文章编号:1000-5641(2014)05-0271-10

一种面向海量分布式数据库的嵌套查询策略

裴欧亚, 刘文洁, 李战怀, 田 征
(西北工业大学 计算机学院,西安 710129)

摘要: 面向大数据分析和处理的 NoSQL 数据库具有非常好的读写性能和可扩展性,但是无法支持完整的 SQL 查询和跨行跨表的事务,对于传统的以关系数据库为主的金融业务在应用上有所限制. OceanBase 是面向海量数据查询的分布式数据库,结合了关系数据库和非关系数据库的优势,同时支持关系查询和跨行跨表事务,具有可扩展性. 但是,目前 OceanBase 只支持简单的、非嵌套子查询的 SQL 语句,无法很好地支持金融应用. 本文在研究 OceanBase 架构和查询策略的基础上,提出了一种基于 BloomFilter 和 HashMap 的查询策略,实验表明该策略能够提高和改善现有查询策略的不足,在实现嵌套查询的基础上,可提高查询性能.

关键词: OceanBase; NoSQL; 嵌套查询; 大数据
中图分类号: TP311.133.1 **文献标识码**: A **DOI**:10.3969/j.issn.1000-5641.2014.05.024

A nested query strategy oriented massive distributed database

PEI Ou-ya, LIU Wen-jie, LI Zhan-huai, TIAN Zheng
(*School of Computer, Northwestern Polytechnical University, Xi'an* 710129, *China*)

Abstract: NoSQL databases have very good read/write performance and scalability in Big data analysis and processing, but they cannot support complete SQL queries and transactions across tables or rows, which limits the application of financial business based on the traditional relation databases. OceanBase, a distributed database, combines the advantages of relational databases and non-relational databases, supporting relational queries and transactions across tables or rows. However, at present OceanBase only supports simple and non-nested queries which cannot meet the needs of financial business. After studying the OceanBase architecture and query strategy, a new strategy based on BloomFilter and HashMap is proposed in this paper. Experiments show that the strategy can improve the existing query strategy and enhance query performance.

Key words: OceanBase; NoSQL; nested queries; big data

收稿日期:2014-06
基金项目:国家 973 课题(2012CB316203); 国家 863 课题(2012AA011004);国家自然科学基金重点项目(61033007); 国家自然基金青年项目(61303037)
第一作者:裴欧亚,男,博士研究生,研究方向为数据管理. E-mail: peiouya2013@mail.nwpu.edu.cn.
通信作者:刘文洁,女,副教授,研究方向为软件理论、自律计算、高可用性系统.
E-mail: liuwenjie@nwpu.edu.cn.

0 引 言

随着云计算、Web2.0 等技术的进一步发展,NoSQL 数据库不断发展壮大. NoSQL 数据库放弃了传统关系型数据库严格的事务一致性和范式约束,采用弱一致性模型,支持分布式和水平扩展,满足了海量数据管理的需求.

为了信息安全及降低数据库系统升级维护费用,国内银行开始推进"去 IOE 化"战略. NoSQL 数据库相较于传统关系型数据,具有超高的性价比和良好的可扩展性. 这些特质促使 NoSQL 数据库成为国内银行业应对海量数据的首选数据库.

OceanBase 是阿里集团研发的一个海量分布式关系数据库系统,采用了 NoSQL 数据库的架构,基于横向扩展模式,可以通过动态在线增加/减少服务器的方式调整系统负载,具有良好的可扩展性. 而且,系统实现了关系数据库的重要特征,支持 SQL 语言查询,相对于其它的 NoSQL 数据库,更好地满足了金融业务的需求.

OceanBase 的可扩展性、标准 SQL 查询和事务的强一致性功能,在应对银行业的金融业务上,具有很大优势. 金融业务的一个特点是大量使用嵌套子查询,但是目前,OceanBase 只支持简单的非嵌套的 SQL,对于复杂的的嵌套子查询尚未实现,因此阻碍了金融业务的导入.

