Improving Large-Scale Fingerprint-Based Queries in Distributed Infrastructure

Shupeng Wang[1], Guangjun Wu[1(✉)], Binbin Li[1], Xin Jin[2], Ge Fu[2], Chao Li[2], and Jiyuan Zhang[1]

[1] Institute of Information Engineering, CAS, Beijing 100093, China
wuguangjun@iie.ac.cn

[2] National Computer Network Emergency Response Technical Team/Coordination Center of China (CNCERT/CC), Beijing 100031, China

Abstract. Fingerprints are often used in a sketching mechanism, which maps elements into concise and representative synopsis using small space. Large-scale fingerprint-based query can be used as an important tool in big data analytics, such as set membership query, rank-based query and correlationship query etc. In this paper, we propose an efficient approach to improving the performance of large-scale fingerprint-based queries in a distributed infrastructure. At initial stage of the queries, we first transform the fingerprints sketch into space constrained global rank-based sketch at query site via collecting minimal information from local sites. The time-consuming operations, such as local fingerprints construction and searching, are pushed down into local sites. The proposed approach can construct large-scale and scalable fingerprints efficiently and dynamically, meanwhile it can also supervise continuous queries by utilizing the global sketch, and run an appropriate number of jobs over distributed computing environments. We implement our approach in Spark, and evaluate its performance over real-world datasets. When compared with native SparkSQL, our approach outperforms the native routines on query response time by 2 orders of magnitude.

Keywords: Big data query · Data streams · Distributed computing Memory computing

1 Introduction

In recent years, plenty of applications produce big and fast datasets, which are usually continuous and unlimited data sets [8,11]. Many applications have to analyze the large-scale datasets in real time and take appropriate actions [3,5,9,13,14]. For example, a web-content service company in a highly competitive market wants to make sure that the page-view traffic is carefully monitored, such that an administer can do sophisticated load balancing, not only for better performance but also to protect against failures. A delay in detecting a response error can seriously impact customers satisfaction [10]. These applications require a continuous stream of often key-value data to be processed. Meanwhile, the data is continuously analyzed and transformed in memory before it is stored on a disk. Therefore, current analytics of big data are more focused on research for efficient processing key-value data across a cluster of servers.

The fingerprints-based structures, such as standard bloom filter (SBF) and dynamic bloom filters (DBF) [7], are usually considered as an efficient sketching mechanism for processing the key-value data, and they can map the key-value items into small and representative structure efficiently by hash functions [1,4]. Fingerprint-based queries can be used as important tools for complex queries in big data analytics, such as membership query, rank-based queries, and correlationship queries [3]. Whereas when confronting large-scale data processing, the current sketching mechanism framework not only decreases the speed of data processing because of the fixed configured policies, but also increases collisions of fingerprint-based applications for dynamic datasets [2,6,7]. In this paper, we consider the problem of deploying large-scale fingerprint-based applications over distributed infrastructure, such as Spark, and propose an efficient approach to automatically reconfigure data structure along with the continuous inputs, as well as run a reasonable number of jobs in parallel two meet the requirements of big data stream analytics. The contributions of our paper are as follows:

1. We present distributed sketching approach, which mainly constitutes global dyadic qdigest and local dynamic bloom filters (DBFs) [7], which can provide capability of data processing with high system throughput. The global dyadic qdigest is constructed via collecting the minimal rank-based qdigest structure from local sketch and can be scaled efficiently. We can also tail the operations of distributed query processing by running an appropriate number of jobs over the local sites, which contain the result tuples.
2. We present detailed theoretical and experimental evaluation of our approach in terms of query accuracy, storage space and query time. We also design and implement our approach in Spark, and compare it with the state-of-the-art sampling techniques and the native system under production environments using real-world data sets.

Spark is often considered as a general-purpose memory computing system, which can provide capability of supporting analytical queries with fault-tolerant framework. We implement prototype of our approach in Spark, and compare it with the native systems (e.g., Spark) and general-purpose sketching method (Spark with Sampling) over real-world datasets. The experimental results validate the efficiency and effectiveness of our approach. Our approach only costs 100 ms for continuously membership queries over 1.4 billion records.

