# 概述MySQL优化器

## 原理和实践

周振兴/orczhou.com@淘宝 For 2013华东数据库技术大会

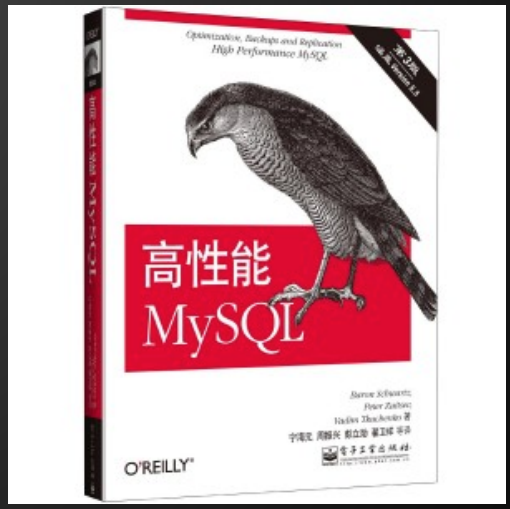
# 关于作者

关于作者：周振兴 / @orczhou / http://orczhoul.com

来自阿里巴巴核心系统数据库开发团队

四年MySQL数据运维管理和性能调优经验

<高性能MySQL>第三版译者



# 目录

- **原理概述**
- Explain Explain
- 更高效的SQL
- 关于子查询

# 原理概述

- 优化器做什么
- MySQL优化器的主要工作
- MySQL案例--优化器如何工作

# 术语和名词

| | | |
|---|---|---|
| Relational | RDBMS | 通常称作 |
| Tuple | Row | 记录/行 |
| Attribute | Column | 列/字段 |
| Restrict | Predicate | WHERE条件/限制/谓词 |
| ... | ... | ... |
| | Row-id/Rowid | 记录的唯一引用/Row-id |
| | ROR(Rowid-ordered Retrieval) | ROR/参考 |

# 什么是优化器?

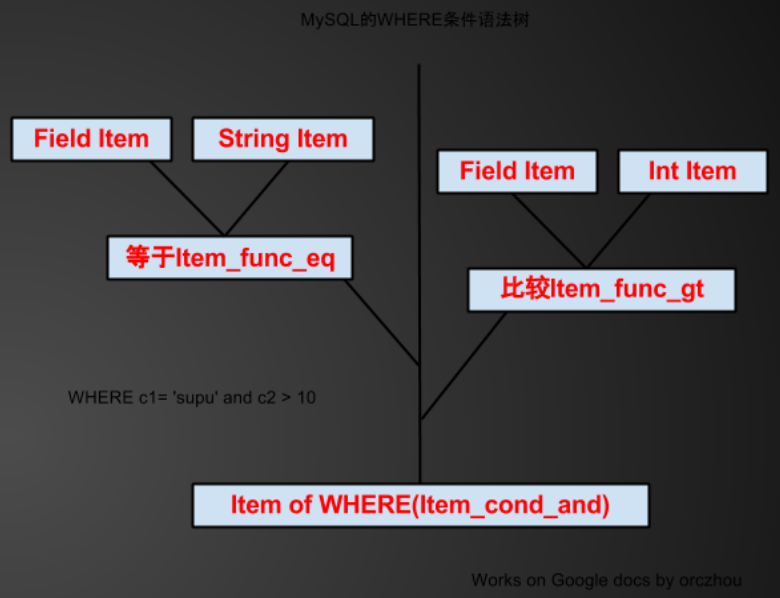
SQL Parse Tree

统计信息

Rule && Cost

优化器

Query Execute Plan (QEP)




Table/Index statistics

顺序/访问方式

interface:read_first/read_next

# Code show

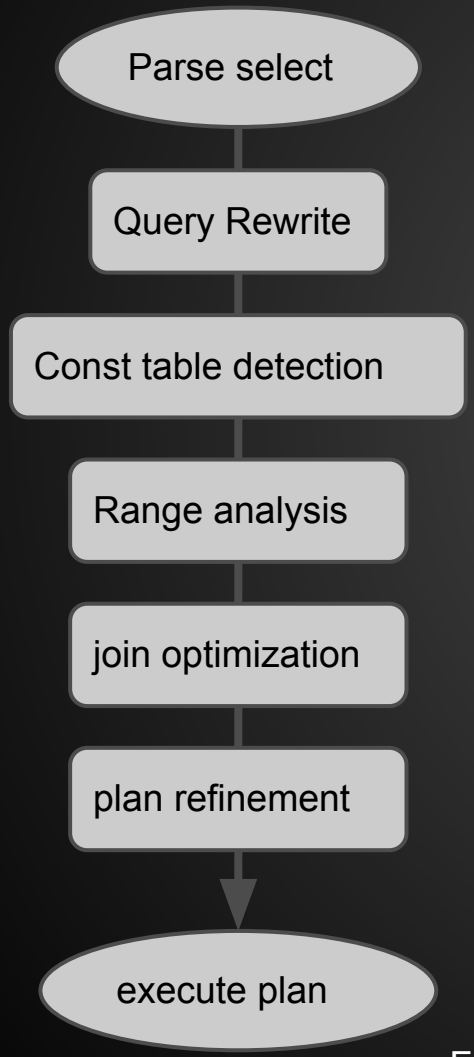
```
new JOIN(Lex);
JOIN::prepare();
JOIN::optimize();
JOIN::exec();
```

顺序和访问方式

if subquery in MySQL 5.1

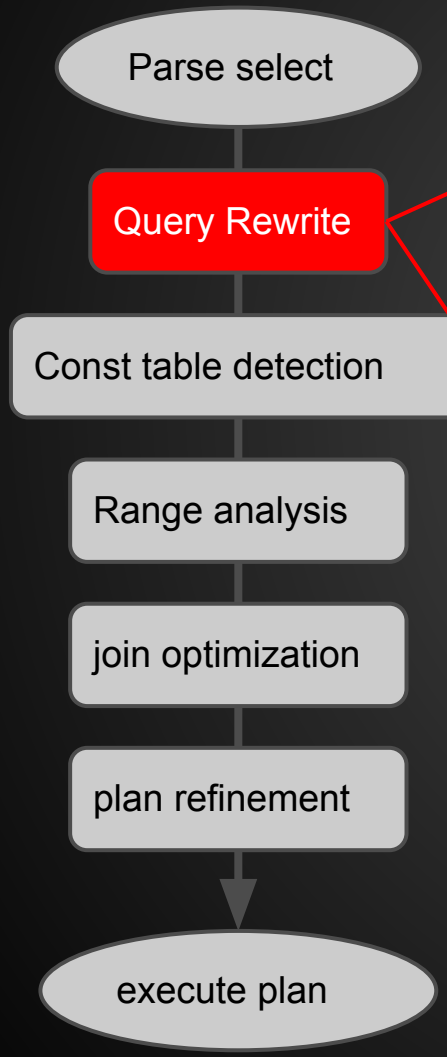
```
new JOIN(Lex);
JOIN::prepare();
JOIN::optimize();
JOIN::exec();
```

