BF-MapReduce :A bloom filter Based Efficient Lightweight Search

Zi-long Tan, Ke-ren Zhou, Hao Zhang, Wei Zhou
School of Software
Yunnan University
Kunming, China

***Abstract*—MapReduce is an attractive programming model for large-scale data-parallel applications. However, the original MapReduce framework also needs some optimizations to improve its performance. In this paper, we propose a novel bloom filter based lightweight MapReduce index (BF-MapReduce). Instead of scanning the whole dataset, our approach uses an auxiliary index to quickly skip unnecessary data segments, which can efficiently degrade the processing cost at map phase. Moreover, in order to deal with multi-dimension dataset, a converting schema is proposed. It can map multi-dimension data into one-dimension index. The experimental results show that our approach is efficient and lightweight. It can reduce the task running time dramatically with a little storage and maintenance cost.**

Keywords—MapReduce; Bloom Filter; Distributed data storage

I. INTRODUCTION

Recently, active research has focused on tuning the performance of MapReduce [1] based applications. Although many existing MapReduce systems have provided multi-objective optimizations for both distributed data storage and task scheduling, task running time varies depending on the input data and serving application which is often neglected by task designer. For example, it is commonly observed that there is conspicuous difference in the magnitude between input and output data for a search process, which results in, sometimes, merely a few records from large scale input. In order to perform the search, a naive MapReduce search task often examines the whole input data set to obtain desired items. This process is quite ineffective and time consuming.

A typical MapReduce task involves two steps: first, map input dataset to several map nodes so that each map node can concurrently process a segment. Next, output from all map nodes is collected and is then passed to reducer nodes. The reducer nodes combine the intermediate data and produce the final results. It is shown by the author [1] that many real-world applications can be described in such a manner. Of particular interest is the selection operation, which generally scans the whole input. Let us first consider a typical selection example: the map nodes select records from the input and left them to the reducer nodes to merge the result. Clearly, reducer node is optional since merging is not indispensable. The running time cost for such a task is primarily resides in the map nodes, because we know a task has to scan the whole input for desired items. Depending on the number of matches, the cost for reducer node varies. Since all matches need to be examined by reducer nodes, improving reducer node performance is sometimes difficult.

In this work, we propose a general index solution to improve the performance by effectively scheduling map nodes. Contrast to previous mapping strategies, our approach can skip unnecessary input segment by using a lightweight index. We demonstrate that this simple scheme can reduce the task running time dramatically with a little storage and maintenance cost.

The paper is structured as follows: Section 2 gives some related works. Section 3 analyzes the data model of Hadoop. Section 4 shows the design and analysis of bloom filter based auxiliary MapReduce index. Section 5 gives some experiment results and analysis. Finally, summarizes the conclusion.

II. RELATED WORKS

Internet applications face more challenges with the advent of terabyte-scale data. In order to process large-scale data effectively, distributed and parallel processing architectures are essential and of great values. Google propose MapReduce, which is an effective programming model that automatically parallelizes the computation across large-scale clusters of machines. Hadoop [2] is the open-source implementation of MapReduce, which widely adopts in Industrial and research. However, evidence shows that current MapReduce implementations seem to be inefficient. (e.g. require far more hardware than traditional relational databases to complete similar task.) Several methods have been proposed to promote the performance of MapReduce. In [3], Hiemstra and Hauff test new retrieval approaches with 12TB of data on MapReduce, showing that sequential scanning is a viable approach to running large-scale information retrieval experiments with little effort. A deep-in performance study is discussed in [4]. The authors identify five design factors that affect the performance of Hadoop, which is helpful for improving tasks' performance. Manimal [5] present an automatic optimization tool which can analyzes MapReduce programs and applies appropriate data-aware optimizations. Sandholm and Lai [6] use regulated dynamic prioritization to improve MapReduce's scheduling, which achieves 11-31% improvement in completion time. In [7], chen propose Tiled-

MapReduce (TMR), which partition a large MapReduce jobs into a number of small sub-jobs and iteratively processes one subjob at a time. This method leads to more efficiently use of resources. Meanwhile Bloom filter is also introduced to the optimization of MepReduce. Paper[9] adopt bloom filter solution to filter join operations and reduce network traffic. Paper[10] use bloom filter to get efficient graph similarity joins. Differ from all above works, our contribution is to setup a bloom filter based light weight auxiliary index and integrate it into MapReduce produce. A converting schema is proposed to deal with multi-dimension dataset.