本文通过分析 OceanBase 的架构及现有查询策略,提出了一种基于 BloomFilter 和 HashMap 的子查询实现策略,以改进现有查询策略在查询性能上的不足和 SQL 功能上的缺陷,为金融业务提供更好的支持. 实验验证该策略与现有的 Oceanbase 的查询策略相比,能更好提高查询速度并支持大数据查询.

1 相关研究进展

嵌套子查询是标准 SQL 的一个非常重要的功能. 正是由于嵌套层数的任意性,才使得标准 SQL 具有强大的功能.

为了支持嵌套子查询,传统的关系数据库做了大量工作. IBM 的 WON KIM 最早于 1982 年将嵌套 SQL 分为不同的嵌套类型,并为每一种嵌套类型提出了转化算法,将嵌套 SQL 尽可能转换为等价的非嵌套 SQL,提升查询性能[1]. 加州大学伯克利分校的 Harry K T Wong 对 WON KIM 的转换算法进行了研究,发现了转换算法中关于聚合函数和数据重复计算的 BUG,并给出了改进的转换算法[2]. WON KIM 和 Harry K T Wong 的研究主要集中在重写 SQL,即将嵌套子查询改写为等价的非嵌套查询,为 SQL 重写技术的研究奠定了重要基础. 突飞猛进的并行计算技术促进了 SQL 查询优化的另一个研究分支-查询分解策略,即将查询 SQL 按一定的策略拆分为一系列可并行执行的子查询. Kim, T. K 等人充分利用网格计算和集群计算的并行能力,将嵌套子查询拆分为多个子查询,子查询被分发至网格/集群的不同节点并行执行[3-5]. Kang, Y. J 等人同样提出了复杂 SQL 分解策略,并构建了多 linux PCs 的 OLAP 引擎 HyperDB. TPC-R benchmark 测试验证了 HyperDB 的优异性能[6].

NoSQL 数据库为了追求查询的高性能,都不内置支持嵌套子查询功能. NoSQL 将嵌套子查询留给应用层实现,主要采用 MapReduce 框架实现[7-8]. 但是,也有一些 NoSQL 系统提供了比较好的机制来实现复杂查询,例如,MongoDB 可以设定复杂的查询条件[9]. 部分 NoSQL 系统通过运行 MapReduce,或者与 Hadoop 结合来支持大规模数据分析[10].

OceanBase 为了高效查询,只支持简单的非嵌套 SQL,包含 IN 后接确定值的查询

SQL. OceanBase 的 Filter 条件过滤方式是逐一比对,因此如果表扫描策略为 GET,在有主键索引的条件下,逐一比对不会耗费大量时间,具有很高的性能. 但是如果表扫描策略为 SCAN,数据表的每条记录都会和 Filter 过滤条件逐一比对,会耗费大量的时间,性能较差. 除此之外,OceanBase 不支持子查询,嵌套 SQL 的实现只能由应用程序负责,这不仅不方便使用,还耗费大量时间.

本文基于 OceanBase 的架构及设计思想,提出了一种基于 HashMap 和 BloomFilter 的嵌套子查询实现策略:当子查询结果集不大于阈值 K 时,直接将结果集绑定至主查询的 Filter 内,按 OceanBase 现有策略即可处理,其性能取决于 OceanBase 现有查询策略的性能,因为当阈值 K 足够小时,不大于 K 的子查询结果集的绑定耗时可忽略不计;当子查询结果集大于 K 时,启用 BloomFilter 和 HashMap,BloomFilter 和 HashMap 的构建会耗费一些时间,但是 BloomFilter 和 HashMap 在数据检索方面具有非常高的效率;当表扫描策略为 SCAN、子查询结果集大于 K 且小于 OceanBase 现有的 IN 上限时,假定结果集大小为 N,嵌套查询策略只需经过数次计算即可判定一行记录是否符合要求,而 OceanBase 现有的查询策略则需要每条记录平均比对 $N/2$ 次(最差情况 N 次,最好情况 1 次). 因此在子查询结果集较大时,嵌套查询策略具有较好的性能.

