Scalable Analytics Model Calibration with Online Aggregation

Florin Rusu Chengjie Qin Martin Torres
University of California Merced
{frusu, cqin3, mtorres58}@ucmerced.edu

Abstract

Model calibration is a major challenge faced by the plethora of statistical analytics packages that are increasingly used in Big Data applications. Identifying the optimal model parameters is a time-consuming process that has to be executed from scratch for every dataset/model combination even by experienced data scientists. We argue that the lack of support to quickly identify sub-optimal configurations is the principal cause. In this paper, we apply parallel online aggregation to identify sub-optimal configurations early in the processing by incrementally sampling the training dataset and estimating the objective function corresponding to each configuration. We design concurrent online aggregation estimators and define halting conditions to accurately and timely stop the execution. The end-result is online approximate gradient descent—a novel optimization method for scalable model calibration. We show how online approximate gradient descent can be represented as generic database aggregation and implement the resulting solution in GLADE—a state-of-the-art Big Data analytics system.

1 Introduction

Big Data analytics is a major topic in contemporary data management and machine learning research and practice. Many platforms, e.g., OptiML [4], GraphLab [48, 49, 29], SystemML [3], SimSQL [50], Google Brain [25], GLADE [40, 12] and libraries, e.g., MADlib [26], Bismarck [17], MLlib [15], Vowpal Wabbit [1], have been proposed to provide support for distributed/parallel statistical analytics.

Model calibration is a fundamental problem that has to be handled by any Big Data analytics system. Identifying the optimal model parameters is an interactive, human-in-the-loop process that requires many hours – if not days and months – even for experienced data scientists. From discussions with skilled data scientists and our own experience, we identified several reasons that make model calibration a difficult problem. The first reason is that the entire process has to be executed from scratch for every dataset/model combination. There is little to nothing that can be reused from past experience when a new model has to be trained on an existing dataset or even when the same model is applied to a new dataset. The second reason is the massive size of the parameter space—both in terms of cardinality and dimensionality. Moreover, the optimal parameter configuration is dependent on the position in the model space. And third, parameter configurations are evaluated iteratively—one at a time. This is problematic because the complete evaluation of a single configuration – even sub-optimal ones – can take prohibitively long.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Gradient descent optimization [6] is a fundamental method for model calibration due to its generality and simplicity. It can be applied virtually to any analytics model [17] – including support vector machines (SVM), logistic regression, low-rank matrix factorization, conditional random fields, and deep neural networks – for which the gradient or sub-gradient can be computed or estimated. All the statistical analytics platforms mentioned previously implement one version or another of gradient descent optimization. Although there is essentially a single parameter, i.e., the step size, that has to be set in a gradient descent method, its impact on model calibration is tremendous. Finding a good-enough step size can be a time-consuming task. More so, in the context of the massive datasets and highly-dimensional models encountered in Big Data applications.

The standard practice of applying gradient descent to model calibration, e.g., MADlib, Vowpal Wabbit, MLlib, Bismarck, illustrates the identified problems perfectly. For a new dataset/model combination, an arbitrary step size is chosen. Model training is executed for a fixed number of iterations. Since the objective function is computed only for the result model – due to the additional pass over the data it incurs – it is impossible to identify bad step sizes in a smaller number of iterations. The process is repeated with different step values, chosen based on previous iterations, until a good-enough step size is found. Certain systems, e.g., Google Brain, support the evaluation of multiple step sizes concurrently. This is done by executing independent jobs on a massive cluster, without any sort of sharing.

We consider the abstract problem of *distributed model calibration with iterative optimization methods*, e.g., gradient descent. We argue that the incapacity to evaluate multiple parameter configurations simultaneously and the lack of support to quickly identify sub-optimal configurations are the principal causes that make model calibration difficult. It is important to emphasize that these problems are not specific to a particular model, but rather they are inherent to the optimization method used in training. The target of our methods is to find optimal configurations for the tunable hyper-parameters of the optimization method, e.g., step size, which, in turn, facilitate the discovery of optimal values for the model parameters.

