On Saying "Enough Already!" in MapReduce

Christos Doulkeridis*
Dept. of Computer and Information Science
Norwegian University of Science and Technology
Sem Sælandsvei 7-9
N-7491 Trondheim, Norway
cdoulk@idi.ntnu.no

Kjetil Nørvåg
Dept. of Computer and Information Science
Norwegian University of Science and Technology
Sem Sælandsvei 7-9
N-7491 Trondheim, Norway
Kjetil.Norvag@idi.ntnu.no

ABSTRACT

The MapReduce framework for parallel processing of massive data sets has attracted considerable attention recently, mainly due to its salient features that include scalability, simplicity, and fault-tolerance. However, despite its merits, MapReduce follows a brute-force approach, which often results in performing redundant work. This is particularly evident in the case of rank-aware queries, such as top-k, where a bounded set of k tuples comprise the result set. To process such queries in MapReduce, the input data needs to be accessed in its entirety, in order to produce the correct result set. To address this limitation of lack of early termination, in this paper, we investigate on different techniques that allow efficient processing of rank-aware queries, without accessing the input data exhaustively. We present various individual approaches that can be combined and demonstrate their advantages and shortcomings. Thus, we provide the first steps towards integrating efficient rank-aware processing in MapReduce.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Experimentation, Performance

Keywords

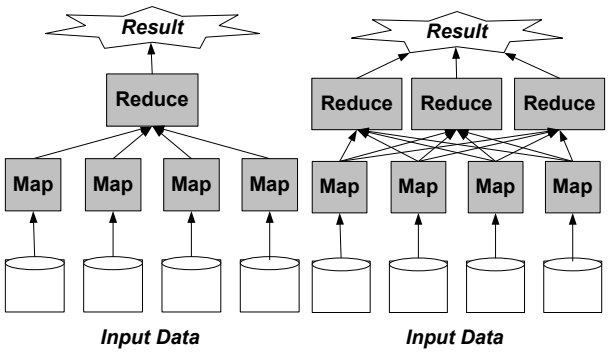
Ranking, top-k query, early termination, MapReduce

1. INTRODUCTION

The advent of cloud computing infrastructures has made feasible the ad-hoc analysis of massive data sets by means of parallel processing. Data analytics constitute a primary candidate application for the cloud due to the complex query processing involved and the vast size of input data to be processed. Rank-aware query processing, e.g., top-k queries, is an essential tool for data analytics, especially for business analysis, as it allows inspection only of the most useful data, rather than overwhelming data sets.

Currently, the most prominent framework for parallel query processing in the cloud is MapReduce [7], mainly due to its merits that include scalability, fault-tolerance, load-balancing, and simplicity. In summary, there exists a map phase where input data is scanned, filtered and the produced intermediate results are given as input to the reduce phase, which usually performs some kind of aggregation. Despite its advantages and popularity, MapReduce has been criticized for its lack of efficiency [17]. This is particularly evident in the case of top-k query processing, where the result to the query consists of a bounded set of k tuples only, which can be orders of magnitude smaller than the size of input data. For such queries, efficient processing would be equivalent to accessing only a subset the input data of small size in order to produce the correct result, instead of scanning the complete input data.

A baseline MapReduce algorithm for top-k processing would be to scan all data during the map phase, and send to the reduce phase only k objects from each partition. The reduce phase would then collect the local top-k tuples from each partition, as depicted in Figure 1(a), and keep only the top-k tuples of this set. In the case of top-k join, where the data come from more than a single relation, multiple reducers can be used, as depicted in Figure 1(b), with each reducer responsible for a different set of values of join attributes. Unfortunately, this baseline technique requires accessing the complete input data, and fails to provide early termination in the case that sufficient data for producing the correct result have been accessed. Thus, support for early termination is a key property for efficient processing of rank-aware queries, and this topic has not been addressed effectively by the research community yet.



(a) Top-k query. (b) Top-k join.

Figure 1: Map and reduce tasks for top-k and top-k join queries.

*C. Doulkeridis was supported under the Marie-Curie IEF grant number 274063.

Cloud-I '12, August 31 2012, Istanbul, Turkey

In this scenario, the main challenge for top-k queries is to restrict the redundant work done at DataNodes, where megabytes of data is read even though only k objects suffice to be reported to the reduce phase. Ideally, only one chunk of data (typically 64MB) should be read from each DataNode, and this chunk should contain the local top-k result.

