

文章编号: 1000-5641(2018)04-0090-09

# 数据存储与处理分离架构下的子链接消除及优化

王彦朝, 胡卉芪, 张 召, 刘小兵, 段惠超

(华东师范大学 数据科学与工程学院, 上海 200062)

**摘要:** 在数据存储与处理分离架构下的 NewSQL 数据库中实现了子链接消除的功能, 使其支持大部分子链接的执行, 减少了从集中式数据库向分布式数据库迁移所需要的 SQL 改造代价, 使得 NewSQL 数据库可以在电信、银行等传统行业投入使用. 同时针对数据存储与处理分离的架构, 对子链接消除之后的执行进行了优化, 尽量减少了不同服务器之间数据的传输量. 实现的结果使得 NewSQL 成功支持了大部分的子链接, 且子链接执行效率也得到了提升.

**关键词:** 分布式数据库; NewSQL; 子链接消除

**中图分类号:** TP311.1 **文献标志码:** A **DOI:** 10.3969/j.issn.1000-5641.2018.04.009

## Sublink elimination and optimization in data storage and processing separation architecture

WANG Yan-zhao, HU Hui-qi, ZHANG Zhao, LIU Xiao-bing, DUAN Hui-chao

(School of Data Science and Engineering, East China Normal University,  
Shanghai 200062, China)

**Abstract:** This paper implemented a sublink elimination optimizer in NewSQL RDBMS (Relational Database Management System). The purpose of the optimizer is to make complex SQL so that it can be executable in New SQL systems, and achieve auto-tuning of complex SQL to reduce the cost of migrating from a centralized database to a distributed database; this would make the NewSQL DBMS (Database Management System) available to telecommunications and banking industries. We also provide a method to minimize data transmission between servers after sublink promotion. Experiments show that the optimization effect of the query optimizer is notable and can replace manual optimization and reduce the workload of application migration.

**Keywords:** distributed database; NewSQL; sublink elimination

## 0 引 言

NoSQL 数据库由于其良好的可扩展性, 成为了广受欢迎的现代 Web 信息系统数据后台存储系统, 比如 bigtable<sup>[1]</sup>、mongodb<sup>[2]</sup>等等. 然而 NoSQL 数据库只能支持基于主键

收稿日期: 2017-06-26

基金项目: 上海市 2017 年扬帆科技人才计划项目 (17YF1427800)

第一作者: 王彦朝, 男, 硕士研究生, 研究方向为数据存储与数据挖掘. E-mail: wyz159753@126.com.

通信作者: 胡卉芪, 男, 助理研究员, 研究方向为数据库. E-mail: hqhu@dase.ecnu.edu.cn.

的put()操作、get()操作、delete()操作, 对比如银行、电信等传统行业中的复杂查询支持有限. 因此, 出现了NewSQL数据库, 期望在保持系统良好可扩展性的前提下, 能快速响应复杂查询请求, 如Spanner<sup>[3]</sup>、OceanBase<sup>[4]</sup>.

但是这些NewSQL数据库系统基本都是互联网公司根据他们的需求进行开发的, 支持的SQL语法只是标准SQL的一个子集, 对于复杂SQL的支持还是比较匮乏, 对于银行、电信等数据存取复杂的行业来说还不能满足需求.

经过对这些行业的SQL进行的调研发现, 大多数不能执行的SQL是子链接的形式. 不过因为子链接原本的嵌套循环执行方式效率太低, 所以本文没有尝试实现子链接的执行, 而是实现子链接的消除, 将子链接转换为子查询与父查询的连接操作. 因此本文在NewSQL分布式数据库中实现了子链接消除的功能, 使得使用现有的系统即可实现子链接的执行, 从而减少了数据库迁移时查询重写的工作负担.

子链接消除是大部分集中式数据库中都实现的查询优化方式, 但是子链接提升的理论描述相对比较分散, 因此本文结合一些论文及系统实现对子链接消除的方案进行了归纳. 由于实际使用的子链接中的子查询绝大多数是简单查询(即只有选择、投影、连接(join)等操作, 没有集合运算、聚集、排序等操作), 所以本文对于这种情况下的子链接消除, 选择了最适合在本系统进行实现子链接消除的方式.

在子链接消除之后, 本文针对数据存储与处理分离的架构特点, 实现了一种子链接执行的优化方法, 通过减少数据传输来提高查询性能.