# 优化器的工作




From: Sergey.P-2009-Understanding and Control of MySQL Query Optimizer

# 优化器的工作-Rewrite



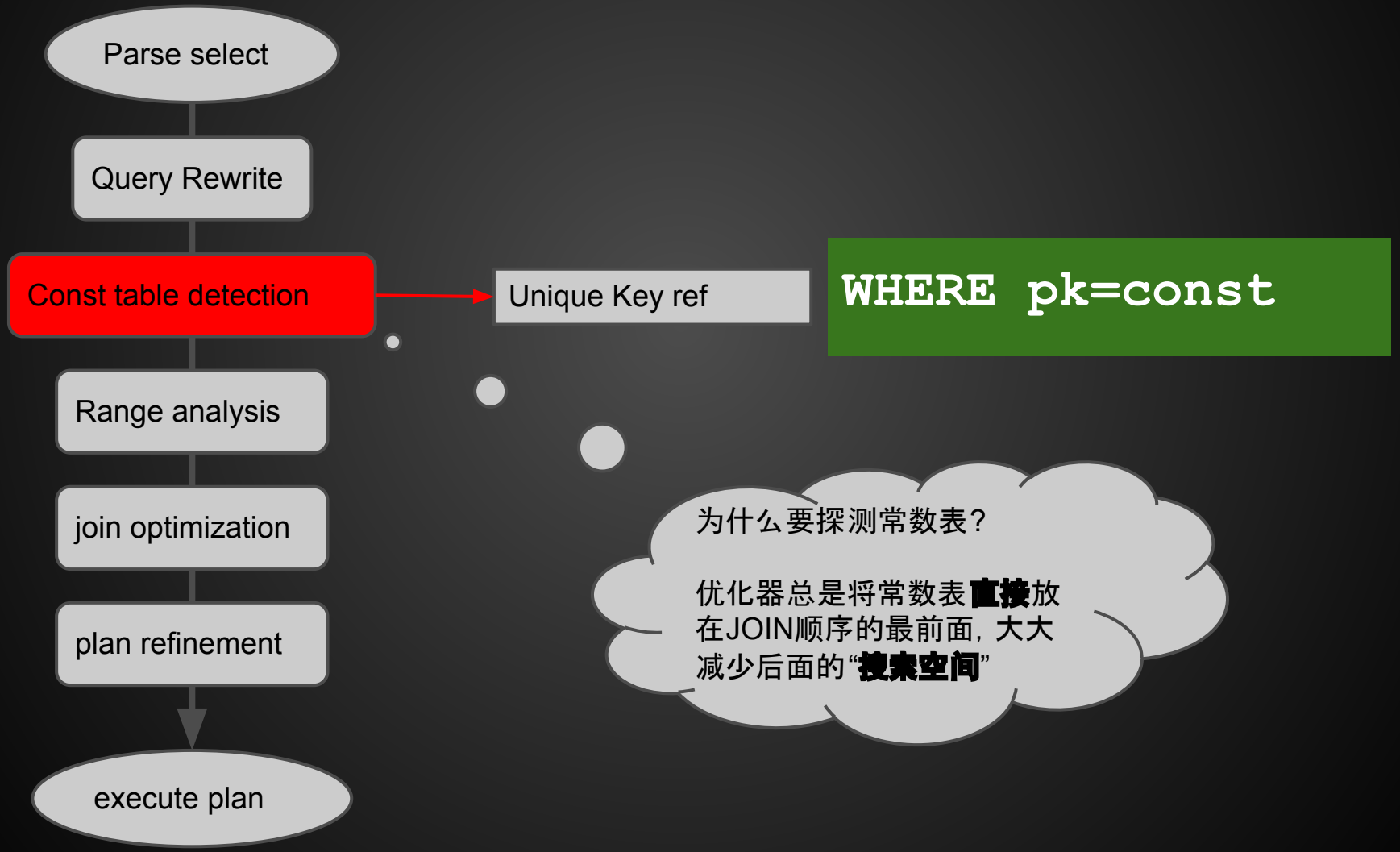

Outer -> Inner JOIN

```
SELECT * FROM
A LEFT JOIN B
WHERE B.col = 5
...(null-rejected)
```

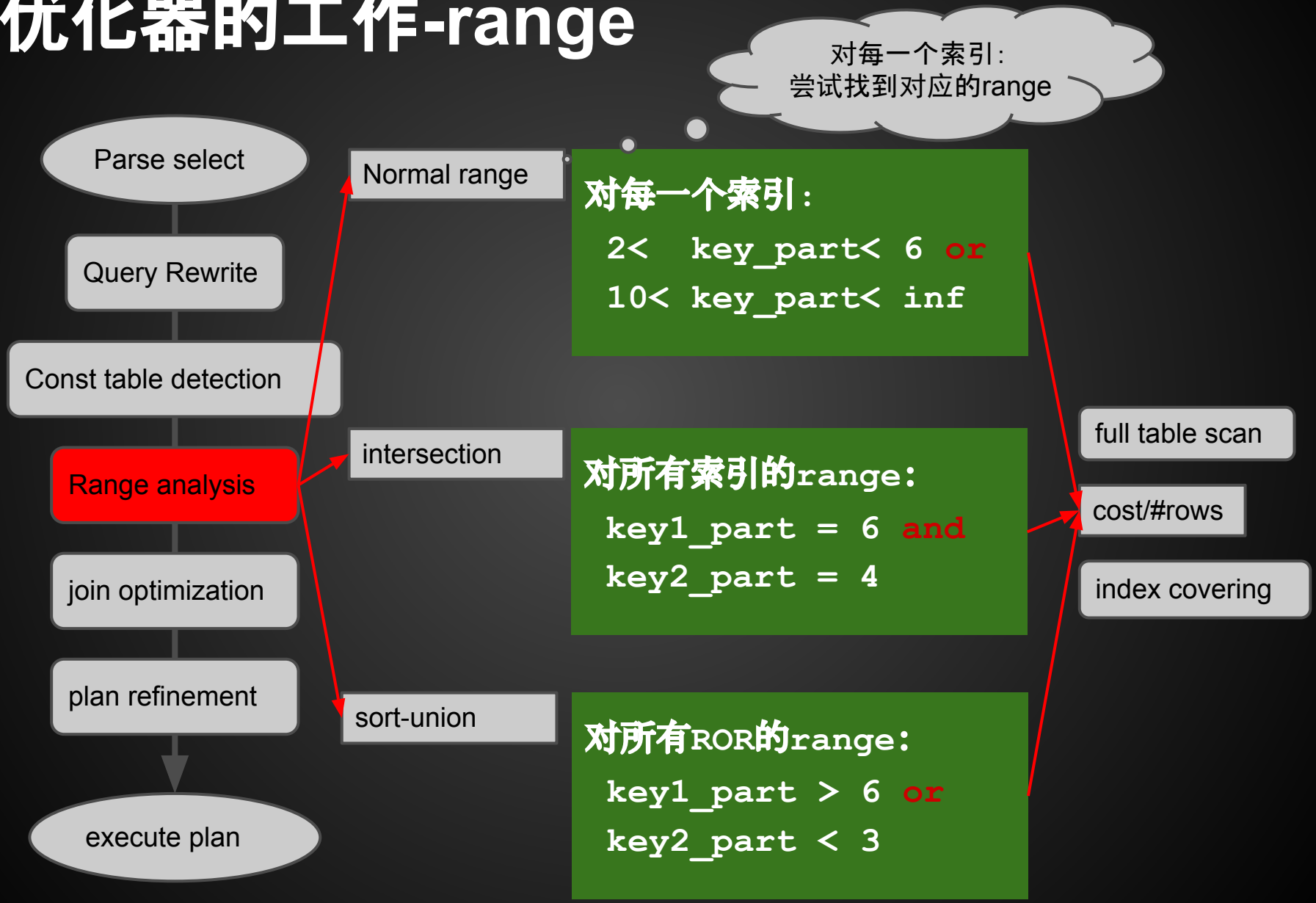
Rewrite subquery

```
IN -> Exists
(push down)
```