In this paper, we describe for the first time approaches for rank-aware query processing using MapReduce that are being developed as part of the *CloudIX* (cloud-based indexing and query processing) project, where the aim is to improve the performance of advanced query processing in the cloud. For simplicity, we restrict the discussion to basic top-k and top-k join queries, although many of the techniques are also applicable for more advanced rank-aware query operators. Thus, the challenge of our research is *how to perform top-k and top-k join queries as efficient as possible utilizing the scalability provided by the MapReduce framework*.

The rest of this paper is structured as follows: Section 2 reviews the related work in the field. In Section 3, we provide the necessary background. In Section 4 we describe sort-based and synopse-based methods for rank-aware query processing using MapReduce. Finally, in Section 5, we conclude and outline promising new directions for rank-aware query processing using MapReduce.

2. RELATED WORK

Despite its merits, MapReduce processing may entail significant amounts of redundant work, when applied for complex query processing tasks. Guided by this observation, recent papers have appeared that try to improve some features of MapReduce processing. CoHadoop [10] tries to place data on data nodes intentionally and exploit locality, so that similar data are stored on the same datanode. This approach greatly improves the performance of query processing, since data that need to be accessed together are usually placed on the same data node. Hybrid systems, such as HadoopDB [1], have been recently proposed, aiming to exploit the best features of MapReduce and parallel DBMSs. In HadoopDB, each datanode hosts a DBMS and uses its query processing and optimization features during local processing. Other approaches that employ indexing techniques at local level on datanodes have also been proposed including B-tree indexing [20] and Hadoop++ [8] that injects indexing information in data files. RanKloud [4] is the only existing work that addresses rank-aware queries (top-k joins) in a MapReduce context. RanKloud follows an interesting approach that computes statistics (at runtime) during scanning of records, and uses these statistics to compute a threshold for termination of query processing. However, one important limitation is that it cannot guarantee the retrieval of k results, i.e., may retrieve fewer results, which is completely different to our approach that aims on correctness and completeness of result.

Several papers have dealt with the issue of top-k query processing in centralized database management systems [5, 6, 13]. In distributed systems, approaches for distributed top-k query processing can be classified in two main categories based on data partitioning: *horizontal* where each server stores a fraction of the available data but all attribute values, and *vertical* where each server stores only some attributes of all available data. Most approaches using vertical partitioning try to improve some limitations of the Threshold Algorithm [11]. In the following, we focus on the case of horizontal partitioning, as it is more close to the philosophy of MapReduce, whereas vertical partitioning requires coordinated access (sorted and/or random) to the data, which would impede the parallel and independent processing of map jobs.

Balke *et al.* [3] try to minimize the data object traffic induced by top-k processing, but the approach is optimized for reoccurring identical queries, which is unlikely as there is an infinite number of potential queries posed by different users. A similar approach for unstructured P2P systems is presented in [2], where the main technique is a variant of flooding, followed by a merging score-list step at intermediate peers. Zhao *et al.* [21] rely on result caching to prune network paths and answer queries without contacting all peers. Ryeng *et al.* [16] studied caching of top-k results and the use of remainder queries to answer future top-k queries. The applicability of the skyline operator for efficiently routing top-k queries over a super-network was studied in [19]. In [14], an approach is proposed that tries to minimize the users' waiting time of top-k results, at the expense of multiple phases of data transmission. Recently, in [9], distributed statistics are exploited for supporting top-k joins efficiently.

3. PRELIMINARIES

In this section we give a brief overview of MapReduce and HDFS, and define the type of queries we will focus on.

3.1 MapReduce and HDFS

Hadoop is an open-source implementation of MapReduce [7], providing an environment for large-scale fault-tolerant data processing. Hadoop consists of two main parts: the HDFS distributed file system and MapReduce for distributed processing.

