Top-k Computations in MapReduce: A Case Study on Recommendations

Vasilis Efthymiou
ICS-FORTH & Univ. of Crete, Greece vefthym@ics.forth.gr
Kostas Stefanidis
ICS-FORTH, Greece kstef@ics.forth.gr
Eirini Ntoutsi
LMU, Germany ntoutsi@dbs.ifi.lmu.de

***Abstract*—Top-k is a well-studied problem in the literature, due to its wide spectrum of applications, like information retrieval, database querying, Web search and data mining. In the big data era, the volume of the data and their velocity, call for efficient parallel solutions that overcome the restricted resources of a single machine. Our motivating application is recommenders, which typically deal with big numbers of users and items, but other applications might benefit as well, like keyword search. In this paper, we propose a parallel top-k MapReduce algorithm that, unlike existing MapReduce solutions, manages to handle cases in which the k results do not fit in memory.**

I. Introduction

Efficient top-k processing is a crucial requirement in many interactive environments, such as the Web and multimedia search, that involve huge amounts of data (for surveys on algorithms for top-k computations, see [4], [11]). As a case study, consider that with the growing complexity of the Web [1], users often find themselves overwhelmed by the mass of choices available. To facilitate users in their selection process, recommender systems offer suggestions of potential interest on items [8]. In particular, recommender systems aim at providing recommendations to users or groups of users by estimating their item preferences and recommending those items featuring the maximal predicted preference. The prerequisite for determining such recommendations is historical information on the users' interests, e.g., the users' purchase history.

Collaborative filtering [10] is a widely used recommendations technique, in which the key concept is to locate users, for example, the top-k ones, with similar rating behavior to the query user; the preferences of these users are employed then for issuing recommendations for the query user. The simplest, naïve approach for finding similar users is by linear scanning the whole user base. Clearly, this is a costly process for large recommender systems, in which there exist millions of users.

Nowadays, there are more efficient approaches for identifying users sharing similar preferences, such as approaches that build user models and employ these models for rating prediction. User models might be derived through, for instance, clustering users into groups of similar users. For example, [6] applies full-dimensional clustering to organize users into clusters and employs these clusters, instead of a linear scan of the database, for predictions.

Nevertheless, full-dimensional clustering is not the best option for the recommendations domain due to the high dimensionality of the data; typically, there exist thousands to millions of items in a recommendation application. Due to the high dimensionality, it is difficult to locate similar users with respect to the whole high-dimensional feature space. Alternatively, it is easier to locate such users in a subset of features or subspace of the original feature space. Solutions that search for user models in such subspaces exist; for example, [7] employs subspace clustering to detect both the users that belong to a cluster and the items that define this cluster.

However, in both full-dimensional and subspace clustering, the quality of recommendations depends heavily on the partitioning strategy. Based on recent studies [7], although a clustering approach is much faster than the naïve approach, the naïve is superior in terms of quality.

Luckily, nowadays, even the naïve approach of linear scanning the database to locate similar users might be implemented efficiently through distributed computing, allowing to resorting to non-approximate solutions. Our goal in this work, is to investigate an efficient parallel implementation for top-k computations by leveraging the MapReduce [2] programming model. Note that when considering group recommendations (e.g., [9], [6]), i.e., recommendations for groups of users instead of single users, the complexity is aggravated, as the heavy computational task of identifying similar users should be repeated for each group member.

We emphasize on the case in which k results, users in our example, do not fit in memory, which has not been previously addressed in a MapReduce environment. This problem arises when, for example, a big company (i.e., with millions of customers) wants to upgrade the services that it offers to its oldest customers, or when a government's tax office wants to spot the citizens with the highest income. The most similar work is the top-k algorithm presented in [5], in which k is small enough to fit, for each data partition, the top-k objects in memory; then, local top-k objects are aggregated to produce the final output. In our experiments,

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

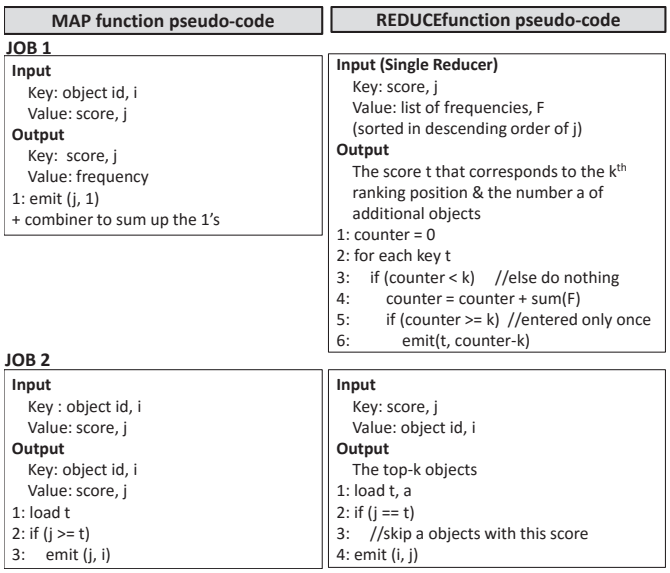

Figure 1. Pseudo-code for the top-k algorithm.

we highlight the scalability of our approach on a cluster of 15 machines, using synthetic big data.

