流数据 Top-K 关键字查询算法

郑诗敏 秦小麟 刘 亮 周 倩
(南京航空航天大学计算机科学与技术学院 南京 210016)

摘 要 基于 Spark Streaming 计算框架的分布式 Top-K 关键字查询是统计流数据中所有关键字的热点研究问题。多数研究通过限定存储空间来实现 Top-K 关键字查询，并假设关键字集合已知。针对这个问题，提出一种可应用于关键字集合未知情况的分布式 Top-K 关键字查询算法，根据监测到的关键字动态地调整存储空间，通过更新策略的优化提升其精度。实验结果表明，该算法的性能在关键字集合未知的情况下比现有算法更优。

关键词 Top-K 关键字查询，流数据，云计算，Spark Streaming
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.8.030

Algorithm for Top-K Keyword Query in Data Streams

ZHENG Shi-min QIN Xiao-lin LIU Liang ZHOU Qian
(College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

Abstract Distributed Top-K keyword query based on the framework of Spark Streaming is a hot research issue. It is used to count all keywords in data streams. Most studies of Top-K keyword query limit storage space and assume that the keywords set is known. To solve this problem, we presented a distributed Top-K keyword query algorithm which can be used in cases where the keywords set is unknown. This algorithm dynamically adjusts the size of storage space according to monitored keywords and optimizes the updated strategy to improve precision. Experimental results show that the proposed algorithm under the condition of unknown keywords set has better performance.

Keywords Top-K keyword query, Data streams, Cloud computing, Spark streaming

1 引言

随着互联网技术的飞速发展及新闻、博客、社交网络的兴起，网络给人们生活带来方便的同时，也带来了数据的爆炸式增长。信息时代的到来，尤其在社交应用的普及之后，每个人每天都在生产数据。在海量数据的时代背景下，人们更关注的是数据中的价值。如何在大数据中快速提取有用的信息，是目前大数据的主要问题之一。Top-K 关键字查询在社交网络、搜索引擎等诸多应用领域中是一种极其常见的查询类型[1,2]。由于海量的流数据中蕴含的关键字集合庞大，对每个词进行统计不切实际且没有意义。Top-K 关键字查询算法能够使用少量的存储空间监测流数据中的关键字，同时根据用户的查询条件，返回出现频数最高的前 k 个关键字。

在许多实际的应用场景中，关键字集合往往是未知的。例如微博，每个用户发送的内容中出现的关键字未必存在于现有的字典中，新的关键字随着社会事件而不断地产生，因此无法预知关键字集合。在搜索引擎的热门搜索中，由于用户检索的随机性，同样有新的关键字出现。在这些场景中，关键字集合大小并非确定的，而是动态变化和未知的。许多限定存储空间来实现 Top-K 关键字查询的算法相继被提出，但在关键字集合未知的情况下，算法的精度和性能还不尽如人意。流数据中关键字集合的大小是影响精度和性能的主要因素之一。经典 Top-K 关键字查询算法[3,4]使用固定大小的存储空间来进行频数的统计和更新，每监测到一个新词时，如果其在存储结构中已经存在，则进行增量操作；反之，则根据不同的策略替换已有的关键字。由于存储空间的限定，当关键字集合庞大时更新的过程将带入误差，直接影响到 Top-K 的结果。因此在经典算法中，通常假设数据呈倾斜分布且关键字集合已知，才能够保证 Top-K 的查询结果具有较高的精度和有序性。但在大数据背景下，关键字集合的大小通常是未知的。同时经典算法未考虑数据划分、合并方法等问题，从而导致算法应用到分布式环境后精度和性能下降。

为了解决上述问题，本文提出一种基于 Spark Streaming 计算框架的分布式 Top-K 关键字查询算法。该算法首先对数据进行划分，将相同的关键字划分到同一个分区，然后每个分区根据监测到的关键字动态地调整存储空间，并通过设置检查点使 Top-K 的结果不引入误差。最后提出合并检查点的方法，保证 Top-K 关键字查询结果具有较高的精度。

本文第 2 节介绍 Top-K 关键字查询和已有的一些相关工作；第 3 节介绍分布式 Top-K 关键字查询算法；第 4 节给出

到稿日期：2015-07-02 返修日期：2015-09-18 本文受国家自然科学基金项目(61373015,61300052)，江苏高校优势学科建设工程资助项目(PAPD)，江苏省重大科技成果转化基金项目(BA2013049)资助。

