Scaling Data Mining in Massively Parallel Dataflow Systems

Sebastian Schelter
Technische Universität Berlin
sebastian.schelter@tu-berlin.de
Supervisor: Volker Markl
Expected graduation date: December 2014

ABSTRACT

The demand for mining large datasets using shared-nothing clusters is steadily on the rise. Despite the availability of parallel processing paradigms such as MapReduce, scalable data mining is still a tough problem. Naïve ports of existing algorithms to platforms like Hadoop exhibit various scalability bottlenecks, which prevent their application to large real-world datasets. These bottlenecks arise from various pitfalls that have to be overcome, including the scalability of the mathematical operations of the algorithm, the performance of the system when executing iterative computations, as well as its ability to efficiently execute meta learning techniques such as cross-validation and ensemble learning.

In this paper, we present our work on overcoming these pitfalls. In particular, we show how to scale the mathematical operations of two popular recommendation mining algorithms, discuss an optimistic recovery mechanism that improves the performance of distributed iterative data processing, and outline future work on efficient sample generation for scalable meta learning.

Early results of our work have been contributed to open source libraries, such as Apache Mahout and Stratosphere, and are already deployed in industry use cases.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Data Mining

Keywords

scalable data mining, distributed processing

1. INTRODUCTION

In recent years, the cost of acquiring and storing data of large volume has dropped significantly. As the technologies to process and analyze these datasets are being developed, we face interesting new possibilities. Businesses can apply advanced data analysis for data-driven decision making, and scientists can test hypotheses on data several orders of magnitude larger than ever before. The analysis is often conducted using machine learning or graph mining approaches. Due to the size and complexity of the data, scaling-out the analysis to parallel processing platforms running on large, shared-nothing commodity clusters is popular [1, 2].

Although there is a high demand for this kind of analysis, scalable data mining is still a very hard problem in real-world applications and an active area of research [2, 3]. We use an example to illustrate the difficulty of the problem: *Principal Component Analysis (PCA)*, which gives insight to the internal structure of a dataset and constitutes a standard analysis technique in a wide range of scientific fields (e.g. signal processing, psychology, quantitative political science). PCA projects the data onto a lower dimensional space, such that the variance of the data is maximized [4]. Algorithm 1 shows the three steps that a typical PCA implementation conducts to compute the first k principal components, the basis of this space. The dataset to analyze consists of N observations, which are represented by the rows of a matrix X. At first, the data has to be centered, i.e., the mean of every column is subtracted from the values of that column (cf. line 1). Next, we compute the matrix multiplication $XX^\top$ and divide the result by the number of observations N to obtain the covariance matrix C (c.f. line 2). In line 3, we compute the k first eigenvectors of the covariance matrix C which correspond to the k first principal components of X.

Algorithm 1: Computing principal components of a matrix X

1	$X = X - \frac{1}{N}\sum_i x_i$	`# centering of the data`
2	$C = \frac{1}{N} X X^\top$	`# covariance matrix`
3	$P = \text{eigs}(C, k)$	`# principal components`

1.1 Pitfalls in Scalable Data Mining

Imagine a real-world application, where we want to apply PCA to a large, sparse dataset such as a text corpus in 'bag-of-words' representation to gain insights about the relationships of term occurrences. It is not sufficient to implement the simple steps from Algorithm 1 on a MapReduce-based system such as Hadoop [5] to get a scalable and efficient PCA implementation. The reasons why it is hard to scale the three simple lines of PCA very well illustrate some major points about scaling-out data mining algorithms.

