第十六讲：联表查询 Join 优化器的设计与实现

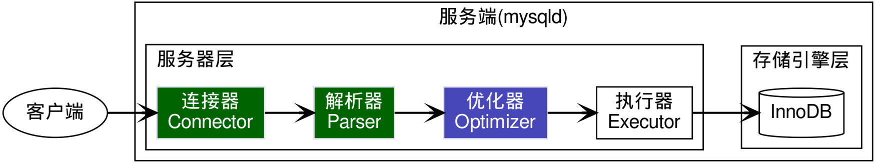
知春路遇上八里桥

<2024-06-29 Sat>

1

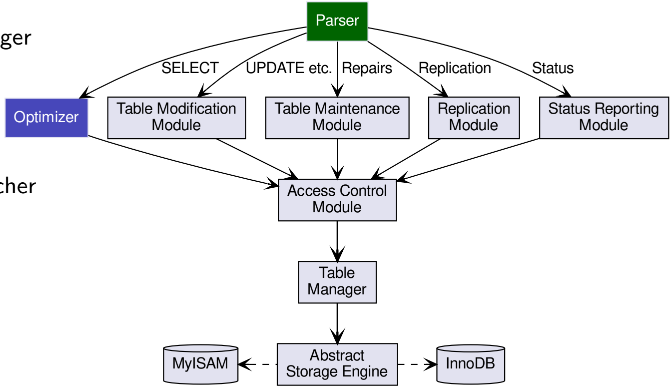
前情提要

执行流程


服务端(mysqld)
服务器层
存储引擎层
客户端
连接器
Connector
解析器
Parser
优化器
Optimizer
执行器
Executor
InnoDB

本节内容

- 连接器
  - ☑ 连接管理器 Connection Manager
  - ☑ 线程管理器 Thread Manager
  - ☑ 用户模块 User Module
- 解析器
  - ☑ 网络模块 Net Module
  - ☑ 派发模块 Commander Dispatcher
  - ☑ 词法分析 Lexical Analysis
  - ☑ 语法分析 Syntax Analysis
- 优化器
  - ☑ 准备模块 Prepare Module
  - ☑ 追踪日志 Optimizer Trace
  - ☐ 优化模块 Optimize Module



2

Join

Join 介绍

- 为了方便编写查询语句, MySQL 提供若干类型的 Join 语句 [1]
- `inner join` 返回左表和右表都包含的行，这里左表和右表的次序可以交换
- `left join` 和 `right join` 表的顺序不可以交换，其中
  - `left join` 返回数据中左表必选，右表可选
  - `right join` 返回数据中右表必选，左表可选
- `straight join` 的逻辑和 `join` 一样, 只不过左表总是在右边之前读取
  - `straight join` 提供了控制执行器读取表先后顺序
  - 它的实现参考 `Optimize_table_order::optimize_straight_join()`
- 一些 Join 的例子

```
select * from t1, t2;
select * from t1 inner join t2 on t1.id = t2.id;
select * from t1 left join t2 on t1.id = t2.id left join t3 on t2.id = t3.id;
select * from t1 left join t2 on t1.id = t2.id;
select * from t1 left join t2 using (id);
```

[1] https://dev.mysql.com/doc/refman/8.0/en/join.html

查询结果实例 (壹)

表 t1 和 t2 中的数据

```
mysql> select * from t1;
+------+------+
| id   | c    |
+------+------+
|    1 | aaa  |
|    2 | bbb  |
+------+------+

mysql> select * from t2;
+------+------+
| id   | c    |
+------+------+
|    1 | xxx  |
|    3 | yyy  |
+------+------+
```

- join 连接

```
mysql> select * from t1 join t2;
+------+------+------+------+
| id   | c    | id   | c    |
+------+------+------+------+
|    2 | bbb  |    1 | xxx  |
|    1 | aaa  |    1 | xxx  |
|    2 | bbb  |    3 | yyy  |
|    1 | aaa  |    3 | yyy  |
+------+------+------+------+
```