郑诗敏(1990－)，男，硕士生，主要研究方向为云环境下数据查询处理，E-mail：shiminzheng@nuaa.edu.cn；秦小麟(1953－)，男，教授，博士生导师，主要研究方向为分布式数据管理与安全；刘 亮(1985－)，男，博士后，讲师，主要研究方向为传感器网络数据库、流数据库等；周 倩(1983－)，女，博士生，主要研究方向为网络安全、传感器网络等。

实验结果及分析；最后总结全文，并指出下一步的研究方向。

2 问题定义和相关工作

2.1 Top-K 关键字查询定义

将流数据记为 $S=\{q_1,q_2,\cdots\}$，关键字集合记为 $O=\{e_1,e_2,\cdots\}$，其中 $q_i\in O$。Top-K 关键字查询：给定整数 $k>0$，时间范围 $I=[t_s,t_e]$，返回在 I 内 S 中出现次数最高的 Top-K 关键字集合，记为 $R=Query(k,I)$。该集合包含两类关键字：G 和 A。G 表示在 I 中出现频数最高的关键字，A 则表示出现的频数近似最高的关键字。当 $|G|=k$ 时，Top-K 查询结果完全准确。

该查询适用于实时性要求高、无须返回精确结果的场景。所提出的 Top-K 查询算法返回的集合中，$|G|$ 近似于 k，A 中的关键字属于或近似接近 Top-K 关键字。

2.2 Spark Streaming 简介

Spark Streaming[5] 是 UC Berkeley AMP 实验室提出的一种基于内存的分布式实时计算框架。Spark Streaming 将输入的实时流数据划分成多段的数据（Discretized Streams，Dstreams），每一段数据存储为分布式弹性数据集（Spark RDD）。然后从 Dstreams 的转换操作中生成对 RDD 的转换操作，执行产生的中间结果可以存储在内存中以进行迭代计算或存储到磁盘中。图 1 展示了 Spark Streaming 的总体架构。

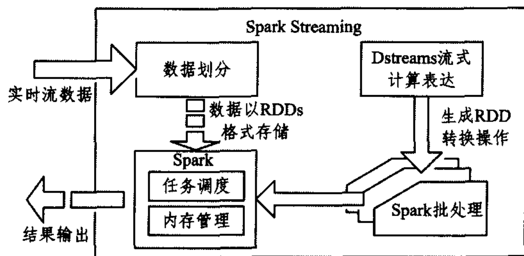

图 1 Spark Streaming 系统总体架构

Spark Streaming 拥有 Mapreduce[6] 的优点，不同点在于 Job 的中间结果可以保存在内存中，从而不需要反复读写 HDFS，因此其性能效率高于 Mapreduce。Mapreduce 适用于大规模的离线计算[7,8]，但在流数据实时计算中，往往不能满足性能上的需求；而 Spark Streaming 在实时性、处理速度方面具有突出的性能优势，因此得到了广泛的应用[16-19]。

2.3 相关工作

流数据 Top-K 关键字查询最初产生于互联网广告领域，该问题可以简单抽象为：在关键字集合大小为 N 的流数据 S 中，查询出现频数最高的 k 个关键字。针对这样的问题，许多研究者提出了相关的方法。这些方法可以分成两类：1）Counter-based方法；2）Sketch-based 方法。

Counter-based 方法使用固定数量的 counters 来监测流数据中的关键字，其中 counters 的数据结构形如 (e,f,Δ)，e 表示关键字，f 表示 e 的频数，Δ 表示 f 的最大可能误差。当监测到 counters 中的关键字时，相对应的 counters 就更新，否则不作任何处理或者替换当前 counters 中的关键字。文献[9]提出了 Sticky Sampling 和 Lossy Counting 算法，该算法虽然简单、直观，但需要扫描所有的 counters 才能返回Top-K 的结果。Demaine 针对 Hot Items 问题提出频繁项算法[10]，该算法使用大小为 k 的 counters 来监测关键字。当 counters 中的关键字出现时，相对应 counters 的频数加 1，否则所有 counters 的频数减 1。当 counters 中的某个关键字的频数等于 0 时，它将被新的关键字替代。尽管算法效率比较高，但精度受数据分布和 counters 数量的影响较大。文献[4]提出了 SpaceSaving 算法，该算法用大小为 m 的 counters（称为 Summary 的数据结构）来监测流数据 S 中不同的关键字。当监测到新关键字时，替换当前 counters 中频数最低的关键字 e_j，然后置其频数为 $count_j+1$ 和最大可能误差 $count_j$。算法能够保证 Top-K 查询具有较高的性能。但在关键字集合比较庞大时，固定的存储空间使得算法精度比较低。

