RanKloud: A Scalable Ranked Query Processing Framework on Hadoop*

K. Selçuk Candan
CIDSE, Arizona State University
Tempe, AZ, 85287, USA
candan@asu.edu

Parth Nagarkar
CIDSE, Arizona State University
Tempe, AZ, 85287, USA
nagarkar@asu.edu

Mithila Nagendra
CIDSE, Arizona State University
Tempe, AZ, 85287, USA
mnagendra@asu.edu

Renwei Yu†
CIDSE, Arizona State University
Tempe, AZ, 85287, USA
renwei.yu@asu.edu

ABSTRACT

The popularity of batch-oriented cluster architectures like Hadoop is on the rise. These batch-based systems successfully achieve high degrees of scalability by carefully allocating resources and leveraging opportunities to parallelize basic processing tasks. However, they are known to fall short in certain application domains such as large scale media analysis. In these applications, the *utility* of a given data element plays a vital role in a particular analysis task, and this *utility* most often depends on the way the data is collected or interpreted. However, existing batch data processing frameworks do not consider *data utility* in allocating resources, and hence fail to optimize for *ranked/top-k* query processing in which the user is interested in obtaining a relatively small subset of the best result instances. A naïve implementation of these operations on an existing system would need to enumerate more candidates than needed, before it can filter out the k best results. We note that such waste can be avoided by utilizing *utility-aware* task partitioning and resource allocation strategies that can prune unpromising objects from consideration. In this demonstration, we introduce `RanKloud`, an efficient and scalable *utility-aware* parallel processing system built for the analysis of large media datasets. `RanKloud` extends Hadoop's MapReduce paradigm to provide support for *ranked* query operations, such as k-nearest neighbor and k-closest pair search, skylines, skyline-joins, and top-k join processing.

*This work is partially funded by the HP Labs Innovation Research Program Grant "*Data-Quality Aware Middleware for Scalable Data Analysis.*"

†Authors are listed in an alphabetical order.

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

General Terms

Theory, Implementation

Keywords

MapReduce, Top-K, Skyline, KNN, Parallel Processing

1. INTRODUCTION

A significant class of data processing applications that are easy to describe, and in many cases, easy to parallelize can be expressed in terms of a small set of primitives. This has led to frameworks such as MapReduce [5, 1], Scope [3], etc. These systems have been successfully applied in domains such as data processing, mining, and information retrieval [6, 8, 11]. Given an atomic task, these rely on the simple semantic properties of the task to partition the work onto many machines. More complex data processing tasks are represented as workflows. Significant savings in execution times are obtained by independently parallelizing each step of the workflow and executing them in a batched manner. Frameworks such as Hadoop [13], which is based on Google's MapReduce [5], have produced impressive results in many data processing and analysis application domains [6, 8, 11]. On the other hand, in others domains (including those involving aggregation and join tasks, which are hard to parallelize) these batch processing systems lag behind traditional DBMS solutions [10].

A particular shortcoming of batch-based data processing frameworks is that they are not effective in domains where the utility of the data elements to a particular task varies from data instance to data instance, and users are interested not in all the data, but the ones that are best suited for the given task. Locating such high utility data is often known as *ranked/top-k* query processing. The applications in which the data utilities vary include decision support, and text and media analysis. In text analysis, for example, the TF-IDF values or the frequency of the words can be considered as the utility scores of the data. While batch-oriented