Algorithm Scalability: The first and most fundamental aspect is the scalability of the mathematical operations of the algorithm itself. Careful investigation of the applied operations reveals two scalability bottlenecks in the algorithm. The first bottleneck is caused by the centering of the data. A lot of modern data analysis tasks deal with very large and at the same time very sparse matrices (e.g. adjacency matrices in social network analysis [6] or matrices representing interactions between users and items in recommendation mining [7]). Often, the ratio of non-zero entries to

SIGMOD'14 PhD Symposium Snowbird, USA
ACM 978-1-4503-2924-8/14/06 ...$15.00.
http://dx.doi.org/10.1145/2602622.2602631.

the overall number of cells in the matrix is one one-thousandth or less [6, 7]. These large matrices are efficiently stored by only materializing the non-zero entries. Applying the first step in PCA (line 1 in Algorithm 1), which centers the data, to such sparse matrices would have devastating consequences. The majority of zero entries become non-zero values and the materialized data size increases by several orders of magnitude, thereby posing a serious scalability bottleneck. The second bottleneck is caused by the matrix multiplication $XX^\top$ conducted to compute the covariance matrix in line 2 of Algorithm 1. Its runtime is quadratic in the number of observations that our dataset contains (which is equal to the number of rows of the input matrix X). Computations with quadratic runtime quickly become intractable on large inputs. Therefore, a direct implementation of the standard PCA algorithm will not scale [1]. The described scalability bottlenecks illustrate the properties of a scalable data mining algorithm. Its operations must preserve data sparsity, a data-parallel execution strategy on partitioned input must exist and their runtime must be at most linear.

Efficient Execution of Iterative Algorithms: Many data mining algorithms exhibit an *iterative* nature. Such algorithms start from an initial state and use the data to iteratively refine this state until a convergence criterion is met. In our example, we need a scalable way to compute the k first eigenvectors of the covariance matrix in the implementation of the *eigs* function in line 3 of Algorithm 1. The iterative Lanczos algorithm is the predominant scalable method to compute these eigenvectors [9]. A parallel processing system suitable for data mining must be able to efficiently execute iterative programs. When executing such computations in large clusters, guaranteeing fault tolerance is a mandatory task. Current systems for executing iterative tasks in large scale clusters typically achieve fault tolerance by checkpointing the result of an iteration as a whole [10, 11, 12]. In case of a failure, the system halts execution, restores a consistent state from a previously saved checkpoint and resumes execution from that point. This approach is problematic as it incurs a high overhead. In order to tolerate machine failures, every machine has to replicate its checkpointed data to other machines in the cluster. Figure 1 shows the runtime spent for checkpoint replication and for actual computation during an iteration of computing PageRank on a large graph [13]. We see that more than eighty percent of the time is needed for guaranteeing fault tolerance. Furthermore, due to the pessimistic nature of the approach, the overhead is always incurred, even if no failures happen during the execution, because the checkpoints must be written in advance. It is highly desirable to find a mechanism that guarantees fault tolerance for iterative computations and at the same time avoids overhead in failure-free cases.

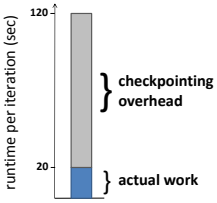


Figure 1: Checkpointing overhead.

Scalable Meta Learning: Research on scalable data mining very often solely focuses on finding efficient ways to execute algorithms for training machine learning models. However, in order to apply a lot of these models to the real world, additional work is necessary. Most models have hyperparameters that need to be carefully chosen as they heavily influence the quality of a model (parameter selection). Another important task is feature selection, where we aim to select a subset of the features (or a subset of combinations of the features) that provides sufficient model quality. In PCA, we might for example want to eliminate columns from the data matrix X that have negligible impact on the resulting principal components [14]. These tasks can be solved via meta learning techniques such as cross-validation that typically operate in two steps. First, samples are generated from the data, then models are trained and evaluated on the sample data. While efficient techniques for the execution of the model training during meta learning are known [15], the efficient generation of samples from large datasets is problematic, especially in cases where those datasets are stored and processed as block-partitioned matrices [2, 16]. Due to the size of the data, it is crucial to avoid redundant copying when generating training data. Furthermore, it is necessary to efficiently support sampling techniques such as sampling with replacement from block-partitioned matrices.