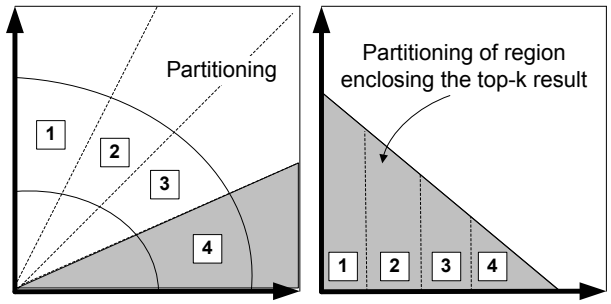
Files in HDFS are split into a number of large blocks which are stored on DataNodes, and one file is typically distributed over a number of DataNodes in order to facilitate high bandwidth and parallel processing. The maintenance of mapping from file to block, and location (DataNode) of block, is handled by a separate NameNode. One important aspect of HDFS important for this paper, is that HDFS is optimized for streaming access of large files, and as a result random access to parts of files is significantly more expensive than sequential access.

A task to be performed using the MapReduce framework has to be specified as two steps: the *Map* step as specified by a map function takes input (typically from HDFS files), possibly performs some computation on this input, and distributes it to worker nodes, and the *Reduce* step which processes these results as specified by a reduce function. An important aspect of MapReduce is that both the input and output of the Map step is represented as Key/Value pairs, and that pairs with same key will be processed as one group by the reducer. It is important to note that since Reduce simply processes incoming data until end of the stream, approaches for limiting the amount of data to be read have to be performed during the Map step.

3.2 Top-*k* Queries

A *top-k* query $q(k, f)$ returns the k most interesting query results, based on a monotone scoring function f. The most important and commonly used case of scoring functions is the weighted sum function, also called linear. This function has an associated query-dependent weight $w[i]$ for each of the n scoring attributes $\tau[i]$ of the database object τ, and the score of an object (or tuple) τ is the sum of the individual scores given by $f_w(\tau) = \sum_{i=1}^{n} w[i] \cdot \tau[i]$.

A frequent operation in rank-aware processing is the combination of join followed by selecting the top-k results from the join. For efficiency reasons this should be executed as one operator, which interleaves the join with ranking, and this is named a *top-k* join query. We focus on binary many-to-many joins, where the relations R_1 and R_2 are joined on a join attribute and a combination of scoring attributes of both relations is used as input to the scoring function f in order to produce the top-k ranking.



(a) Intelligent data placement. (b) Partitioning used in [4].

Figure 2: Data placement on DataNodes.

4. RANK-AWARE QUERY PROCESSING

In this section we describe various constituent blocks of efficient rank-aware processing in a MapReduce context and discuss their advantages and disadvantages. The efficiency of the different approaches depends on a number of parameters, including availability of sorted access, data partitioning to servers, and which one(s) to use can be decided by user/application or automatic tools.

4.1 Sorting

In centralized databases, efficient processing of top-k queries is accomplished by means of *sorted access* to data. In sorted access, the tuples are accessed sequentially ordered by some scoring predicate. This sorted access is accomplished either *directly* by storing the data sorted on disk or *indirectly* by means of a secondary index.

In the case of parallel processing with MapReduce, there exist two alternative ways of providing sorted access. The straightforward way for a MapReduce programmer to sort data is to perform sorting right before the actual data processing. This approach follows the spirit of MapReduce, where all processing takes place as part of the job execution at hand. In this case, each time a top-k query $q(k, f)$ is issued, a MapReduce task is initiated that sorts data based on the query function f. MapReduce provides ways to sort data. After having a global sorted data set based on f, the result to the top-k query is easily obtained by reporting the first k tuples. The main limitation of this technique is that each incoming top-k query $q(k, f)$ causes sorting of the input data based on f. Clearly, this overhead imposed for each query is high, and it is important to find a more efficient solution.

An alternative technique is to store data sorted on each DataNode based on a monotone function f', which in the general case is different from f. It can be easily shown [12] that the top-k local tuples to a query $q(k, f)$ from a DataNode can be retrieved by accessing only a bounded set of the first k' tuples of the stored data, where $k' \geq k$. Storing the data sorted on DataNodes can be achieved either at load time or by a separate MapReduce program that needs to be executed once.

Query Processing. When data is sorted based on the query function f, the top-k result consists of the first k tuples. In the case that a different function f' is used for sorting, the correct top-k result is retrieved when the following condition holds: the best possible score based on f of the next tuple in the sorted order is worse than the score of the k-th best tuple retrieved so far.

The advantage of using sorting is the cheap cost of query processing, as the top-k result of each DataNode will be (with high probability) in the first chunk of data. Its disadvantage is that the cost that sorting entails may be important, especially in the case of massive data sets.