具体实现是基于华东师范大学数据科学与工程学院(简称“DaSE”)对OceanBase 0.4.2进行功能扩充和性能优化后的CEDAR 0.2<sup>[5]</sup>版本.

本文第1节介绍子链接提升的理论方案; 第2节介绍子链接执行的优化方案; 第3节介绍子链接执行所必需的半连接的物理实现; 第4节测试子链接的优化方案的效果; 第5节总结全文.

## 1 子链接消除的理论方案

### 1.1 子链接的相关概念

所谓子链接是形如 *select...from... where 列或表达式+运算符+(select 查询)* 的查询结构.

子链接按照运算符分类, 可分为3类: in类型子链接、ALL/ANY/SOME类型子链接、exist类型的子链接<sup>[6]</sup>.

子链接按照关联性分类, 可分为相关子链接和非相关子链接两类<sup>[7]</sup>. 非相关子链接即子查询中没有引用父查询中的值. 例如

```
select * from t1 where t1.col1 > (select count(*) from t2);
```

相应地, 相关子链接即在子查询中引用了父查询中的值. 例如

```
select * from t1 where t1.col1 > (select count(*) from t2 where t2.col2 = t1.col2);
```

在此查询中, 子查询中引用了父查询中的表t1的列col2.

### 1.2 子链接的执行方式及优化方案

对于如下子链接

```
select * from t1 where t1.col1 in (select col2 from t2);
```

其原始执行方式是对于父查询中  $t_1$  的每一行数据, 执行一次  $t_1.col1 \text{ in } (\text{select } col2 \text{ from } t2)$  的判断, 即  $t_1$  中有多少行即执行子查询多少次, 这是非常低效的. 因此, 要对原始的子链接执行方式进行优化.

子链接优化的理论大体可以分为两种: 一是将子链接中的子查询结构消除, 将子链接表达式变为连接表达式, 直接变为一层连接, PostgreSQL 中采用的就是这种方式<sup>[7]</sup>, 这种方式适用于子查询是简单子查询(即只有选择、投影、连接等操作, 没有集合运算、聚集、排序等操作, 可以被提升到父查询中)的情况, 因为如果子查询不可被提升, 则此种优化方式就无法执行; 二是将子查询中引用的父查询的值下压到子查询中, 将子查询与父查询的关联关系消除<sup>[8]</sup>, 此种方式适用范围更加广泛, 也能比较有效的提升查询性能, 不过由于其并没有将子链接结构消除, 仍然需要数据库系统支持子链接.

由于实际使用的子链接中的子查询绝大多数是简单查询, 而且考虑到对不支持子链接的数据库系统的适用性, 本文采用的是第一种子链接优化方式, 即子链接消除.

对于子查询为复杂查询的子链接, 使用子链接消除方式的数据库系统无法对其查询计划进行调整. 如果系统后续实现了子链接的执行功能之后, 那么可以通过“重复不变式”<sup>[9]</sup>的方式对复杂子链接的执行进行优化, 这样就可以避免对已有的子链接消除逻辑的调整了.

子链接的消除需要根据子链接运算符的不同采用不同的方法. 由于  $\text{all}/\text{any}/\text{some}$  类型的子链接较难消除, 在许多商用数据库如 MySQL、PostgreSQL 中都没有进行优化, 因此本文也主要关注  $\text{exist}$  类型和  $\text{in}$  类型的子链接消除.

下面分别给出这两种子链接的消除方式.

#### 1.2.1 $\text{exist}$ 类型的子链接消除

对于非相关子链接,  $\text{exist}$  类型的一般没有什么意义, 因为对于每一行都是判断一个相同的  $\text{exist}$  条件, 这意味着要么选出所有行, 要么一行也选不出. 实际应用中基本不会出现  $\text{exist}$  类型的子链接. 所以查询优化器忽略对  $\text{exist}$  类型非相关子查询的优化.

对于  $\text{exist}$  类型的相关子链接, 可以直接将子查询和父查询合并来进行消除. 例如

```
select * from t1 where exist (select * from t2 where t1.col1 = t2.col1);
```

消除后为

```
select * from t1, t2 where t1.col1 = t2.col1;
```

不过要注意, 这里并不是完全等价的, 如果  $t_2$  的  $\text{col1}$  中有重复的数据, 则提升后的 SQL 语句会对于同一个  $t_1.col1$  生成多个结果行, 而原始的 SQL 只会生成一行.