III. HADOOP DATA MODEL ANALYSIS

As an open-source implementation of MapReduce, the Hadoop project offers a lot of additional features such as distributed file system (the HDFS), locality sensitive task scheduling and etc. It isolates the distributed file system layer from MapReduce task, so a task manipulates the file as if on a local file system.

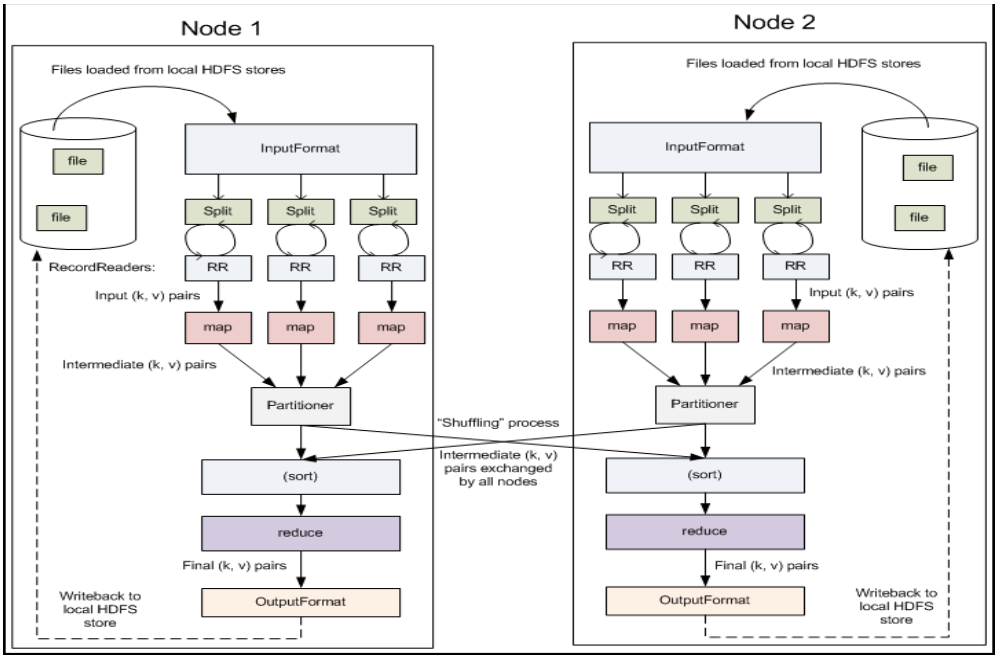

Fig. 1. MapReduce on HDFS

Fig. 1 depicts the MapReduce flow on Hadoop. We mainly concern the partition of input dataset. Traditional MapReduce uses RecordReader to split the whole input dataset. For example, a LineRecordReader is used to parse plain text datasets where each record is terminated by an End-Of-Line '\n' character. Roughly, the job controller will split the input dataset with align to record boundaries so that each mapper has an equal share of the dataset. This naive partition paradigm actually tries to balance the load between mappers. However, the naïve partition scheme becomes insufficient for search tasks over large scale of dataset. Since reading the whole datasets consists of not only local disk interactions but also a lot of network transmission, recalling that dataset is stored on HDFS, an effective partition scheme is in need for increasing popular database applications and information systems.

To remedy this problem, our approach is to setup an auxiliary index file for the input dataset. In particular, both the dataset and its index are used during the task. We hope the index is small so that is doesn't require too much space to store. In addition, such an index should be lightweight and is convenient to maintain. For these considerations, we use the bloom filter as underlying structure for the index implementation. As a probabilistic data structure, bloom filter has many advantages over others, such as linked list, AVL-tree, trie and hash table. Most of these require storing at least the data items themselves, hence seems to be space-inefficient. Bloom filter takes advantage of compactness inherit from array and its probabilistic nature, leading in 1% error but requires only about 9.6 bits per element.

