Beyond Online Aggregation: Parallel and Incremental Data Mining with Online Map-Reduce

Joos-Hendrik Böse
Intl. Comp. Sci. Institute
Berkeley,USA
jboese@icsi.berkeley.edu

Artur Andrzejak
Zuse Institute Berlin (ZIB)
Berlin, Germany
andrzejak@zib.de

Mikael Högqvist
Zuse Institute Berlin (ZIB)
Berlin, Germany
hoegqvist@zib.de

ABSTRACT

There are only few data mining algorithms that work in a massively parallel and yet online (i.e. incremental) fashion. A combination of both features is essential for mining of large data streams and adds scalability to the concept of Online Aggregation introduced by J. M. Hellerstein *et al.* in 1997. We show how an online version of the Map-Reduce programming model can be used to implement such algorithms, and propose a solution for the "hardest" class of these algorithms - those requiring multiple Map-Reduce phases. An experimental evaluation confirms that the proposed methods can substantially accelerate interactive analysis of large data sets and facilitate scalable stream mining.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Data Mining

General Terms

Algorithms, Measurement, Performance

Keywords

Map-Reduce, Stream Mining, Machine Learning

1. INTRODUCTION

Since the early days of computing, users were interested in reducing time between job submission and obtaining results. For example, in interactive data analysis the operator typically chooses next exploration steps based on the result of the last query [18, 5]. The overall speed of such an analysis depends critically on the *Time-to-Solution*, i.e. the query response time.

Parallel and distributed processing has been the major tool to face the challenges of reducing this time, especially when very large data sets or computationally-intensive algorithms are involved. The *Map-Reduce* programming model has recently become a primary choice for fault-tolerant and massively parallel data crunching [7, 3]. The drawback of established Map-Reduce implementations is that they work only in batch mode and do not allow stream processing or exploiting of preliminary results.

Online Aggregation is an alternative approach to reduce the Time-to-Solution [14, 13]. This technique, developed in the context of databases, allows monitoring of preliminary results of an aggregation query along with estimates on the result's error intervals. Among others, this empowers a user to stop or modify a running task if the accuracy of a preliminary result is sufficient, or conversely, if the result is likely to be meaningless. The current implementation of this technique (notably, as a prototype for PostgreSQL) has only limited scalability as parallel processing is not used. Furthermore, it requires potentially complex analytical formulas to compute error intervals for each type of aggregation operator.

Combining both approaches - i.e. the Map-Reduce-based processing and exploiting of preliminary results - gives a data analyst two knobs for reducing the Time-to-Solution: either by increasing the degree of parallelism or by a more aggressive usage of early results. If used together, these tools promise scalability beyond the capacities of today's systems. This goal is targeted by a recent extension of the open-source Hadoop framework which allows *online* distributed computations under the Map-Reduce model [6].

We propose an alternative implementation (devised and developed independently) of an online Map-Reduce framework under the shared-memory architecture. The focus of our study is not on large-scale framework architecture (contrary to [6]) but on the challenges of parallel data analytics. These challenges include:

- Support for estimating the convergence of the preliminary results in an algorithm-independent way and detailed progress reporting.
- Adaptation of data mining algorithms for incremental processing with Map-Reduce. Special focus is given to algorithms which require multiple phases, such as k-means [4].

Specifically, our contributions are the following:

- We propose a shared-memory Map-Reduce framework which works in an online fashion. In addition to finite data sets it is capable to process data streams.
- We provide a set of architectural extensions to estimate the convergence of preliminary results and to monitor

MDAC '10, April 26, 2010 Raleigh, NC, USA

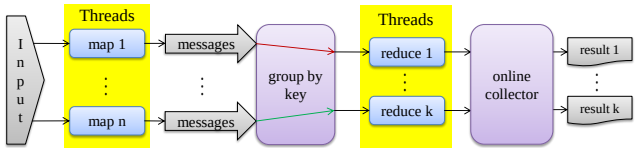

Figure 1: Architecture of the online Map-Reduce framework

scheduling fairness via detailed reporting of the processing progress. Contrary to aggregation-specific estimation of error intervals in [14] our approach works with any algorithm and does not make any assumptions on the data distribution.