II. MapReduce Top-k Computation

MapReduce is a programming model that allows parallel processing of large datasets on a computer cluster. A dataset, given as input to a MapReduce job, is split into chunks, which are then processed in parallel. Abstractly, a map function, emitting intermediate (key, value) pairs for each input split, and a reduce function that processes the list of values of an intermediate key, coming from all the mappers, should be defined. Optionally, a combine function can be provided, to process the output of each mapper, like a local, mini-reducer, and decrease the amount of data transferred through the cluster network. Many problems that cannot be solved in a single MapReduce job can utilize a chain of several jobs, as is the case in our suggested solution.

In this work, we propose the algorithm sketched in Figure 1 for top-k computations. This algorithm takes as input a set of pairs (i, j), representing objects i associated with scores j, and outputs the k pairs with the highest scores. In case of recommendations, i stands for a user u' and j for the similarity between our reference user u and u'. Our goal is to detect the top-k users with the higher similarity scores.

To implement that, we use two MapReduce jobs. The first job (*Job 1*) identifies the maximum score threshold t, such that, at least k objects have a score j greater than, or equal to t. The second job (*Job 2*) outputs exactly k objects with a score j greater than, or equal to t.

In detail, Job 1 takes as input (i, j) pairs. For each pair, the first mapper emits a (j, 1) pair (Mapper 1, Line 1). A combiner then sums up the 1's that correspond to the same score j, provided by the same mapper, practically counting the number of objects with score j.

For each score j, given in descending order (using a custom key comparator), the single reducer keeps a counter for the number of objects that have a score greater than, or equal to the current score (Reducer 1, Line 1). When this counter reaches or exceeds k (Reducer 1, Line 5), the reducer stops, after emitting the current score t and the number a of additional objects that have the same score (Reducer 1, Line 6), namely, how many more objects than k, have the emitted score t. After that, the reducer performs no further processing (Reducer 1, Line 3).

When $a = 0$, the mapper of the final job emits a (i, j) pair that it is given as input, when $j \geq t$. In that case, exactly k objects are emitted from the mapper and the job skips the reduce phase. This case, occurring when it is rather unlikely that many objects will have the same score, is not depicted in Figure 1. When $a > 0$, the mapper emits a (j, i) pair (Mapper 2, Line 3) for a (i, j) pair that it is given as input, when $j \geq t$ (Mapper 2, Line 2). In that case, $k+a$ objects are emitted from the mapper. The reducer then skips a objects from those with score t (Reducer 2, Lines 2-3) and emits all the rest (Reducer 2, Line 4).

III. Evaluation

For our experiments, we used a cluster of 15 Ubuntu 12.04.3 LTS servers (virtual machines), each with 8 AMD 2.1 GHz CPUs, 8 GB RAM and 60 GB of hard disk capacity, placed in a local network with speed around 500 MB/s. One of the nodes served as a master node and the rest were slave nodes. Each slave node could run simultaneously 4 map or reduce tasks, while each task had a heap size of 1,250 MB available. Each experiment was repeated twice and the average time was considered, to eliminate external factors effects (e.g., network overhead). Finally, we used the ~okeanos[1] GRNET cloud service, Apache Hadoop 1.2.0 and Java version 1.7.0_25 from OpenJDK. The source code, baselines, and dataset generator are publicly available at https://github.com/vefthym/Top-k_in_MapReduce.

To show the generality of our algorithm and test it with a bigger volume of data than the available recommender systems datasets we could find[2], we use synthetic datasets, consisting of ($object$, $score$) pairs, where $object$ is a unique numeric id and $score$ is a random number in $[0, 100]$. Figure 2 (left) presents the running time and the speedup of our algorithm on a dataset of 300M randomly-scored objects when asking for top k=100M, using 1, 5, 10 and 14 slave nodes. After a point, using more nodes is not improving the execution time much, since there is an overhead of using MapReduce, e.g., data transferring and job setup. However, in bigger datasets, as that of Figure 2 (right), consisting

[1] https://okeanos.grnet.gr

[2] For example, the largest version of MovieLens consists of 72K users [3].