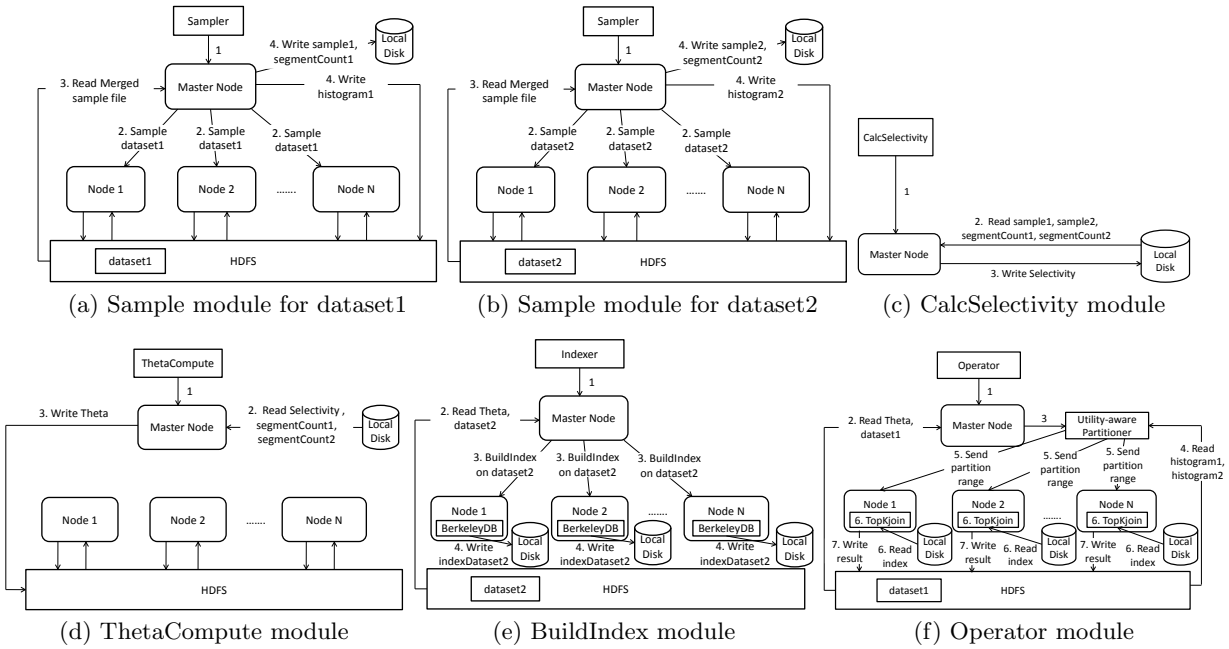

Figure 1: RanKloud's execution engine: RanKloud leverages and extends the open-source Hadoop infrastructure

systems promise large scale parallelism, they are unable to optimize for *ranked* query processing since they do not consider variations in data utility. In order to avoid wasted work, these data processing systems need to employ *utility-aware* task partitioning and resource allocation strategies that can prune unpromising objects from consideration.

Motivated by the above observations, this paper introduces RanKloud, an efficient and scalable, utility-aware, parallel processing system developed for the analysis of large scale media datasets. RanKloud builds on Hadoop's MapReduce paradigm and provides: (1) adaptable, utility and rank-based data processing (*map*, *reduce* and *merge*) primitives, (2) waste and unbalance-avoidance strategies for incremental batched processing, and for utility-aware data partitioning and resource allocation, and (3) media processing workflow scheduling based on the data and utility characteristics discovered at runtime. In particular, we demonstrate uSplit[1], a data partitioning strategy that we developed for processing top-k join queries in batch-oriented cluster environments. We show how uSplit adaptively samples datasets in order to allocate resources in a work-balanced and wasted-work avoiding manner for top-k join processing. In addition to uSplit, we also demonstrate efficient implementations of the skyline and k-nearest neighbor search operators. Our future research involves extending RanKloud to support join-based skyline and k-closest pair queries.

The rest of the paper is structured as follows: in Section 2, we give an overview of the existing work. Section 3 presents the overall system architecture of RanKloud. In Section 4, we discuss the demonstration scenarios. Lastly, we conclude and give a sketch of future research directions in Section 5.

[1]The details of this work can be found in [15]

2. RELATED WORK

A top-k query can always be processed by simply enumerating all results and then picking the best k. But when the number of candidate results is large, and when the k is much smaller than the number of candidate results, it is *wasteful* to rely on processing strategies that would require the system to enumerate all possible candidate results. Instead, for *ranked* processing operations such as k-nearest neighbor search, k-nearest neighbor and k-best-nearest neighbor joins, skylines and top-k skylines, top-k joins, top-k group and top-k co-group operations, data processing systems need to employ data structures and algorithms that can prune unpromising data objects from consideration, without having to evaluate them.

Most existing algorithms that implement the above operations assume that one or both of the following data access strategies are available: (a) sorted access – the system is able to access the input data incrementally, in the non-increasing order of utility scores. For example, top-k join algorithms such as FA, TA, and NRA [7] assume that the input data is available in utility sorted order – pipelined access to sorted data helps identify candidate matches and prune non-promising results faster; (b) indexed/clustered access – for any candidate object, the system is able to quickly identify the corresponding utility score by using an index structure for efficient utility score look up. The index structure also helps identify similar objects quickly and prune irrelevant objects faster. For instance, algorithms to tackle k-nearest neighbor and k-closest pair queries assume that the data is ordered using space filling curves [14] or partitioned using index structures (such as R-trees, KD-trees, and Voronoi diagrams) [4] or hash functions [2]. Skyline algorithms also include sorting-based and index-based algorithms [12]. However, implementing these operators on a

```
#!/bin/bash
bin/hadoop jar TopKoperator.jar DataAnalyzer.Sample input_params(dataset1 JoinAttr1 UtilityAttr1 dataset1_distribution SampleBudget1 numPartitions1 rangeScale1 sampleScale1 sampleType1)
                                output_params(segmentCount1 histogram1 sample1)
bin/hadoop jar TopKoperator.jar DataAnalyzer.Sample input_params(dataset2 JoinAttr2 UtilityAttr2 dataset2_distribution SampleBudget2 numPartitions2 rangeScale2 sampleScale2 sampleType2)
                                output_params(segmentCount2 histogram2 sample2)
bin/hadoop jar TopKoperator.jar DataAnalyzer.CalcSelectivity input_params(segmentCount1 segmentCount2 sample1 sample2 numPartitions1 numPartitions2)
                                output_params(Selectivity)
bin/hadoop jar TopKoperator.jar DataAnalyzer.ThetaCompute input_params(Selectivity segmentCount1 segmentCount2 numPartitions1 numPartitions2 rangeScale1 rangeScale2 K mergeFunction)
                                output_params(Theta)
bin/hadoop jar TopKoperator.jar Operator.BuildIndex input_params(dataset2 JoinAttr2 UtilityAttr2 Theta)
                                output_params(indexDataset2)
bin/hadoop jar TopKoperator.jar Operator.TopKjoin input_params(dataset1 JoinAttr1 UtilityAttr1 histogram1 histogram2 indexDataset2 K mergeFunction Theta)
                                output_params(queryResult)
```