# 优化器的工作-const table



# 优化器的工作-range



# Range-analysis

- Normal range

```
WHERE
   2< key1< 6 and
   4< key2< inf
```

cost/#rows

key1: 121 / 100

key2: 178 / 147

- intersection

```
WHERE
   key1 = 6 and
   key2 = 10
```

key1 and key2:
(Both ROR)
254.3 / 68

- sort-union

```
WHERE
   2< key1< 6 or
   4< key2< inf
```

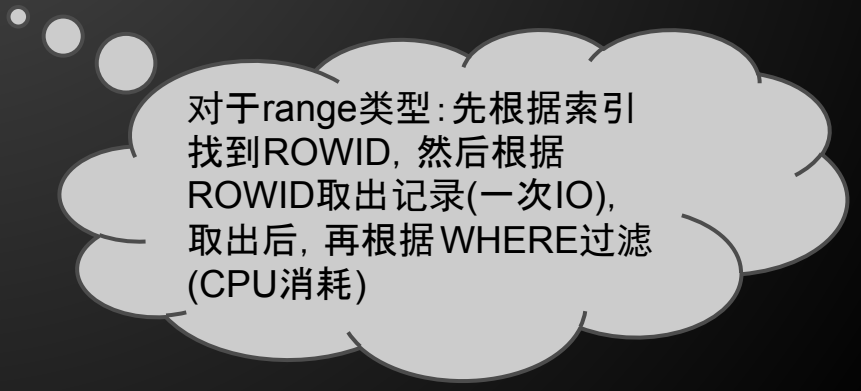
key1 or key2:
354.3 / 188

# Cost计算

总成本 := cpu cost + io cost

MySQL如何读取/处理记录

```
info->read_record(info);          # IO COST
evaluate_join_record(join,...); # CPU COST
```



# Range-analysis：normal range

- Normal range

cost/#rows

```
WHERE
   2< key1< 6 and
   4< key2< inf
```

key1: 141/100

key2: 208 / 147

table scan:150

- cost 计算sample

key1:
- storage预估返回记录数，100
- io cost = 100
- cpu cost = (#rows/5)*2 = 40

# Range-analysis：normal range

- sample

```
explain select * from tmp_range
where key2_part1 > 89 and key2_part1 < 100\G
           id: 1
  select_type: SIMPLE
        table: tmp_range
         type: range
possible_keys: ind2
          key: ind2
      key_len: 4
          ref: NULL
         rows: 25
        Extra: Using where

show status like '%Last_query_cost%';
+-----------------+-----------+
| Last_query_cost | 36.009000 |
+-----------------+-----------+
```

cost: #rows + (#rows/5)*2
25 + 25/5*2 = 35

# Range-analysis：intersection/交集

- intersection range

cost/#rows

```
WHERE
   key1 = 6 and
   key2 = 10
```

key1: 141/100

key2: 208 / 147

table scan:150

- 场景

  - 两个索引range返回rowid都很多
  - 两组rowid交集很少
  - 两个range都是ROR的(即返回的rowid都是已经排序好的)
  - 两个索引能够覆盖

# Range-analysis：intersection range

- sample (参考)

```
explain select count(*) from tmp_index_merge where
(key1_part1 = 4333 and key1_part2 = 1657) and (key3_part1 = 2877)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: tmp_index_merge
         type: index_merge
possible_keys: ind1,ind3
          key: ind3,ind1
      key_len: 4,8
          ref: NULL
         rows: 3622
        Extra: Using intersect(ind3,ind1); Using where; Using index
```