- We propose a further modification of the online Map-Reduce architecture to handle algorithms which require multiple Map-Reduce phases. As a proof-of-concept we design and evaluate a version of the k-means clustering algorithm which runs in an incremental mode in our framework.

- We evaluate the potential of exploiting preliminary results under Map-Reduce and the convergence estimation tools on several algorithms (word count, linear regression, PCA, Naive Bayes, k-means). Our evaluation indicates that (for our data sets) using preliminary results can reduce the processing time by 50%-70% with a moderate quality degradation.

Paper structure. In Section 2 we present the architecture of our shared-memory online Map-Reduce framework. Section 3 describes data mining algorithms and their adaptation to our framework. Section 4 is devoted to the experimental evaluation. Section 5 discusses related work. We conclude with Section 6.

2. ONLINE MAP-REDUCE FRAMEWORK

The Map-Reduce programming model [7] has recently become the tool of choice for fault-tolerant distributed processing of massive data sets. Google's proprietary implementation was followed by open-source projects such as Hadoop and extensions like Dryad [15]. In essence, there are three processing steps under this model. First, input data is partitioned and processed independently by *map* tasks, with each one emitting a list of $<key, value>$ pairs as output. In the second step these pairs are grouped by keys, yielding for each unique key k a list $value_1, \ldots, value_t$ of all values belonging to the same key k. In step three, the "per-key" lists are processed independently by *reduce* tasks, which collectively create the final output of one Map-Reduce phase. Note, that many algorithms require multiple such phases to complete. The primary reason why existing Map-Reduce implementations work only in batch mode is that the output of map tasks is completely written to the file system before commencing step two (grouping by keys).

2.1 Architecture

In order to access and utilize preliminary processing results (i.e. those obtained on a part of the input data) we have implemented a shared-memory Map-Reduce framework, see Figure 1. While this type of architecture allows only limited scalability, our framework is primarily intended as a testbed for implementing and evaluating incremental, parallel data mining algorithms. As these algorithms are designed to work under any degree of parallelism, the presented approaches will achieve full scalability and fault-tolerance when deployed on distributed frameworks such as Hadoop Online [6].

Our framework has several differences compared to a batch system. First, input data for the map tasks can be either read in from files or from data streams. These mappers output $<key\, k, value>$ pairs periodically and as events. The "group by key" step is implemented as a hash map (dictionary) mechanism, where each pair is immediately assigned to the input queue of a unique reducer corresponding to k. Also reducers output their results regularly (after having processed a specified amount of input). These partial results are collected in a new component called "collector" which computes a preliminary (or final) result used for convergence estimation and optional visualization. The collector is also responsible for estimating the progress of computation in non-streaming mode, see Section 2.2. The number of individual results collected or aggregated before emitted by a mapper (or reducer) is called *mapper* (*reducer*) *bucket size*.

For the implementation we used the actor library of the Scala programming language. This allows to have a very high number of map tasks (on the order of millions). The same mechanism is applied to reducers, allowing the convenience of having only one key per reducer.

2.2 Progress and Convergence Estimation

When deployed in a non-streaming mode, our framework allows a user to take decisions about the current processing step before all input data is processed. The corresponding actions might include:

- Cancelling of the whole processing if already preliminary results are meaningless or of low-quality.

- Accepting an intermediate result (and stopping further processing) if its accuracy is sufficient.

- Preparation of further analysis steps based on preliminary results.

In order to balance the risks of using a preliminary result against the benefits of an accelerated computation, two questions need to be answered: "To what degree is the result likely to change if all data is processed?" and "How much time can be saved if the current result is used?". Our framework helps to answer these questions by providing mechanisms for monitoring the convergence of the results and the processing progress. To use these mechanism, an algorithm's designer (called subsequently a user) needs to provide (a small amount of) own code which takes care of algorithm's specifics.