所以, 优化后的 SQL 语句的等号表示的不应该是内连接, 而应该是左半连接 (left semi-join). 左半连接对于每一个左表的值, 只从右表中寻找一个匹配的值, 匹配到之后就停止匹配这一个左表的值并输出到结果集, 同时继续进行下一个左表值的匹配. 这样就可以保证结果集中的数据行数不会因右表中出现重复而改变.

相应地, 对于  $\text{not exist}$  类型的相关子查询, 消除的方式相同, 不过提升后的连接方式为左反半连接. 左反半连接对于每一个左表的值, 如果它不能匹配到任何一个右表的值, 则输出到结果集; 如果能够匹配到右表中的任意一个值, 则不输出到结果集.

因此, 通用的  $\text{exist}$  子链接消除流程总结如下.

(1) 对于非相关  $\text{exist}$  子链接, 不优化.

(2) 对于相关  $\text{exist}$  子链接, 优化前为

```
select ... from outertable, ... where exist(select ... from innertable, ... where outertable.col1 = innertable.col2 and inner_conditions) and outer_conditions;
```

优化后为

```
select ... from outertable, ..., innertable, ... where outertable.col1 = innertable.col2  
and inner_conditions and outer_conditions;
```

其中, innertable 为子链接中的表, outertable 为父查询中的表, inner\_conditions 为子链接中除了关联条件之外的其他约束条件, outer\_conditions 为父查询中除了子链接之外的其他约束条件.

### 1.2.2 in 类型的子链接消除

in 类型的非相关子链接可以直接转换为子查询与父查询的连接操作. 例如对于如下非相关 in 子链接

```
select * from outertable where outertable.col1 in (select col1 from innertable);
```

可以转换为

```
select * from outertable, (select col1 from innertable) as tmp1 where outertable.col1 =  
tmp1.col1;
```

其中, innertable 为子链接中的表, outertable 为父查询中的表, col1 为两个表储存相同类型数据的列.

可以看到, 将子查询 (select col1 from innertable) 提升到了父查询中作为一个临时表 tmp1, 再将 innertable 与 tmp1 进行连接.

同 exist 的消除一样, 由于 in 类型子链接的语义也是找到一个即停止, 也就是相当于对右表的去重, 所以消除之后的连接运算符也需要是半连接. 对于 not in, 消除之后的连接运算符是反半连接.

这里子查询提升上来之后, 可以选择将这个子查询与父查询进行合并, 也可以不进行进一步的处理. 因为这两种方式的执行方式基本是一样的, 都是两表分别过滤之后进行连接, 出于降低实现复杂度的考虑, 本文在这里选择了后者.

in 类型的相关子链接可以转换为 exist 类型的相关子链接, 然后按照 exist 类型相关子链接提升的方式进行提升. 例如

```
select * from outertable where outertable.col1 in (select col1 from innertable where  
innertable.col2 = outertable.col2);
```

可以转换为

```
select * from outertable where exist(select col1 from innertable where innertable.col2  
= outertable.col2 and outertable.col1 = innertable.col1);
```

本文将 outertable.col1 in (select col1 from innertable) 转化为了子查询中的一个关联连接条件 outertable.col1=innertable.col1, 这里将一个具有去重含义的 in 转换为连接, 按理说也需要使用半连接运算符. 不过由于转换后的 exist 本身也包含有去重的含义, 所以这里的半连接可以省去, 直接使用内连接即可.

相应地, 对于 not in 类型的相关子链接, 可以转换为 not exist 类型的相关子链接.

in 类型的子链接提升方式总结如下.