IV. BLOOM FILTER BASED SEARCH INDEX

In this section, we introduce an index-based partition scheme that help the master skip unnecessary input segments before launching mappers. Comparing to the naïve partition strategy, our approach benefits the performance with similar or usually less mappers.

A. The Naïve Partition

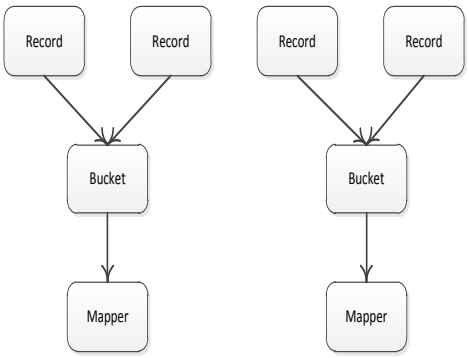

Fig. 2. The naïve partition

Let us first consider the naïve approach. Input records are first split into buckets, with each bucket contains roughly equal number of records as illustrated in Fig. 2. Each bucket corresponds to a mapper, where it is the input data for the mapper.

The naïve partition provides a simple load balance mechanism. It is assumed that each record needs a bounded time to process by the underlying algorithms. Since each mapper receives approximately an equal share of records, and is executed independently, the overall running time, to some degree, is guaranteed. Let the random variable c be the time to process a record, we may assuming c is normally distributed. Thus a mapper j would take $t_j = \sum c$ to finish processing all its records. We know t_j is also normally distributed. Assume $t_j \sim N(\mu, \sigma^2)$. It follows the running time for all mappers can be written as: $\emptyset = \max\{t_1, t_2, \ldots\}$. Designate the probability for $\emptyset = x$ by $P(\emptyset)$. The expected time for all mappers to complete their sub-tasks is therefore given by

$$E[\phi] = \sum_{j=1}^{+\infty} j \sum_{k=1}^{m} \binom{m}{k} P(X=j)^k P(X<j)^{m-k} = \sum_{j=1}^{+\infty} j \left[P(X<j+1)^m - P(X<j)^m \right] \quad (1)$$

where m is the number of mappers. Note that this expected running time increases when there are more mappers. In other words, the overall running time is expected to drop with fewer mappers even if each mapper processes a fixed number of records.

B. Same Input, Fewer Buckets

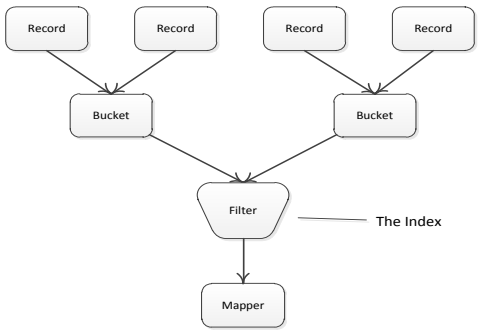

Fig. 3. Partition with bloom filter

We have noticed the potential benefits of using fewer mappers. However, it is not possible to reduce the number of mappers for the naïve partition without augmenting mapper input. Because the input size is fixed, using fewer mappers means each mapper works on more data.

Intuitively, we may test whether a bucket has the desired items before passing the bucket to mapper. The method we used is to provide a filter index, which is a bundle of bloom filters. Similar to the naïve partition, we first make a logical partition of the input records. Now we obtain several buckets, respectively B1,B2,… Before launching mappers for these buckets, the subscripts for buckets are checked against our filter index. Only those buckets whose subscripts are inside filter index will be passed to the mappers. Thus some buckets are discarded and we have fewer mappers.

The modified procedure is described in Fig. 3. In practice, the filter index can be a file stored on HDFS together with the dataset. Hence it would introduce little maintenance effort.