To support estimation of processing progress, we provide as a part of the collector API, information on the relative progress for all existing reducers. Note, that the set of all reducers corresponds to the set of all keys emitted by any mapper (up to current time). For a reducer with key k, a user's code can query (among others) its relative progress: an estimate which fraction of all values $value_1, \ldots, value_t$ corresponding to k has been processed so far by this reducer.

To compute this estimate we monitor (i) the size of all input data already processed by (all) mappers, and (ii) (for each k) how many pairs with key k are emitted (collectively) by the mappers per 1 kbyte of processed input.

This information is much more detailed than the progress metric provided e.g. by Hadoop [6] but it is beneficial if (relative) progress of reducers is heterogeneous. For example, in our linear regression algorithm we only have two keys A and B, representing left and right side of a matrix equation to be solved [4]. If the relative progress for A is 0.75 and for B only 0.25, the preliminary result represents only 25% of all data. In this case, user's code should compute the minimum of all "per-key" progress values to report the total progress, while for other algorithms different aggregation functions (average, maximum) might be preferred.

2.2.1 *Convergence Estimation*

Our framework provides online plots to help the user in estimating expected changes of the results in course of the processing. These plots termed *convergence curves* show a metric dif (explained below) against the processing progress (x-axis), see Figure 4. The metric dif() ≥ 0 expresses an algorithm-specific difference between two results. In more detail, our collector API allows periodical computation and storing of *signatures* sig(R) of an (intermediate) result R. The signatures are introduced to save memory, and are implemented by the user. They are either a full result (e.g. d coefficients in linear regression, d small), or a compact representation of a result or its quality (e.g. frequencies of most common 100 words in a word count algorithm). The metric dif() takes as arguments two signatures and computes their "distance" as a scalar. If sig(R) is a vector, dif is likely to be implemented as an Euclidean distance. For scalar signatures, dif is the absolute difference.

Currently we plot values dif(sig(R_i), sig(R_F)), where R_F is the last known result, and R_i iterates over periodically sampled intermediate results from the oldest one until R_F. A user typically uses the slope of the convergence curve to assess the degree by which the result is still likely to change. While this approach is generic enough to work with any type of algorithm, it has also a serious drawback: it does not support to assess the absolute accuracy of the result via mechanisms such as error intervals. We will address this problem as a part of the future work.

2.3 Handling Multi-Phase Algorithms

Several popular machine learning algorithms have been successfully adapted to the (batch) Map-Reduce model [4]. While conversion to online processing mode is easy for algorithms which require only *one* Map-Reduce phase (see Section 3.1), algorithms with multiple such phases (such as k-means, EM, SVM) pose a challenge. Consider the following naive approach of adapting to online processing. Let R be a preliminary result of the 1st phase. In online mode, this result is immediately used as the input to the 2nd phase. When the processing of the 1st phase progressed, a new output R' is issued. If this output is used as a new input for the 2nd phase, all the processing of R in the 2nd and subsequent phases is wasted.

The idea of our approach is to "re-submit" preliminary results to the mappers as a part of the input while having only one running Map-Reduce phase. The implementation requires only minor modifications of the framework shown in Figure 1: the APIs of the mappers are changed to receive preliminary results (or some variations thereof) in form of events. This solution is suitable for algorithms which require multiple phases due to some iterative computing process. It is also beneficial for efficiency if the "re-submitted" data has small size compared to the input data. Both conditions apply in the case of the k-means clustering algorithm (Section 3.2).