2　OceanBase 现状

2.1　OceanBase 架构

OceanBase 的整体架构如图 1 所示.

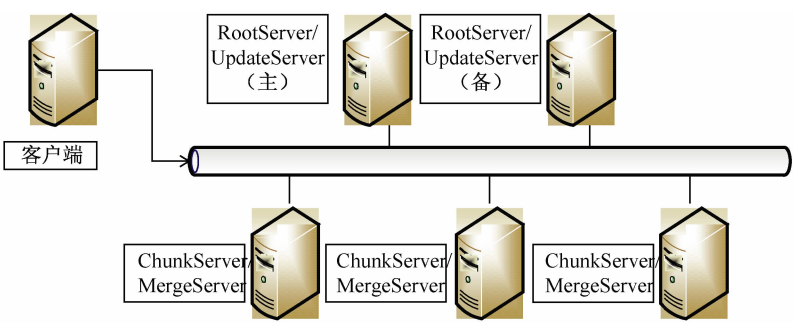

图 1　OceanBase 整体架构图

Fig. 1　Architecture of OceanBase

OceanBase 根据其设计目的,并结合淘宝业务特点,采用了"主—从"系统架构.

主节点,即 RootServer,唯一,管理集群中的所有从节点服务器,子表数据分布以及副本管理. 为了规避单主节点宕机的风险,RootServer 采用了一主一备的结构,主备之间采用数据强同步策略,并通过心跳实现软件高可用性.

从节点被分为以下三种类型的节点:

(1) 更新服务器,即 UpdateServer,唯一,存储增量数据,其实现类似单机的内存数据库,高效支持跨行跨表事务. 为了保证高可用性,UpdateServer 同样采用了一主一备的结构. 为了保证系统的高性能,UpdateServer 主备间可配置不同的模式,异步模式主要用于异地容灾,异步模式支持最终一致性.

(2) 基线数据服务器,即 ChunkServer,多台,存储基线数据.

(3) 合并服务器,即 MergeServer,对外提供标准的 SQL 访问接口,对内合并多台 ChunkServer 返回的数据集.

2.2 OceanBase 现有查询策略

Oceanbase 将数据分为基线数据和增量数据,分别存储在 ChunkServer 和 UpdateServer. 对于任何一次查询 SQL 请求,OceanBase 都需要执行 ChunkServer 和 MergeServer 相关数据的合并.

OceanBase 关于查询 SQL 的数据库结构如图 2 所示.

