文章编号: 1000-5641(2016)05-0036-09

基于 LSM Tree 的分布式索引实现

隆 飞, 翁海星, 高 明, 张 召

(华东师范大学 数据科学与工程研究院, 上海 200062)

摘要: 近年来 Log-Structured-Merge(LSM) Tree 在 NoSQL 系统中得到了广泛地应用. 主要是因为 LSM Tree 架构提出了延迟更新和批量写入的算法, 将随机写转换为批量写, 减少了磁盘臂的移动开销, 从而大大地提升了数据库的写入性能. 然而, 读性能却也因此受到影响. LSM Tree 和 B Tree 之间的本质区别使得 NoSQL 系统不适宜直接引用 B Tree 作为辅助索引结构. 本文实现了 LSM Tree 下的一种分布式辅助索引结构, 提出针对这种读写分离架构的索引批量加载策略, 并对 LSM Tree 的查询计划树进行了缓冲优化, 避免了重复的查询解析, 使得索引读的性能得到了相应的提升.

关键词: 辅助索引; 日志结构合并树; NoSQL

中图分类号: TP31 **文献标识码**: A **DOI:** 10.3969/j.issn.1000-5641.2016.05.005

Distributed secondary index based on LSM Tree

LONG Fei, WENG Hai-xing, GAO Ming, ZHANG Zhao

(*Institute for Data Science and Engineering, East China Normal University, Shanghai* 200062, *China*)

Abstract: In recent years, Log-Structured-Merge Tree has been widely used in NoSQL systems. This is mainly because it has proposed two algorithms: update delayed and batch write, convert random write to batch write, reducing the cost of moving the disk arm therefore the write performance of database has been enhanced greatly. However, the read performance of database has also been affected negatively. The essential difference between LSM Tree and B Tree makes NoSQL not suitable for using B Tree as index structure directly. This paper implements a distributed secondary index based on LSM Tree, and proposes a bulk loading method in this read and write separation architecture. We also do lots of works on the optimization of index query plan to avoid repeatly query parsing IO so that the performance of index read has been greatly improved.

Key words: Secondary Index; LSM Tree; NoSQL

收稿日期: 2016-05

基金项目: 国家 863 计划项目(2015AA015307); 国家自然科学基金(U1401256, 61402180, 61402177); CCF-腾讯联合研究基金(AGR20150114); 上海市自然科学研究基金(14ZR1412600)

第一作者: 隆 飞, 男, 硕士生, 研究方向为分布式数据库. E-mail: 451858158@qq.com.

通信作者: 张 召, 女, 副教授, 硕士生导师, 研究方向为数据库. E-mail: zhzhang@sei.ecnu.edu.cn.

0 引　　言

近些年, 伴随着数据采集设备的快速发展, 用户成为产生数据的主体, 产生的数据总量往往会达到 TB 甚至 PB 级别. 集中式数据库在处理这些数据时, 代价高昂. 因此, 水平扩展良好的 NoSQL 数据库应运而生, 并迅速发展起来. 大多数 NoSQL 系统都是基于 key-value 形式的存储架构, 例如 HBase[1], LevelDB, Apache Cassandra[2]等. 它们都是搭建于廉价的硬件设施之上, 往往适用于对数据库的伸缩性, 高性能, 高可用和低成本有更高要求的应用.

无论是传统的关系型数据库抑或是新型 NoSQL 数据库, 更高的数据查询速度始终是不变的追求. 典型的 NoSQL 系统只支持 key 上的查询操作, 对于 value 上的查询操作支持都不是很友好, 但许多应用需要快速查询 value 列的某一行, 因此它们对于 NoSQL 上的辅助索引的需求也越来越迫切. 为了满足 NoSQL 数据库对于辅助索引的需求, 本文实现了解决大规模数据量的分布式辅助索引功能, 设计了针对 Log-Structured-Merge (LSM)[3]体系的批量加载策略以解决基线数据上索引的构建任务, 通过使用索引列进行查询, 能够大大提高对数据表中非主键数据的查询性能.

分布式存储引擎下的索引实现对于 NoSQL 类系统有着重大的意义, 同时它的研究已经成为查询效率研究的热点之一. 例如, 华为公司实现在 HBase 上建立辅助索引 HIndex[4], Google Spanner[5]中支持高并发跨数据中心情况下满足事务一致性的索引, Chen et al[6]提出了一种云端类 DBMS 的索引框架, 华东师范大学数据工程研究院在阿里的数据库 Oceanbase 上实现辅助索引[7]等等.