The three listed pitfalls *algorithm scalability*, *efficient execution of iterative algorithms*, and *scalable meta learning* illustrate why scalable data mining is such a hard problem: There is no 'silver bullet' solution to the problem. Scalability has to be achieved in many different aspects at once that all require skills and knowledge from separate fields of computer science. For algorithm scalability, a deep understanding of mathematics and machine learning is required. In order for systems to efficiently execute iterative algorithms, a background in distributed systems, data management and system engineering is necessary. The aforementioned pitfalls categorize the research conducted thus far during the PhD as well as the plans for the remaining time. In the following, we will detail the results achieved to date and lay out directions for the remaining time.

2. ACHIEVED RESULTS

With regard to *algorithm scalability*, we developed a parallel, functional formulation of two popular algorithms from the field of recommendation mining [17, 18].

2.1 Scaling Collaborative Filtering Algorithms

Collaborative filtering (CF) is a popular approach in recommender systems which analyzes the historical interactions between a user and an arbitrary kind of item. Based on patterns found in the historical data, a CF algorithm recommends new, potentially highly preferred items to the users. Let A be a $|C| \times |P|$ matrix holding all known interactions between a set of users C and a set of items P. If a user i interacted with an item j, then a_{ij} holds a numeric value representing the strength of this interaction.

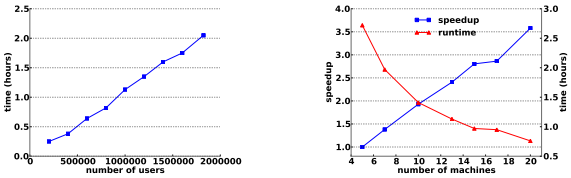
Scalable Item-Based Collaborative Filtering: So-called 'Item-Based Collaborative Filtering' [19] is among the most popular approaches to CF. The idea is to find items with similar interaction patterns. Mathematically, this means computing a $|P| \times |P|$ similarity matrix S from pairwise comparisons of the columns of A with a similarity measure. We started with a simplified view of the problem by imagining that A is binary (interactions either happened or they did not happen) and used the number of co-interacting users as our similarity measure. Then the computation of S becomes the simple matrix multiplication $S = A^\top A$. This view is very useful as it catches the computational shape of the problem. Analogous to the computation of the covariance matrix in our example from Section 1.1, a naïve execution of this matrix multiplication does not scale as the runtime increases quadratically with the number of columns (the number of items $|P|$). The key to scaling out item-based CF is finding an efficient and parallelizable way to compute $A^\top A$ on partitioned data.

[1] randomized algorithms provide scalable PCA [8]

The *row-outer-product* formulation [20] for computing S is efficiently parallelizable in a MapReduce environment, as it models S as the sum of outer products of the rows of A. We proposed to store A in a row-partitioned manner in the distributed filesystem and to compute S in a single pass over the data as follows: mappers compute outer products of rows and reducers sum those up to form S. Our paper holds details on how to generalize this computation to a wide variety of similarity measures by embedding canonical functions [17]. This mode of computation already fulfills one of the three conditions from Section 1.1 for scalable algorithms, as it conducts the matrix multiplication in a way that is parallelizable on partitioned input. A crucial question however is the handling of sparsity and the runtime of the operation. The algorithm described in our paper exploits sparsity by only emitting the non-zero rows of the rank one matrices built from the rows of A in the mapper. The effort for computing the individual rank one outer-product matrices is quadratic in the number of non-zeros of the row used as input. That means that the computation is dominated by the densest rows of A ('power users' with an unproportionally high amount of interactions, which we found in all CF datasets we looked at). We mitigated this effect with negligible impact on recommendation quality, by selectively down-sampling the interaction histories of the 'power users'. We experimentally evaluated our approach on a large dataset consisting of 700 million interactions [7]. The results showed that our algorithm scales linearly with the number of users in the dataset as well as with the number of machines in the cluster (c.f. Figure 2). For further details about the evaluation, please refer to [17].