- 笛卡尔积

```
mysql> select * from t1, t2;
+------+------+------+------+
| id   | c    | id   | c    |
+------+------+------+------+
|    2 | bbb  |    1 | xxx  |
|    1 | aaa  |    1 | xxx  |
|    2 | bbb  |    3 | yyy  |
|    1 | aaa  |    3 | yyy  |
+------+------+------+------+
```

- straight join 直接连接

```
mysql> select * from t1 straight_join t2;
+------+------+------+------+
| id   | c    | id   | c    |
+------+------+------+------+
|    2 | bbb  |    1 | xxx  |
|    1 | aaa  |    1 | xxx  |
|    2 | bbb  |    3 | yyy  |
|    1 | aaa  |    3 | yyy  |
+------+------+------+------+
```

- inner join 内连接

```
mysql> select * from t1 inner join t2;
+------+------+------+------+
| id   | c    | id   | c    |
+------+------+------+------+
|    2 | bbb  |    1 | xxx  |
|    1 | aaa  |    1 | xxx  |
|    2 | bbb  |    3 | yyy  |
|    1 | aaa  |    3 | yyy  |
+------+------+------+------+
```

查询结果实例 (贰)

表 t1 和 t2 中的数据

```
mysql> select * from t1;
+------+------+
| id   | c    |
+------+------+
|    1 | aaa  |
|    2 | bbb  |
+------+------+

mysql> select * from t2;
+------+------+
| id   | c    |
+------+------+
|    1 | xxx  |
|    3 | yyy  |
+------+------+
```

- 内连接

```
mysql> select * from t1 inner join t2 on t1.id = t2.id;
+------+------+------+------+
| id   | c    | id   | c    |
+------+------+------+------+
|    1 | aaa  |    1 | xxx  |
+------+------+------+------+
```

- 左连接

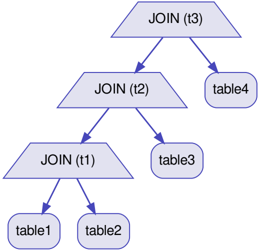
```
mysql> select * from t1 left join t2 on t1.id = t2.id;
+------+------+------+------+
| id   | c    | id   | c    |
+------+------+------+------+
|    1 | aaa  |    1 | xxx  |
|    2 | bbb  | NULL | NULL |
+------+------+------+------+
```

- 右连接

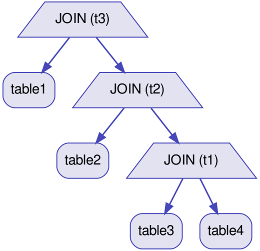
```
mysql> select * from t1 right join t2 on t1.id = t2.id;
+------+------+------+------+
| id   | c    | id   | c    |
+------+------+------+------+
|    1 | aaa  |    1 | xxx  |
| NULL | NULL |    3 | yyy  |
+------+------+------+------+
```

连接树

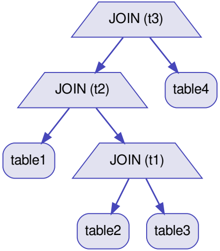
- 多个表进行连接时，执行过程可以表示成树形结构，即连接树 (Join Tree)
  - 左深树 (Left Deep Join Tree) 的每个连接的右节点都是一个表
  - 右深树 (Right Deep Join Tree) 的每个连接的左节点都是一个表
  - 浓密树 (Bushy Join Tree) 的左节点或右节点都有可能不是表



左深树



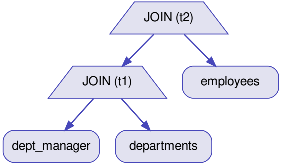
右深树



浓密树

多表 Join 的左深树结构

- n 个表会有排列数 [2] A_n^n 种做 Join 情况. 对 a, b, c 三个表, $A_3^3 = 6$ 种情况
  - (a, b, c) / (a, c, b) / (b, a, c)
  - (b, c, a) / (c, a, b) / (c, b, a)
- MySQL 多表 Join 的查询语句采用数组来表示左深树结构




```sql
select
  e.first_name, e.last_name, a.from_date
from
  dept_manager a
  join departments d on a.dept_no = d.dept_no
  join employees e on a.emp_no = e.emp_no
where
  d.dept_no = 'd001'
```