Sketch-based 方法与 Counter-based 方法不同，其通过哈希对所有的关键字进行近似估计。每个关键字通过哈希函数映射到 counters，对每次产生的冲突做相应的更新。由于哈希碰撞，使得关键字频繁度的估计精度降低。文献[3]提出的 Count Sketch 算法解决了 FindApproxTop$(S,k,\in)$ 问题，准确度为$(1-\delta)$。文献[11]提出的 Group Test 算法主要针对 Hot Items 问题，它出错的概率为常量 δ。Group Test 算法准确率高，但是其空间复杂度大且不能保证结果的有序性。文献[12]提出的 Multistage filters 方法与 Group Test 算法相似。Sketch-based 方法对关键字做估计，使用的存储空间不受关键字集合大小的限制。然而，每次冲突都会产生高额的开销，并且该类方法不能保证查询结果的准确性。

在两类方法中，Sketch-based 方法每次插入的代价比较高，而 Counter-based 方法更新简单，更适用于大规模的实时数据。文献[13]比较了现有的经典算法，结果表明 SpaceSaving算法具有较高的性能，在找出 Top-K 关键字的同时，能通过记录最大可能误差来保证结果的有序性；但算法精度在关键字集合未知时较低。

3 分布式 Top-K 关键字查询算法

现有 Top-K 关键字查询算法在关键字集合未知或在分布式环境中，整体的精度和性能较低。针对这问题，提出基于 Spark Streaming 框架的分布式 Top-K 关键字查询算法（Time Supported Top-K Term Query，TSTop-K）。TSTop-K 改进了 SpaceSaving 算法，先对数据进行划分，将相同的关键字划分到同一个分区，然后提出 Summary 动态更新的策略和设置检查点的方法，保证 Top-K 关键字中不引入误差（$\Delta=0$）。针对 Top-K 关键字查询 Query(k,I)，首先在每个分区中查找 I 内产生的检查点，然后将其合并产生局部结果，最后将每个分区的局部结果合并，返回最终查询结果集$G\cup A$。图 2 描述了 TSTop-K 算法的执行流程。

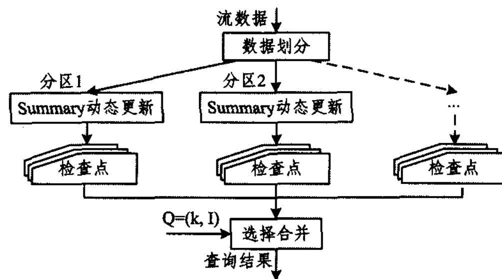

图 2 TSTop-K 算法的执行流程

3.1 数据划分

云环境中，数据划分的基本原则是尽可能地将数据均匀划分到各个机器节点上。但流数据中的关键字数量、分布都

是未知的，采用抽样的方法不能很好地解决这个问题，同时增加了额外的开销。算法在统计关键字的过程中，为了实现分布式并行化，最直观的方法是对数据进行分区。TSTop-K 采用散列函数的方法，将具有相同哈希值的关键字划分到同一个分区。

通过数据的分区，使得不同分区中的关键字不存在相同项，这样可以避免多个分区的 Top-K 关键字结果合并时可能出现的错误。在图 3 中，白色部分代表每个分区存储的Top-2 关键字，阴影部分代表舍弃的关键字。两个分区合并后的 Top-2 结果为{*a. count*=5,*b. count*=2}，而正确的结果为{*a. count*=5,*c. count*=3}。数据划分之后可以避免这种错误的发生，同时达到并行化的目的。

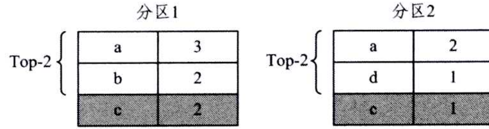

图 3　不同分区存储的 Top-2 关键字

3.2　动态 Summary 更新策略和检查点的设置

现有算法一般采用静态 Summary 作为存储结构，Summary 中包含固定数量的 counters。在关键字集合已知的情况下，可以人为选择适量的 counters，从而保证算法具有较高的精度和性能。但在关键字集合大小未知时，使用少量 counters 的静态 Summary 的算法精度较低。比较直观的方法是使用大量的 counters，然而由于不确定使用的数量或者使用过多的存储空间，使得该方法无法解决问题。因此提出一种动态 Summary 的更新策略，即根据监测到的关键字动态调整 Summary 的大小，保证 Top-K 结果在不引入误差的同时，又能节省大量的存储空间。

动态 Summary 更新策略允许算法在开始时以少量的 counters 初始化，随着监测到的关键字增多，当更新过程影响到 Top-K 结果精度时，动态地扩充 counters 数量。算法 1 描述了每个分区关键字的更新过程。