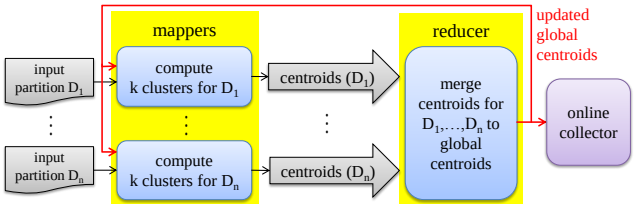

Figure 2: Overview of the online k-means algorithm

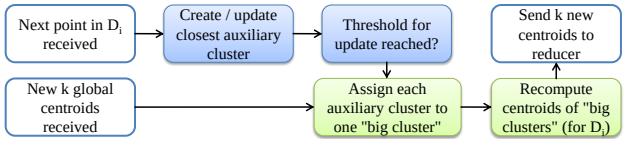

Figure 3: Mapper computation in online k-means

3. ALGORITHMS

3.1 One-Phase Algorithms

The design of data mining algorithms which require only one Map-Reduce phase follows [4]. The general idea is to decimate the size of the input files via computing some small-size intermediate representation in the mappers. This compact representation is forwarded to the reducers for aggregation, and finally to the collector for computation of the final result. This is illustrated on the example of the Principal Component Analysis (PCA).

Let matrix X with n rows and d columns represent our input data, i.e. n data points of dimension d. A PCA task boils down to computing the eigenvectors of the covariance matrix $Y = \frac{1}{n}X^TX - \mu^T\mu$ (a d-by-d matrix), where μ is a 1-by-d vector with means for every column of X. We compute $A = \frac{1}{n}X^TX$ and $M = \mu^T\mu$ separately and in parallel. Assuming that $n \gg d$, the "traditional" matrix multiplication (i.e. component-wise multiplication of ith row of X^T with jth column of X to obtain $y_{i,j}$) is not scalable, as at most d^2 mappers could be used concurrently. Therefore, for each data row x_i (ith row of X) we compute a d-by-d matrix $P_i = x_i^Tx_i$. This computation can be parallelized by at most n mappers. The resulting matrices P_i are sent "online" to the reducer 1, which simply adds them up and periodically forwards this matrix sum $\sum_1^k P_i$ (and the matrix count k) to the collector. To compute μ, we just send the x_i's from all the mappers to the reducer 2, which aggregates them as S_j and records their current number j. In the collector we compute the preliminary covariance matrix $Y' = \frac{1}{k}\sum_1^k P_i - (\frac{1}{j}S_j^T)(\frac{1}{j}S_j)$, and the eigenvectors of Y'

(note that we might have $j \neq k$). As d is small, the latter computation is fast. We further optimize the communication cost: instead of sending each matrix P_i and each data row x_i, we aggregate them already in mappers according to the mapper bucket size (Section 2.1).

3.2 Online K-Means Algorithm

The k-means algorithm groups a set of (numerical) d-vectors into k clusters, where k is part of the input. This algorithm is inherently interactive and requires in batch mode multiple Map-Reduce phases [4]. The version proposed here is an example of the approach described in Section 2.3. The idea is based on the parallel k-means presented in [8]. Figure 2 gives an overview of the processing.

We assume that the output consists of k *centroids* (d-vectors representing cluster centers) and not the grouping of the data set (albeit this could be added easily). Input data is partitioned into sets $D_1, \ldots, D_n$, each assigned to a separate mapper. A result of the incremental processing (described below) of a single mapper m is a set of k preliminary centroids for the already received part of D_m. These results are periodically sent to the (single) reducer. In more detail, instead of a centroid of the ith cluster C_i^m ($i = 1, \ldots, k$) of mapper m, we send the vector sum $C_i^m.\boldsymbol{sum}$ of all points assigned to C_i^m and their count $C_i^m.num$, see [8]. The reducer computes from this data the *global* centroids: for the ith of the k clusters its centroid is given as

$$\frac{1}{\sum_{m=1}^{n} C_i^m.num} \sum_{m=1}^{n} C_i^m.\boldsymbol{sum}.$$

This data is periodically forwarded to the collector and also "re-submitted" to the mappers as shown in Figure 2.

The essence of a single iteration in a mapper is analogous to the batch algorithm in [8]. Given a local data set (part of D_i) and k global centroids, (1) assign each data point to the closest centroid, and (2) for each cluster $i = 1, \ldots, k$ update its centroid according to the (possibly new) point assignment, see Figure 3. The problems of this schema in the online setting are: we might not have enough memory to record all points when processing streams, and computation might become too expensive if the number of stored points grows and input updates are frequent.