cost: 每个索引读取成本 + ROWID合并成本 + 合并后记录读取成本

# Range-analysis：sort-union/并集

- union range

```
WHERE
   2< key1< 6 or
   4< key2< inf
```

cost/#rows

key1: 141/100

key2: 208 / 147

table scan:150

- 场景
  - 走两个索引的成本都很小

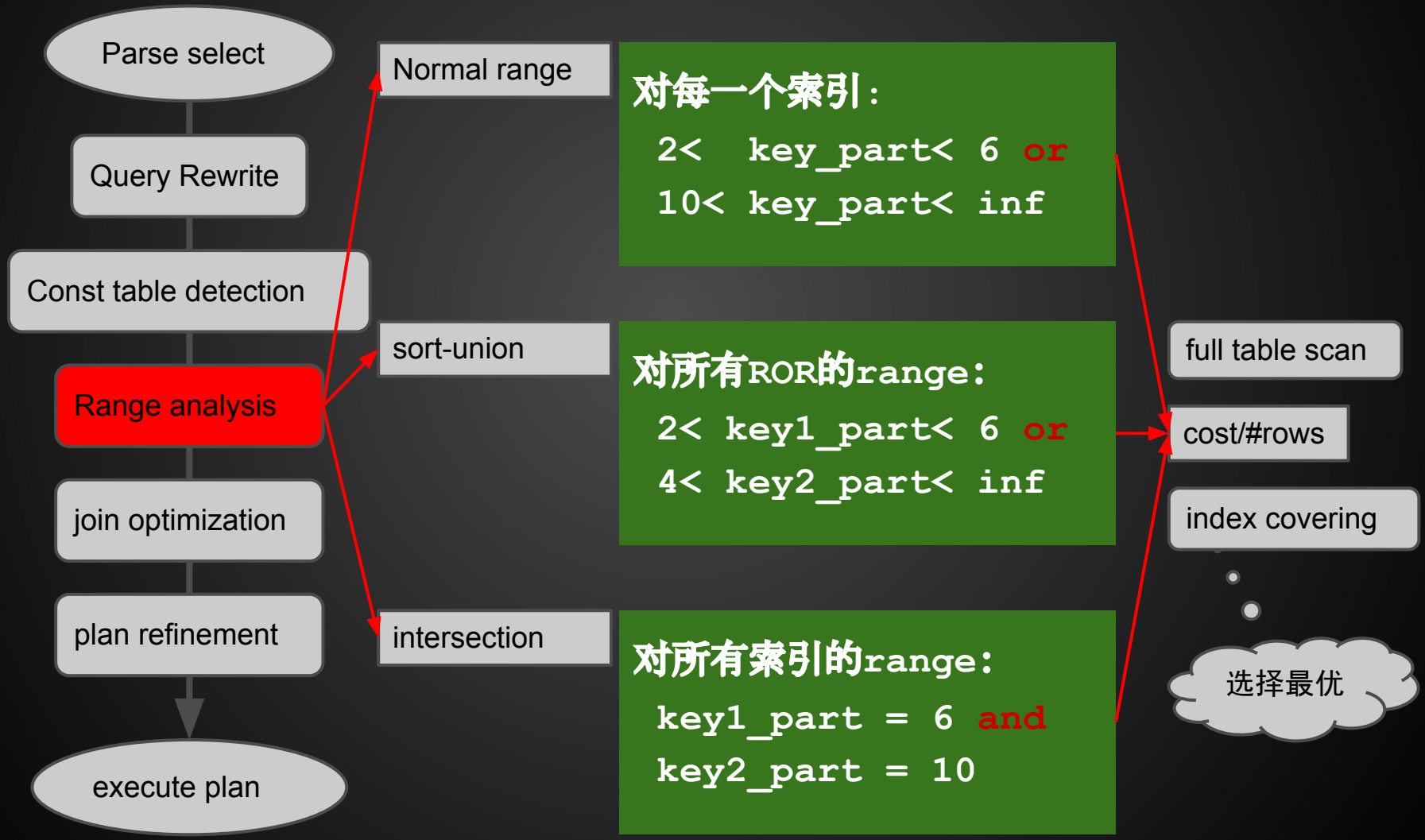
# Range-analysis：sort-union

- sample (参考)

```
explain select * from tmp_index_merge where (key1_part1 = 4333 and
key1_part2 = 1657) or (key3_part1 = 2877)\G
           id: 1
  select_type: SIMPLE
        table: tmp_index_merge
         type: index_merge
possible_keys: ind1,ind3
          key: ind1,ind3
      key_len: 8,4
          ref: NULL
         rows: 2
        Extra: Using union(ind1,ind3); Using where
```

cost: 每个索引读取成本 + ROWID排序成本(非ROR)
+ 合并成本 + 合并后记录读取成本

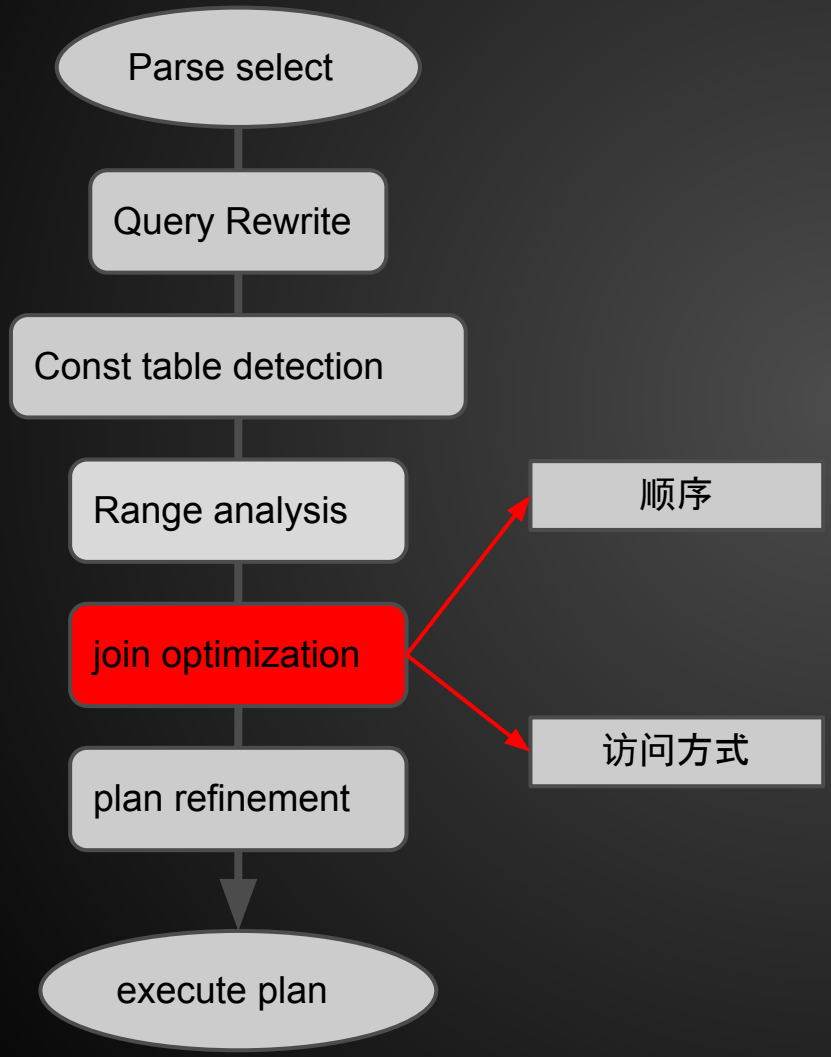
# 优化器的工作-range



# Range-analysis：其他

- range无法直接使用索引统计信息
- MySQL目前没有直方图，只能每次调用存储引擎借口(某些场景会是系统瓶颈)
- 使用存储引擎抽样借口，可以避免数据分布不均匀的问题
- 等值表达式也按照range计算，多个range一次计算