4.2 Intelligent Data Placement

Data placement to DataNodes is carried out transparently in Hadoop, without taking into account the data content. However, the way data is placed significantly affects load balancing and the performance of query processing. With respect to load balancing, it is important to balance the useful work to DataNodes and avoid redundant processing that will not produce any top-k results.

Existing partitioning schemes used for data placement are oblivious to the nature of top-k queries. Therefore, we propose the use of an intuitive partitioning method, termed *angle-based partitioning* [18], which can improve query processing, especially if combined with sorted access. The partitions produced by angle-based partitioning are illustrated in Figure 2(a). The advantage compared to the built-in partitioning of Hadoop is clear. In addition, this partitioning method offers advantages over the proposed partitioning in [4], shown in Figure 2(b), since it splits the useful work to DataNodes fairly. As illustrated, intuitively, the angle-based scheme splits the region near the axes to all partitions, assuming minimum values are preferable. In contrast, the partitioning shown in Figure 2(b) will assign more work to partition 1 with high probability. In addition, the angle-based partitioning can be generalized in higher dimensions (i.e., number of scoring attributes) in a straightforward way.

Moreover, the angle-based partitioning can be combined with sorting on DataNodes to improve the performance of local query processing further. Each partition needs to be split and stored to multiple chunks on each DataNode. It is beneficial to split the partition in such a way that the best tuples for any incoming top-k query are located at the first chunk. Then, the remaining chunks can be pruned from further processing. The splits are defined based on the distance of a tuple to the origin of the data space. Graphically, this is depicted in Figure 2(a) by means of arcs with increasing radius centered at the origin of the axes. The radius of each arc is set so that the corresponding split contains as many tuples as can fit in a chunk.

4.3 Use of Synopses

In the case that sorted access to data is not provided, our premise is to create and exploit data synopses about the stored data that will allow to identify when the accessed data suffice to produce the correct top-k result set. Then, we can cease accessing the remaining underlying data, thus achieving performance gains.

4.3.1 Construction

Different types of synopsis can be employed to maintain summary information about the underlying data. The synopsis should satisfy some requirements, such as being compact in size, concise, and should be easy to build, i.e., using at most a single pass of the data. To this end, we propose the use of multidimensional histograms as data synopsis, where dimensions are defined by the scoring attributes. Each histogram bin is described by two n-dimensional values, the lower boundary and the upper boundary of the bin, which determine the range of scores for all database objects enclosed in that bin. In addition, the number of such database objects is stored in the bin. Furthermore, a bin maintains the chunks' addresses where the enclosed objects are stored. Figure 3 depicts an example of a 2-dimensional histogram stored on a DataNode and summarizing the locally stored data.

The histograms are built seamlessly during the loading phase of data to HDFS in a single pass, e.g., using techniques such as [15]. Each histogram corresponds to the data stored at one DataNode, and the histogram is also stored at the DataNode. Essentially, the histogram serves as a local index to the DataNode, indicating which

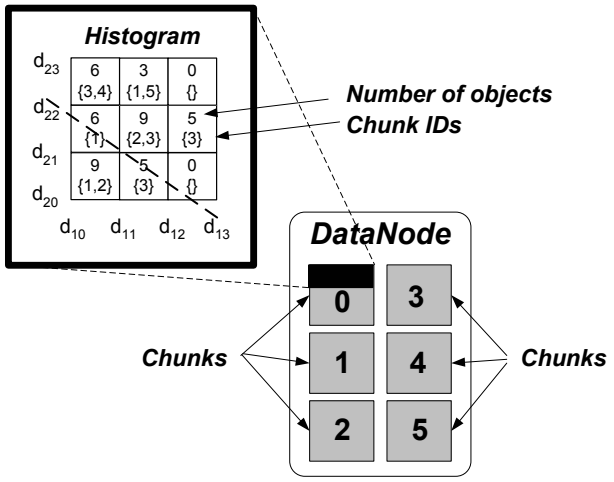


Figure 3: Storage of data synopses on DataNodes.

chunks of data are potentially useful for producing the query result. The representation of the histogram is much smaller than the data, so the storage overhead imposed is minimal.