C. An Incremental Index Structure

We have introduced a new partition strategy based on filter index. Now it is time to investigate its internal constitution. As we have discussed in previous sections, the filter index is a set of bloom filters. In particular, each bloom filter works as a projection corresponding to a segment of input dataset. Let the sequence D denote input dataset byte stream. Assume we are performing a fair logical split of D, that is, approximately dividing D into equal-size segments $d_1,d_2,\ldots,d_k$, which are aligned to record boundaries. This process can be done immediate since the input size is known. Accordingly, we allocate k bloom filters of capacity m. Our goal is to generate projections of $d_1,d_2,\ldots,d_k$ into, respectively, those bloom filters. Let us focus on generating projection for one segment. It can be seen as mapping multi-dimensional vectors into bloom filter buckets, where records are represented by vectors. A typical phonebook record dataset D = {('John', 5551234), ('David', 5554793)}, for example, is easily described using two-dimensional vector (name, no.). Such vector description methodology also applies to many other datasets. Since mapping a set of single values is well introduced in many works of bloom filters [8], we omit repetition here. Above all, our major objective is to map multi-dimensional vectors to bloom filter.

An intuitive method for the vector mapping is transform vectors to single values. In the phone book example above, we may replace D with D' = {'John', 'David', 5551234, 5554793}. As we know, mapping D' to bloom filter is fairly easy. Obviously, any vector set can be transformed to its single value form in similar manner: vector set S = {(x1,x2,…),(y1,y2,…),…} is transformed into S' = {x1,x2,…,y1,y2,…}. We show that this simple transformation meets our needs. Applying the transformation to a segment d_j containing $|d_j|$ l-dimensional vectors, we obtain a set of at most $n=l|d_j|$ elements. If the number of hash functions for bloom filter is chosen at optimum, i.e. $m \ln 2 / n$, this give rise to an false positive probability of

$$p = 2^{-m \ln 2 / n} \quad (2)$$

Consider a query Q which would result in r records. Assume those r records are equally distributed on the k segments, and a map bucket contains t segments. Each bucket thereby is expected to contain rt/k records. Accordingly, the probability for a bucket does not contain any of the rt records is given by

$$q = (1-p)^{rt/k} \quad (3)$$

Hence, approximately q-percent map nodes can be saved due to prior knowledge from their filter indices. In response to this setting, k filter indices corresponding to the k segments are needed, which take up km/8 bytes.

V. EVALUATION

We evaluated our bloom filter based index (BF-MapReduce) with Hadoop cluster in Baidu Inc. The servers run Redhat AS (64bit) OS, with 48GB memory and Intel Xeon 2.27GHz (8 cores) CPU. The dataset is 10 TB. To guarantee correctness, each experiment runs for 5 times, and use the average data as final result.

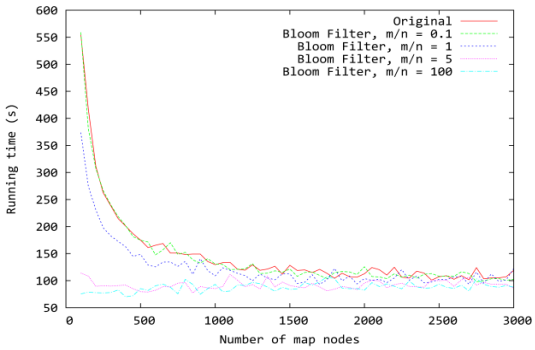

Fig. 4. Performance Analysis

Fig. 4 gives the performance comparisons with different index settings. As described above, the most effective factors in our search approach mainly focus on bloom filter's arguments. The map node will quickly skips unnecessary bucket with probability $\left(1-2^{-m\ln 2/n}\right)^{rt/k}$, hence the capacity m of each bloom filter definitely affect query performance.

As showed above, it is not difficult to understand that MapReduce tasks' processing time decrease with the augment of map nodes when searching given keys in same large dataset. However, using of our filter index will elevate process performance especially that map nodes are not very plentiful. From the figure 4, we can see that when map nodes are less than 500, assist with the filter index, the search's processing time can definitely be reduced even better than using 2000-3000 map node. This result is remarkable because it shows that we can achieve same performance with dramatically less resource.

As depicted in Fig.4, the search performances will be diverse with different capacity of bloom filter. With less capacity such as m/n=1, the search performance can be optimized but not achieve best. Our experiment shows that when capacity is 5 times of each segment's elements sum, it maybe achieve best balance with proper space cost. This is because more false positive will be aroused if the capacity is less, hence cannot efficiently skip unnecessary bucket as expect. If the capacity is great, more space will be cost, but it is not helpful to further improve whole performance.