算法 1　Insert (element e_i)

输入：关键字 e_i

输出：插入操作状态

```
1. T ← monitored terms，error ← false；
2. if e_i∈T then
3.    count_i←count_i+1；
4. else
5.    if |T|<size then //size 为 counters 的数量
6.       T←T∪{e_i}；
7.    else
8.          e_j←min_{e_j∈T} count_j；
9.          if count_k<count_j+1 then //count_k 代表 T 中第 k 大的频数
10.            if error|times≥maxTimes then //counters 中带入误差
11.               return false；
12.            size←size+k；//增加 k 个 counters
13.            times←times+1；
14.            T←T∪{e_i}；
15.         else
16.            count_i←count_j+1；
17.            Δ_i←count_j；
18.            T←T∪{e_i}\{e_j}；
19.            error←true；
20. if count_k<count_i&Δ_i>0 then
21.    rollback；
22.    return false；
23. else
24.    return true；
```

T 代表 counters 数量为 *size* 的 Summary，当监测到的关键字 e_i 属于 *T* 时，相应的 $count_i$ 加 1；反之，如果不属于且|*T*|< *size*，那么将关键字加入到 *T* 中，并置其频数为 1 及置最大可能误差 Δ_i 为 0(第 2—6 行)。当|*T*|=*size* 时，找出 Summary 中频数最小的关键字 e_j 和按频数降序后的第 *k* 个关键字 e_k。通过比较两者之间频数的关系，判断关键字 e_i 替换现有的关键字后是否会将误差带入到 Top-K 的结果中。如果会，那么判断是否需要动态扩充 counters 的数量，判断的依据是 *T* 中所有 counters 的最大可能误差为 0 且扩展次数小于上限 *maxTimes*，若满足，则为 *T* 动态扩充 *k* 个 counters(第 8—14 行)；如果不会，那么采取 SpaveSaving 算法的更新策略，同时置 *error* 为真，代表 *T* 中引入了大于零的最大可能误差(第 16—19 行)。Summary *T* 中可能存在关键字 e_i，满足 $count_i \leq count_k$ 和 $\Delta_i>0$，当更新过程中增加 e_i 的频数时，可能会出现 $count_i>count_k$ 的情形，使 Top-K 结果出现误差，在这种情况下算法采取回滚的方法(第 20—21 行)。

定理 1　给定 Summary *T*，动态扩充 counters 数量，不影响现有 counters 的频数和最大可能误差。

证明：动态扩充的条件是 *T* 中任意的 counters 最大可能误差都为 0 且扩充次数小于上限 *maxTimes*。因此在扩充时，counters 中记录的是关键字精确的出现次数，新增加的 counters 并不影响已监测到的关键字。

当一个关键字的更新影响到 Top-K 结果，同时 *T* 中的 *error* 为真或到达 *maxTimes* 上限时，不采取扩展 Summary 的方式。如果进行扩展，那么新插入的关键字将影响现有的 counters，出现数据不一致的情形。为了解决这个问题，提出设置检查点的方法。该方法保存当前 Summary 的状态，然后用一个大小为 *size*+*k* 的空 Summary 重新监测关键字。

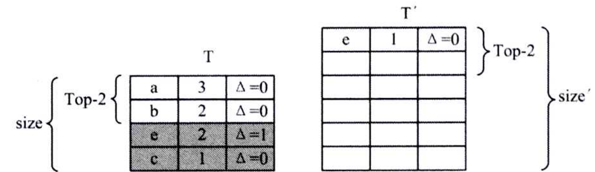

图 4　检查点的设置流程

假设初始用大小为 2 的 Summary 支持 Top-2 查询，给定关键字序列为{*a*,*a*,*b*,*a*,*b*,*c*,*d*,*e*,*e*}，当监测到 *c* 时，Summary 动态扩充增加了 2 个 counters。当监测到第一个 *e* 时，Summary *T* 的状态如图 4 所示。当监测到第二个 *e* 时，如果直接更新 *e* 的频数，那么 *e* 的频数为 3，最大可能误差为 1，这时误差将带入到 Top-2 的查询结果中。因此，在有可能影响到正确结果时，将采用设置检查点的方法来避免。在图 4 中，监测到第二个 *e* 时，保存 Summary *T* 的状态(不包括阴影部分)，同时用大小为 *size*′的空 Summary *T*′来替换，并将回滚的 *e* 插入到 *T*′中。设置检查点的条件：1)Summary 中有非零的最大可能误差 Δ，新插入的关键字影响到了 Top-K 的结果；2)由时间粒度决定，例如时间粒度为 1min，那么每隔 1min 设置一