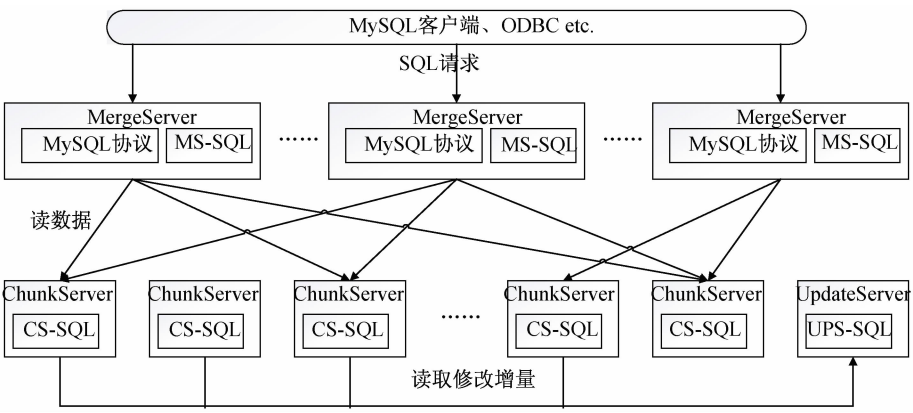

图 2 Oceanbase 数据库功能层整体结构

Fig. 2 Overall structure of OceanBase database function layer

用户通过 MySQL 客户端、ODBC 等方式将 SQL 请求发送给某台 MergeServer,MergeServer 的 MySQL 协议模块从 SQL 请求中解析出 SQL 语句,并交给 MS-SQL 模块进行词法/语法解析,并生成逻辑计划和物理计划. MergeServer 首先根据物理计划定位请求的数据所在的 ChunkServer,接着将物理计划发往相应的 ChunkServer. 每个 ChunkServer 调用各自的 CS-SQL 模块完成 SQL 查询. 如果需要增量数据,ChunkServer 的 CS-SQL 模块自动从 UpdateServer 读取修改增量,完成增量数据与本地基线数据的融合. ChunkServer 最终将查询结果返回给请求的 MergeServer. MergeServer 的 CS-SQL 模块执行多个 ChunkServer 返回结果的合并,获取最终结果.

OceanBase 支持的 SQL 语句比较简单,绝大部分针对单张表格,只有很少一部分操作涉及多张表格,例如 join 操作. OceanBase 目前只对主键有索引,OceanBase 表扫描策略分为 GET 和 SCAN 两种,GET 策略当且仅当过滤条件包含全部主键列时启用. SQL 执行本地化亦是 OceanBase 查询策略的基本设计原则,即尽量支持 SQL 计算本地化,保持数据节点和计算节点一致.

3 基于 OceanBase 的嵌套 IN 子查询策略

本文主要针对非相关的嵌套 IN 子查询. 非相关的嵌套 IN 子查询是指外查询依赖于内查询,内查询不依赖外查询且可以独立先执行(下文的嵌套查询策略就是嵌套 IN 子查询策略).

嵌套查询策略包含:嵌套查询 SQL 的查询树构建;查询树的执行引擎;两阶段数据过滤.

3.1 查询树

策略没有采用传统关系数据库的 SQL 重写技术,而是采用“内查询先执行,外查询绑定内查询的结果(集)后执行”的方案.该方案实现简便,而且相较于 SQL 重写技术,降低了传送到 MergeServer 的数据量,节省了带宽,减小了嵌套查询对并发查询的影响.

查询树的节点的部分主要数据结构如下.

```
struct QT_Node {
        char * sub_query;//保存本节点对应的 SQL 语句
        PhysicalPlan * phy;//保存本节点对应的 SQL 语句的物理计划
        List<QT_Node * > * next_child;//保存本节点依赖的子节点指针
        List<HashMap * > child_hashmap;//保存子查询结果集
        List<BloomFilter * > child_bf;//保存子查询结果集的指纹信息
```

注:phy 内保存子节点执行结果填充位置标记;phy 内的位置标记和 next_child 一一对应.

查询树可以借助传统关系型数据库的 SQL 语句解析结果进行构建.示例嵌套 SQL 和其查询树如图 3 所示.

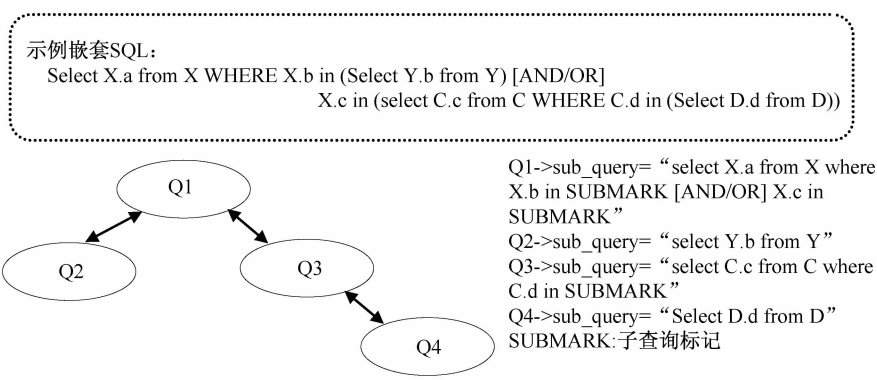

图 3　示例 SQL 及其查询树

Fig. 3　Sample SQL and its query tree

3.2　执行引擎

执行引擎的主要功能就是按照一定的策略执行查询树.依据策略构建的查询树,具有“兄弟节点相互独立”和“父节点依赖子节点”的特性.查询引擎根据查询树的特性,采用从叶节点到根节点的递归计算算法.