An implementation of our approach has been contributed to the popular open-source machine learning library *Apache Mahout* [21]. It serves as the basis for many large-scale, real-world recommender systems (e.g. place recommendation at *Foursquare* [22], scientific article recommendation in the academic network *Mendeley* [23] and diverse recommendations in *ResearchGate* [24], the world's largest social network for researchers).

Scalable Latent Factor Models: Next, we worked on parallelizable formulations for so-called 'latent factor models', approaches to CF that leverage a low-rank matrix factorization of the interaction data [25]. The idea is to approximately factor the sparse $|C| \times |P|$ matrix A into the product of two rank r feature matrices U and M, such that $A \approx UM$. The $|C| \times r$ matrix U models the latent features of the users (the rows of A), while the $r \times |P|$ matrix M models the latent features of the items (the columns of A). A prediction for the strength of the relation between a user and an item is given by the dot product $u_i^\top m_j$ of the vectors for user i and item j in the low-dimensional feature space. Popular techniques to compute such a factorization are stochastic gradient descent (SGD) and alternating least squares (ALS) [25, 26]. ALS rotates between re-computing the rows of U in one step and the columns of M in the subsequent step. Analogously to other distributed implementations [10], we chose ALS as our optimization technique for our parallel implementation. Although it is computationally more expensive than SGD, it naturally lends itself to parallelization, because during the re-computation of the user feature matrix U for example, u_i, the i-th row of U, can be re-computed by solving a least squares problem only including a_i, the i-th row of A, which holds user i's interactions, and all the columns m_j of M that correspond to non-zero entries in a_i. This re-computation of u_i is independent from the re-computation of all other rows of U and therefore, the re-computation of U is easy to parallelize if we manage to guarantee efficient data access to the rows of A and the corresponding columns from M. We demonstrated how to execute ALS on MapReduce through a series of broadcast-joins and evaluated our implementation on a large dataset consisting of more than 5 billion interactions [18]. Our implementation is part of the popular open-source machine learning library *Apache Mahout* [21] and used by the machine learning server *PredictionIO* [27].



(a) increase of user base (b) increase of cluster size

Figure 2: Scalability of our item-based CF algorithm.

2.2 Optimistic Recovery for Distributed Iterative Data Processing

Negative experiences with the deficiencies of the MapReduce programming model (and Hadoop's execution model) for iterative computations led to work on the *efficient execution of iterative algorithms*, using the general dataflow system Stratosphere [28], a massively parallel data analysis system. Stratosphere provides a runtime with dedicated support of iterative data flows [29]. We demonstrated the applicability of this runtime implementation for data mining and network analysis [30].

As outlined in Section 1.1, traditional pessimistic approaches to fault tolerance in iterative computations incur a high overhead. Even worse, they always incur this overhead, even if no failures occur. Lineage-based recovery has been proposed to avoid checkpointing [12], but due to the complex data dependencies of many iterative algorithms, this approach usually results in a full recomputation of the algorithm state. Therefore, we proposed an optimistic recovery mechanism that does not cause any overhead in failure-free cases [13]. It is applicable to a wide range of fixpoint algorithms. Typically such algorithms refine an initial state iteratively until a convergence criterion is met, as illustrated by the left graphic in Figure 3. The graphic in the center shows how a failure is handled by a traditional, pessimistic approach. By writing checkpoints, it records all steps that are taken, and exactly repeats them in case of a failure.

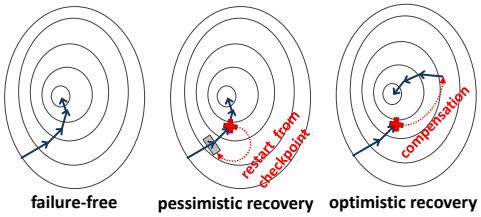


Figure 3: Convergence behavior.