Both problems can be solved by limiting the number of "historical" points to be stored. To this purpose we introduce *auxiliary clusters* (AC) which represent the input points (the other k clusters are called now "big" clusters). Upon arrival of a new point, either a new AC is created, or it is added to a closest existing AC. Cluster creation happens only if the maximum number of ACs is not reached, and the distance of the new point is larger than the average distance between two ACs. If a point is added to an existing AC, the vector sum of all points in this AC and their count are updated.

When the update of the "big" clusters is initiated (see Figure 3) and the computations (1) and (2) need to be performed, the centroids of the ACs are treated as data points yet weighted by the cardinality of respective AC. The number of ACs per mapper is usually much larger than k. This number allows balancing between the degree of approximation and memory/CPU requirements.

4. EXPERIMENTAL RESULTS

Name	Algorithm	Data Set	Size (MB)	dif type
wc-g	word count	G	17000	MSE
lr-s	lin. regression	S1	7600	MSE
pca-s	PCA	S2	3800	MSE
nb-m	Naive Bayes	M	1.3	MR
nb-u	Naive Bayes	U	250	MR
km-c	k-means	C	363	MSE

Table 1: Pairs (algorithm, data set) used for experiments

Our experiments consisted in running an algorithm with one or more data sets (both specified below) in online mode while recording values of the metric dif (Section 2.2.1) and the time passed for each 1% of processed data. We have also run these experiments in the offline mode (i.e. with a very large mapper bucket size; for k-means in Matlab) and recorded the total time. Apart from the k-means algorithm this corresponds to batch processing as outlined in [4]. Table 1 lists the pairs (algorithm, data set) used in the experiments, where the data sets are:

G Texts of English Gutenberg Books[1] from 03/23/2010

S1 Synthetically generated 10^8 data rows, each with a random point x in $\mathcal{R}^4$ and a dependent variable generated by adding noise to product $x\,[1, 2, 3, 4]^T$

S2 Synthetically generated $5 \cdot 10^7$random points in $\mathcal{R}^4$

M Spambase[2] from UCI Machine Learning Repository

U URL Data Set from [16]

C US Census Data[3] (1990) from UCI Machine Learning Repository.

We have used the following types of the dif metric for measuring distance between two results: MSE - Mean Squared Error between result signatures represented as vectors (e.g. for linear regression 4 approximated coefficients); MR - misclassification rate on a test set. For experiments we used a system with 8 quad-cores (2.4 GHz) and 256 GB main memory running under Red Hat Enterprise Linux 5.1. Our framework has been deployed on Java 1.6 (JDK 6u18, 64-bit) and Scala 2.7.7.

4.1 Evaluation of Convergence and Time

Figure 4 shows the convergence curves of all six experiments. Here the x-axis shows the fraction of processed data, and the y-axis values of the dif metric between current preliminary result and the final result. While these curves are available only after all data is processed (and so not available online), we show them to illustrate the potential of exploiting preliminary results. Except for `wc-g` and `nb-u`, preliminary result approximates the final result "reasonably well" after only 20% to 40% of all data is processed.

To quantify what "reasonably well" means, we introduce the following definitions. For a given experiment, let *dif-range* D be the 95-percentile highest value of all dif-values

[1] http://www.gutenberg.org
[2] http://archive.ics.uci.edu/ml/datasets/Spambase
[3] http://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)