2 Approach Design

2.1 Sketch Design

The data structure of our sketch includes two parts: (1) A top level dyadic qdigest, which accepts items of out-of-order data stream and compresses them into a rank-based structure. We design a hashMap-based structure to implement the dyadic qdigest to improve the speed of the data processing. (2) A precise fingerprints structure, which map keys within a range of the dyadic qdigest into DBFs. Also, we design efficient query processing techniques to boost the performance of data processing over large-scale and dynamic datasets.

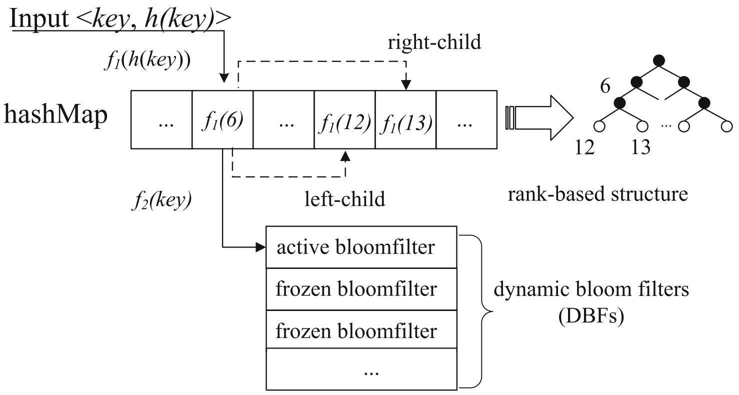


Fig. 1. Data structure for a local sketch.

We extend the traditional qdigest [12] to dyadic qdigest, which is used to divide the value space of input items into dyadic value range and compress them into a memory constrained rank-based structure for high-speed data streams processing. A dyadic qdigest can be considered as a logarithmic partition of space $[0, \phi - 1]$, and any interval $[a, b]$ can be divided into $\log_2(b - a)$ intervals. In order to improve the speed of data processing, we design a hash-based dyadic qdigest to maintain rank-based sketch structure, while maintaining the binary tree semantics of traditional qdigest [12]. Suppose the rank-based structure maintains summaries in value space $[0,\phi - 1]$. A node is represented by id, and the id is 1, iff. the node is the root node, otherwise, ids of leaf nodes equal to $value + \phi$. We can maintain a binary tree semantics for a node with id via following formulations:

1. Left child id of the node is $id \times 2$;
2. Right child id of the node is $id \times 2 + 1$;
3. Parent id of the node is $\lfloor id/2 \rfloor$.

The mapping between the hashMap and input item key can be computed by hash function f_1. A simple implementation is that we can compute the function via $f_1(h(key)) = h(key) \bmod \phi$, where h is a hashcode computing function such as MD5. For example, if $\phi = 8$, $f_1(h(key)) = 6$, the right child id is $2 * f_1(h(key)) = 12$ and left child id is $2 * f_1(h(key)) + 1 = 13$. Hence, we can compute the traditional binary tree structure over domain ϕ by hash-based computing easily.

When an item with key inserts into the sketch structure, we first calculate the node id of the item via $f_1(h(key))$, search the corresponding node by the id, and increase the number of items in the nodes. As with the DBFs, we arrange them at each node of the dyadic qdigest to record the fingerprints of input items, such that we insert key into the active filter of DBF through $f_2(key)$. The basic idea of DBFs is the filters array calculated with standard bloom filters (SBF), and the bloom filter that is currently written into is called active bloom filter. If the number of keys maintained at the active filter exceeds the threshold of collisions, we freeze the active filter as a frozen filter and create a new filter as an active filter.

In general, the query and inserting efficiency for a binary tree with M nodes is $O(\log M)$. While the hash-based implementation of a binary tree structure can improve the query time to $O(1)$. Also, it can be seen from Fig. 1 that the writing time of an item into DBF is $O(k)$, where k is the number of hash functions used in the active filter. At the same time, the above structure can also be compressed and scaled flexibly to meet the requirements of the distributed memory computing framework.

2.2 Query Processing

In a distributed computing infrastructure, the driver of query application usually deployed at query site and supervise the continuous queries over distributed local sites. In our approach, the large-scale fingerprint-based queries include two stages: *the initial stage*, at which we collect information from local sites and build the global dyadic qdigest at query site using constrained space; *query processing stage*, at which we send the query to local sketches which are related to a query in a batch mode guided by the global qdigest, and push-down the query processing into each local sketch. At the end of the stage we just exchange and collect the minimal information from local sites for final results. We next present the details of the two stages.