对于 in 类型的非相关子链接, 优化前为

```
select ... from outertable, ... where outertable.col1 in (select col2 from innertable where  
inner_conditions) and outer_conditions;
```

优化后为

```
select ... from outertable, ..., (select col2 from innertable where inner_conditions) as
tmp1 where outertable.col1=tmp1.col2 and outer_conditions;
```

对于 in 类型的相关子链接, 优化前为

```
select ... from outertable, ... where outertable.col1 in (select col2 from innertable where
outertable.col3=innertable.col4 and inner_conditions) and outer_conditions;
```

优化后为

```
select ... from outertable, ... where exist
(select col2 from innertable, ... where outertable.col1=innertable.col2 and out-
ertable.col3=innertable.col4 and inner_conditions) and outer_conditions;
```

以上总结了子链接的消除方式, 这与商用集中式关系数据库实现的子链接提升比较类似. 按照这种方式进行优化之后, 大部分的子链接语句已经可以执行了. 不过在实际使用中, 很多数据量大的子链接的执行效率是比较低的. 原因在于, 在集中式数据库中, 通过使用连接替代嵌套循环执行方式来提升执行效率, 对于这种查询的优化效果比较明显; 而分布式数据库中此种查询的主要时间花费在了数据传输上, 因此还要在之前优化的基础上尽量减少数据的数据传输代价. 基于此种考虑, 本文提出了以下的优化方案.

## 2 子链接执行的难点及优化方案

在数据存储与处理分离的架构中, 数据存放于存储服务器, 而查询在处理服务器上执行.

一般的子链接的执行过程是: 处理服务器接受 SQL 查询请求, 解析生成物理查询计划后, 将单表的扫描操作及其约束条件发送到存储服务器端, 由存储服务器完成表的扫描和基本的筛选再发送给处理服务器; 处理服务器接收到所有存储服务器的请求之后, 再进行连接(join)等多表操作; 最后将查询结果返回给客户端. 过程如图 1 所示.

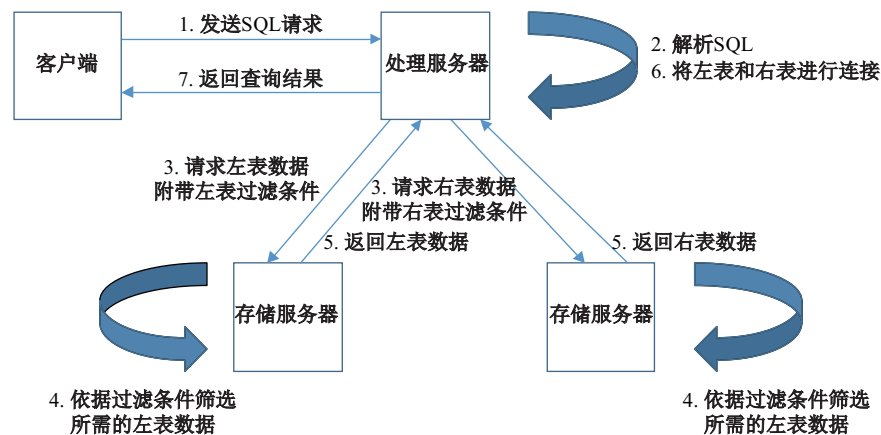


图1 未优化的子链接执行流程

Fig. 1 Unoptimized sublink execution flow

可以看出, 当涉及连接操作时, 需要将待连接的两表的数据都传输到处理服务器上才可以执行, 而如果其中一个或两个表很大, 则传输会耗费大量的时间.

在之前的论述中, 子链接消除方法的执行结果都是变为两个表连接, 而一般来说子链接的内表的数据量会比较小, 外表数据量会比较大, 连接结果一般会比较小. 因此利用这个特点, 本文提出了一种子链接执行的优化方式: 子链接消除之后将数据量较小的表传输到较大的表上进行过滤, 过滤掉无法产生连接结果的外表数据, 以此来减少传输大表数据所需的时间. 例如, 子

链接的提升结果对应的SQL

```
select t1.* from t1, t2 where t1.col1 = t2.col1 and inner_condition and outer_condition;
```

无优化的执行流程是将t1、t2都拉到处理服务器再进行连接. 经过优化之后, 会把t2的连接列col1的数据打包发送给t1所在的存储服务器, 筛选t1表需要传输的数据, 减少t1表需要传输的数据量. 过程如图2所示.