递归算法如下:

```
Input: the rootNode of QT_Tree
QT_EXECUTE(pNode):
    If NULL ! = pNode && pNode is leaf QT_Node then
      If rootNode = = pNode then
        execute pNode to generate resultset;
      Else
        execute pNode to generate resultset;
        If resultset.count() > threshold then
          Generate HashMap and BloomFilter with resultset;
          Assign HashMap and BloomFilter to the parent Node;
      Else
        assign the resultset to the parent Node;
      Erase current node from QT_Tree;
    else
      For i = 0 to next_child->count() then
          CALL QT_EXECUTE with parameter pNode->next_children->at(i);
END
```

算法的核心:串行执行每个节点;除根节点外,每个节点执行结束,将本节点从查询树移除,以确保查询树的正确执行.

算法中的 threshold 控制着是否启用 HashMap 和 BloomFilter. threshold 为可变参数,可变化范围为(0,511],因为 OceanBase 的 IN 操作符支持的操作数上限不大于 511 组. 当子查询结果集不大于 threshold 时,直接将子查询结果集写入主查询的物理计划内,接下来的物理计划的执行等处理遵循 OceanBase 现有的查询处理流程. 当子查询结果集大于 threshold 时,将主查询的物理计划连同子查询结果集生成的 BloomFilter 一起发送至 ChunkServer 处理,MergeServer 利用子查询结果集生成的 HashMap 过滤以获取最终的结果集.

3.3 两阶段数据过滤

两阶段数据过滤:首先 ChunkServer 进行非严格的 BloomFiter 过滤,获得最终结果集的超集;其次 MergeServer 进行严格的 HashMap 过滤,获得最终结果集.

两阶段数据过滤如图 4 所示.

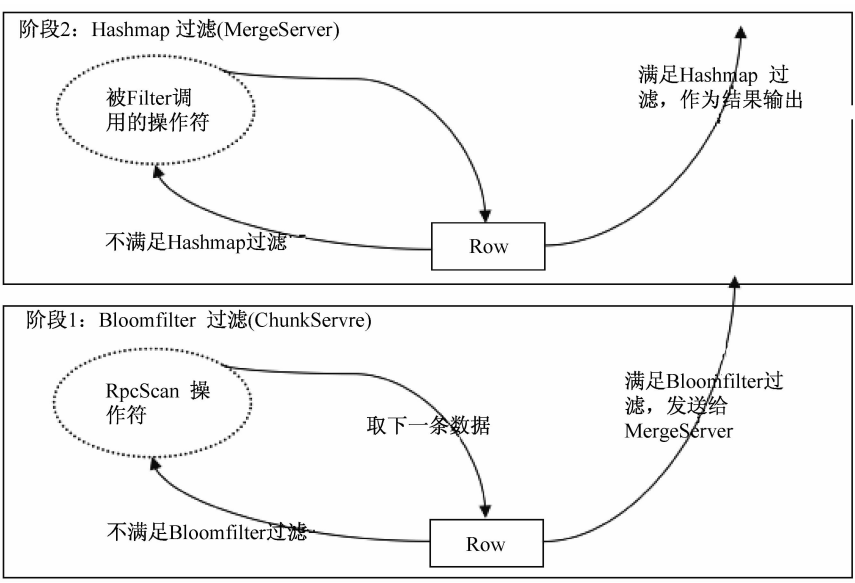

图 4 两阶段数据过滤

Fig. 4 Two-phase data filtering

ChunkServer 进行数据表扫描时,每读取一行,都执行 BloomFilter 检查,检查通过则发送至 MergeServer,否则继续读取下一行,直至读取完毕. MergeServer 对 ChunkServer 发送来的每一条数据,都执行 HashMap 过滤,将过滤生成的结果返回给用户. 因为 BloomFilter 的固有的误报特性,ChunkServer 发送给 MergeServer 的结果集是包含最终结果集的超集,因此 MergeServer 必须进行一次严格过滤,去除误报记录,获取最终结果.