本文的章节安排如下: 第 1 节首先介绍了背景及其研究现状, 包括LSM Tree 模型和其存储架构; 第 2 节描述了辅助索引的结构, 阐述了辅助索引的单点存储方式和分布式存储方式; 第 3 节是介绍了分布式存储架构下辅助索引的实现细节; 在第 4 节, 本文做了一些实验来验证辅助索引为查询带来的性能提升, 并针对本文对辅助索引所做的一些优化进行了实验验证; 第 5 节作为全文的总结并分析了未来工作.

1 基于 LSM-Tree 的分布式存储系统架构

LSM 模型诞生于 1996 年. 自 Google 公司的 BigTable[8]论文发表之后, LSM模型受到大量的关注. LSM 对于数据的变化采用的是延迟和批量写的算法, 将内存与磁盘中的数据级联在一起. 与传统的 B Tree 相比, LSM会先将一段时间内的插入操作缓冲在内存, 当内存达到阈值时, 再将内存中的数据和磁盘上的数据做合并, 并顺序写入一个或多个磁盘页. 这种顺序写相比于随机写, 能够明显地减少磁盘臂的移动, 会快很多; 因此 LSM Tree 能够提升数据库的写入性能. 比较适合 LSM-Tree 的应用场景是: insert 数据量大, 读数据量和 update 数据量不高且读一般针对最新数据, 例如历史库和日志文件. Diff-index[9]论文指出 LSM 是针对写作了优化的而B Tree 对读和写都只做了一些简单的优化.

LSM Tree 由许多位于磁盘上的树结构成分和一个驻留在内存当中的成分组成. 数据按照主键排序, 以 SSTable[8](Sorted String Table)的形式存储在磁盘组件中. 如图1所描述, 分布式 LSM 架构将数据分为增量数据和基线数据, 分别存储在更新服务器和基线服务器上. 图中包含一个内存组件和三个磁盘组件. 当客户端发起更新请求时, 首先会在磁盘上追加一行操作日志, 然后, 将更新的数据写入到 C0 Tree 当中. 当内存达到存储阈值, LSM Tree会发起合并流程. 为了取回数据的实际值, 读操作不得不对每个磁盘中的树结构上进行查询.