4.3.2 Query Processing and Optimization

Prior to the initiation of the mappers, processing of the histograms takes place in order to identify which chunks contain useful tuples that will produce the top-k result. Then, these chunks can be accessed with random access, instead of accessing all chunks sequentially. In this process, some chunks will be eliminated and will not be accessed, as they cannot contribute to the final result set. The output of histogram processing is a set of chunk IDs that are guaranteed to contain the correct top-k tuples. However, it should be noted that as the cost of random access is higher than sequential access in HDFS, a cost model is necessary to estimate when it is beneficial to use random access (*query optimization*).

The remaining question is how to use the histogram to produce the desired chunk IDs. To achieve this goal, the histogram bins are accessed progressively, starting from the best (minimum) scores to the worst (maximum), until it is guaranteed that k tuples are enclosed in the accessed bins. When this occurs, we collect the chunk IDs contained in the accessed bins, and these correspond to the chunks that are guaranteed to contain the top-k tuples. In the example of Figure 3, assume a top-k query has been posed with k=5 and let the dashed line over the bins define which area contains the top-k result. The algorithm will access the two dimensions until d_{13} and d_{21} respectively. In the six produced bins, 34 tuples are contained (more than 5), and only chunks with IDs 1,2 and 3 need to be accessed.

Several optimizations are possible under this framework. One possibility is to use histograms of varying width, e.g. equi-depth, to have more detailed information about score values that have a higher probability to appear in the top-k. Another possibility is to explore different methods for computing the score bounds for each dimension, e.g., by examining more bins from the dimension that is most promising to produce the top-k results faster. Also, as already mentioned, cost-based optimization needs to be employed to decide which is the most cost-effective access method.

On a final note, the data synopses can also be combined with the aforementioned approaches (sorting and deliberate data placement), in order to boost the performance of query processing.

5. CONCLUSIONS AND OUTLOOK

In this paper, we described techniques for efficient execution of top-k queries using MapReduce. We are currently implementing and evaluating these techniques, and will in future work refine the methods as well as support other flexible query operators. Several interesting research directions open up for rank-aware query processing in MapReduce. First, devising analytical cost models that determine which strategy will lead to smaller processing cost. Then, finding a provably optimal partitioning scheme for top-k queries and top-k joins. Other interesting directions include support for more complex functions (non-linear or even non-monotonic), and single pass algorithms for computing data synopses with accuracy guarantees. Last, but not least, we plan to extend our techniques to be applicable for intermediate results produced by other query operators as part of a query plan. This will enable treating top-k processing as a first-class citizen in the cloud.

6. REFERENCES

[1] A. Abouzeid et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] R. Akbarinia, E. Pacitti, and P. Valduriez. Reducing network traffic in unstructured P2P systems using top-k queries. *Distributed and Parallel Databases*, 19(2-3):67–86, 2006.

[3] W.-T. Balke et al. Progressive distributed top-k retrieval in peer-to-peer networks. In *Proc. of ICDE'2005*, 2005.

[4] K. S. Candan et al. RanKloud: Scalable multimedia data processing in server clusters. *IEEE MultiMedia*, 18(1):64–77, 2011.

[5] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of SIGMOD'1997*, 1997.

[6] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proc. of VLDB'1999*, 1999.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of OSDI'2004*, 2004.

[8] J. Dittrich et al. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.

[9] C. Doulkeridis, A. Vlachou, and K. Nørvåg et al. Processing of rank joins in highly distributed systems. In *Proc. of ICDE*, 2012.

[10] M. Y. Eltabakh et al. CoHadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of PODS'2001*, 2001.

[12] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proc. of SIGMOD'2001*, 2001.

[13] I. F. Ilyas et al. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst.*, 31(4):1257–1304, 2006.

[14] W. Kokou et al. ASAP top-k query processing in unstructured P2P systems. In *Proc. of P2P'2010*, 2010.

[15] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of SIGMOD'1998*, 1998.

[16] N. H. Ryeng, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient distributed top- query processing with caching. In *Proc. of DASFAA'2011*, 2011.

[17] M. Stonebraker et al. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[18] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Prof. of SIGMOD'2008*, 2008.

[19] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. On efficient top-k query processing in highly distributed environments. In *Proc. of SIGMOD'2008*, 2008.

[20] S. Wu et al. Efficient B-tree based indexing for cloud data processing. *PVLDB*, 3(1):1207–1218, 2010.

[21] K. Zhao, Y. Tao, and S. Zhou. Efficient top-k processing in large-scaled distributed environments. *Data Knowl. Eng.*, 63(2):315–335, 2007.