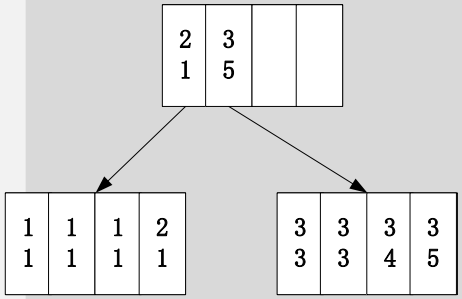
```
Create table t1 (a int primary key, b int, c int, d int) engine = innodb;
create index idx_bc on t1 (b, c);

(1,1,1,1),(2,1,1,2),(3,2,1,3),(4,3,3,4),(5,3,3,5),(7,3,4),(8,3,5),(9,1,1)
```

primary




idx_bc




```
stat_n_rows = 8;

stat_n_diff_key_vals = 8;

stat_clustered_index_size = 3;

rec_per_key[a] = 1;
```

```
stat_n_diff_key_vals[b] = 3;

stat_n_diff_key_vals[b,c] = 5;

rec_per_key[b] = 8/3;

rec_per_key[b,c] = 8/5;
```

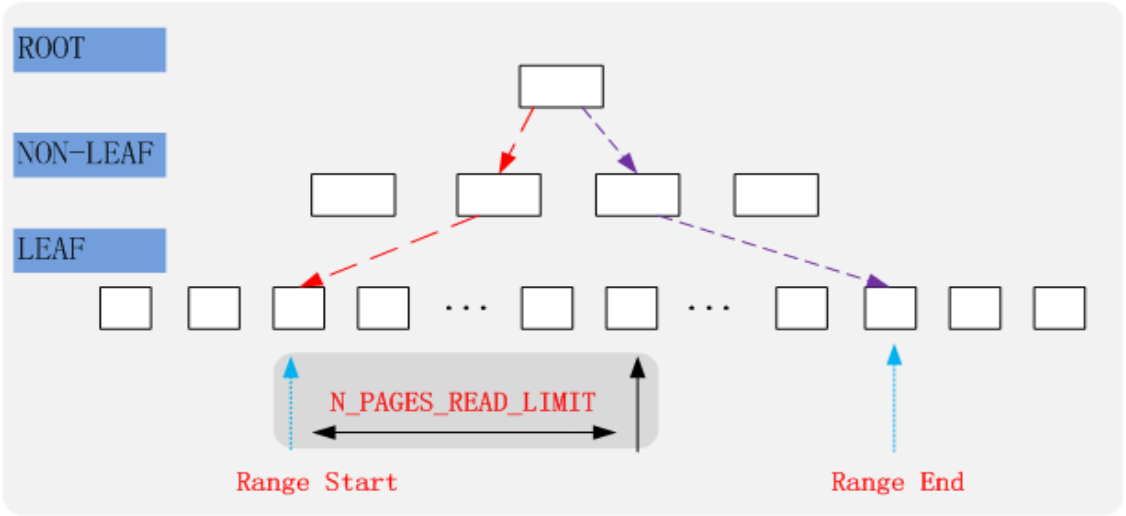
# Statement级统计信息

- 语句级统计信息
  - 到目前为止，MySQL/InnoDB尚缺少哪些统计信息呢？
  - **stat_clustered_index_size (已有)**
  - **Ranges (根据where条件分析得出)**
  - **Records/Rows (无)**
  - **stats_block_size (已有)**
  - **key_info[keynr].key_length (MySQL上层维护)**
  - **rec_per_key (已计算)**

- rows统计信息
  - **功能**
    - 聚簇索引范围查询
    - 二级索引覆盖范围扫描
    - 二级索引非覆盖范围扫描
  - **无rows(records)统计信息**
    - 无法进行范围查询

# Statement级统计信息

- records_in_range
  - *每个范围查询，在查询优化阶段，针对每一个可选的索引，都会调用存储引擎层面提供的records_in_range函数，计算查询范围中的记录数量：rows/records*
- 算法简析
  - Range Start
  - Range End
  - N_PAGES_READ_LIMIT
    - **default: 10**
    - 越大：越精确，性能越差
    - 越小：...
  - 输出
    - **estimate rows between [range start, range end]**
  - rows
    - **records_in_range = records_in_upper_level(叶页面数) * records_per_leaf**