In this paper, we apply parallel online aggregation to identify sub-optimal configurations early in the processing by incrementally sampling the training dataset and estimating the objective function corresponding to each configuration. We design concurrent online aggregation estimators and define halting conditions to accurately and timely stop the execution. We provide efficient parallel solutions for the evaluation of the concurrent estimators and of their confidence bounds that guarantee fast convergence. The end-result is online approximate gradient descent—a novel optimization method for scalable model calibration. We show how online approximate gradient descent can be represented as generic database aggregation and implement the resulting solution in GLADE—a state-of-the-art Big Data analytics system. Our major contribution is the conceptual integration of parallel multi-query processing and online aggregation for efficient and effective large-scale model calibration with gradient descent methods. This requires novel technical solutions as well as engineering artifacts in order to bring significant improvements to the state-of-the-art.

The article is organized as follows. We begin with a brief overview of GLADE—a parallel data processing system targeted at aggregate computation (Section 2). GLADE provides a series of features that are essential for designing and implementing online approximate gradient descent. Section 3 provides an overview of parallel online aggregation and how it is supported in GLADE. Section 4 presents online approximate gradient descent—the novel optimization method for analytics model calibration. A concrete application to SVM classification is given in Section 5. We conclude with future directions in Section 5.

2 A Brief Overview of GLADE

GLADE is a parallel data processing system executing any computation specified as a Generalized Linear Aggregate (GLA) [40, 12] using a merge-oriented strategy. It provides an infrastructure abstraction for parallel processing that decouples the algorithm from the runtime execution. The algorithm has to be specified in terms of the clean GLA interface, while the runtime takes care of all the execution details including data management,

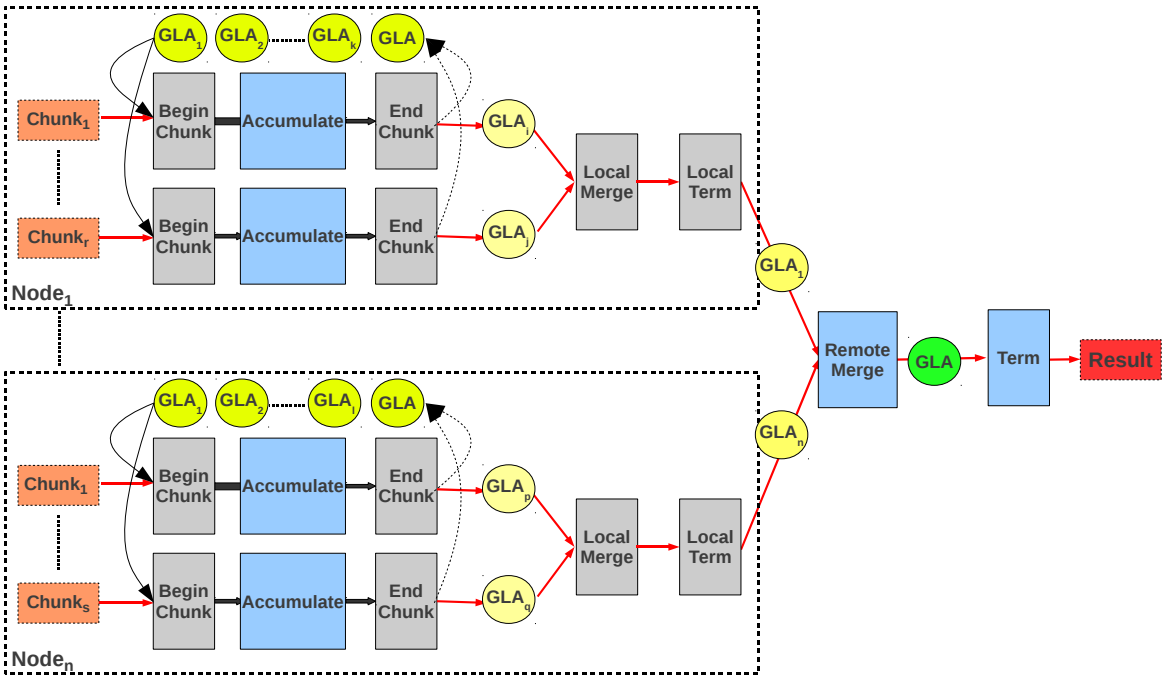


Figure 1: GLADE execution model implements the GLA abstraction.

memory management, and scheduling. GLADE is architecture-independent. It uses thread-level parallelism exclusively inside a processing node while process-level parallelism is used only across nodes. There is no difference between these two, though, in the GLADE infrastructure abstraction.