The main idea of our proposed approach is to exploit the robust nature of many fixpoint algorithms used in data mining, which converge to the correct solution from many possible intermediate states. Therefore, the system actually does not need to checkpoint the exact steps taken. In case of a failure, it is sufficient to restore the intermediate state in such a way that the algorithm will still converge to the correct solution. We think of this as making the algorithm "jump" to a different (but valid) intermediate state in case of a failure, as depicted by the rightmost illustration in Figure 3. In

order to ensure the validity of the intermediate state, the user has to implement an algorithm-specific *compensation* function. In case of a failure, the system applies this function and continues the execution afterwards. The compensation function re-initializes the failed parts of the state, and potentially adjusts the non-failed parts. This re-initialization is the "jump" to a new state. The compensation function must ensure that the algorithm still converges to the optimal answer and its application may result in additional iterations. This proposed mechanism eliminates the need to checkpoint intermediate state. In our paper, we outlined two general logical plans for executing fixpoint algorithms with Stratosphere. Then, we described how to extend the programming model of Stratosphere to allow users to specify compensation functions. We made the compensation function part of the execution plan, and execute it only in case of failures. Interestingly, we found that a map operator is adequate to compensate a wide class of algorithms.

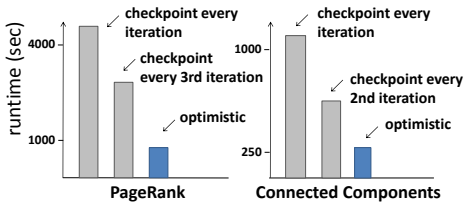


Figure 4: Failure-free performance.

In order to demonstrate the applicability of our approach, we explored three classes of problems that require distributed, iterative data processing. We started by looking at approaches to carry out link analysis and to compute centralities in large networks. Many of these algorithms operate by computing and interpreting the principal eigenvector of a large sparse matrix that represents the network. The computation involved in these algorithms reduces to a general technique, called the *Power Method* [20], for which we gave a compensation function that acts as a template for this class of problems. Next, we looked at path problems in graphs which include standard problems, such as reachability and shortest paths. We again discussed how to solve those with a general algorithm [31] and gave a template for a compensation function for that algorithm. Last, we described how to compensate the computation of latent factor models introduced in Section 2.1.

In an experimental evaluation on a cluster of 25 machines, we compared our proposed optimistic strategy against a pessimistic approach that writes checkpoints. We computed PageRank and the connected components of two large graphs with billions of edges. For details about the algorithms and the experimental setup, please refer to [13]. We first looked at the failure-free performance of optimistic and pessimistic recovery, in order to measure the overhead introduced by distributed checkpointing. Figure 4 illustrates the results. For the measured checkpoint intervals, the pessimistic approach increased the runtime by a factor of up to five. Note that picking an appropriate checkpoint interval for pessimistic approaches upfront is hard in general, as it depends on the load of the cluster and the number of iterations the algorithm needs to converge. In the failure-free case, the performance of our optimistic approach was equal to running without fault tolerance, as it only incurs overhead in case of a failure (in form of additional iterations). Next, we evaluated the recovery performance of our proposed recovery mechanism by simulating failures and making our prototype apply the compensation function and finish the execution. Again, we compared this to a pessimistic strategy that writes checkpoints. In the majority of cases, the optimistic mechanism provided faster recovery. However, we also noticed that the amount of additional iterations caused by our compensation functions strongly depends on the executed algorithm and the time of the failure. There were cases where later failures caused a lot of additional work as the algorithm was already close to the fixpoint. In these cases, the optimistic mechanism can take longer to recover than a pessimistic approach. In future work, we therefore plan to investigate how to mitigate this effect by automatically switching to a pessimistic strategy for later stages of the execution.