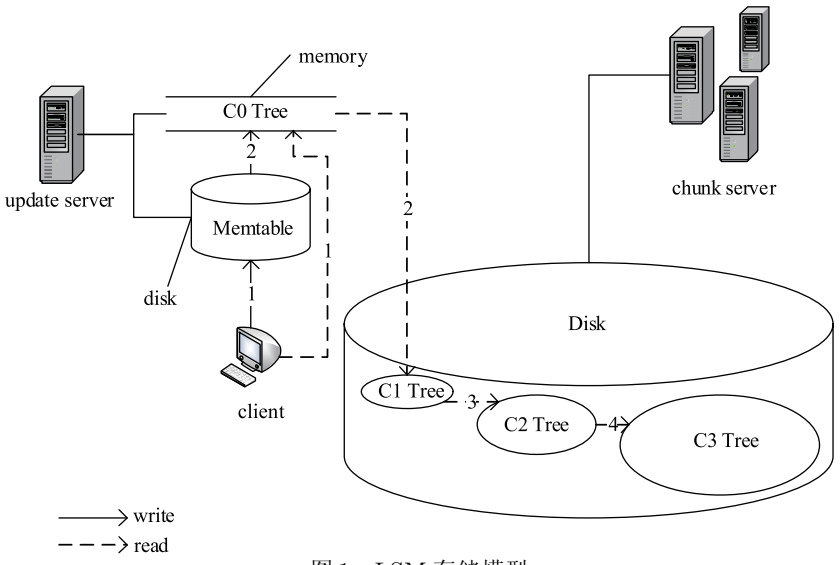

图 1 LSM 存储模型

Fig. 1 LSM storing model

在一些大规模数据分析的应用当中, 相对于整个数据库, 更新数据量往往不是很多. 对于这部分数据, 可以直接存放在内存当中. 如图2 所示, 在读写分离情况下, 内存数据往往存放在一个单独的服务器当中, 称为 UpdateServer(UPS), 对应LSM中的 C0 组件. 基线数据存储在 Chunkserver(CS)中. 根据应用的规模, 可以对 CS 水平扩展. 在 CS 上实现 LSM 的不同级别的存储. Rootserver(RS)是整个集群的管理者, 管理集群中的所有服务器, 数据分布, 元数据信息以及副本信息. 为保证 RS 的高可用性, 通常RS采取主备架构, 主备之间强一致. MergeServer(MS), 是 SQL 解析模块, 产生查询计划, 并将每个 Tablet 的读取请求发送到相应的 CS. CS 首先读取磁盘中包含的基准数据, 接着请求 UPS, 获取相应的增量数据, 并将基准数据与增量数据融合后得到最终结果, 返回给 Client[10].

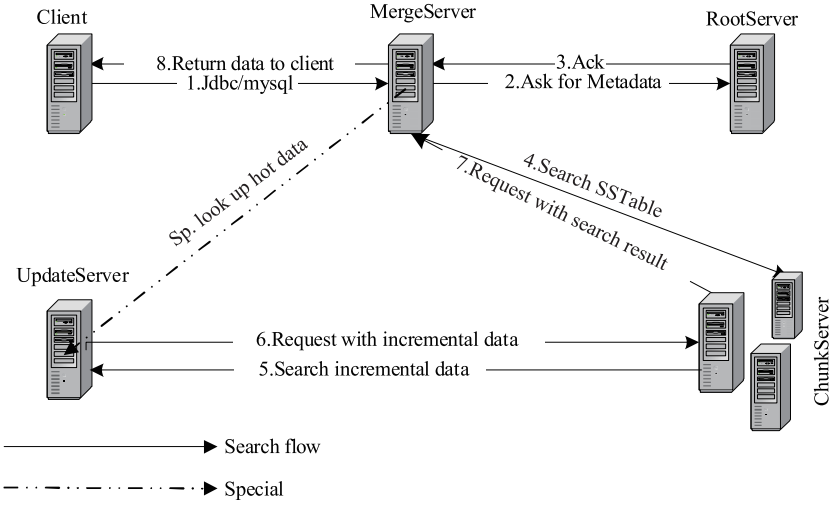

图 2 分布式存储架构图

Fig. 2 Distributed system architecture

2　辅助索引结构描述

2.1　索引存储结构

对于许多 NoSQL 系统而言, 数据是以 key-value 对的形式组织在磁盘当中的. Key-value 对是最简单的非基本数据模型之一, 一些其他更复杂的数据模型则都是在它的基础之上建立的. 由此本文定义了数据的存储模式: $D = \{t|t = \langle \text{key}, \{\text{values}\}\rangle\}$. 此模式将数据库定义为元组的集合, 元组以⟨key, {values}⟩的形式存储在磁盘上, 并按照 key 值对数据进行排序, values 是多个 value 的集合. key 类似于传统数据库中的主键, 不允许重复, 因此 key 所在列构成 NoSQL 系统中的主索引.

随着越来越多的应用采取 NoSQL 系统作为数据管理系统, 而这些应用的负载大多是以 value 列为查询列, 因此主索引无法满足应用的需求, 建立 value 列上的辅助索引的需求也越来越强烈. 传统的辅助索引定义形式如下: $M = \{t|t = \langle \text{search\_key}, \text{pointer}\rangle\}$. 传统的辅助索引包括 BTree 索引和 Hash 索引等, 都是将索引组织在内存当中, 并以 pointer 的方式定位到磁盘中的数据. 这种方式对处理海量数据的分布式数据库并不适用[11], 主要原因有: 维护代价大; 制约了数据库的水平扩展性等.

本文提出一种基于读写分离式 LSM 架构的索引结构, 其存储模式如下所示: MD={D_mem+D_disk|D_mem={⟨search_key+primary_key,{covering}⟩}, D_disk={⟨search_key+primary_key,{covering}⟩}}. 索引分为两个部分: 位于 UPS 中 D_mem 部分和位于磁盘中的 D_disk 部分. 其存储结构中摒弃了传统索引的指针部分, 而将 search_key 和 primary_key 作为索引存储结构的联合主键. 与传统数据库类似, 使用冗余列来实现覆盖索引(covering index), 避免二次查询操作, 从而提高查询效率.