3.3.1 BloomFilter 过滤

分布式架构下,将作为主查询过滤条件的超大的子查询结果集分发至不同的数据节点的方案会占用大量传输带宽. 为了降低带宽占用率且加速查找,嵌套查询策略使用了一种多哈希函数映射的快速查找数据结构——布隆过滤器. 相较于其他的数据结构,布隆过滤器在空间和时间方面都有巨大的优势,特别适合于海量数据集的表示和查找.

策略所构建的 BloomFilter 采用如下的公式:

$$k = -\ln(p) \div \ln(2),$$

$$m = (n * k) \div \ln(2).$$

其中 p:误判率. m:位数组大小. n:总数据数目. k:所需哈希函数数目.

BloomFilter 的构建由 MergeServer 负责,构建算法如下.

Input:子查询结果集 S

①依据上述公式及 S、误报率 p(默认),计算 BloomFilter 所需位数组大小 m,所需哈希函数数目 k.

②读取 S 的一条记录 R,如果 R 为 NULL,转⑤.

③将 R 依次带入 k 个哈希函数 $H_1(R),\ldots,H_k(R)$得到 k 个值 $V_1,\ldots,V_k$.

④将 BloomFilter 的位数组的 $V_1,\ldots,V_k$ 位设置为 True,转②.

⑤构建结束,返回 BloomFilter.

BloomFilter 的查找算法如下.

①读入一条记录 R.

②将 R 依次带入 k 个哈希函数 $H_1(R),\ldots,H_k(R)$得到 k 个值 $V_1,\ldots,V_k$.

③比对 BloomFilter 的位数组的 $V_1,\ldots,V_k$ 位. 如果 k 个位全为 True,则返回查找成功,否则返回查找失败.

3.3.2　HashMap 过滤

由于 BloomFilter 的误报特性,MergeServer 得到的结果集是最终结果集的超集. 因此 MergeServer 必须进行严格的数据过滤,以获得最终结果集.

MergeServer 严格的数据过滤条件就是海量的子查询结果集. 如何组织子查询结果集,以提供高效的查找是一个关乎性能的重要问题. 在当今服务器普遍支持大内存的状况下,嵌套查询策略采用全内存的 HashMap 存储子查询结果集.

HashMap 的高效查找依赖于哈希函数的均匀散列和低冲突率. 均匀散列保证每一个桶内的数据检索时间大致相同;低冲突率保证快速定位. 本文设计的 HashMap 采用链表法解决地址冲突,链表的每一个节点只保存 key.

MergeServer 负责构建 HashMap,且利用构建的 HashMap 进行严格的数据过滤.

HashMap 的构建算法如下.

Input:子查询结果集 S

①初始化 HashMap,分配哈希桶空间.

②读取 S 的一条记录 R,如果 R 为 NULL,转⑤.

③将 R 带入哈希函数 $H(R)$,依据得到的哈希值确定待插入的哈希桶 BUCKET BT.

④将 R 以链表的形式挂在 BT 的链表末尾,转②.

⑤构建结束,返回 HashMap.

HashMap 的查找算法如下.

①读入一条记录 R.

②将 R 带入哈希函数 $H(R)$,依据得到的哈希值确定待查询的哈希桶 BUCKET BT.

③遍历 BT 内的链表节点,逐个比对. 如果相同则返回查找成功,否者返回查找失败.

查询树的每一个非叶子节点的执行都需要两阶段数据过滤,即首先根据孩子节点的结果集构建 HashMap 和 BloomFilter,接着将 BloomFilter 连同本节点的物理计划分发至数据

节点,数据节点依据物理计划及过滤条件 BloomFilter,将最终结果集的超集返回给 MergeServer,最后 MergeServer 利用 HashMap 执行最后的严格的数据过滤,获得最终结果集.