# Range Query-Possible Keys

- 我们已经能计算什么？
  - 以下各种访问路径的Cost：CPU + IO
    - 全表扫描
    - 聚簇索引范围扫描
    - 二级索引扫描
    - 二级索引范围扫描(Index Coverage)
    - 二级索引范围扫描(No Coverage)

- 尚缺少什么？
  - 对于一个Range Query，哪些索引是可选索引？
    - Possible Keys
  - 以where中的部分算子作用列打头的索引
    - 可选算子：>, >=, =, <, <=, in…
    - select * from t1 where a 算子 …;
      - **若有以a打头的索引idx_a，则idx_a即为一个possible key**

# Range Query Optimizer流程

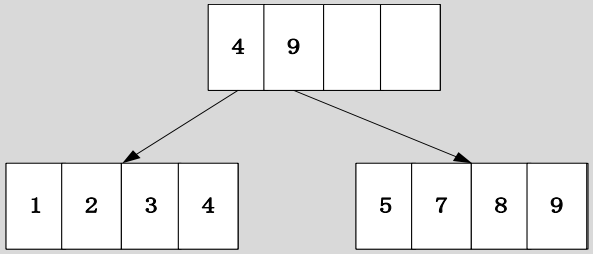
- Range Query Optimizer主流程
  - 1. 根据查询条件，计算所有的possible keys；
  - 2. 计算全表扫描代价
    - **Cost_all**
  - 3. **计算最小的索引范围访问代价；**
    - 对于每一个possible keys(可选索引)，调用records_in_ranges函数，计算范围中的rows；
    - 根据rows，计算二级索引访问代价；
    - 获取Cost最小的二级索引访问：**Cost_range**；
  - 4. 对比全表扫描代价与索引范围代价
    - Cost_all > Cost_range → 全表扫描；
    - Cost_all < Cost_range → 索引范围扫描；
- 流程分析
  - Range Query Optimizer，最慢的在于步骤3
    - 减少possible keys；
    - 减少records_in_range调用；

# Range Query Optimizer – 举例

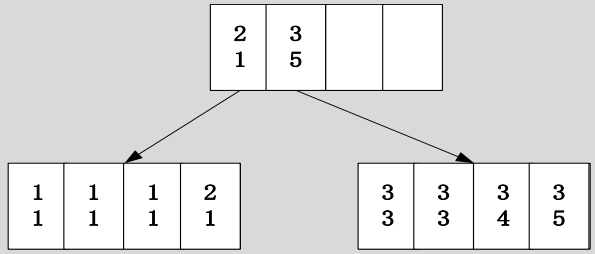
```
Create table t1 (a int primary key, b int, c int, d int) engine = innodb;
create index idx_bc on t1 (b, c);

(1, 1, 1, 1), (2, 1, 1, 2), (3, 2, 1, 3), (4, 3, 3, 4), (5, 3, 3, 5), (7, 3, 4), (8, 3, 5), (9, 1, 1)
```

primary




idx_bc




```
stat_n_rows = 8;

stat_n_diff_key_vals = 8;

stat_clustered_index_size = 3;

rec_per_key[a] = 1;
```

```
stat_n_diff_key_vals[b] = 3;

stat_n_diff_key_vals[b,c] = 5;

rec_per_key[b] = 8/3;

rec_per_key[b,c] = 8/5;
```

SQL举例

```
select * from t1;
select b,c from t1 where b = 1;
select * from t1 where b = 1;
select * from t1 where b = 3 and c = 4;
select * from t1 where c = 3;
```

# Range Query Optimizer – 举例

- Cost计算
  - **Cost = CPU Cost + I/O Cost**
- SQLs
  - select * from t1;
    - 聚簇索引全扫描：**Cost = 8/5 + *1(微调)* + 3 + *1.1(微调)***
  - select b,c from t1 where b = 1;
    - idx_bc (**Index Coverage Scan**)：**Cost = 3/5 + 1(IO Cost)**
  - select * from t1 where b = 1;
    - idx_bc (**Non Coverage Scan**)：**Cost = 3/5 + (1 + 3)**
  - select * from t1 where b = 3 and c = 4;
    - idx_bc (**Non Coverage Scan**)：**Cost = 1/5 + (1 + 1)**
  - select * from t1 where c = 3;
    - 聚簇索引全扫描