在实现中, 索引表按照联合主键水平切分成多个 tablet. 每个 tablet 按照一定的策略均匀分布到集群中的 CS 上, 满足基线和增量数据分离特性, 并由 RS 执行负载均衡. 实验证明这种适应于架构的索引实现方法可以很好地满足非主键列查询需求.

必须强调的是, 本文的实现方法与传统的索引实现差别较大, 但是能够满足索引高可扩展性以及高可用性的需求. 从本质上说, 索引表就是存储在数据库中的另一张数据表. 图 3 中给出存储的一个示例, 辅助索引表 Si1(Secondary index 1)将索引列和原数据表 T1(Table

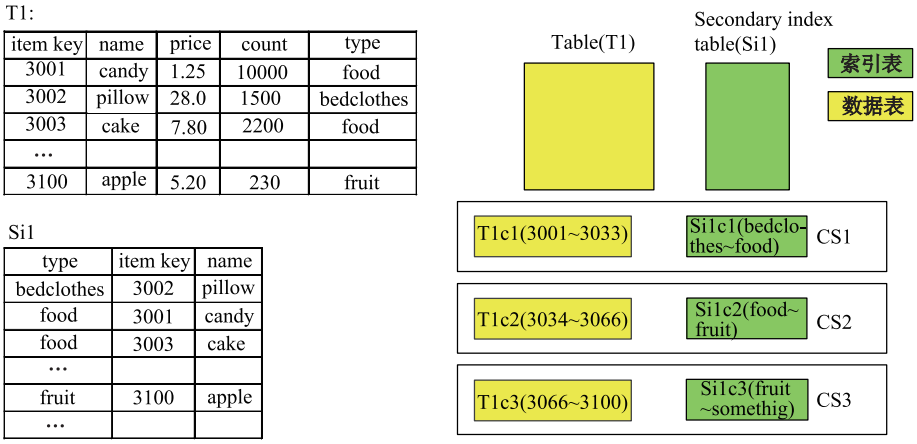

图 3　索引表存储模型

Fig. 3　Storage model of secondary index

1)的主键列作为自身的联合主键, 同时添加一些附加列信息. T1物理上按照主键列进行排序, 并水平切分成为 T1c1, T1c2 和 T1c3 三个部分, 分别存储在CS1, CS2 和 CS3上. Si1 按联合主键在磁盘上重新排序, Si1c1, Si1c2 和 Si1c3 是其对应的水平切片. T1 的第一主键是 itemkey, 而 Si1 的第一主键是 type.

辅助索引表构建时, 并不构造出整张表. 而是直接在每个 CS 上直接构造辅助索引的每个部分. 并将构造完成的每个 SSTable 的主键范围信息汇报给 RS, 然后由 RS 发起拷贝副本的操作.

2.2 索引特征分析

存储代价: 分布式数据库中引入索引对原来的系统的影响主要体现在更新数据表时带来的额外代价和索引存储的额外代价. 然而, 索引表一般只存储数据表的某些列, 其存储空间远远小于数据表, 属于可接受范围. 因为更新操作均在 UPS 的内存中完成, 时间成本也远远小于磁盘读操作, 因此其对于性能的影响也可以接受.

可扩展性和弹性: 本文实现的索引功能支持水平扩展, 不会影响数据库的可扩展性. 通过增加集群当中的 CS 和 MS 的数量, 集群的整体性能"线性可扩展". 并且索引支持在集群运行过程当中, 动态增加结点, 弹性良好.

可用性**:** 创建索引表的时候可以指定索引表在集群中的副本数目. 在索引批量加载的过程中会在不同的 CS 上创建多份副本, 以保证索引数据的可用性. 目前, 在构建索引过程中, CS 发生故障会影响到索引的构建过程, 每日合并结束后, 索引状态不可用. 索引创建失败不会影响集群的正常使用.

3 索引实现及维护

索引实现分为两个部分, 一部分是索引构建完成后动态维护部分, 这部分内容比较简单, 实现时只需保证原数据表和索引表之间的数据一致性即可; 另一部分是对原表上已经存在的数据的处理, 必须将此部分数据同步到索引表中. 很自然的想法是将原数据表中的数据一条条插入到索引表当中, 但是此种方法效率难以接受. 考虑到索引表在磁盘中是以 SSTable 的形式进行存储的, 可以实现一种批量加载的方法, 构造出索引表数据的 SSTable. 批量加载效率很高, 但是难点在于如何收集分布在集群各个部分的 CS 上的数据, 也在于如何对收集的数据进行排序. 3.1 将介绍批量加载实现, 3.2 节简单介绍索引一致性模型.