个检查点，查询 Query(k,I)中的时间范围 I 大于或等于算法设定的时间粒度。

定理 2 $\forall e_i \in Summary$，当 $i \leqslant k$ 时，$\Delta_i = 0$。

动态扩展 Summary 的方法使得算法在关键字集合未知时可以以少量 counters 初始化，通过监测到的关键字动态扩充，提高了算法的精度。但不断扩充必定会造成不必要的浪费，因此提出缩减 Summary 的方法。当新的时间区间开始时，假如上一时间段(例如上一分钟)没有生成检查点，并且最近的 Summary 中无误差的关键字($\Delta=0$)的个数大于两倍的 k 值，那么在新的 Summary 中减少 k 个 counters 的数量。

3.3 Top-K 查询算法

采用动态的 Summary 更新策略之后，在监测关键字的过程中，会产生许多的检查点。这些检查点保存的 Summary 可以进行合并，同时能够保证 Top-K 查询结果的正确性和有序性[14]。当接收到查询 Query(k,I)后，在每个分区 partition s 中找到时间范围 I 内产生的 Summary 集合，然后将属于同一个分区的 Summary 集合合并得到局部的查询结果，最后将每个分区的结果合并得到最终的 Top-K 关键字查询结果。算法 2 描述了查询处理的过程。

算法 2 Query(k,I)

输入：k 值和时间范围 I

输出：Top-K 结果集

```
1. part-K←∅;
2. for each partition s∈S do
3.   partitionSummaries←find(k,I);
4.   part-K←part-K∪partitionMerge (k,partitionSummaries);
5. return Merge (part-K);
```

每个分区保存的 Summary 包含了不同的关键字集合，简单地合并分区中的 Summary 可能会产生图 5 中的错误。图中白色部分表示检查点保存的部分，阴影为舍弃的部分。Top-2 查询的结果为{$a.count=8$,$b.count=3$}，而实际的正确结果为{$a.count=8$,$c.count=4$}。

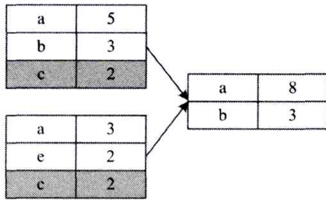

图 5 Summary 的合并

出现上述问题的原因是舍弃部分的关键字影响到了 Top-K 的结果，从而无法判断 Top-K 的准确性。针对这个问题，提出增加 k 值上限的方法，即在更新过程中实际取的 k 值为 k'，满足 $k'=k+1$。例如计算 Top-2 时，检查点中的 Summary 保存 Top-3 的结果，多使用一个 counter 作参考。该方法可以将 Top-K 查询结果集划分成 G 和 A 两个集合，G 代表 Top-K 关键字，A 代表近似 Top-K 关键字，其中 $|G|$ 近似于 k。算法 3 描述了每个分区合并 Summary 集合的过程。

算法 3 partitionMerge (k,SummarySet S)

输入：k 值和需要合并的 Summary 集合 S

输出：分区 Top-K 结果集和有序状态

```
1. T←∅,order←true;
2. for each s∈S do
3.   for each e∈s do
4.     if e∈T then
5.       T[e].count←T[e].count+e.count;
6.     else
7.       sum_{k+1}←0;
8.       for each s'∈S do
9.         if e∈s' then
10.          continue;
11.        sum_{k+1}←sum_{k+1}+s'.count_{k+1};
12.      e.count←e.count+sum_{k+1};
13.      e.Δ←sum_{k+1};
14.      T←T∪{e};
15.  boundary←boundary+s.count_{k+1};
16. K←T 中 Top-K 关键字;
17. e_{k+1}←T 中第 Top-k'个关键字
18. for each e∈K do
19.   if e.count≥boundary&e.count−e.Δ≥e_{k+1}.count then
20.     G←G∪{e};
21.   else
22.     A←A∪{e};
23.     order←false;
24. return (G∪A,order);
```

计算任意关键字 $e \in s$ 的频数和最大可能误差需要遍历 Summary 集合 S。如果 Summary 中含有关键字 e，那么累加 e 的频数；反之，关键字 e 的频数和最大可能误差同时加上 Summary 中第 $k+1$ 个关键字的频数(第 2−14 行)，*boundary* 代表 Top-K 之外关键字最大可能的频数。当任意关键字 $e \in K$ 满足条件时，关键字 e 属于 Top-K 关键字，反之为近似 Top-K 并且 Top-K 查询结果的排列顺序可能存在误差(第 19−23 行)。