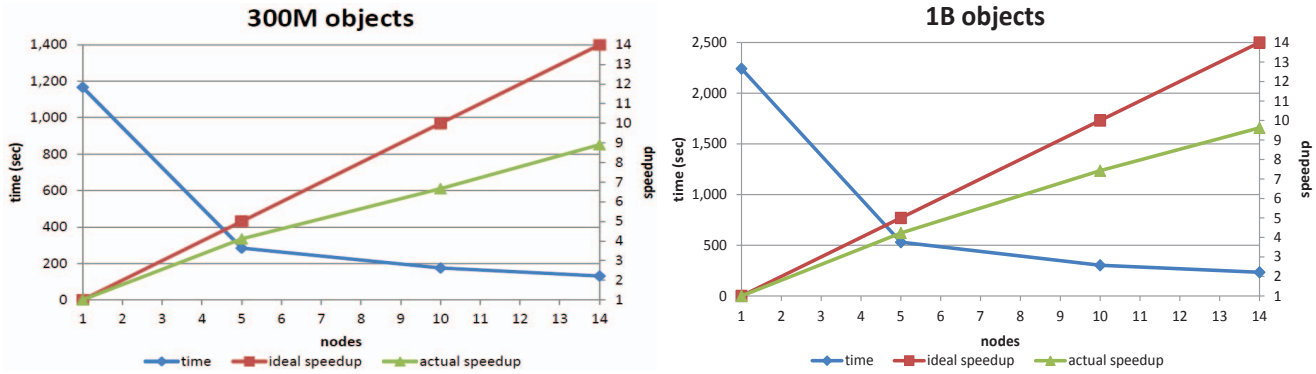

Figure 2. Time and speedup of the top-k algorithm with different number of nodes for the 300M dataset (left) and 1B dataset (right). For both cases, top-k=100M.

of 1B randomly-scored objects, the number of nodes, after which execution time is not significantly improved, is slightly higher. We note that k is not affecting the execution time or speedup of our algorithm, as much as the size of the input data, since the most expensive task is sorting the objects.

As an indication of the running time that a sequential algorithm needs for the same task, we executed external merge sorting on a single core based on scores, and then kept the top-k pairs. Merge sorting took around 1,537 and 5,971 seconds for the 300M and 1B datasets, respectively. This means that our approach, for the 300M dataset, requires 25% less time, even when a single node is used, and 91% less time when 14 nodes are used. For the 1B dataset, the corresponding reductions in time are 62.6% and 96.1%, respectively.

Moreover, our algorithm performs better than parallel merge sorting, when the latter is run on 4 cores of a single machine. For example, for the 1B dataset, our algorithm outperforms merge sorting by 28% and 92.5%, when 1 and 14 nodes are used, respectively.

IV. Summary

In this work, we have presented an efficient and scalable top-k MapReduce algorithm that is capable of managing the case in which k results do not fit in memory. To the best of our knowledge, this is the first time that a MapReduce algorithm can handle this case.

So far, we experimented with simulated recommendation datasets; our next step will be to experiment with real data. We plan to broaden the spectrum of input data types and consider, e.g., the users' behavior and friendships in social networks. Another extension can be in terms of grouping similar scores using some, e.g., hashing function. So far, the different scores are treated as different values, however in a real recommender, small variations in the scores might be neglected.

Acknowledgements

This work was partially supported by the EU FP7 IdeaGarden (#318552), the H2020 PARTHENOS (#654119) and the DFG FOR 1670 projects. We would also like to thank GRNET, for providing the ~okeanos cloud service, on which we ran our experiments.

References

[1] V. Christophides, V. Efthymiou, and K. Stefanidis. *Entity Resolution in the Web of Data*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2015.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[3] Grouplens-Research. MovieLens. http://grouplens.org/datasets/movielens/, 2015. [Online; accessed 05-January-2015].

[4] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[5] D. Miner and A. Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, Inc., 1st edition, 2012.

[6] E. Ntoutsi, K. Stefanidis, K. Nørvåg, and H. Kriegel. Fast Group Recommendations by Applying User Clustering. In *ER*, pages 126–140, 2012.

[7] E. Ntoutsi, K. Stefanidis, K. Rausch, and H. Kriegel. "Strength Lies in Differences": Diversifying Friends for Recommendations through Subspace Clustering. In *CIKM*, pages 729–738, 2014.

[8] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, editors. *Recommender Systems Handbook*. Springer, 2011.

[9] S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Space efficiency in group recommendation. *VLDB J.*, 19(6):877–900, 2010.

[10] J. J. Sandvig, B. Mobasher, and R. D. Burke. A survey of collaborative recommendation and the robustness of model-based algorithms. *IEEE Data Eng. Bull.*, 31(2):3–13, 2008.

[11] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM Trans. Database Syst.*, 36(3):19, 2011.