3.1 基线服务器上索引的批量加载

3.1.1 批量加载实现

批量加载的难点在于对收集的数据进行排序, 为了尽可能避免全表扫描, 本文将排序分为局部排序和全局排序两个阶段. 批量加载过程充分利用架构特点, 在集群中的每个 CS 上单独进行, 可以避免大量数据迁移. 同时使用多线程技术, 并行处理数据表的不同 tablet 数据, 加快批量加载流程的速度.

在数据表上创建索引表的时候, 必须将数据表中已经存在的数据批量加载到索引表当中, 之后, 索引表才能够正常提供服务. 合并子模块定期将 UPS 中的增量数据合并到 CS 中, 论文[12]描述了一种具体的合并模块的工作流程, 称为每日合并模块, 采取排序合并算法进行数据合并. 因为在索引批量加载过程当中, 会阻断所有的 DDL 操作, 并且索引表批量加载时必须获得最新的数据表静态数据, 所以为了尽量减小索引构建过程对于整个系统的影响, 本文将索引的批量加载安排普通的每日合并流程之后. 批量加载主要分为以下四个阶段: 第一阶段, 进入批量加载的准备阶段. 为了使得索引表能够获得最新的数据, 我们在构建索引表

之前需要进行一次每日合并. 因此, 合并过程成为批量加载的准备阶段. 系统没有执行索引批量加载流程时, 所有索引构建线程被阻塞, 见算法 1.

算法1 批量加载算法—Schedule

```
输入: tablet信息或者range信息
输出: 索引构建信息
1.  for every thread in each CS do
2.      get_tablets_ranges()
3.      if (get nothing)
4.          pthread_cond_wait(&cond_, &mutex_);
5.      else if(get tablets)
6.          Call construct_local(tablet); //完成数据tablet准备, 进入局部阶段
7.      else
8.          Call construct_global(range); //收到Range信息, 进入全局排序阶段
9.      end if
10. end for
```

其次, 进入局部索引构建阶段. 如算法 2 所示, 当普通的每日合并完成之后, 便开始索引的静态数据部分的构建. 当 CS 接收到了 RS 的创建索引的信号之后, 获得一个数据表的 tablet, 对这个 tablet 按索引列进行排序, 并写到一个局部的 sstable 中. 完成以上步骤之后, CS 向 RS 汇报局部索引 sstable 的信息.

算法2 批量加载算法－局部排序(construct_local(type& tablet))

```
输入: 局部阶段开始信号
输入: 局部有序SStable文件
1.  get_local_index_handler //唤醒工作线程
2.  for each local index handler do
3.      create new sstable //用于存储排好序的SSTable
4.      sort_.get_next_row(row) //排序
5.      Add information into index_reporter_
6.      Calc column checksum for data
7.      tablet->add_local_index_sstable //SSTable暂存在数据表的tablet上
8.      if (success)
9.          Report information to RS
10.     else
11.         Push this into local failed list
12.     end if
13. end for
```

第三阶段, 进入全局索引构建阶段. 如算法 3 所示, RS 接收到了 CS 发过来的采样信息之后, 将索引列划分为若干个 range, 尽量使得每个 CS 上的 tablet 的数量相等, 以达到负载均衡. RS 将切分后的 range 信息发送给 CS. CS 根据这个 range 信息互相之间拉取数据. 这个阶段完成后, 每个 CS 便完成了全局的索引构建. 最后一个阶段是索引表批量加载完成阶段. 这个阶段

完成复制全局索引备份, 检查数据校验和, 修改索引表状态为可用.

算法 3 批量加载算法–全局排序(construct_global(type& range))

```
输入: 全局阶段开始信号, RS发送过来的Range
输出: 全局有序SSTable文件
1.  get_global_index_handler
2.  for each global index handler do
3.      start_agent //agent从其他CS或自己上获取某个range内的数据
4.      execute get_data plan
5.          root_.get_next_row() //调用local_agent_和interactive_agent_ 获取数据
6.      do column checksum
7.      construct_index_tablet_info //构造索引表的tablet信息
8.      close_sstable //saving sstable in disk
9.      if (success)
10.         release sstable //释放局部阶段SStable信息
11.         do data checksum, modify index status
12.         report information to RS
13.     else
14.         Push this into global failed list
15.     end if
16. end for
```