Execution Model Figure 1 depicts the GLADE execution model expressed in terms of the GLA interface abstraction [13]. GLA extend database User-Defined Aggregates (UDA) with methods for vectorized parallel processing. The standard UDA interface defines the following four methods: `Init`, `Accumulate`, `Merge`, and `Terminate`. These methods operate on the `state` of the object. While the interface is standard, the user has complete freedom when defining the `state` and implementing the methods.

GLADE evaluates GLAs by calling the interface methods as follows. `Init` – not displayed in Figure 1 – initializes the state of the object. `BeginChunk` is invoked before the data inside the chunk are processed—once for every chunk. `EndChunk` is similar to `BeginChunk`, invoked after processing the chunk instead. These two methods operate at chunk granularity. For example, data can be sorted for more efficient processing in `BeginChunk`, while temporary state can be discarded in `EndChunk`. `Accumulate` is invoked for every input tuple and updates the state according to the user-defined code. Merging is intended for use when the input is partitioned and multiple GLAs are computed—one for each partition. It takes as input two GLAs and merges their state into a combined GLA. Merging is invoked in two places. `LocalMerge` puts together local GLAs created on the same processing node, while `RemoteMerge` combines GLAs computed at different nodes. This distinction provides different optimization opportunities for shared-memory and shared-nothing parallel processing, respectively. `Terminate` is called after all the GLAs are merged together in order to finalize the computation, while `LocalTerminate` is invoked after the GLAs at a processing node are merged. `LocalTerminate` allows for optimizations when processing is confined to each node and no data transfer is required. It is important to notice that not all the interface methods have to be implemented for every GLA. We provide an illustrative GLA example in Section 5.

3 Parallel Online Aggregation

The idea in online aggregation is to compute only an estimate for an aggregate query based on a sample of the data [24]. In order to provide any useful information, though, the estimate is required to be accurate and statistically significant. Different from one-time estimation [18, 20] that may produce very inaccurate estimates for arbitrary queries, *online aggregation is an iterative process* in which a series of estimators with improving accuracy are generated. This is accomplished by including more data in estimation, i.e., increasing the sample size, from one iteration to another. The end-user can decide to run a subsequent iteration based on the accuracy of the estimator.

Method	**Usage**

`Init ()`
`Accumulate (Item` d`)`
`Merge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`
`Terminate ()`
	Basic UDA interface

`BeginChunk ()` `EndChunk ()`	Vectorized processing

`LocalMerge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`
`LocalTerminate ()`
`RemoteMerge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`
	Heterogeneous parallelism

`EstimatorTerminate ()` `EstimatorMerge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`	Partial aggregate computation
`Estimate (`$estimator$`,` $lower$`,` $upper$`,` $confidence$`)`	Online estimation

Table 1: Estimation-enhanced GLA interface.

Overlap Query Processing and Estimation A more efficient alternative that avoids the sequential nature of the iterative process is to *overlap query processing with estimation* [16, 2]. As more data are processed towards computing the query result, the accuracy of the estimator improves accordingly. For this to be true, though, data are required to be processed in a statistically meaningful order, i.e., random order, to allow for the definition and analysis of the estimator. This is typically realized by randomizing data during the loading process. The apparent drawback of the overlapped approach is that the same query is essentially executed twice—once towards the final aggregate and once for computing the estimator. However, with the ever increasing number of cores available on modern CPUs, the additional computation power is not utilized unless concurrent tasks are found and executed. Given that the estimation process requires access to the same data as normal processing, estimation is a natural candidate for overlapped execution.