# Join Optimizer

- Join Optimizer简介
  - 根据给定的join查询，计算代价最小的查询计划
    - **表的join顺序最优**
    - **每张表的执行路径最优**
  - 递归穷举所有可能的组合与执行路径
  - optimizer_search_depth
    - 控制递归穷举深度
    - optimizer_search_depth >= join tables —> 执行计划全局最优，代价高
    - optimizer_search_depth < join tables —> 执行计划局部最优，代价低
  - **rec_per_key**
    - 根据此参数，计算对于Join内表中的一条记录，外表有多少Join到的记录；

# Optimizer-分析

- **Range Optimizer**
  - IO代价较高，possible_keys越多，随机IO代价越高
  - records_in_range结果不稳定，导致range查询优化的结果不稳定
- **Join Optimizer**
  - CPU代价较高
  - join的tables越多，穷举最优执行计划的代价越高
- **OLTP使用**
  - 更应该关注range查询优化代价，尽量较少possible_keys

# Optimizer – 优化

- **Count 优化**
  - 可优化SQL
    - select count(*) from ...;
    - select count(primary key) from ...;
    - select count(index key) from ...;
  - 优化方案
    - **将全表扫描(聚簇索引全扫描)转换为索引键值最短的可选索引；**
    - *sql_select.cc::make_join_readinfo();*
  - 优化原理
    - 1. 索引键值越短，需要读取的页面越少，IO Cost越小；
    - 2. 二级索引存储Primary Key；
    - 3. Primary Key列，在二级索引不会重复存储；

# Optimizer – 优化

- **ROR 优化**
  - ROR 定义
    - **Rowid Ordered Retrieval**；
    - 若二级索引的部分前导列与聚簇索引的部分前导列完全一致，则说明二级索引回聚簇索引的顺序与聚簇索引键值顺序基本一致，此时二级索引回表的代价较小，可进行优化；
    - *思索：是否有点类似于Oracle的Cluster Factor(聚簇因子)？*
  - ROR 优化
    - 满足ROR条件的二级索引，索引非覆盖扫描的代价，使用新的计算公式；
    - 降低ROR索引的Cost，增加二级索引访问概率；
  - ROR Cost
    - Coverage Scan：与原有一致；
    - Non-Coverage Scan
      - **ROR Cost = Index Coverage Scan Cost + Cluster Index Range Scan Cost**
      - *opt_range.cc::get_best_ror_intersect();*

# Optimizer – 优化

- **In 优化**
  - In查询中所有的取值，均组成一个Range查询；
  - In的值越多 → 范围查询越多
  - **Index Dive vs No Index Dive (MySQL 5.6)**
    - Index Dive
      - 使用records_in_range，估算每个in值范围的rows；
      - 优势：精确
      - 劣势：I/O，慢；
    - No Index Dive
      - 直接使用rec_per_key，估算每个in值的rows；
      - 优势：快，Optimizer代价小；
      - 劣势：不精确；
  - 参数
    - **eq_range_index_dive_limit**

# Optimizer – 优化(续)

- **Semi-join 优化**
  - 将subquery，转换为semi-join，加速执行效率 (MySQL 5.6)
  - 转换条件(部分)
    - IN or = ANY；非NOT IN;
    - Subquery中没有GROUP BY or HAVING;
    - …
- Semi-join算法
  - **FisrtMatch**
  - DuplicateWeedout
  - **Materialization (**Uncorrelated**)**
  - LooseScan

# Optimizer – 优化

- Explain 优化
  - 优化方案
    - 运行Explain时，Optimizer打印所有可选路径的Cost，帮助DBA定位执行计划出错的原因；
    - 类似于Oracle的10053 trace；
    - 详见：Optimizer tracing：Query Execution Plan descriptions beyond EXPLAIN

# 参考资料


1. *MySQL* Internal Details of MySQL Optimizations
2. *何登成* MySQL InnoDB查询优化实现分析
3. *MySQL* MySQL Optimizer Team blogs
4. *Percona* Optimizer Standoff MySQL 5.6 vs MariaDB 5.5
5. *Percona* A case for MariaDB's Hash Joins
6. *IGOR* Notes of an optimizer reviewer
7. *Surajit Chaudhuri* An Overview of Query Optimization in Relational Systems

Q & A

谢谢大家！