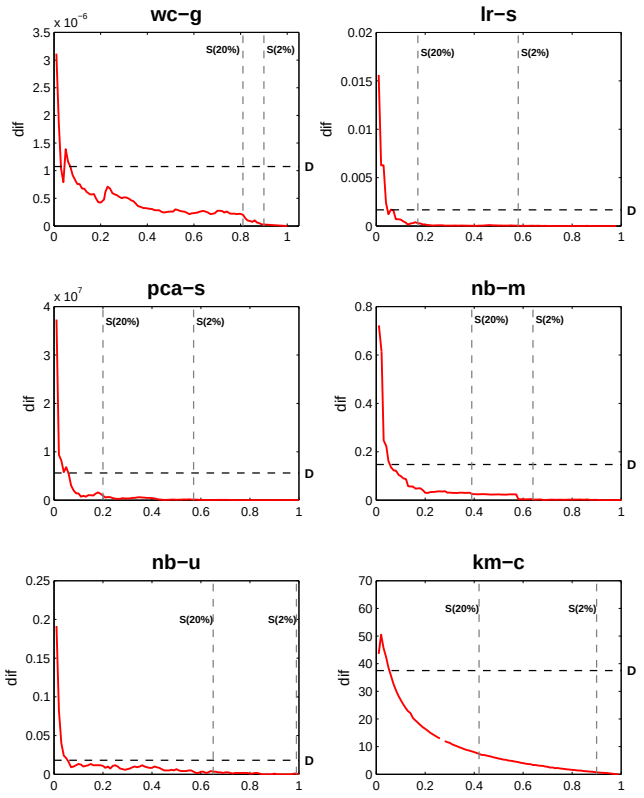

Figure 4: Convergence curves for all experiments (x-axis: fraction of processed input, y-axis: value of dif between preliminary and final result)

encountered during the processing (i.e. in an experiment D is the highest value among the dif-values after removing the highest 5% of them), see horizontal lines in Figure 4. For a given percentage y% called *error level* we define $S(y)$ as the smallest fraction r of the data which needs to be processed such that all values of dif encountered afterward are no larger than $\frac{y}{100}D$. In other words, if we remove from a convergence curve the 5% highest dif values, $S(y)$ is the smallest value on the x-axis such that the convergence curve right of $S(y)$ is never larger than y% of the whole y-axis range (which is D).

Table 2 shows these input fractions for error levels of 20%, 10%, 5%, 2% and 1%. For example, to achieve the error level of 10% (or less) we need to process at most 36% of the data for `pca-s` and `lr-s`, while `nb-m` requires 58% and `km-c` 62% of the input to be processed. Experiments `nb-u` and `wc-g` require almost all data to be processed, which is also confirmed by Figure 4.

Table 3 shows (in minutes) the total time T_{off} needed by the offline algorithm and times T_{on} of the online algorithm to process all data. We have expected the offline algorithm to be more efficient as the need for inter-component communication and event creation is much lower. However, in all cases execution time of the online algorithm is smaller. We attribute this to the overhead caused by allocating much larger amount of heap memory in the mappers (in the offline algorithms). This is confirmed by case `nb-m` - the small size of the data set (1.3 MB) makes here effects of memory allocation negligible, and consequently $T_{\text{off}} = T_{\text{on}}$.

Name	$S(20\%)$	$S(10\%)$	$S(5\%)$	$S(2\%)$	$S(1\%)$
wc-g	0.81	0.83	0.88	0.90	0.90
lr-s	0.17	0.19	0.47	0.58	0.68
pca-s	0.2	0.36	0.42	0.57	0.7
nb-m	0.39	0.58	0.58	0.64	0.87
nb-u	0.65	0.72	0.84	1.0	1.0
km-c	0.42	0.62	0.78	0.90	0.96

Table 2: Minimum fraction of input necessary for decreasing error levels

Name	wc-g	lr-s	pca-s	nb-m	nb-u	km-c
T_{off}	46.7	5.01	11.2	1.8	42.0	-
T_{on}	42.4	4.26	7.6	1.8	34.7	37.5

Table 3: Time of executing offline and online algorithms (in minutes)

4.2 Accuracy Evaluation

For `wc-g`, `lr-s`, `pca-s`, `nb-m`, and `nb-u` online and offline algorithms yield identical final results. In contrast, the final results of the online k-means differ from those of the standard (offline) k-means due to a modified approach. We analyze this difference in terms of the sum U of Euclidean distances between each point and its closest centroid. The offline version of k-means (as implemented in Matlab 2007a) yielded for the final result $U_{\text{off}} = 1.52905 \cdot 10^9$ while our online algorithm gives $U_{\text{on}} = 1.98541 \cdot 10^9$ with 1000 auxiliary clusters (ACs), see Figure 5 (left). The accuracy of the final result improves with the number of ACs up to 1000 but not beyond this number. Figure 5 (right) relates the difference between U_{off} and U_{on} to values of U for intermediate results of our online k-means algorithm.