# 优化器的工作-join optimization

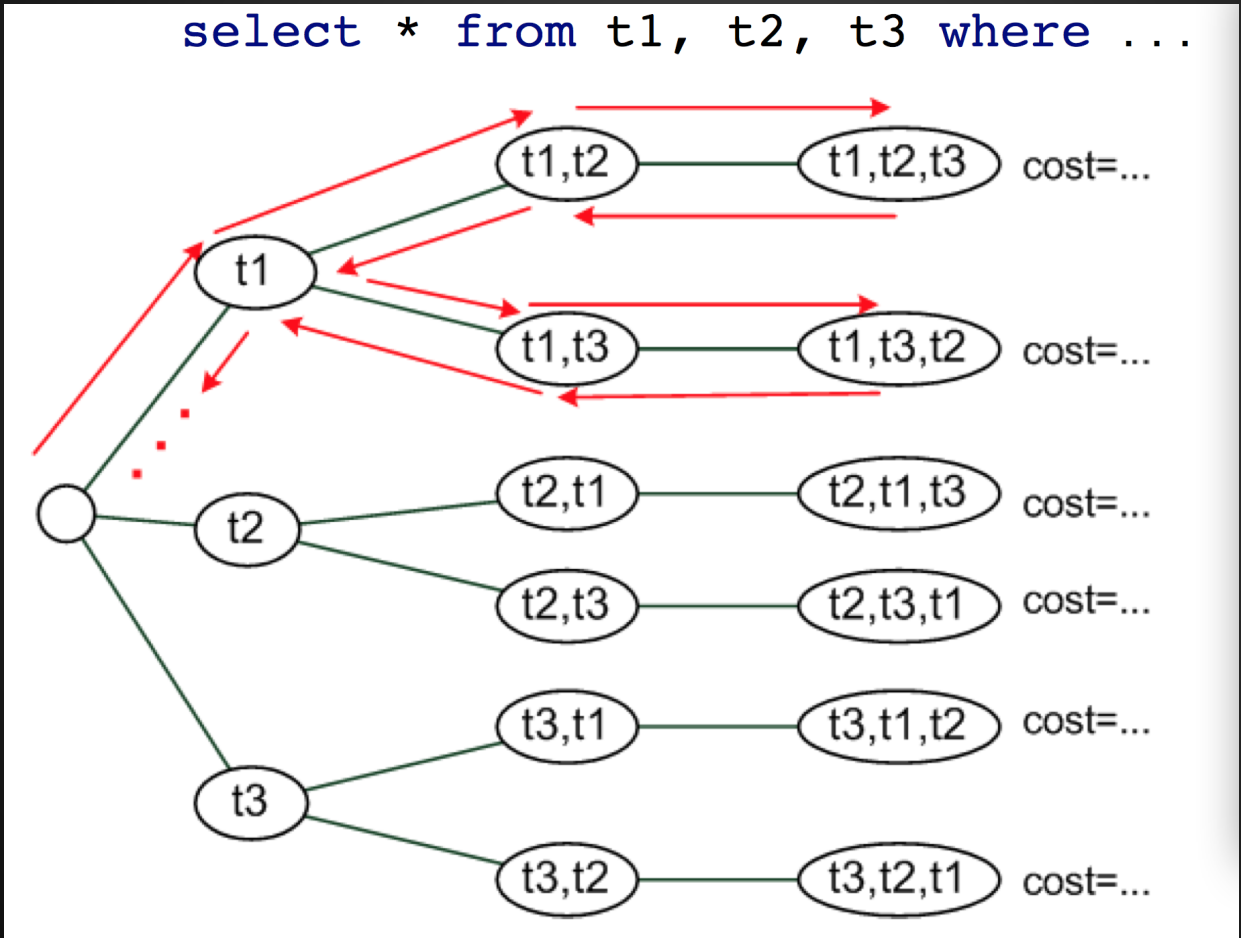



贪婪搜索--蜕化后为穷举搜索

**顺序如何影响成本？**

**访问方式有哪些？**

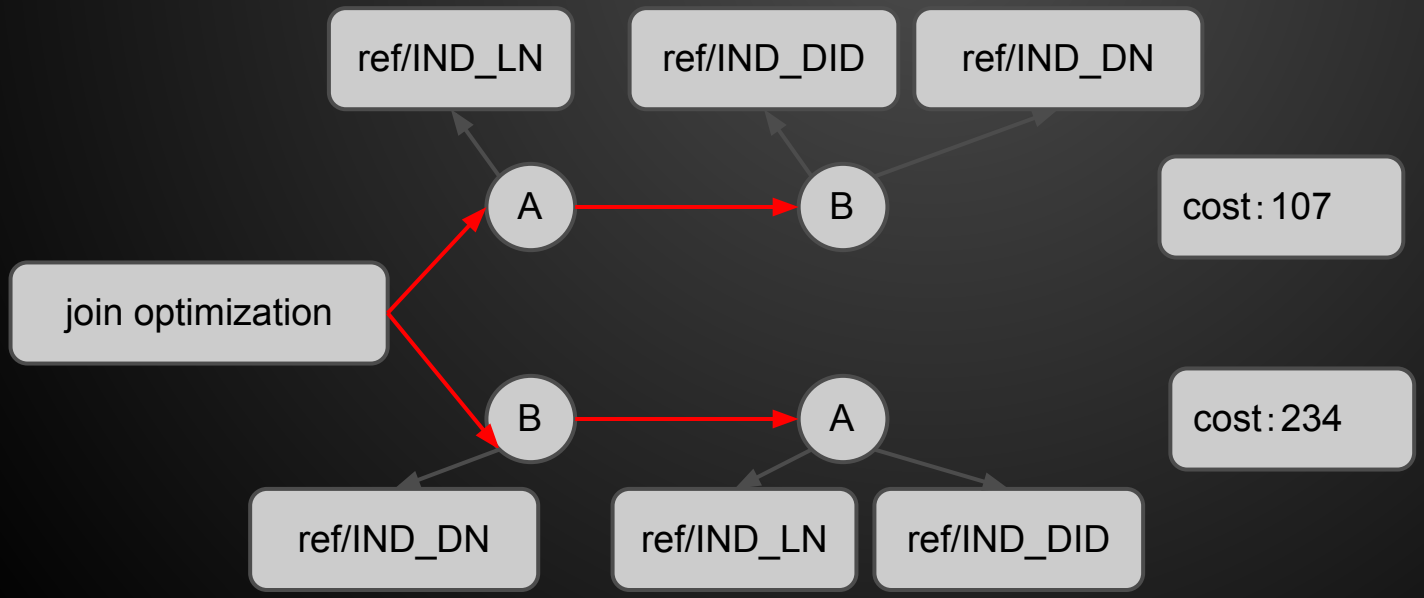
# 优化器的工作-join optimization




图片来源:参考文献2

# 优化器的工作-join optimization案例

```
explain
select *
from
  employee as A,department as B
where
      A.LastName = 'zhou'
  and B.DepartmentID = A.DepartmentID
  and B.DepartmentName = 'TBX';
```



# join optimization：其他

- outer join处理时，总是转换为left join
- JOIN只遍历left-deep tree
- 如果可能，outer join都转换为inner join

# 目录

# Explain explain

- 概述/walkthrough

- 一些注意事项
  - keylen
  - select type
  - Extra

# Explain - walkthrough

```
explain ...

********** 1. row **********

           id: 1

  select_type: SIMPLE/UNION/PRIMARY/SUBQUERY

        table: A

         type: ref/const/eq_ref/ref/...

/range/index/ALL

possible_keys: IND_L_D,IND_DID

          key: IND_L_D

      key_len: 43

          ref: const

         rows: 1

        Extra: Using where    /using index/using
```

select_type: select类型

type: 数据访问方式

rows: 预估需要扫描的记录

Extra:其他信息

# Explain - key_len

```
CREATE TABLE `department` (
  `DepartmentID` int(11) DEFAULT NULL,
  `DepartmentName` varchar(20) DEFAULT NULL,
  KEY `IND_D` (`DepartmentID`),
  KEY `IND_DN` (`DepartmentName`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk;
********** 1. row **********
           id: 1
  select_type: SIMPLE
        table: B
         type: ref
possible_keys: IND_D,IND_DN
          key: IND_D
      key_len: 5
          ref: test.A.DepartmentID
         rows: 1
        Extra: Using where
```

key_len: 5 = INT(4 bytes) + NULL(1)