Top-K 查询需要将每个分区产生的 Top-K 关键字进行合并。由于 3.1 节采用了数据划分的方法，因此在合并分区结果时不会出现图 5 中的问题。算法 4 描述了合并所有分区 Top-K 关键字查询结果的过程。

算法 4 Merge (SummarySet S)

输入：需要合并的 Summary 集合 S

输出：Top-K 结果集

```
1. T←∅;
2. for each s∈S do
3.   for each e∈s do
4.     T←T∪{e};
5. K←T 中 Top-K 关键字;
6. return K
```

3.4 代价分析

下面对流数据 Top-K 关键字查询中的更新、合并算法分别做代价分析。

3.4.1 更新代价

在每个分区中执行 Insert 更新算法，监测到关键字后或者增加现有频数或者插入、替换。假设每个分区监测到的关键字数量为 n，那么更新过程中的时间复杂度为 O(n)。Summary 会在更新的过程中动态地扩展，最多扩展 *maxTimes* 次，其中每次增加 k 个 counters，因此在最坏情况下的空间复杂度为 $O((maxTimes+1)k)$。

3.4.2 合并代价

假设分区中需要合并的 Summary 数量为 m，包含的关键

字集合大小为 n。算法 3 在合并时，对每个不同的关键字需要扫描一次分区中所有的 Summary，因此最坏情况下的时间复杂度为 $O(k(mn+1))$。在算法执行过程中，用集合 T 来存储不同的关键字，那么空间复杂度为 $O(n)$。假设分区数为 p，算法 4 在合并每个分区 Top-K 查询结果时，采用遍历统计的方法，总的时间复杂度和空间复杂度都为 $O(pk)$。

4 实验结果与分析

4.1 实验设置

实验在一个由 4 个节点组成的集群上进行，其中 1 个节点作为 master 节点，其余 3 个节点作为 worker 节点。节点配置如下：CPU：Xeon E5-2620（双核 2.00GHz），内存：8GB，硬盘：2TB，操作系统：CentOS6.4（64bit），节点上运行 spark-1.0.1 版本。

实验数据采用来自芬兰赫尔辛基大学的真实数据集 kosarak[13,15]。数据集中的每条记录由若干整数项组成，实验中将每一项作为关键字，以流的方式接入集群。数据集总共包含 8019015 项，其中不同的有 41270 项。为了测试性能并与其他算法进行对比，从数据集中划分出 4 个子集，数据量分别为 1M，2M，4M 和 8M，其中 M 代表百万个关键字。

4.2 结果分析

实验从 TSTop-K 算法的精度、存储开销、检查点数量、吞吐量和查询效率 5 个方面进行性能测试，并通过与 SpaceSaving算法进行对比，体现算法的优势。

（1）k 对查询精度的影响

图 6 示出了在不同 k 值条件下 TSTop-K 和 SpaceSaving 算法的精度对比。TSTop-K 算法初始的 counters 数量为 200，分区数为 4。从图 6 中可以看出，SpaceSaving 算法的精度随着 k 值的增大而降低，随着 counters 数量（图中用 m 表示）的增大而有所提高，而 k 值的变化对 TSTop-K 算法并没有显著的影响。当 $k=10$ 时，两种算法的精度都为 1；当 $k=50$ 时，SpaceSaving 算法的精度明显下降，m 值越小，精度越低。TSTop-K 算法由于采用了动态变化的策略，使得算法精度不受初始 counters 数量的影响，因此其精度高，表现出非常好的适应性。

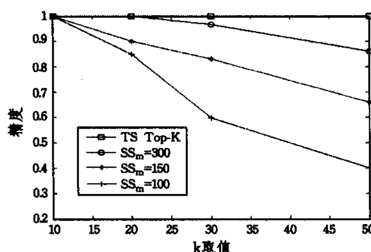

图 6 不同 k 值时 TSTop-k 与 SpaceSaving 算法的精度比较

（2）数据量对存储空间大小的影响

图 7 示出了在 $k=20$ 时数据量递增的情况下算法占用内存空间大小的对比，其中 wordCount 算法表示统计所有的关键字，并将其作为基准。从图 7 中可以看出，wordCount 算法的空间开销非常大，而其他两种算法所占用的内存空间较少。当数据量为 1M 时，wordCount 算法使用 25343 个 counters 的空间，当数据量为 8M 时，使用的数量上升至 41255 个，而 TSTop-K 算法所需的存储空间基本与 SpaceSaving（$m=300$）一致。因此 TSTop-K 算法除了精度高之外，所需的内存空间也较少。