3. FUTURE DIRECTIONS

In the remainder of the PhD, we intend to work on the ability of a system to efficiently conduct *scalable meta learning*.

3.1 Efficient Sample Generation for Scalable Meta Learning

As introduced in Section 1.1, meta learning is necessary to apply machine learning models to real world use cases. Such techniques involve cross-validation, where we estimate the quality of a model, and ensemble learning, where we build a strong model from a set of weaker models. A common cross-validation technique is k-fold cross-validation [32]. Here, the data is randomly divided into k disjunct subsets and we execute k experiments. In every experiment, we train a model with a given set of hyperparameters on all but the k-th subset and test its prediction quality on the data from the k-th subset. In the end, the overall prediction quality for the chosen set of hyperparameters is computed from the outcomes of the individual experiments. A common technique for ensemble learning is bagging [33], where we train a set of models on bootstrap samples of the input data and combine their predictions afterwards.

On a single machine with random access to the data, these techniques are straightforward to implement. In a shared-nothing architecture, the efficient execution of meta learning becomes more difficult, especially when the data is represented as block-partitioned matrices stored on different machines [2, 16]. Once the samples for training and testing are generated from the input data, the model creation and evaluation is executed with common data- and task-parallel techniques [15]. Typically, the input data is a matrix whose rows correspond to observations. That means, for the generation of training and test data, we have to sample rows from this matrix. On block-partitioned matrices, the content of a row is spread over several blocks that are stored on different machines, which makes sampling more difficult. Furthermore, creating copies of the data becomes costly: A naïve implementation of k-fold cross validation would create k training samples consisting of $k-1$ subsets of the data, which would result in the materialization of $k-1$ copies of the whole dataset. For large inputs, this again poses a scalability bottleneck. Another aspect is that sampling becomes more difficult in a distributed environment [34]. Bagging, for example, employs sampling with replacement, which is hard to accomplish in parallel in a distributed environment, as determining the number of occurrences per row in a data sample requires the distributed generation of a partitioned multinomial random variate, whose components are correlated. One possible approach is to generate a table of assignments of observations to data samples upfront and join this table with the data [35]. However, this technique is difficult to adapt to block-partitioned data and furthermore requires join primitives, which are not available in many systems proposed for large scale data mining [2, 16]. We aim to develop an efficient and general framework for generating samples from large datasets stored as block-partitioned matrices in shared-nothing clusters. Ideally, the framework should be general enough to allow the implementation of a wide variety of sampling techniques through user-defined functions.

4. CONCLUSION

Mining large datasets on shared-nothing clusters will continue being an interesting and challenging field of research in the years to come. Scalability has to be achieved in many different aspects, ranging from adjusting the underlying mathematical operations of the algorithms to building systems that efficiently execute the prevalent modes of computation. In this paper, we presented our work on a selection of concrete problems connected to these aspects.

We started with work on algorithm scalability, where we developed parallel formulations of two recommendation mining algorithms, and described how to efficiently implement them in MapReduce. On a large dataset, we could demonstrate linear scalability of our item-based CF algorithm with the number of users in the dataset as well as the number of machines in the cluster. For latent factor models, we chose an optimization technique that lends itself to parallelization and showed scalability to datasets with billions of interactions. Next, we focused on a systems aspect, namely optimistic recovery for iterative data processing. In contrast to existing approaches, our proposed mechanism does not incur overhead during failure-free execution. It leverages the robust, self-correcting nature of a large class of fixpoint algorithms, which converge to the correct solution from various valid intermediate states. We illustrated the applicability of this approach for three classes of problems, described how to implement it in a dataflow system and conducted an experimental evaluation on large datasets. Finally, we motivated future work on efficient sample generation for meta learning in a shared nothing environment, when the data is represented as block-partitioned matrices.