- NULL 需要额外一个字节
- VARCHAR 变成需要两个字节
- 多字节字符集：gbk*2 utf8*3
- 其他：

| | |
|---|---|
| int | 4 |
| datetime | 8 |
| bigint | 8 |
| CHAR(M) | M*w... |

参考：Data Type Storage Requirements

# Explain - key_len总是取最大的

```
C

REATE TABLE `tmp_keylen` (
  `id` int(11) NOT NULL,
  `nick` char(10) DEFAULT NULL,
  `address` char(20) DEFAULT NULL,
  `color` char(10) NOT NULL,
 KEY `ind_t` (`id`,`nick`,`address`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1

explain select * from tmp_keylen
where
      id >= 1
  and nick = 'zx'\G
********** 1.row **********
        table: tmp_keylen
         type: range
          key: ind_t
      key_len: 15
        Extra: Using where
```

```
id >=1 and nick = 'zx'
```

对MySQL来说，这是两个Range：

```
id > 1

id = 1 and nick = 'zx'
```

对应的key_len分别是 4和15

总是取最大的，所以， key_len 是15

参考：Data Type Storage Requirements

# Explain - key_len看出执行计划正确性

```
explain select *
from
  tmp_users
where
      uid     = 9527
  and l_date >= '2012-12-10 10:13:17'\G
*************** 1. row ********************
           id: 1
  select_type: SIMPLE
        table: tmp_users
         type: ref
possible_keys: ind_uidldate
          key: ind_uidldate
      key_len: 4
          ref: const
         rows: 418
        Extra: Using where
```

```
CREATE TABLE `tmp_users` (
  `id` int(11) NOT NULL
AUTO_INCREMENT,
  `uid` int(11) NOT NULL,
  `l_date` datetime NOT NULL,
  `data` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `ind_uidldate` (`uid`,`l_date`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk;
```

解决:使用force index

Bug#12113

# Explain - type

**type：JOIN过程中，单表访问方式**

| | | |
|---|---|---|
| const | 只有一条记录，如唯一索引的常数引用 | WHERE primary_key=1; |
| ref/eq_ref/ref_or_null | 引用 / 唯一索引引用 / (or null) | key = 1 / A.un_key = B.col3 |
| range / index_merge | 索引范围扫描 / 多个索引交集、并集 | key > 10 |
| index | 全索引扫描 | 若有using index，则 索引覆盖扫描<br>否则，是按索引顺序扫描，再回表 |
| ALL | 全表扫描 | |
| unique_subquery /<br>index_subquery | IN子查询，改写成EXISTS后，<br>使用唯一/索引做first match扫描 | |
| full-text | 使用全文索引 | |
| system | MyISAM表/且单表只有一条记录 | |

# Explain - type - index_merge

使用多个索引访问数据

```
SELECT *
FROM
  tmp_index_merge
WHERE
     key1_part1 = 2
  or key2_part1 = 4
```

同时使用key1和key2获取rowid，然后merge后回表查询；一般满足：

- 两个索引访问成本都低
- 合并后成本也低于全表扫描

```
select key1_part2,key3_part1
from
  tmp_index_merge
where
     (  key1_part1 = 4333 and
        key1_part2 = 1657
     )
  and key3_part1 = 2877\G
```

同时使用key1和key2获取数据，然后merge后获得结果；需要满足：

- 所有索引访问都是ROR的
- 多个索引合并后需要是覆盖

# Explain - type - index

**全索引扫描**

```
          type: index
possible_keys: NULL
           key: ind_uidldate
       key_len: 12
           ref: NULL
          rows: 824
         Extra: Using index
explain select uid from tmp_users force index
(ind_uidldate)\G
```

如果Extra有using index，表示这是一个索引覆盖扫描，无需回表；

```
          type: index
possible_keys: NULL
           key: ind_uidldate
       key_len: 12
           ref: NULL
          rows: 824
         Extra: Using where
explain select data,uid from tmp_users force index
(ind_uidldate) where data = 3 order by uid\G
```

否则，这是一个按照索引顺序的全表扫描，仍然需要回表

# Explain - type - index_subquery

**子查询**

```
explain
select * from tmp_t1
where
  id in
     (
     select id from tmp_t2
     where age = 3
     );
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: tmp_t2
         type: index_subquery
possible_keys: ind_id,ind_age
          key: ind_id
      key_len: 5
          ref: func
         rows: 1
        Extra: Using where
```

index_subquery会使用first match原则，性能较好

子查询需要满足如下条件才会使用index_subquery

- 子查询格式：left_exp in (Subquery)
- 优化阶段发现子查询恰好使用REF/EQ_REF
- select list对应的列也恰好是ref的字段

本案例中都满足：

- id in (...) 格式
- 子查询使用索引ind_id(id)，为ref
- select list中的列id，正是ref索引的列

unique_subquery类似于此，只是使用的唯一索引

# Explain - Extra - using where/index

Using where 需要过滤

```
          type: index
possible_keys: NULL
           key: ind_uidldate
       key_len: 12
           ref: NULL
          rows: 824
         Extra: Using where;using index;
```

explain select uid from tmp_users force index (ind_uidldate) where uid = 3 order by uid\G

可能回表过滤;

也可能是索引过滤;(using index)

Using index 覆盖扫描

Using filesort 索引不排序，需要再做排序

# 高效SQL-案例

- 原理概述
- Explain Explain
- **更高效的SQL**
- 关于子查询

# 高效SQL-案例

- 案例一-- 覆盖索引
- 案例二-- deferred JOIN
- 案例三-- 子查询

# 高效SQL-覆盖索引

- 优点：
  - 索引通常更小，所以只需更少IO，对缓存和IO都有好处
  - 通常都且这是顺序IO
  - 对InnoDB来说，覆盖索引则少了一次(主键)索引扫描

- deferred join和覆盖索引
  - 可以用在分页

# 高效SQL-覆盖索引-性能对比

- 索引覆盖扫描 vs 回表

```
select
  count(*)
from
  t1
     force index(IND_DATAIDGMT)
where
  is_main = 1 and
  (
    the_type=100 or
    the_type=200 or
    the_type=300 or
    the_type=500
   ) and
  data_id=631389273 and
  status=0 and
  gmt_modified >= '2012-03-28' and
  gmt_modified <= '2012-03-29';
```

原索引:

```
KEY `IND_DATAIDGMT` (
  `DATA_ID`,
  `GMT_MODIFIED`
);
```

新索引:

```
KEY`IND_DATAIDGMT` (
  `DATA_ID`,
  `GMT_MODIFIED`,
  `the_type`,
  `status`,
  `is_main`);
```

# 高效SQL-覆盖索引-性能对比

- 两个索引逻辑读对比

相同的SQL，当覆盖索引和非覆盖索引时候的性能（以逻辑读评估）：

| No. | Not Cover | Cover all |
|---|---|---|
| 1 | 578186 | 14529 |
| 2 | 444781 | 14529 |
| 3 | 362091 | 14529 |
| 4 | 358816 | 14529 |
| 5 | 358816 | 14529 |
| 6 | 358816 | 14529 |
| 7 | 358816 | 14529 |
| 8 | 358816 | 14529 |
| 9 | 358823 | 14529 |
| 10 | 358816 | 14529 |
| AVG | 389678 | 14529 |