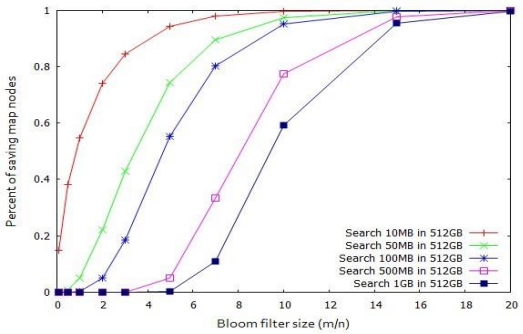

Fig. 5: Cost Analysis

Fig. 5 is the additional space cost analysis of the auxiliary bloom filter index. We randomly choose search data from 512GB data, then generate index with different size. Obviously, search performance enhances with the increase of bloom filter's capacity. This is because larger capacity leads to lower false positive probability that can efficiently skip needless map nodes. Compare to the input dataset, the index's space cost of proper capacity is very tiny and even can be neglect. Using phonebook as example, designate m/n=5, we expect each record with 2 items, and each item is 32Byte. If each segment contains one million records, the index cost of dataset with 512GB will be 10GB. Correspond to the dramatically performance enhancement of using auxiliary index, this cost is acceptable and valuable.

VI. CONCLUSION

Based on performance analysis of MapReduce, we find that the overall running time is expected to drop down even with fewer mappers. Hence, to decrease mappers, we use auxiliary filter index to reduce input dataset size of each map node. In our design, the whole dataset are split into some data segments which align to record boundaries. Each segments built a bloom filter to index data inside. When search operation performs, each map node firstly checks corresponding bloom filter of segment to decide whether this segment is needed to load and process. The bloom filter based index is small so that it doesn't require too much space to store. 10 TB dataset and about 3000 number of map nodes are used. The experiment results show that our approach is efficient and valuable for resource saving.

ACKNOWLEDGMENT

This work is supported by the National Nature Science Foundation of China (Grant No.61363021, No.61363084) and the Open Foundation of Key Laboratory in Software Engineering of Yunnan Province (Grant No.2012SE304).

REFERENCES

[1] Jeffrey Dean, Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM - 50th anniversary issue: 1958 - 2008 CACM Homepage archive Volume 51 Issue 1, January 2008 pp. 107-113

[2] A Bialecki, M Cafarella, D Cutting, O OMalley, Hadoop: a framework for running applications on large clusters built of commodity hardware, 2005, http://hadoop.apache.org

[3] Djoerd Hiemstra, Claudia Hauff, MIREX: MapReduce Information Retrieval Experiments, LNCS 2010, Volume 6360/2010, pp. 64-69

[4] Dawei Jiang, Beng Chin Ooi, Lei Shi, Sai Wu, The performance of MapReduce: an in-depth study, Proceedings of the VLDB Endowment VLDB Endowment Hompage archive, Volume 3 Issue 1-2, September 2010, pp. 472-483

[5] Eaman Jahani, Michael J. Cafarella, Christopher Ré, Automatic optimization for MapReduce programs, VLDB, Vol. 4 Issue 6, Mar 2011 pp. 385-396

[6] Thomas Sandholm, Kevin Lai, MapReduce optimization using regulated dynamic prioritization, SIGMETRICS '09, pp. 299-310

[7] Rong Chen, Haibo Chen, Binyu Zang, Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling, PACT '10, pp.523-534

[8] A. Broder, M. Mitzenmacher, Network Applications of Bloom Filters: A Survey, Internet Mathematics Vol.1, Issue 4, Jan. 2004, pp. 485-509

[9] Bhushan M, Banerjea S, Yadav S K. Bloom filter based optimization on HBase with MapReduce[C]//Data Mining and Intelligent Computing (ICDMIC), 2014 International Conference on. IEEE, 2014: 1-5.

[10] Chen Y, Zhao X, Ge B, et al. Practising Scalable Graph Similarity Joins in MapReduce[C]//Big Data (BigData Congress), 2014 IEEE International Congress on. IEEE, 2014: 112-119.