4 实验结果

OceanBase 单服务器部署.服务器由 1T 硬盘,16 G 内存,16 核 CPU,一块网卡组成.服务器操作系统是 Red Hat6.2,内核是 2.6.32-220.el6.x86_64.

4.1 实验一

该实验衡量小规模子查询数据集状况下嵌套 IN 子查询策略的性能.测试表 test,共计 100 万条记录,包含 id、name 共计两个字段,其中 id 为主键列.启用 BloomFilter 和 HashMap 的阈值设置为 20,即子查询结果集不大于 20 条.

测试 SQL 语句模板如下所示.

①:一层嵌套 SQL:Select count(*) from test where [id/name] in (select [id/name] from test Where id < ConstValue)

②:与①等价 SQL:Select count(*) from test where [id/name] in (ConstValue,ConstValue,...)

①和②等价:如果①的子查询结果和②绑定的 ConstValue 完全相同.

OceanBase 现有查询策略由于不支持子查询,为了和嵌套查询策略作对比,因此执行嵌套子查询的等价 SQL,即②形式的 SQL.

小规模子查询数据集下,有主键索引的 OceanBase 已有查询策略性能测试结果及嵌套查询策略性能测试结果如表 1 所示.嵌套查询 SQL 已转化为 OceanBase 支持的非嵌套 SQL.

表 1 小规模子查询数据集下两种策略的结果,有主键索引

Tab.1 Results of two strategies with small subquery dataset,primary key index

子查询结果集/条记录	OceanBase 现有策略耗时/s	嵌套 IN 子查询策略耗时/s
1	0.01	0.91
20	0.01	3.48
21	0.01	1.14
100	0.01	1.11
200	0.02	1.11
500	0.05	1.11

小规模子查询数据集下,无主键索引的 OceanBase 已有查询策略性能测试结果及嵌套查询策略性能测试结果如表 2 所示.

表 2 小规模子查询数据集下两种策略的结果,无主键索引

Tab.2 Results of two strategies with small subquery dataset,without primary key index

子查询结果集/条记录	OceanBase 现有策略耗时/s	嵌套 IN 子查询策略耗时/s
1	0.94	0.96
20	4.04	4.05
21	4.19	1.16
100	17.37	1.16
200	34.06	1.17
500	超时	1.17

表 1 和表 2 表明:嵌套 IN 子查询的性能虽然低于 OceanBase 现有的有主键索引的查询

性能,但是却远远高于 OceanBase 现有的非主键列的查询性能.

4.2　实验二

该实验衡量大规模子查询数据集状况下嵌套子查询策略的性能,实验环境同实验一. 大规模子查询数据集下的嵌套查询策略的性能测试结果及 Mysql5.1.52 的性能测试结果如表 3 所示.

表 3　大规模子查询数据集下嵌套查询策略的结果

Tab.3　Results of nested query strategy with massive subquery dataset

子查询结果集/条记录	嵌套 IN 子查询策略耗时/s	Mysql/s
1	1.17	26.95
10	1.73	26.85
20	1.96	27.20
50	3.21	27.09
80	4.84	26.87
100	6.29	26.85

表 3 验证了嵌套 IN 子查询的高性能,在同等条件下,其耗时远远低于 Mysql 耗时.

5　总　　结

NoSQL 数据库内置复杂 SQL 语句查询功能是一个重要的趋势. 本文的主要目的就是提出一种查询策略,用于支持复杂 SQL 的子集——嵌套子查询,且改进 OceanBase 现有的查询策略.

嵌套查询策略通过构建查询树和查询引擎实现了嵌套子查询功能. 嵌套子查询策略还通过两阶段数据过滤,即 HashMap 过滤和 BloomFilter 过滤,改善了 OceanBase 现有的查询策略,并保证了查询的高效. 实验一和实验二结果表明,通过采用嵌套子查询策略,OceanBase 实现了对嵌套子查询功能的支持,同时其嵌套查询性能优于 Mysql 的嵌套查询.