- 最终优化后的 Join 顺序为 d/a/e ，并非输入顺序 a/d/e

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	d	NULL	const	PRIMARY	PRIMARY	16	const	1	100.00	Using index
1	SIMPLE	a	NULL	ref	PRIMARY,dept_no	dept_no	16	const	2	100.00	Using index condition
1	SIMPLE	e	NULL	eq_ref	PRIMARY	PRIMARY	4	employees.a.emp_no	1	100.00	NULL

[2]排列数 A_m^n 表示从 n 个不同的元素中任取 m ($m \leq n$) 个所有排列的个数. $A_m^n = n!/(n-m)!$

Json 格式详细执行计划


```
{
  "query_block": {
  "select_id": 1,
  "cost_info": {
    "query_cost": "2.90"
  },
  "nested_loop": [
    {
      "table": {
        "table_name": "d",
        "access_type": "const",
        "possible_keys": [
          "PRIMARY"
        ],
        "key": "PRIMARY",
        "used_key_parts": [
          "dept_no"
        ],
        "key_length": "16",
        "ref": [
          "const"
        ],
        "rows_examined_per_scan": 1,
        "rows_produced_per_join": 1,
        "filtered": "100.00",
        "using_index": true,
        "cost_info": {
          "read_cost": "0.00",
          "eval_cost": "0.10",
          "prefix_cost": "0.00",
          "data_read_per_join": "184"
        },
        "used_columns": [
          "dept_no"
        ]
      }
    },
```

```
{
  "table": {
    "table_name": "a",
    "access_type": "ref",
    "possible_keys": [
      "PRIMARY",
      "dept_no"
    ],
    "key": "dept_no",
    "used_key_parts": [
      "dept_no"
    ],
    "key_length": "16",
    "ref": [
      "const"
    ],
    "rows_examined_per_scan": 2,
    "rows_produced_per_join": 2,
    "filtered": "100.00",
    "index_condition": "(`employees`.`a`.`dept_no` = 'd001')",
    "cost_info": {
      "read_cost": "0.50",
      "eval_cost": "0.20",
      "prefix_cost": "0.70",
      "data_read_per_join": "64"
    },
    "used_columns": [
      "emp_no",
      "dept_no",
      "from_date"
    ]
  }
},
```

```
{"table": {
    "table_name": "e",
    "access_type": "eq_ref",
    "possible_keys": [
      "PRIMARY"
    ],
    "key": "PRIMARY",
    "used_key_parts": [
      "emp_no"
    ],
    "key_length": "4",
    "ref": [
      "employees.a.emp_no"
    ],
    "rows_examined_per_scan": 1,
    "rows_produced_per_join": 2,
    "filtered": "100.00",
    "cost_info": {
      "read_cost": "2.00",
      "eval_cost": "0.20",
      "prefix_cost": "2.90",
      "data_read_per_join": "272"
    },
    "used_columns": [
      "emp_no",
      "first_name",
      "last_name"
    ]
  }
}
]
}
}
```

另外两种 Join 的写法

- join3_02 第二种 Join 写法：将条件全部写在 where 的后面

```sql
select
  e.first_name, e.last_name, a.from_date
from
  dept_manager a join departments d join employees e
where
  a.dept_no = d.dept_no and a.emp_no = e.emp_no and d.dept_no = 'd001'
```

- join3_03 第三种 Join 写法：笛卡尔积

```sql
select
  e.first_name, e.last_name, a.from_date
from
  dept_manager a, departments d, employees e
where
  a.dept_no = d.dept_no and a.emp_no = e.emp_no and d.dept_no = 'd001'
```

- 后面两种写法和之前的会产生同样的执行计划 [3]

[3] https://github.com/Jeanhwea/mysql-source-course/blob/master/assets/join3-prepare-example.org

尝试一下 straight_join 来控制 Join 顺序

- 使用 straight_join 查询语句实验 [a]

```sql
select
  e.first_name,
  e.last_name,
  a.from_date
from
  dept_manager a
    straight_join departments d
              on a.dept_no = d.dept_no
    straight_join employees e
              on a.emp_no = e.emp_no
where
  d.dept_no = 'd001'
```