- 性能相差约25倍 389678/14529

# 高效SQL-覆盖索引-deferred join

- 有时候没法直接使用覆盖索引

```
select
  ID,SHOW_TITLE,NICK_DATE,ZM,...,WTF
from
  t1
where
  is_main = 1 and
  (
    the_type=100 or
    the_type=200 or
    the_type=300 or
    the_type=500
   ) and
  data_id=631389273 and
  status=0 and
  gmt_modified >= '2012-03-28' and
  gmt_modified <= '2012-03-29';
```

新索引:

```
KEY`IND_DATAIDGMT` (
  `DATA_ID`,
  `GMT_MODIFIED`,
  `the_type`,
  `status`,
  `is_main`);
```

# 高效SQL-deferred join

- 延迟读取一些列

```
SELECT
  A.ID,A.SHOW_TITLE,A.NICK_DATE,ZM,...,A.
WTF
FROM
  tmp_deferred AS A,
  (select
    ID
  from
    tmp_deferred
  where
    is_main = 1 and
    biz_type = 100 and
    status=0 and
    gmt_modified >= '2011-12-28' and
    gmt_modified <= '2012-03-29';
  ) B
WHERE A.ID,B.ID;
```

新索引:

```
KEY`IND_DATAIDGMT` (
  `DATA_ID`,
  `GMT_MODIFIED`,
  `the_type`,
  `status`,
  `is_main`);
```

# 高效SQL-deferred join-性能对比

- 普通和deferred逻辑读对比

```
改写成deferred SQL,性能对比(以逻辑读评估):
|No.  | Not deferred | deferred join |
|1    | 30468        | 1528          |
|2    | 30482        | 1521          |
|3    | 30468        | 1528          |
|4    | 30468        | 1528          |
|5    | 30482        | 1521          |
```

- 非索引WHERE过滤性越强，性能提升越大
  - **本案例过滤后，剩余2%数据**

- 性能相差约20倍 30468/1528

# 目录

# 子查询-subquery

- 关于子查询的一些事实
  - **5.6.5以后子查询将尽可能转换成Semi-join来执行**
  - **5.1 子查询的执行"总是从外到内"** `Outer table -> inner table`
  - **5.1 子查询转换成JOIN速度可能会变慢**

# 子查询-subquery-VS-JOIN

- 考察如下SQL

```
Subquery
SELECT * from tmp_t1
where
  id in (
     select id from tmp_t2 where age = 1
  );


JOIN
SELECT A.* FROM tmp_t1 as A,tmp_t2 as B
WHERE A.id = B.id and B.age = 1;
```

# 子查询-subquery-VS-JOIN

- 对比
  - 改写成JOIN后，A.id可能匹配多条记录
  - 改写成JOIN后，语义发生了变化
  - JOIN不可能使用first Match原则
- 性能
  - 如果inner -> outer更合适，则JOIN更好
  - 如果firstMatch很有用，则subquery更好
  - 如果程序不希望做去重，那么只能subquery

# 子查询-subquery-VS-JOIN

- 不用场景下的性能

```
Subquery
SELECT * from tmp_t1 where  id in ( select id from tmp_t2 where age = 1
);
JOIN
SELECT A.* FROM tmp_t1 as A,tmp_t2 as B  WHERE A.id = B.id and B.age = 1;
```

```
for i in `seq 1 1` ; do  mysql -uroot test -e 'insert into tmp_t2 values
(1,1,2012)'; done;

for i in `seq 1 1000` ; do  mysql -uroot test -e "insert into tmp_t1
values(60000*rand(),char(round(ord('A') + rand()*(ord('z')-ord('A')))))";
done;
```

逻辑读:

| No. | subquery | join |
|---|---|---|
| 1 | 3676 | 11 |

- JOIN更快，约300倍

# 子查询-subquery-VS-JOIN

- 不用场景下的性能

```
Subquery
SELECT * from tmp_t1 where  id in ( select id from tmp_t2 where age = 1
);
JOIN
SELECT A.* FROM tmp_t1 as A,tmp_t2 as B  WHERE A.id = B.id and B.age = 1;
```

```
for i in `seq 1 1000` ; do  mysql -uroot test -e 'insert into tmp_t2
values(1,1,2012)'; done;

for i in `seq 1 10` ; do  mysql -uroot test -e "insert into tmp_t1 values
(5*rand(),char(round(ord('A') + rand()*(ord('z')-ord('A')))))"; done;
```

逻辑读:

| No. | subquery | join |
|---|---|---|
| 1 | 42 | 9060 |

- subquery更快，约215倍

# 子查询-subquery-VS-JOIN@5.6

- 不用场景下的性能

```
Subquery
SELECT * from tmp_t1 where  id in ( select id from tmp_t2 where age = 1
);
JOIN
SELECT A.* FROM tmp_t1 as A,tmp_t2 as B  WHERE A.id = B.id and B.age = 1;
```

| 逻辑读 | 场景1 | 场景2 |
| --- | --- | --- |
| 子查询 | 11 | 38 |
| JOIN | 11 | 3039 |

- **5.6以后子查询性能总是很好**

# 参考文献


1. MySQL Manual/Documentation
2. Understanding and control of MySQL Query Optimizer
   By Sergey. P.
3. New subquery optimizations in MySQL 6.0
   By Sergey. P
4. The MariaDB/MySQL Query Executor In-depth
   By Timour Katchaounov
5. MySQL optimizer overview
   By Olav sandsta MySQL/Oracle
6. Improving MySQL/MariaDB query performance through optimizer tuning
   By Sergey Petrunya Timour Katchaounov
7. MySQL source code

# Q & A

# 附录一

```
CREATE TABLE `tmp_index_merge` (
  `id` int(11) NOT NULL,
  `key1_part1` int(11) NOT NULL,
  `key1_part2` int(11) NOT NULL,
  `key2_part1` int(11) NOT NULL,
  `key2_part2` int(11) NOT NULL,
  `key2_part3` int(11) NOT NULL,
  `key3_part1` int(11) NOT NULL DEFAULT '4',
  PRIMARY KEY (`id`),
  KEY `ind2` (`key2_part1`,`key2_part2`,`key2_part3`),
  KEY `ind1` (`key1_part1`,`key1_part2`,`id`),
  KEY `ind3` (`key3_part1`,`id`)
) ENGINE=InnoDB;
```

# 附录二

```
CREATE TABLE `tmp_t1` (
  `id` int(11) DEFAULT NULL,
  `nick` varchar(32) DEFAULT NULL,
  KEY `ind_id` (`id`),
  KEY `ind_nick` (`nick`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk

CREATE TABLE `tmp_t2` (
  `id` int(11) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `year` int(11) DEFAULT NULL,
  KEY `ind_id` (`id`),
  KEY `ind_age` (`age`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk
```

# 友情提示

- **2013年11月16日于上海举办华东架构师大会**
- 已确定的架构师大会主题及演讲嘉宾
- 去哪儿 唐娟 海量数据的搜集和实时分析架构设计与实践
- 江游科技 时继江 网络游戏一键开服的架构设计与实践
- 金山网络 毛剑 异构数据库的实时数据同步架构设计