Recall that the local sketch structure, including dyadic qdigest and DBFs attached into each range (or called node) of the dyadic structure. At the initial stage of large-scale query processing, the query site collects the dyadic qdigest from local sketches and merge them into a global dyadic qdigest, which can consume constrained space. The operation can be conducted by pairwise merging between two sketches. Since the tree merging procedure just collects and merges between high-level dyadic qdigest summaries, such that the amount of data exchanged between local sites is small. At the final stage of the merging operation, we compress the union tree according to space-constrained parameters k by constrains (1) and (2). Note that the proposed framework can transform the continuously point queries into a batch-processing mode, and run reasonable number of jobs dynamically according to data distribution. Next, we present applications of large-scale fingerprint-based queries utilizing our approach.

Now, we need to formalize the input & output of a query and extract different attributes for sketching. The problem is to search the membership of a set of keys, and returns true or false for the key existence prediction. This query method is usually used in interactive queries or as tools for users interface. The dyadic qdigest extract the hashcode of *key*s from inputs, and predict the existence in the DBFs.

The DBFs based on SBF are compact and provide probabilistic prediction and may return true for an input key. The DBF also can provide false positive error, which reports true for some keys that are not actually members of the prediction. Let m, k, n_a and d be core parameter of a SBF as the previous literature [7]. The DBFs compress the dynamic dataset D into $s \times m$ SBFs matrix. In this paper, we use $f^{SBF}_{m,k,n_a,d}$ and $f^{DBF}_{m,k,n_a,d}$ to describe the false positive error of a SBF and DBFs when the d elements maintained in the sketch. If $1 \leq d \leq n_a$, it indicates that the number of elements in the set A does not exceed the collision threshold n_a, thus $DBF(A)$ is actually a standard bloom filter, whose false positive rate can be calculated in the same way as $SBF(A)$. The error $f^{DBF}_{m,k,n_a,d}$ of a membership query can depicted as

$$f^{DBF}_{m,k,n_a,d} = f^{SBF}_{m,k,n_a,d} = (1 - e^{-k \times d/m})^k. \tag{1}$$

If $d > n_a$, there are s SBFs used in $DBF(A)$ for data set A, meanwhile there are $i(1 \leq i \leq s-1)$ SBFs which contain n_a items in the filter, and false positive rates are all $f^{SBF}_{(m,k,n_a,n_a)}$. We design an active filter for the inserting, and false positive rate of the active filter in DBF(A) is $f^{SBF}_{(m,k,n_a,t)}$, where t is $d - n_a \times \lfloor d/n_a \rfloor$. Therefore, the probability that all bits of the filters in DBF(A) are set to 1 is $(1 - f^{SBF}_{(m,k,n_a,n_a)})^{\lfloor d/n_a \rfloor}(1 - f^{SBF}_{(m,k,n_a,t)})$, and the false positive rate of DBF(A) is shown in Eq. 2. We notice that when there is only one filter in DBF(A), Eq. 2 can transform into Eq. 1.

$$\begin{aligned} f^{DBF}_{m,k,n_a,d} &= 1 - (1 - f^{SBF}_{(m,k,n_a,n_a)})^{\lfloor d/n_a \rfloor}(1 - f^{SBF}_{(m,k,n_a,t)}) \\ &= 1 - (1 - (1 - e^{-k \times n_a/m})^k)^{\lfloor d/n_a \rfloor} \times (1 - (1 - e^{-k \times (d - n_a \times \lfloor d/n_a \rfloor)/m})^k). \end{aligned} \tag{2}$$

3 Experimental Evaluation

In this paper, we implement our approach in Spark, which works on the on YARN with 11 servers. The configuration information of a server of the cluster is shown in Table 1. We use the real-word traffic of page-views nearly 100 GB uncompress datasets. We mainly focus our evaluation on query efficiency and query accuracy of our approach.