3.1.2 范围切分算法

整个批量加载过程中涉及到 CS 和 RS 信息的交互以及 CS 之间的数据拉取. 每台 CS 在完成索引构建局部排序阶段之后, 进行 Range 信息采样, 并汇报给 RS, 采样信息以等高直方图的形式记录. RS 根据采样信息进行 Range 划分, 决定每个CS全局排序的范围. 样本数的设计会影响 Range 切分的均衡性, 样本数越高, Range 切分越均衡. 但是样本数目太多, 采样过于频繁影响性能, 并且网络传输代价也会随之增大. 因此本文实现将样本数定义为 tablet 的数目, 并且设置一个最大值, 当 tablet 数目超过最大值时, 样本数被设置为最大值.

在算法 4 中, max_bucket_num 是预设采样等高直方图桶的数目, 值为 256; tablet_num 是该数据表对应的数据分片的数目, sample_num 是样本数. 采样时, 每隔N行取出此行的主键信息作为 Range 的 End key, N=(tablet 总行数/sample_num). 每个 CS 独立完成采样, 并将采样信息汇报给 RS. RS 接收到所有的来自 CS 的汇报信息之后将切分 Range 信息. 首先 RS 将汇报信息做一个排序, 见算法 4 的第三行; 其次对 Range 进行切分, 算法 4 的九到十三行描述了 RS 切分 RANGE 详细过程.

算法4: RS Range 切分算法

```
输入: 来自 CS 的汇报信息 ReportInfo
输出: 切分结果集Res
1. key=decode ReportInfo
2. if all_reported is true and decoded successfully
3.     sort all range info from ReportInfo //对汇报结果进行排序
4. end if
5. sample_num=min(max_bucket_num,tablet_num) //计算样本数目
6. for iter in [SortedReportInfo.begin(), SortedReportInfo.end()) do
7.     temp range count++
8. end for
9. range step=temp range count/sample num //RS计算切分步长
10. divide range count=(temp range count % range step==0) ? sample num:sample num+1
11. for (int I=0; I< temp range count; I+=range step) do
12.     Res[I].put(key)
13. end for
14. return Res
```

3.2　索引缓存方式调整

LSM Tree架构下的索引与传统的索引比较, 其读操作需要一次内存访问和至少一次磁盘访问, 因此速度会比传统的索引读操作慢一些.

在很多应用场景下, 应用程序处理的语句几乎都是相同, 而变化的都是 where, set 或者 values 后面的变量的具体数值.

针对这种场景, 缓存索引查询的计划树, 动态调整查询参数是非常有意义的. 实现时, 为每个索引查询分配一个 sql_id, 将<sql_id, query_plan>以 hashmap 的形式存储在内存当中. 相同的 sql_id 可以在内存中获取相同的查询计划, 避免了重复的查询解析过程.

4　实验结果与分析

4.1　实验环境

本文在阿里巴巴集团开源的分布式数据库系统 OceanBase[13]上实现了辅助索引功能, 以下简称 OB. OB 是一款增量数据和基线数据分离的 LSM 架构的分布式数据库产品, 主要是为了解决淘宝网面临的大规模数据存储和管理问题. OB 具有良好的可扩展性, 能够实现了数千亿条记录、数百 TB 数据上的跨行跨表事务[14].

本文实验集群搭建在4台服务器上. 每台服务器的硬件环境如下: 64-bit 2.00 GHz 24核Intel(R) Xeon(R) CPU E5-2620, 132 GB 内存, 千兆以太网, 磁盘3.7TB, 操作系统CentOS release 6.5 (Final). 将 RS 和 UPS 配置在同一台服务器上, 另外三台服务器均配置为 MS 和 CS, 数据库的副本数目设为 3.

实验的目的在于验证本文实现的辅助索引对于提高查询速度的带来的贡献, 以及验证本文在索引上所做的一系列优化的效果.