As already mentioned, scalable data mining is a wide topic that touches a lot of scientific fields, therefore a PhD thesis can only cover a limited selection of its aspects. There are several other important aspects that we did not cover. A promising approach to scaling data mining is to execute the algorithms on graph-based systems [36], especially on those that can accelerate convergence through asynchronous execution [10]. Declarative languages that are automatically optimized and compiled to a parallel processing platform [2] will make scalable data mining available to users without a systems background. Another interesting direction is to process large datasets on a single machine (e.g., by using random projections to create sketches that can be efficiently handled in-memory [8]). Single machine, out-of-core algorithms have also been shown to scale to very large datasets in network analysis use-cases [37, 38].

5. REFERENCES

[1] J. Lin and A. Kolcz. Large-scale machine learning at Twitter. *SIGMOD'12*, pp. 793–804.

[2] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. *ICDE'11*, pp. 231–242.

[3] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. MLI: An API for distributed machine learning. *ICDM'13*, pp. 1187–1192.

[4] C. Bishop. Pattern Recognition & Machine Learning. *Springer 2006*.

[5] Apache Hadoop, http://hadoop.apache.org.

[6] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi. Measuring user influence in twitter: The million follower fallacy. *ICWSM'10*.

[7] R2 - Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0.

[8] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM'11*, 53(2):217–288.

[9] U. Kang, B. Meeder, and C. Faloutsos. Spectral analysis for billion-scale graphs: discoveries and implementation. *PAKDD'11*, pp. 13–25.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed Graphlab: A framework for machine learning and data mining in the cloud. *PVLDB'12*, pp. 716–727.

[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. *SIGMOD'10*, pp. 135–146.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI'12*, pp. 2–2.

[13] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All roads lead to rome: optimistic recovery for distributed iterative data processing. *CIKM'13*, pp. 1919–1928.

[14] W. Krzanowski. Cross-validation in principal component analysis. *Biometrics*, 1987.

[15] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB'14*.

[16] B. Huang, S. Babu, and J. Yang. Cumulon: optimizing statistical data analysis in the cloud. *SIGMOD'13*, pp. 1–12.

[17] S. Schelter, C. Boden, and V. Markl. Scalable similarity-based neighborhood methods with mapreduce. *RecSys'12*, pp. 163–170.

[18] S. Schelter, C. Boden, M. Schenck, A. Alexandrov, and V. Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. *RecSys'13*, pp. 281–284.

[19] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. *WWW '01*, pp. 285–295.

[20] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[21] Apache Mahout, http://mahout.apache.org.

[22] A recommendation engine, foursquare style, http://s.apache.org/ee.

[23] Scientific article recommendation & Mahout, http://goo.gl/e1kAMd.

[24] Researchgate uses Mahout, http://s.apache.org/tkz.

[25] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42:30–37, 2009.

[26] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. pp. 337–348.

[27] PredictionIO, http://prediction.io.

[28] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. *SoCC'10*, pp. 119–130.

[29] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB'12*, pp. 1268–1279.

[30] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl. Iterative parallel data processing with stratosphere: An inside look. *SIGMOD'13* (demo track).

[31] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings - New Models and Algorithms*. Springer, 2008.

[32] S. Geisser. The predictive sample reuse method with applications. *Journal of the American Statistical Association*, 70(350),1975.

[33] L. Breiman. Bagging predictors. *Machine learning*, 24(2), 1996.

[34] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. Planet: massively parallel learning of tree ensembles with mapreduce. *PVLDB'09*, pp. 1426–1437.

[35] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *PVLDB'09*, pp. 1481–1492.

[36] R. Xin, D. Crankshaw, A. Dave, J. Gonzalez, M. Franklin, and I. Stoica. GraphX: Unifying data-parallel and graph-parallel analytics. *arXiv:1402.2394*, 2014.

[37] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. *OSDI'12*, pp. 31–46.

[38] P. Boldi and S. Vigna. In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond. *arXiv:1308.2144*, 2013.