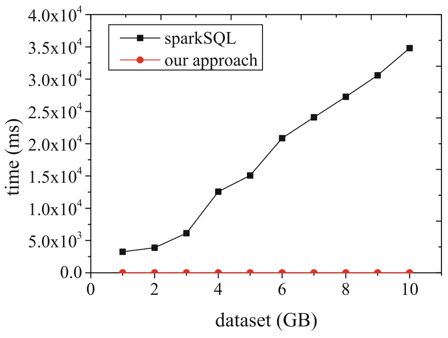
We first compare our prototype with the native system Spark. We use the rank-based queries as the testing cases. The large-scale quantiles query is a more important and complex statistical query method. For quantiles queries, Spark provides the *percentile* function to support quantiles lookups. We use the *percentile* function in SparkSQL to conduct the rank-based queries on Spark platform. The SparkSQL statement, such as “select percentile (countall, array

Table 1. Configuration information in our experiments.

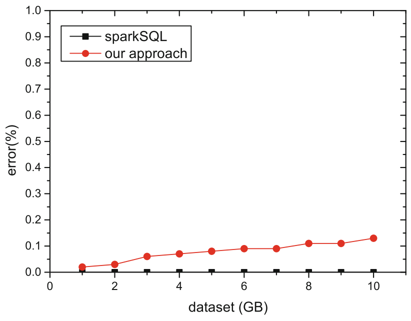
Name	Configuration Information
Operating system	CentOS 6.3
CPU model	Intel(R) Xeon(R) E5-2630@2.30 GHz
CPU cores	24×6
Memory	32 GB
Spark version	2.1.0
JDK version	1.8.0_45-b14
Scala version	2.11.7
Hadoop version	2.7.3

(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1)) from relations", can be used to obtain the ten quantiles from data set exactly. Meanwhile, the data structure of the two approaches, such as sketching structure of our approach, and related RDDs of Spark, are all kept in memory.

The results are shown in Fig. 2(a) and (b). The large-scale rank-based fingerprints queries need to scan the dataset to obtain the global ranking quantiles, which is a time-consuming operation. Our approach can improve the query efficiency through partition and distributed sketching framework, thus it can improve the query efficiency greatly. Under the 10 GB real-world data set testing, our approach can respond a query less than 200 ms, while the memory computing framework Spark costs 35 s to respond the same query.



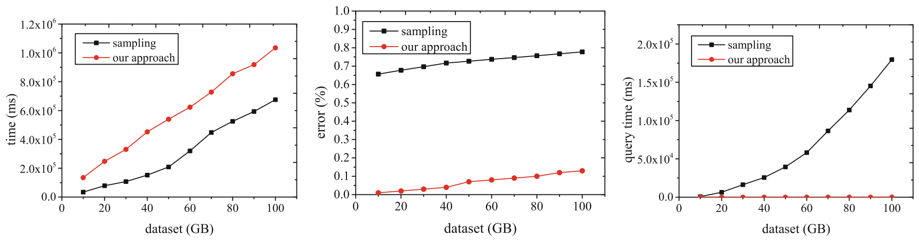
(a) query efficiency.



(b) query accuracy.

Fig. 2. Compared with native Spark.

In order to evaluate the query performance of our approach over large-scale datasets, we conduct the evaluation over 100 GB uncompressed data. When the data set is larger than the maximum space for RDDs in Spark, the data will be spilled into disk, and it impacts the query processing greatly. The Spark provides a *sampling* (denoted as Spark&Sampling) interface to conquer the real-time data processing over large-scale datasets. The Spark&Sampling is a type of Bernoulli sampling with no placement. The sampling method extracts samples from partitions and maintains samples in memory blocks. We improve the accuracy of the sampling method using adjustment weights for samples. For an input item v, we sample it with the probability of p, $p \in (0, 1)$, then the adjusted weight of the item is v/p. We configure the same memory usage for the two approaches, we compare them on three aspects: construction time, query efficiency and query accuracy. The experimental results are shown in Fig. 3(a), (b) and (c).



(a) construction time. (b) query accuracy. (c) query efficiency.

Fig. 3. Compared with Spark with sampling interface.

Our approach can extract samples from large-scale datasets and arrange them into specified sketching structure, while the Spark&Sampling extracts randomized samples from partitions and attaches them in block directly, thus the time costed in our approach is slightly higher than Spark&Sampling method.