4.2　实验结果及分析

实验 1 使用雅虎的标准测试工具 YCSB(Yahoo！Cloud Serving Benchmark)[15]进行测试, 并选择工作负载C, 这个工作负载是 100% 的读操作, Query 执行总次数为 10 万次, 为了与索引读返回列数相同, readallfields 设置为 false. 本文对 YCSB 进行了扩展, 使得 YCSB 核心负载集能够进行非主键列查询. 为简化实验, 本文重新定义数据生成方式, 保证索引列于 PRIMARY_KEY 是一对一的关系. 实验采用数据集为1000万行, 使用 YCSB 实现的 JDBC 数据库层接口逐行导入. 在未添加索引时, 利用 YCSB 对 OB 非主键进行查询, 超出 YCSB 响应时间限制, 未得出实验数据. 实验 1 可以从侧面反映出辅助索引实现后对于查询的性能提升. 考虑到索引的具体实现, 本文将原表索引查询与原表主键查询进行对比, 结果如图 4.

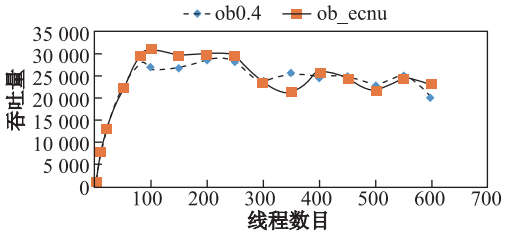

图 4　YCSB 只读查询性能对比

Fig. 4　Only-read performance using YCSB

可以看到, 索引查询速度基本和主键查询速度一致. 两组实验分别在 80 和 100 个线程数时吞吐率达到最优, 这是因为本文所做的实验使用 YCSB 连接到一个 MS 上. 单台 MS 对于连接数目有一定限制, 当超过这个限制的时候, 线程在 MS 发生冲突, 并排队等待, 性能不会随着线

程数目上升. 实验可以得出结论: 索引查询性能能够达到主键查询的性能, 而未实现索引查询之前, 非主键列查询无法完成.

实验 2 测试使用一张商品表 item, 主键为 itemkey, 在 type 属性上建立索引, count 是表中不被索引表覆盖的一列. 数据集大小为 1 000 万行, 891MB. 测试中使用SQL如下所示:

Query1: select itemkey from item where type =

Query2: select itemkey from item where type = ?

Query3: select count from item where type =

Query4: select count from item where type = ?

参数使用随机数产生, 保证一定命中数据库中的数据. ? 表示此版本的数据库实现了索引查询计划的缓存.

基于辅助索引的实现策略, 可以将索引查询分为两类: 回表和不回表, 具体是指查询是否需要从索引表对应的数据表中取出数据. 如图5中所示, NotBack_prepared 代表的是不回表的查询并且会缓存索引查询执行计划(Query2), Notback_Notprepared 代表的是不回表且不会缓存执行计划的查询(Query1), Back_prepared 代表的是回表且缓存执行计划的查询(Query4), Back_Notprepared 代表的是回表且不会缓存执行计划的查询(Query3). 分别比较回表和不回表情况下, 缓存索引查询计划对于性能的提升. 回表情况下性能提升了 6 倍左右, 不回表的情况下性能提升在 3∼7 倍之间.

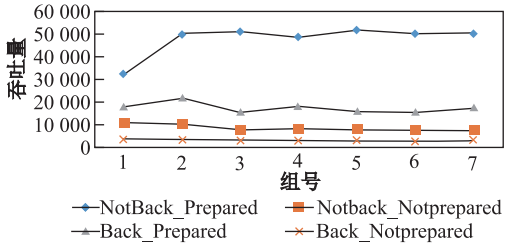

图 5 缓存执行计划性能对比

Fig. 5 Execution plan performance

5 结 语

主索引和辅助索引对于数据库查询优化有重要的意义. 现在绝大多数流行的 NoSQL 数据库一般只支持到主索引, 但是越来越多的应用要求非排序列上的查询操作. 本文根据读写分离架构存储体系的特点设计实现分布式的二级索引方案. 此方案符合一般的 NoSQL 架构, 可以很容易在迁移到HBase等其他架构类似的系统中. 在索引构建过程中, 遇到各种问题时故障恢复工作, 以及分布式索引缓存优化是本文未来将要考虑的工作.

[参 考 文 献]

[1] APACHE ORG. Apache HBase[EB/OL]. [2016-07-07]. https://hbase.apache.org/.

[2] LAKSHMAN A, MALIK P. Cassandra: A decentralized structured storage system[J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.

[3] O'NEIL P, CHENG E, GAWLICK D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[4] HUAWEI. Secondary index in HBase[EB/OL]. [2016-07-07]. https://github.com/Huawei-Hadoop/hindex.

(下转第 66 页)