Figure 2: An example of a user-defined query script

system such as Hadoop that supports high degrees of parallelism requires further care.

Hadoop's MapReduce framework can perform joins between large datasets using either a map-side join or a reduce-side join, but writing the code to do joins from scratch is fairly involved. So rather than writing MapReduce programs, one can opt for higher-level frameworks such as Pig [9], Hive, or Cascading in which join operations are a core part of the implementation. However, these frameworks fail to efficiently support rank-based query processing – each operator processes most of its input data, and top-k selection involves filtering after a large number of candidates have been produced. Time and resources are wasted in producing unnecessary, low utility results; thus, most of the work done is wasted. Hence, via the `RanKloud` framework, we aim to tackle the scalability challenges posed by large scale media analysis applications that existing batched processing systems fail to handle.

3. SYSTEM ARCHITECTURE

Figure 1 illustrates the architecture of the `RanKloud` framework. The `execution engine`, which consists of the `Sample`, `CalcSelectivity`, `ThetaCompute`, `BuildIndex` and the `Operator` modules, is incorporated into the existing Hadoop infrastructure. These modules drive the entire process of executing a rank-based query submitted by the user. Along with these modules, we integrate into Hadoop a `BerkeleyDB`[2] component in order to provide a way to index datasets during query processing. The `query script` is a user-defined UNIX bash script that is sent to the `execution engine` of `RanKloud`, which then invokes a MapReduce job on the Hadoop cluster. The query result is obtained by accessing Hadoop's Distributed File System (HDFS). Below, we use the top-k join operator as an example to describe the details of the architecture.

3.1 Query Script

As discussed earlier, the `query script` is a user-defined UNIX bash script. This gives the user the flexibility to describe a series of operations that are applied to the input data to produce the output. The user can decide which of `RanKloud`'s modules need to be called in order to execute his/her query. Taken as a whole, the operations describe a workflow, which the `RanKloud execution engine` translates into a series of MapReduce jobs.

Figure 2 shows an example query script that executes a top-k join operation on two datasets. This script causes `RanKloud`'s `execution engine` to: (1) invoke the `Sample` and the `ThetaCompute` modules to collect the necessary statistics of the datasets, and then, (2) the `TopKjoin` operator is called to obtain the k best join results.

3.2 Sampling and Lower Bound Computation

One major difficulty we need to deal with when executing complex media analysis workflows is that the statistics of the intermediary, transient data within the workflow are not available in advance. `RanKloud` alleviates this problem by collecting any information needed to estimate data statistics as a function of the data utility. The `uSplit` method described in [15] is, therefore, essential in achieving waste-avoiding ranked query processing.

The `uSplit` sampling approach is implemented through the following modules: (a) the `Sample` module – based on the sampling budget, number of partitions of the utility space, range scale factor, sample scale factor, input data utilities, data distribution, and choice of either map-side or reduce-side sampling, this module randomly picks sample tuples from each of the datasets. It outputs the sample tuples and histograms of the datasets; (b) the `CalcSelectivity` module – this module estimates the join selectivity of the datasets as a function of the histograms and the sample tuples picked by the `Sample` module; (c) the `ThetaCompute` module – this module computes the lower bound, Θ_k, based on the join selectivity estimated by the `CalcSelectivity` module, the merge function, and K (the number of top results). The histograms and the Θ_k value generated by these modules are sent as inputs to `RanKloud`'s `Operator` module.