After the sketch construction, we conduct 100 randomized rank-based fingerprints queries, and compute the average query efficiency and query accuracy of a query. The global sketch is a type of qdigest structure, which predicts the quantiles with some errors. We present the error comparisons of the two approaches. Our sketch can provide the error less than 0.1% for the rank-based quantiles estimation, while the error in Spark&sampling is larger than 0.7%. Meanwhile, our approach can improve the query response time significantly over large-scale datasets. When compared under 100 GB real-world dataset, our approach can provide an answer within 300 ms, while the Spark&sampling needs 180 s to respond the same query. Therefore, our approach is more appropriate for big and fast dataset processing and can provide real-time (or near real-time) response for large-scale fingerprint based queries

4 Conclusion

Large-scale fingerprint-based queries are often time-consuming operations. In this paper, we propose a distributed sketching framework to boost the performance of the large-scale queries, such as continuously membership queries and rank-based queries. Towards minimizing the shuffling time between the local sites, we construct a global dyadic qditest structure at query site using constrained space. For large-scale queries, the global sketch only build at initial stage of the queries, and can supervise the following continuously queries. This design enable our approach to run reasonable number of jobs under data skew scenarios. At local sites, we combine the sketching techniques, such as dyadic qdigest and dynamic bloom filters to build concise structure for dynamic data processing. The experimental results have expose the efficiency and effectiveness of our approach for large-scale queries over distributed environments. In the future, we plan to apply our approach in production environments to improve the efficiency of complex queries, such as group by queries and join queries.

Acknowledgment. This work was supported by the National Key Research and Development Program of China (2016YFB0801305).

References

1. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. ACM (1970)
2. Corominasmurtra, B., Sol, R.V.: Universality of Zipf's law. Phys. Rev. E Stat. Nonlinear Soft Matter Phys. **82**(1 Pt 1), 011102 (2010)
3. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: practically better than bloom. In: ACM International on Conference on Emerging NETWORKING Experiments and Technologies, pp. 75–88 (2014)
4. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Networking **8**(3), 281–293 (2000)
5. Wu, G., Yun, X., Li, C., Wang, S., Wang, Y., Zhang, X., Jia, S., Zhang, G.: Supporting real-time analytic queries in big and fast data environments. In: Candan, S., Chen, L., Pedersen, T.B., Chang, L., Hua, W. (eds.) DASFAA 2017. LNCS, vol. 10178, pp. 477–493. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55699-4_29
6. Guo, D., Wu, J., Chen, H., Luo, X.: Theory and network applications of dynamic bloom filters. In: IEEE International Conference on Computer Communications IEEE INFOCOM 2006, pp. 1–12 (2006)
7. Guo, D., Wu, J., Chen, H., Yuan, Y., Luo, X.: The dynamic bloom filters. IEEE Trans. Knowl. Data Eng. **22**(1), 120–133 (2010)
8. Guo, L., Ma, J., Chen, Z.: Learning to recommend with multi-faceted trust in social networks. In: Proceedings of the 22nd International Conference on World Wide Web Companion, WWW 2013 Companion, pp. 205–206. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2013)
9. Katsipoulakis, N.R., Thoma, C., Gratta, E.A., Labrinidis, A., Lee, A.J., Chrysanthis, P.K.: CE-Storm: confidential elastic processing of data streams. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015, pp. 859–864. ACM, New York (2015)
10. Mishne, G., Dalton, J., Li, Z., Sharma, A., Lin, J.: Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 1147–1158. ACM, New York (2013)

11. Preis, T., Moat, H.S., Stanley, E.H.: Quantifying trading behavior in financial markets using Google trends. Sci. Rep. **3**, 1684 (2013)
12. Shrivastava, N., Buragohain, C., Agrawal, D., Suri, S.: Medians and beyond: new aggregation techniques for sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, pp. 239–249. ACM, New York (2004). https://doi.org/10.1145/1031495.1031524, https://doi.org/10.1145/1031495.1031524
13. Wang, Z., Quercia, D., Séaghdha, D.O.: Reading tweeting minds: real-time analysis of short text for computational social science. In: Proceedings of the 24th ACM Conference on Hypertext and Social Media, HT 2013, pp. 169–173. ACM (2013)
14. Xiaochun, Y., Guangjun, W., Guangyan, Z., Keqin, L., Shupeng, W.: FastRAQ: a fast approach to range-aggregate queries in big data environments. IEEE Trans. Cloud Comput. **3**(2), 206–218 (2015). https://doi.org/10.1109/TCC.2014.2338325