当然,嵌套查询策略在小规模子查询数据集且有主键索引的条件下,相较于 OceanBase 现有策略有明显的性能差异,虽然嵌套查询语句的复杂性是一个重要因素,但是嵌套查询策略在查询优化方面的不足也是一个重要因素. 因此,未来的一个重要工作就是优化嵌套查询策略.

[参　考　文　献]

[1] KIM W. On optimizing an SQL-like nested query[J]. ACM Transactions on Database Systems (TODS), 1982, 7(3): 443-469.

[2] GANSKI R A, WONG H K T. Optimization of nested SQL queries revisited[C]//ACM SIGMOD Record. ACM, 1987, 16(3): 23-33.

[3] KIM T K, JUNG S H, KIM K R, et al.: HyperDB: A PC-based database cluster system for efficient OLAP query processing[C]//Proc of 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2007). Cambridge, USA (November 2007).

[4] KIM T K, OH S K, CHO W S, et al.: A lanlinux-based grid system for bioinformatics applications[C]//Proc. of International Conference on Advanced Communication Technology (ICACT). 2006,3:2187-2192.

[5] KIM T K, KIM K R, OH S K,et al. A Hybrid Grid and Its Application to Clustering Orthologous Groups for Multiple Genomes[C]//Proc International Symposium on Computational Life Science. 2006:20-10.

[6] KANG Y J, TIM T K, YANG K E, et al.: An OLAP query decomposition technique for PC-based database cluster systems[C]//Proc of the 18th IASTED International Conference on Parallel and Distributed Computing and

Systems (PDCS 2009). Innsbruck, Austria (February 2009).
[7] 申德荣，于戈，王习特，等. 支持大数据管理的 NoSQL 系统研究综述[J]. 软件学报，2013，8：008.
[8] POKORNY J. NoSQL databases: a step to database scalability in web environment[C]//The 13th International Conference on Information Integration and Web-based Applications & Services (iiWAS). 2011：278-283.
[9] BROWN A, WILSON G. The Architecture of Open Source Applications[M]. Lulu. com, 2011.

(责任编辑 李 艺)

(上接第 270 页)

[参 考 文 献]

[1] ZAHARIA M, CHOWDHURY M, FRANKLIN M J, et al. Spark: cluster computing with working sets[C]//HotCloud2010. USENIX Association Berkeley, CA:[s. n.], 2010：10-10.
[2] Spark[OL]. http://spark. apache. org/.
[3] Shark[OL]. http://shark. cs. berkeley. edu/.
[4] Spark SQL[OL]. http://spark. apache. org/sql/.
[5] BLANAS S, PATEL J M, ERCEGOVAC V, et al. A comparison of join algorithms for log processing in MaPreduce[C]//SIGMOD2010. New York: ACM, 2010：975-986.
[6] SAKR S, ANNALIU, FAYOUMI A G. The Family of MapReduce and Large-Scale Data Processing Systems[J]. ACM Computing Surveys (CSUR), 2013, 46(1).
[7] KARGER D, LEHMAN E, LEIGHTON T, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide Web[C]//STOC97. New York :ACM, 1997：654-663.
[8] DECANDIA G, HASTORUN D, JAMPANI M, et al. Dynamo: Amazon's highly available key-value Store[C]//SOSP2007. New York: ACM, 2007：205-220.
[9] XIN R S, ROSEN J, ZAHARIA M, et al. Shark: SQL and rich analytics at scale[C]//SIGMOD2013. New York: ACM, 2013：13-24.
[10] THUSOO A, SARMA J S, JAIN N, et al. Hive :a warehousing solution over a map-reduce framework[J]. PVLDB, 2009, 2(2)：1626-1629.
[11] OTHAYOTH R, POESS M. The making of TPC-DS[C]//VLDB2006. New York: ACM , 2006：1049-1058.
[12] POESS M, NAMBIAR R O, WALRATH D. Why you should run TPC-DS:a workload analysis[C]//VLDB2007. New York: ACM , 2007：1138-1149.

(责任编辑 李 艺)