4.3 Scalability Evaluation

Scalability is evaluated by changing the number of map tasks per run for the experiments `lr-s` and `km-c`, see Figure 6. Since each mapper is provided with an own core, this number effectively controls the degree of parallelism. The speed-up is not linear in the number of used cores; in case of `lr-s`, deploying more than 15 mappers does not shorten execution time, which might indicate that this algorithm is constrained by the I/O-bound throughput of our shared-memory machine.

5. RELATED WORK

Processing of massive data. With few exceptions in the domain of supercomputing, most such processing is performed on clusters of (possibly commodity) machines. Here the Map-Reduce programming model popularized by Google [7] turned out successful in coping with issues of fault-tolerance and scalability. Follow-up open-source frameworks are Hadoop and the recent Hadoop Online system [6]. Extensions of the Map-Reduce model are presented under the Microsoft's Dryad project [15] and in [3].

Interactive data analysis. There are several approaches to reduce time of interactive large-scale data analysis and in-

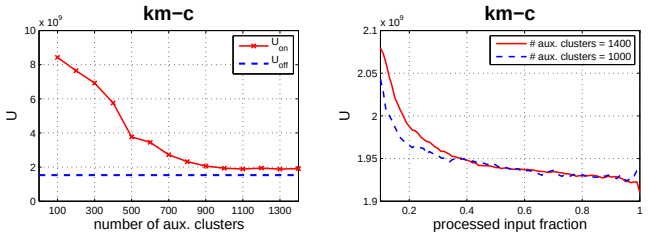

Figure 5: Left: U of the final result for growing numbers of auxiliary clusters; Right: U of intermediate results

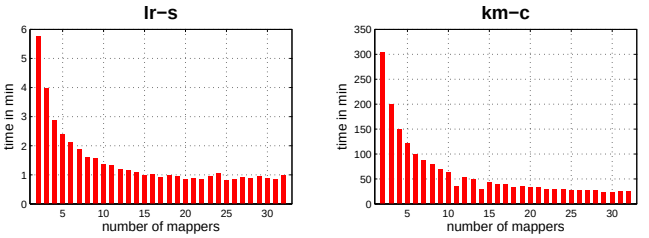

Figure 6: Execution time for increasing levels of parallelism

crease its flexibility. [18] proposes query templates (query preprocessing). Authors of [5] discuss support for flexible storage and analytics of large data sets from DB perspective. Relevant are also languages such as Sawzall [20] and Pig Latin as they facilitate fast programming of queries and data mining studies under Map-Reduce.

Online Aggregation and incremental learning. The concept of exploiting preliminary query results has been introduced 1997 in [14] and refined in [13]. There is a relation to incremental learning, i.e. training of machine learning models with new data added incrementally. Many mining algorithms have corresponding versions, including decision trees [10], SVMs [9], or clustering [11]. Note: in relation to k-means clustering, the term *incremental* frequently label algorithms which try to avoid local minimum, see e.g. [19].

Parallel machine learning. Most parallel machine learning algorithms feature proprietary design and consequently limited scalability due to lack of fault-tolerant large-scale frameworks. Examples include parallel SVMs [12], or clustering [8, 17]. A notable exception is work [4] which proposes implementations of common algorithms in the (batch) Map-Reduce model. To our knowledge there exists no parallel *and* online k-means algorithm prior to this work.

Stream Processing. The algorithms presented here can be also deployed for (parallel) stream mining. Research in this domain includes frameworks (Borealis, STREAM, TelegraphCQ, DataCell/MonetDB), query languages [2], and Complex Event Processing systems.