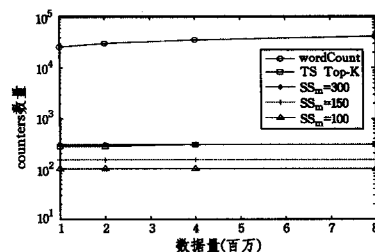

图 7 $k=20$ 时不同数据规模下所需 counters 数量的变化

（3）数据量对检查点数量的影响

图 8 示出了 TSTop-K 算法在不同 k 值下检查点数量随着数据量变化的情况。从图 8 中可以看出，随着 k 值的增长，检查点的数量也增多，但随着数据量的增大，检查点数量变化趋于平稳。当 $k=10$ 且数据量为 1M、2M 和 4M 时，检查点数量为 15，当数据量增长到 8M 时，检查点数量为 18，即数据量从 1M 增长到 8M，检查点数量只增加了 3 个。由于 TSTop-K 算法在运行开始时初始的 counters 数量较少，因此检查点数量较多。随着动态调整，检查点数量明显减少且趋于平稳。

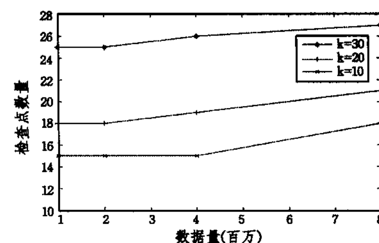

图 8 不同数据规模下检查点数量的变化

（4）k 对吞吐量的影响

图 9 示出在不同 k 值条件下 TSTop-K 算法处理关键字数量的变化情况。从图 9 中可以看出，每毫秒处理的关键字数量随着 k 值的增大而减少。当 $k=10$ 时，每毫秒处理 5019 个关键字，当 $k=50$ 时，每毫秒处理的关键字下降到 4202 个。吞吐量下降的原因是当 k 值增大时算法更新的代价随之增加。

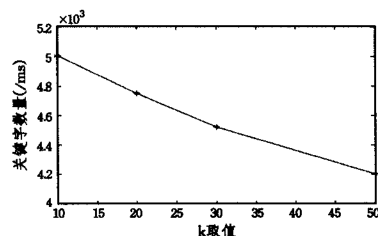

图 9 不同 k 值时吞吐量的变化

（5）时间范围对查询效率的影响

图 10 示出了在查询时间范围不同时算法的查询效率。实验中时间粒度设定为 1min。

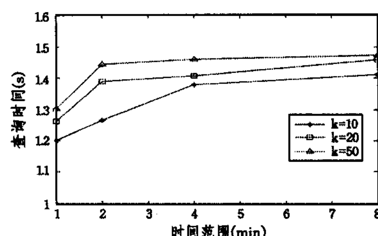

图 10 不同时间范围查询时算法的性能

从图 10 中可以看出，当 k 增大时，查询时间随之增加；当 k 一定时，随着时间范围的增大，所需的查询时间也随之增

加，且查询时间趋于平稳。当 $k=10$ 时，查询 1min 内的 Top-K 关键字需要 1.2s，查询 8min 内的关键字则需要 1.4s。查询时间先是增加而后逐渐平稳，原因在于算法执行初期，生成的检查点数量较多，查询时需要合并的代价较大，但随着检查点数量的平稳，查询的时间也趋于稳定。

结束语 本文研究分布式环境下流数据 Top-K 关键字查询的问题，由于海量的数据和未知庞大的关键字集合，使得现有的算法精度较低。在 Spark Streaming 框架的基础上提出了分布式 Top-K 关键字查询算法 TSTop-K，该算法采用动态 Summary 的更新策略，通过设置检查点的方式，提出相应的合并方法，使得在关键字集合未知时 Top-K 查询结果具有较高的精度。实验表明 TSTop-K 算法能很好地支持实时性强、精度要求高的 Top-K 关键字查询，具有实用价值。在性能对比上，TSTop-K 算法能动态调整存储空间的大小，使之性能和精度高于经典的算法。下一步工作将研究 Top-K 时空关键字查询算法。

参考文献

[1] Chen L, Cong G, Cao X, et al. Temporal spatial-keyword top-k publish/subscribe[C]// 2015 IEEE 31st International Conference on Data Engineering. IEEE, 2015: 255-266

[2] Zheng K, Su H, Zheng B, et al. Interactive top-k spatial keyword queries[C]// 2015 IEEE 31st International Conference on Data Engineering. IEEE, 2015: 423-434

[3] Charikar M, Chen K, Farach-Colton M. Finding Frequent Items in Data Streams[J]. Theoretical Computer Science, 2004, 312 (1): 1530-1541

[4] Metwally A, Agrawal D, Abbadi A E. Efficient Computation of Frequent and Top-k Elements in Data Streams[C]// International Conference on Database Theory. Springer-Verlag, 2005: 398-412