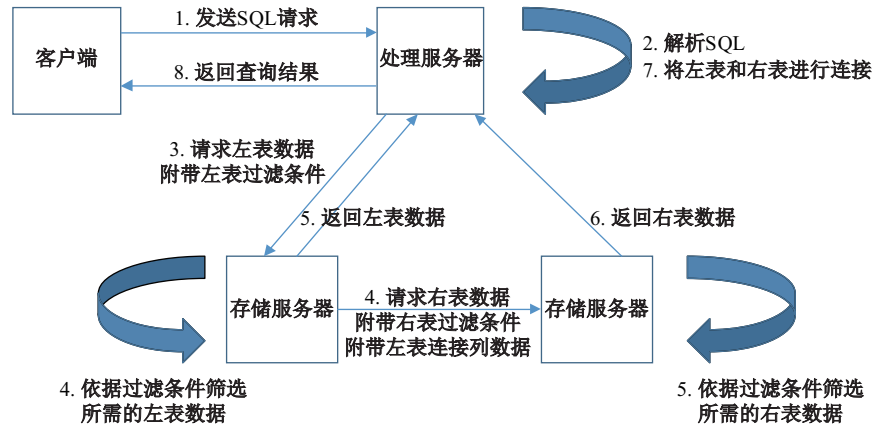


图2 优化的子链接执行流程

Fig.2 Optimized sublink execution flow

### 3 半连接的物理实现

子链接的逻辑提升方式在上文中已经论述得比较详尽了, 子链接提升的实现就是, 将这种提升过程具体为对逻辑查询计划的调整逻辑, 这里不再详述.

之前的提升方式都是将子链接变形为父查询表与子查询表的左半连接操作. 而半连接运算符的主要用途就是处理经过消除优化后的子链接, 所以不支持子链接的NewSQL基本都没有实现半连接运算符. 所以还要对左半连接和左反半连接进行实现. 半连接运算符一般是实现在具体连接类中(例如MergeJoin、HashJoin), 作为对左右表每行的连接进行判断的函数. 以下提供了一种在MergeJoin中实现半连接的方式.

左半连接的语义是对于每一个左表的值, 只从右表中寻找一个匹配的值, 匹配到之后就停止匹配这一个左表的值并输出到结果集, 同时继续进行下一个左表值的匹配. 由于MergeJoin是先排序后连接, 所以对于一条左表数据, 只要扫描下一个右表的行, 如果左边大于右边值, 则左边数据不变, 取下一个右表数据; 如果左边等于右边, 则输出左边行, 读取下一个左表数据; 如果左边大于右边, 则左边值匹配不到右表中的值, 直接取下一个左表值继续. 这个过程的伪代码如下.

```

//半连接 LEFT_SEMI_JOIN
Function left_semi_join(const Row *&row) // 这里的row即为加入结果集中的数据行
Row *left_row = NULL; //存储左表当前参与连接的行
Row *right_row = NULL; //存储右表当前参与连接的行
left_op->get_next_row(left_row); //通过左表扫描运算符获取左表一行数据
if (last_right_row_有效) //上一个匹配到的右表值
    right_row = last_right_row_;
    last_right_row_ = NULL;
else

```

```

    right_op->get_next_row(right_row); //通过右表扫描运算符获取右表一行数据
while (两个子运算符还有数据没有迭代完)
if (left_row 和 right_row 在等值 join 条件上相等)
    row = left_row; //输出左表数据
    last_right_row_ = right_row;
    break;
else if(left_row 在等值 join 条件上< right_row)
    left_op->get_next_row(left_row);
else
    // left_row 在等值 join 条件上> right_row
    right_op->get_next_row(right_row);

```

其中, row 存储的是每一行的数据, left\_op\_是 join 运算左边的子物理运算符, right\_op\_为 join 运算右边的子物理运算符, get\_next\_row 获取该运算符的下一行数据.

左反半连接的语义是对于每一个左表的值, 如果它不能匹配到任何一个右表的值, 则输出到结果集; 如果能够匹配到右表中的任意一个值, 则不输出到结果集. 因此对于一个左表值和一个右表值: 如果左大于右, 则左边没有匹配到任何右表数据, 输出左边的行; 如果左等于右, 则左边匹配到了, 抛弃此左边行, 读取下一左边行; 如果左小于右, 继续读取下一右表行. 过程的伪代码如下.

```

// 反半连接 LEFT_ANTI_SEMI_JOIN
Function left_anti_semi_join(const Row *&row)
    Row *left_row = NULL;
    Row *right_row = NULL;
    left_op->get_next_row(left_row);
    if (last_right_row_ 有效)
        right_row = last_right_row_;
        last_right_row_ = NULL;
    else
        right_op->get_next_row(right_row);
    while (两个子运算符还有数据没有迭代完, 既 left_row 和 right_row 都有效)
    if (left_row 和 right_row 在等值 join 条件上相等)
        left_op->get_next_row(left_row);
    else if(left_row 在等值 join 条件上< right_row)
        row = left_row;
        last_right_row_ = right_row;
        break;
    else
        // left_row 在等值 join 条件上> right_row
        right_op->get_next_row(right_row);
    if (没有找到行要输出&& left_row 有效)
        row = left_row; // left_op_ 还有数据;