6. CONCLUSIONS

We presented a set of data mining algorithms for parallel, incremental processing under online Map-Reduce. In particular we introduced an approach to handle iterative algorithms which need multiple Map-Reduce phases, and proposed a version of the k-means clustering algorithm based on our approach. Further results are tools for estimating progress and convergence of the preliminary results. Experiments show that our algorithms are capable to combine benefits of Online Aggregation with parallelism.

Future work includes adapting other iterative data mining algorithms (SVM, EM) [4] to online processing. We will also refine convergence monitoring, and work on tools to handle concept drift in data. Further topics are redundant online data processing under unreliable resources [1].

7. ACKNOWLEDGMENTS

This work is carried out in part under the EC projects SELFMAN (FP6-034084) and eXtreemOS (FP6-033576).

8. REFERENCES

[1] Andrzejak, A., Kondo, D., and Anderson, D. P. Exploiting non-dedicated resources for cloud computing. In *12th IEEE/IFIP Network Operations & Management Symposium (NOMS 2010)* (Osaka, Japan, Apr 19–23 2010).

[2] Arasu, A., Babu, S., and Widom, J. The CQL continuous query language: semantic foundations and query execution. *VLDB J 15*, 2 (2006), 121–142.

[3] Chen, S., and Schlosser, S. W. Map-reduce meets wider varieties of applications. Tech. Rep. IRP-TR-08-05, Intel Labs Pittsburgh Tech Report, May 2008.

[4] Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G. R., Ng, A. Y., and Olukotun, K. Map-reduce for machine learning on multicore. In *NIPS* (2006), pp. 281–288.

[5] Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J. M., and Welton, C. MAD skills: New analysis practices for big data. *PVLDB 2*, 2 (2009), 1481–1492.

[6] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. Mapreduce online. Tech. Rep. UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[7] Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. *CACM 51*, 1 (2008), 107–113.

[8] Dhillon, I. S., and Modha, D. S. A data-clustering algorithm on distributed memory multiprocessors. In *Workshop on Large-Scale Parallel KDD Systems, SIGKDD* (London, UK, 2000), pp. 245–260.

[9] Domeniconi, C., and Gunopulos, D. Incremental support vector machine construction. In *ICDM'01* (Washington, DC, USA, 2001), pp. 589–592.

[10] Domingos, P., and Hulten, G. Mining high-speed data streams. In *Knowledge Discovery and Data Mining* (2000), pp. 71–80.

[11] Ester, M., Kriegel, H.-P., Sander, J., Wimmer, M., and Xu, X. Incremental clustering for mining in a data warehousing environment. In *VLDB'98* (San Francisco, CA, USA, 1998), pp. 323–333.

[12] Graf, H. P., Cosatto, E., Bottou, L., Durdanovic, I., and Vapnik, V. Parallel support vector machines: The cascade SVM. In *NIPS* (2004).

[13] Hellerstein, Avnur, Chou, Hidber, Olston, Raman, Roth, and Haas. Interactive data analysis: The control project. *COMPUTER: IEEE Computer 32* (1999).

[14] Hellerstein, J. M., Haas, P. J., and Wang, H. J. Online aggregation. In *Proc. ACM SIGMOD Int. Conf. Management of Data* (13–15 May 1997), vol. 26(2), pp. 171–182.

[15] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007), ACM, pp. 59–72.

[16] Ma, J., Saul, L. K., Savage, S., and Voelker, G. M. Identifying suspicious URLs: an application of large-scale online learning. In *ICML* (2009), vol. 382, ACM, p. 86.

[17] Olson. Parallel algorithms for hierarchical clustering. *PARCOMP: Parallel Computing 21* (1995).

[18] Olston, C., Bortnikov, E., Elmeleegy, K., Junqueira, F., and Reed, B. Interactive analysis of web-scale data. In *CIDR* (2009), www.crdrdb.org.

[19] Pham, D. T., Dimov, S. S., and Nguyen, C. D. An incremental k-means algorithm. *Journal of Mechanical Engineering Science 218*, 7 (2004), 783–795.

[20] Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4 (2005), 277–298.