[5] Zaharia M, Das T, Li H, et al. Discretized streams: fault-tolerant streaming computation at scale[C]// Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 423-438

[6] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Commun. ACM (CACM), 2008, 51(1): 107-113

[7] Ci Xiang, Ma You-zhong, Meng Xiao-feng. Method for top-K query on big data in cloud[J]. Journal of Software, 2014, 25(4): 813-825(in Chinese)
慈祥，马友忠，孟小峰. 一种云环境下的大数据 Top-K 查询方法[J]. 软件学报, 2014, 25(4): 813-825

[8] Song Jie, Hao Wen-ning, Chen Gang, et al. Research of Distributed ETL Architecture Based on MapReduce[J]. Computer Science, 2013, 40(6): 152-154(in Chinese)
宋杰，郝文宁，陈刚，等. 基于 MapReduce 的分布式 ETL 体系结构研究[J]. 计算机科学, 2013, 40(6): 152-154

[9] Manku G, Motwani R. Approximate frequency counts over data streams[C]// Proceedings of the 28th International Conference on Very Large Data Bases. 2002: 346-357

[10] Demaine E D, Lopez-Ortiz A, Munro J I. Frequency estimation of internet packet streams with limited space[C]// Proceedings of the 10th Annual European Symposium on Algorithms. 2002: 348-360

[11] Cormode G, Muthukrishnan S. What's hot and what's not: tracking most frequent items dynamically[J]. TODS, 2005, 30 (1): 249-278

[12] Estan C, Varghese G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice[J]. ACM Trans. Comput. Syst., 2003, 21(3): 270-313

[13] Manerikar N, Palpanas T. Frequent items in streaming data: An experimental evaluation of the state-of-the-art[J]. DKE, 2009, 68(4): 415-430

[14] Agarwal P K, Cormode G, Huang Z, et al. Mergeable summaries [C]// PODS. 2012: 23-34

[15] Frequent Itemset Mining Dataset Repository, University of Helsinki, 2008[OL]. http://fimi.cs.helsinki.fi/data

[16] Kim J M, Park Y T. Scalable OWL-Horst ontology reasoning using SPARK[C]// International Conference on Big Data and Smart Computing. IEEE, 2015: 79-86

[17] Armbrust M, Xin R S, Lian C, et al. Spark sql: Relational data processing in spark[C]// Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015: 1383-1394

[18] Lin C Y, Tsai C H, Lee C P, et al. Large-scale logistic regression and linear support vector machines using Spark[C]// 2014 IEEE International Conference on Big Data. IEEE, 2014: 519-528

[19] Akgün B. Streaming Linear Regression on Spark MLlib and MOA [C]// Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015. ACM, 2015: 1244-1247

(上接第 127 页)

[7] Cannière C D, Rechberger C. Finding SHA-1 Characteristics: General Results and Applications [M]// Advances in Cryptology-ASIACRYPT 2006. Berlin Heidelberg: Springer-Verlag, 2006: 1-20

[8] Cannière C D, Mendel F, Rechberger C. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search [M]// Selected Areas in Cryptography. Berlin Heidelberg: Springer-Verlag, 2007: 56-73

[9] Grechnikov E A. Collisions for 72-step and 73-step SHA-1: Improvements in the Method of Characteristics: Report 2010, 413 [R/OL]. Cryptology ePrint Archive, http://eprint.iacr.org/2010/413.pdf

[10] Grechnikov E A, Adinetz A V. Collision for 75-step SHA-1: Intensive Parallelization with GPU: Report 2011, 641 [R/OL]. Cryptology ePrint Archive, http://eprint.iacr.org/2011/641

[11] Adinetz A V, Grechnikov E A. Building a collision for 75-round reduced SHA-1 Using GPU Clusters[M]// Euro-Par 2012 Parallel Processing. Berlin Heidelberg: Springer-Verlag, 2012: 933-944

[12] Sugita M, Kawazoe M, Perret L, et al. Algebraic Cryptanalysis of 58-Round SHA-1 [M]// Fast Software Encryption. Berlin Heidelberg: Springer-Verlag, 2007: 349-365

[13] Pramstaller N, Rechberger C, Rijimen V. Exploiting Coding Theory for Collision Attacks on SHA-1 [M]// Cryptography and Coding. Berlin Heidelberg: Springer-Verlag, 2005: 78-95

[14] Joux A, Peyrin T. Hash Functions and the (Amplified) Boomerang Attack[M]// Advances in Cryptology-CRYPTO 2007. Berlin Heidelberg: Springer-Verlag, 2007: 244-263