3.3 Operator

3.3.1 Utility-aware Partitioner

The `Utility-aware Partitioner` explained in [15] is a custom partitioner that is integrated into the existing Hadoop infrastructure to enable top-k join queries to be processed in parallel. This component forms a part of the `Operator` module, and its main function is to estimate the amount of work that needs to be done to find the top-k results. The partitioner estimates the join work by evaluating the histograms produced by the `Sample` module, and based on this, it creates work-balanced partitions of the

[2]Oracle's Berkeley DB Java Edition 3.3.87 – http://www.oracle.com/technology/software/products/berkeley-db/je/index.html

input data that are sent to the servers for join processing. If the histograms preexist, then the user has the option of reusing them for future query processing.

3.3.2 Top-K Join Operator

Our implementation of the top-k join operator has the following modules: (a) the `BuildIndex` module – this module is connected to the `BerkeleyDB` component to enable each server to create the necessary index structure in order to support efficient join processing. If an index exists, then the user has the option of reusing it; (b) the `TopKjoin` module – each server calls this module in order to execute its portion of the join task based on the the merge function, K, and the lower bound, Θ_k. The results from the individual servers are then combined to select the top-k results.

4. DEMONSTRATION SCENARIOS

This section describes the demo set up and the scenarios. We use real (IMDB[3] and Flickr images[4]) and synthetic datasets of varying sizes and data distributions. `RanKloud` is integrated into Hadoop running on the Ubuntu operating system. This demonstration presents three possible scenarios.

4.1 Two-way Top-k Join Processing

This scenario demonstrates how `RanKloud`'s `uSplit` approach is designed to handle joins between two datasets. A complete user-defined query, similar to the one shown in Figure 2, is executed to show how this workflow is executed on our framework. We will compare our technique to Hadoop and Pig-Latin join algorithms to demonstrate the efficiency of our method. Lastly, we will show how `uSplit`'s data statistics and indexes can be reused in `RanKloud` for top-k join queries that are executed repeatedly.

4.2 Multi-way Top-k Join Processing

Through this scenario, we demonstrate how the `uSplit` partitioning strategy is extended to handle joins between multiple datasets. The sampling and lower bound computation modules have the ability to collect the necessary data statistics and estimate the lower bound, Θ_k, even for high dimensional join processing. The top-k join operator is extended to efficiently process joins between multiple datasets based on the statistics it receives.

4.3 Other Ranked Processing Operators

In this scenario, we show `RanKloud`'s ability to handle skyline and k-nearest neighbor search operations on large datasets. Parallel versions of these operators are implemented to build a complete ranked query processing framework. Our future research goals involve extending `RanKloud` to support join-based skyline and k-closest pair queries. While the overall adaptive approach is similar to the top-k join operator, the specifics of work partitioning and pruning strategy will differ from one operator to another.

5. CONCLUSION

This demonstration introduces a new data processing framework on Hadoop called `RanKloud`. `RanKloud` supports ranked query operations on large media datasets in a waste-avoiding manner by treating the utilities of the data to be an integral part of the analysis process. We demonstrate the effectiveness of `uSplit` in handling top-k join operations. `uSplit` considers the ranked semantics of the analysis operations, as well as the data and utility characteristics discovered at runtime, in deciding on a data partitioning and resource allocation strategy. We also demonstrate the effectiveness of `RanKloud` in supporting other ranked operators such as the skyline and k-nearest neighbor search operators. Our future research will involve extending `RanKloud` to support join-based skyline and k-nearest neighbor queries, in which the data points on which the skyline/k-nearest neighbor query is executed are not available, but an explicit join operation needs to be carried out in order to discover these points.

[3]http://www.imdb.com/interfaces

[4]http://press.liacs.nl/mirflickr/

6. REFERENCES

[1] Amazon. Elastic MapReduce. http://aws.amazon.com/elasticmapreduce.

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[3] R. Chaiken, B. Jenkins, P. A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, 2008.

[4] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[6] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with MapReduce. In *HLT*, pages 265–268, 2008.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[8] J. J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In *SIGIR*, pages 155–162, 2009.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[10] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[11] S. Singh, J. Kubica, S. Larsen, and D. Sorokina. Parallel large scale feature selection for logistic regression. In *SDM*, pages 1171–1182, 2009.

[12] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel Skyline computation. In *SIGMOD Conference*, pages 227–238, 2008.

[13] Yahoo! Hadoop. http://hadoop.apache.org, 2008.

[14] B. Yao, F. Li, and P. Kumar. K nearest neighbor queries and kNN-joins in large relational databases (almost) for free. In *ICDE*, pages 4–15, 2010.

[15] R. Yu, M. Nagendra, P. Nagarkar, K. S. Candan, and J. W. Kim. Workload-balanced processing of top-K join queries on cluster architectures. *Technical Report: ASUCIDSE-CSE-2010-001*, 2010.