- 通过估算得到 cost 值是: $3.35 > 2.90$


```json
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "3.35"
    },
    "nested_loop": [
      {
        "table": {
          "table_name": "a",
          "access_type": "ref",
          "possible_keys": [
            "PRIMARY",
            "dept_no"
          ],
          "key": "dept_no",
          "used_key_parts": [
            "dept_no"
          ],
          "key_length": "16",
          "ref": [
            "const"
          ],
```


[a] 见 JOIN_ORDER Hint 的用法

对比 straight_join 实际执行耗时

```
mysql> set profiling=1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> ... 执行查询

mysql> show profiles\G
*************************** 1. row ***************************
Query_ID: 1
Duration: 0.00922100
   Query: select /*+ rtc */
  e.first_name, e.last_name, a.from_date
from
  dept_manager a straight_join departments d straight_join employees e
where
  a.dept_no = d.dept_no and a.emp_no = e.emp_no and d.dept_no = 'd001'
*************************** 2. row ***************************
Query_ID: 2
Duration: 0.00867425
   Query: select /*+ rtc */
  e.first_name, e.last_name, a.from_date
from
  dept_manager a join departments d join employees e
where
  a.dept_no = d.dept_no and a.emp_no = e.emp_no and d.dept_no = 'd001'
2 rows in set, 1 warning (0.00 sec)
```

Join 执行算法

- MySQL 执行器默认使用 NLJ [4] 算法来实现 Join
  - 假设有三个表 `t1`, `t2` 和 `t3` 对于的 Join 类型如下
    1. `t1` range 范围扫描
    2. `t2` ref 引用
    3. `t3` ALL 全表扫描
  - NLJ (Nested-Loop Join) 算法的伪代码如下

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions, send to client
    }
  }
}
```

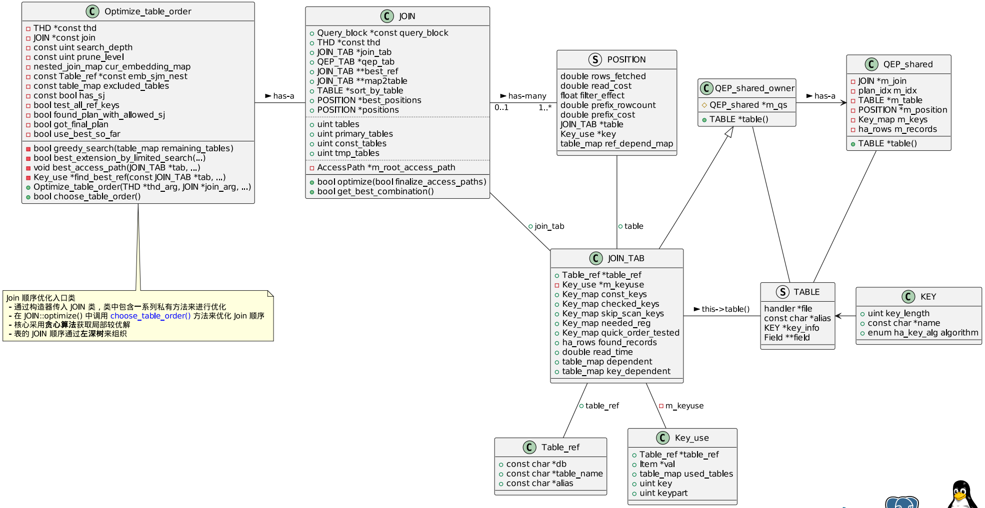
- NLJ 的思路是：先选定驱动表，然后根据过滤条件来查询被驱动表的数据
  - 如果过滤条件在 Join 操作之前，NLJ 算法的性能很好
  - 通常经过优化器后会满足小表驱动大表的原则

[4] https://dev.mysql.com/doc/refman/8.0/en/nested-loop-joins.html

3

代码分析

Join Order 数据结构



条件断点调试过程

1. 设置条件断点，如果包含 "rtc" 字串则触发断点

```
(gdb) break JOIN::optimize if strstr(thd->query().str, "rtc")
(gdb) c
```

2. 添加标识 /*+ rtc */ 后发送给 mysqld