```

其中, row 存储的是每一行的数据, left\_op\_是 join 运算左边的子物理运算符, right\_op\_为 join 运算右边的子物理运算符, get\_next\_row 获取该运算符的下一行数据.

#### 4 实 验

实验基于华东师范大学在 OceanBase 基础上开发的 CEDAR. CEDAR 中原本不支持子链接,子链接 SQL 只有经过本文中的查询优化器的处理之后才可以执行,所以没有办法比较子链接提升的优化效果.因此在本实验部分,重点测试子链接消除之后优化方案的性能提升效果.

在 CEDAR 项目的前期工作中,已经实现了这种优化的连接方式,因此逻辑查询优化器直接在子链接的消除时加入指定使用这种优化的 hint.但是之前的优化是针对内连接的,所以左表过滤右表时是按照内连接运算符来进行过滤,而子链接的执行是半连接的运算符,所以本文也将左半连接和左反半连接加入连接优化的执行逻辑中.

#### 4.1 实验环境

使用4台服务器组成的集群作为测试环境,每台服务器的配置相同,包括4核1.2 GHz 主频 CPU、100 GB 内存、3 000 GB 磁盘.服务器上安装了 CentOS release 6.5 系统,相互之间通过千兆以太网连接.集群中的一台服务器被配置为 RootServer、MergeServer 和 UpdateServer,另外3台服务器被配置为 ChunkServer.实验采用的数据是使用数据生成器随机生成的数据.

执行测试的机器通过 MySQL 客户端连接集群,使用 MySQL 客户端的查询返回时间作为查询执行的总时间.客户端的配置为 Intel Xeon E5-1603 4核 2.80 GHz、16 GB 内存、500 GB 硬盘.

#### 4.2 实验数据集

表 sbtest1 作为子链接中的内表,数据量设置了 1 000 条和 1 000 万条两种情况.在测试的时候通过在内表添加约束条件更改内表中实际筛选出来的数据量大小.

表 sbtest2 作为子链接中的外表,数据量设置了 1 000 条、10 万条、100 万条和 1 000 万条 4 种情况.

实验使用的 SQL

```
select * from sbtest2 where sbtest2.id in (select id from sbtest1 where sbtest1.id < [内表数据量]);
```

这条 SQL 会被查询优化器优化为 sbtest2 与 sbtest1 的连接操作,其中内表数据量为测试时使用的变量.

#### 4.3 实验结果

实验结果如图3所示.由图3可以看出,当右表数据较大时,优化效果比较明显;不过在数据较小时虽然优化的方式没有不优化的方式快,但因为数据量小,依然可以很快地查出来,所以本文对于子链接消除之后的执行默认使用了这一种优化方式.后续的工作中可以通过统计信息来获取两表大小,从而自动地使用合适的子链接执行方式.对于两个较大的表连接,可以使用布隆过滤器<sup>[10]</sup>来替代传输左表连接列数据的过程,减少传输量.

## 5 结 论

本文设计的子链接消除功能使得 NewSQL 数据库成功地支持了更多的复杂 SQL,基本满足了一些传统行业的数据查询需求,使得分布式数据库的应用场景更加广泛.如电信、银行等查询复杂的应用场景可以部署分布式数据库,而不需要对原来应用的很多 SQL 进行改造,这就极大地节省了企业的数据存储成本,同时也提高了数据存储的可靠性和可扩展性.

在性能方面,本文针对数据存储与处理分离架构设计了子链接执行优化方案,此方案对于实际生产环境中很多性能较差的查询都有不错的优化效果,对于小的查询所带来的额外代价也可以接受.因此可以减少企业应用从集中式数据库环境迁移到分布式数据库环境后,对性能较



差SQL的手工优化代价.