会议网址：http://atcc.mysqlops.com/ 新浪微博：@mysqlops

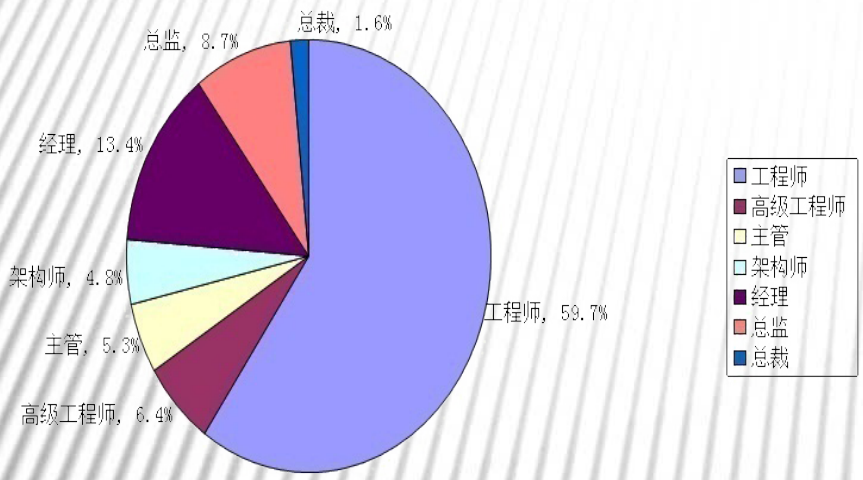
# 联系方式

- 联　系 人(一):会议组织者
- 姓　　名:金　官　丁
- 联系电话:136 6166 8096
- 邮箱地址:mysqlops@sina.com
- 即时通信:172010148(QQ), @mysqlops(新浪微博)

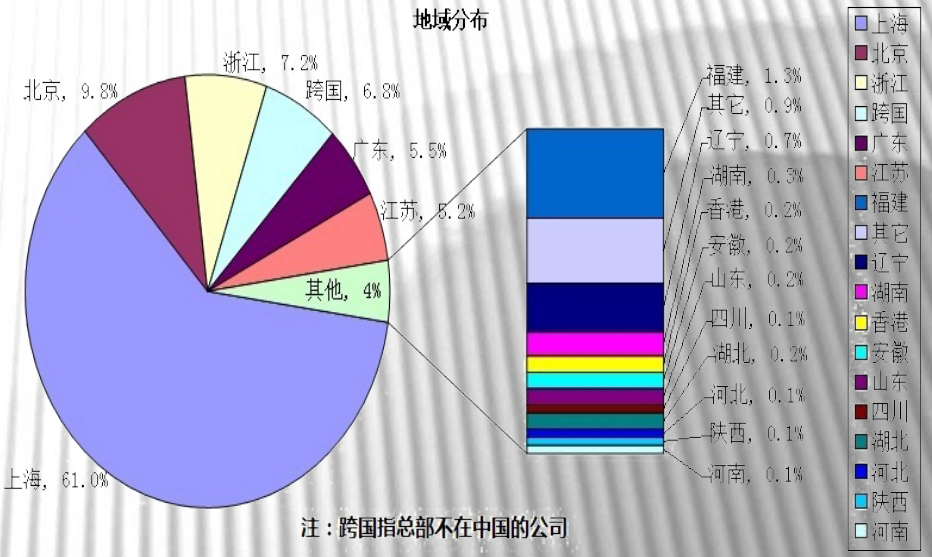
- 联　系 人(二):会议秘书
- 姓　　名:朱　颖　丹
- 联系电话:136 5197 9898
- 邮箱地址:vera_zhuyd@163.com
- 即时通信: 378091820 (QQ), @戇小囡猪猪(新浪微博)

# 2013华东数据库技术大会

职位级别分布

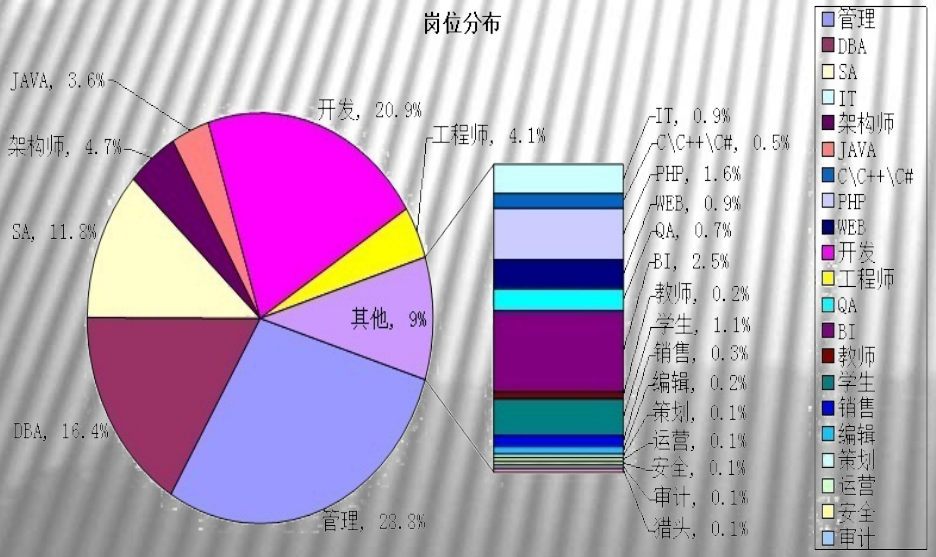



地域分布

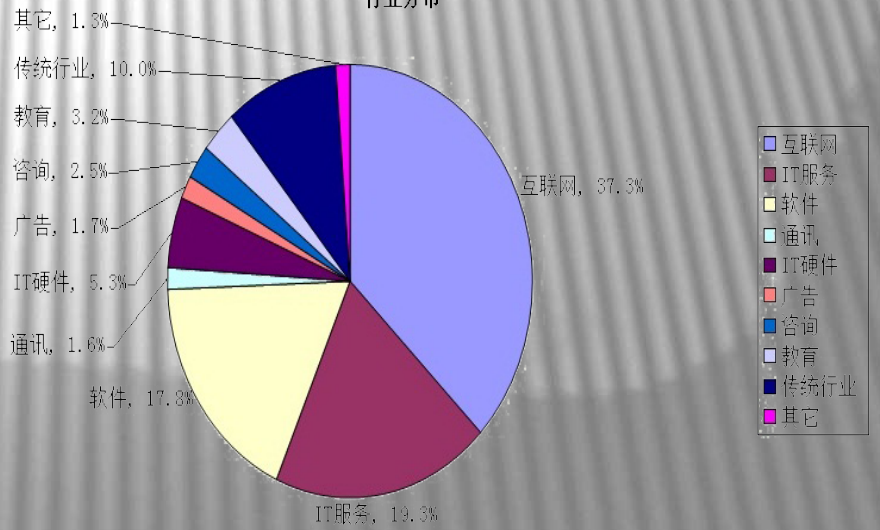



注：跨国指总部不在中国的公司

岗位分布




行业分布



# 2013华东数据库技术大会

主办单位：



协办单位：





捐助单位：




媒体支持：



# 2013华东数据库技术大会

# 2013华东数据库技术大会

# 2013华东数据库技术大会

感谢您的一路相随，我们一起携手走向未来！