```
(gdb) printf "%s\n", thd->query().str
select /*+ rtc */
  e.first_name, e.last_name, a.from_date
from
  dept_manager a join departments d join employees e
where
  a.dept_no = d.dept_no and a.emp_no = e.emp_no and d.dept_no = 'd001'
(gdb)
```

Optimize_table_order::choose_table_order()

- choose_table_order() 选择多表 Join 的顺序
- greedy_search() 穷举式 + 贪婪式算法，在尽可能小的搜索空间内得到相对较优的查询计划
  - 算法输入 remaining_tables 剩余待计算的表
  - 算法输出 pplan (partial plan) 当前最优的计划
  - 算法实现的伪代码如下

```
pplan = <>;
do {
  (t, a) = best_extension(pplan, remaining_tables);
  pplan = concat(pplan, (t, a));
  remaining_tables = remaining_tables - t;
} while (remaining_tables != {})
return pplan;
```

- best_extension_by_limited_search() 最终找到近似最优解的连接排列组合
  - 对于 n 个表的搜索量为 A_n^n 整体搜索空间比较大，所以会进行剪枝
    1. pruned_by_cost：根据代价剪枝
    2. pruned_by_heuristic：根据启发式规则剪枝
  - 限制最大可搜索的深度 [5]，记录在 Optimize_table_order::search_depth 中
- best_access_path() 获取最优路径

[5] 最大搜索深度参数 optimizer_search_depth=62

贪心算法递归搜索过程

- 包含 5 个表的 Join 搜索过程

```sql
select /*+ rtc */
  d.dept_no, t.title, e.first_name, s.from_date, s.salary
from
  dept_manager a join departments d join employees e join salaries s join titles t
where
  a.dept_no = d.dept_no and a.emp_no = e.emp_no and a.emp_no = s.emp_no
  and a.emp_no = t.emp_no and d.dept_no = 'd001';
```

- 递归调用过程，当 pplan_cost 值过大时发生剪枝 [6]

```
$ grep cost_for_plan /tmp/mysqld.trace
T@10: | | opt: cost_for_plan: 0.461096           => POSITIONS: d           <= 搜索
T@10: | | | opt: cost_for_plan: 0.451096         => POSITIONS: d a         <= 搜索
T@10: | | | | opt: cost_for_plan: 2.64926        => POSITIONS: d a e       <= 搜索
T@10: | | | | | | opt: cost_for_plan: 4.99109    => POSITIONS: d a e t     <= 搜索
T@10: | | | | | | | opt: cost_for_plan: 10.7898  => POSITIONS: d a e t s   <= 目前为止最优解
T@10: | | | | | | opt: cost_for_plan: 6.58652    => POSITIONS: d a e s     <= 剪枝
T@10: | | | | opt: cost_for_plan: 2.79293        => POSITIONS: d a t       <= 剪枝
T@10: | | | | opt: cost_for_plan: 4.38835        => POSITIONS: d a s       <= 剪枝
T@10: | | | opt: cost_for_plan: 28466.5          => POSITIONS: d e         <= 剪枝
T@10: | | | opt: cost_for_plan: 44874.5          => POSITIONS: d t         <= 剪枝
T@10: | | | opt: cost_for_plan: 281270           => POSITIONS: d s         <= 剪枝
```

[6] https://github.com/Jeanhwea/mysql-source-course/blob/master/assets/join5.mysqld.trace

Optimize_table_order::find_best_ref()

- find_best_ref() 获取最优的索引来进行 ref 访问，使用以下优先级进行选取

  1) A clustered primary key with equality predicates on all keyparts is always chosen.
  2) A non nullable unique index with equality predicates on all keyparts is preferred over a non-unique index, nullable unique index or unique index where there are some keyparts without equality predicates.
  3) Otherwise, the index with best cost estimate is chosen.

- 索引选取优先级
  - 等值聚簇索引
  - 非空唯一等值索引
  - 非空普通等值索引
  - 可空唯一等值索引
  - 唯一非等值索引
  - Cost 最小的索引

结束