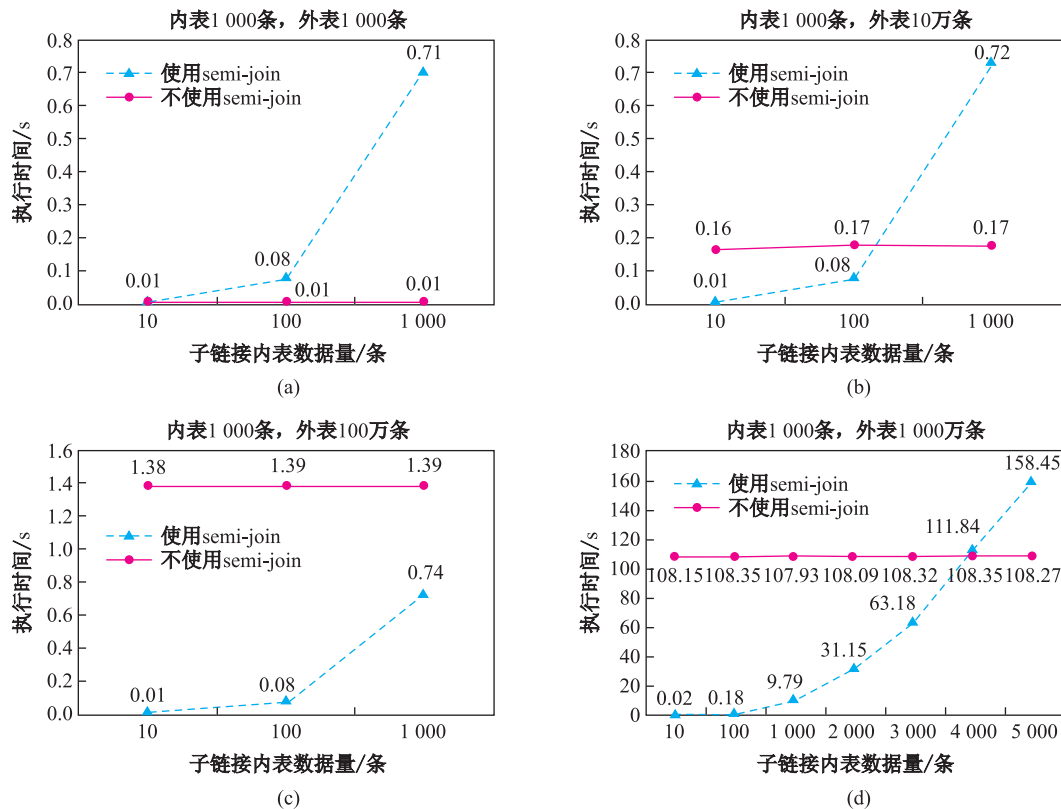


图3 实验结果

Fig.3 Experimental results

### [参 考 文 献]

- [1] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data [C]//Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2006, 205-218.
- [2] CHODOROW K. MongoDB: The Definitive Guide [M]. [S.l.]: O'Reilly Media Inc, 2013.
- [3] CORBETT J C, DEAN J, EPSTEIN M, et al. Spanner: Google's globally distributed database [C]//Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2012, 251-264.
- [4] ALIBABA. OceanBase [EB/OL]. [2017-05-02]. <https://github.com/alibaba/oceanbase>.
- [5] daseECNU. Cedar [EB/OL]. [2017-05-02]. <https://github.com/daseECNU/Cedar>.
- [6] STONEBRAKER M, HANSON E, HONG C H. The design of POSTGRES rules system [C]//1987 IEEE 3rd International Conference on Data Engineering. IEEE, 1987: 365-374.
- [7] 李海翔. 数据库查询优化器的艺术: 原理解析与SQL性能优化 [M]. 北京: 机械工业出版社, 2014: 2-263.
- [8] SESHADRI P, PIRAHESH H, LEUNG T Y C. Complex query decorrelation [C]//Proceedings of the 12th International Conference on Data Engineering. IEEE, 1996: 450-458.
- [9] RAO J, ROSS K A. Reusing invariants: A new strategy for correlated queries [J]. ACM SIGMOD Record, 1998, 27(2): 37-48.
- [10] BURTON H. Bloom, space/time trade-offs in hash coding with allowable errors [J]. Communications of the ACM, 1970, 13(7): 422-426.

(责任编辑: 李 艺)