Design Space An online aggregation system provides estimates and confidence bounds during the entire query execution process. As more data are processed, the accuracy of the estimator increases while the confidence bounds shrink progressively, converging to the actual query result when the entire data have been processed. There are multiple aspects that have to be considered in the design of a parallel online aggregation system. First, a mechanism that allows for the computation of partial aggregates has to be devised. Second, a parallel sampling strategy to extract samples from data over which partial aggregates are computed has to be designed. Each sampling strategy leads to the definition of an estimator for the query result—estimator that has to be analyzed in order to derive confidence bounds. Each of these aspects is discussed in detail elsewhere [38]. In the

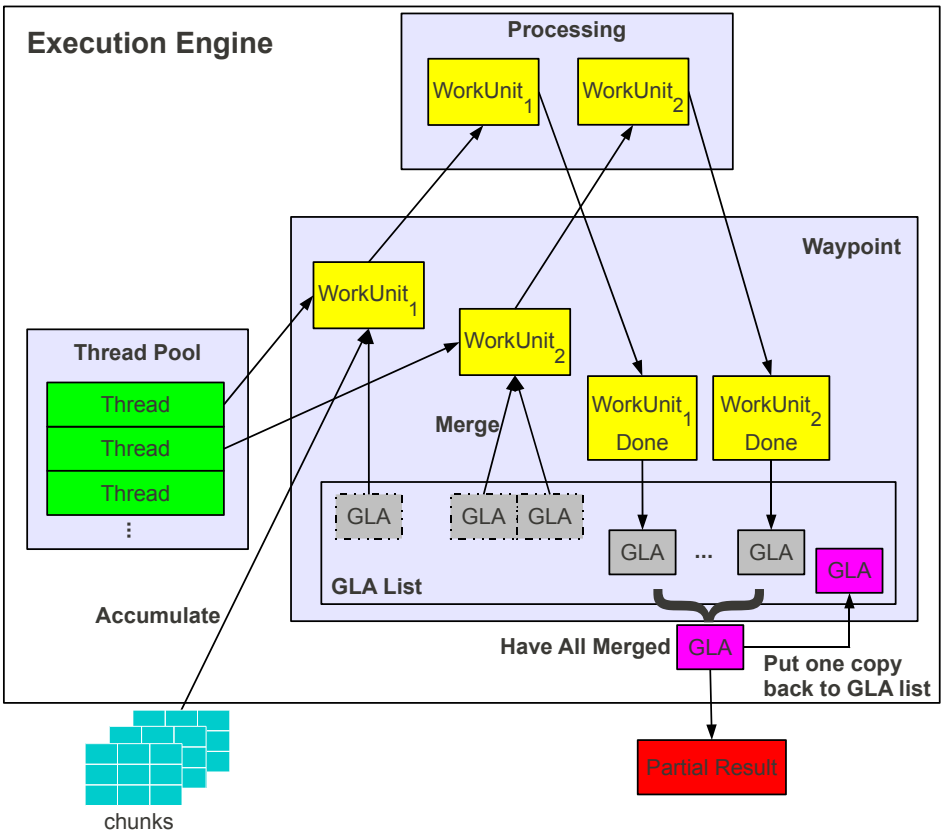


Figure 2: Execution strategy for parallel online aggregation in GLADE.

following, we present how online aggregation is supported in GLADE by enhancing the GLA abstraction with estimation capabilities.

3.1 Estimation-Enhanced GLA Abstraction

To support estimation, the GLA state has to be enriched with additional data on top of the original aggregate. Moreover, the GLA interface has to be also extended in order to distinguish between the final result and a partial result used for estimation. At least two methods have to be added—`EstimatorTerminate` and `EstimatorMerge`. `EstimatorTerminate` computes a local estimator at each node. It is invoked after merging the local GLAs during the estimation process. `EstimatorMerge` is called to put together in a single GLA the estimators computed at each node by `EstimatorTerminate`. It is invoked with GLAs originating at different nodes. Essentially, `EstimatorTerminate` and `EstimatorMerge` correspond to `LocalTerminate` and `RemoteMerge`, respectively. However, they contain exclusively estimation logic. The third method we add to the GLA interface is the `Estimate`. It is invoked by the user application on the GLA returned by the framework as a result of an estimation request. `Estimate` computes an estimator for the aggregate result and corresponding confidence bounds.

Table 1 summarizes the extended GLA interface we propose for parallel online aggregation. This interface abstracts both the aggregation and estimation processes in a reduced number of methods, releasing the user from the details of the actual execution in a parallel environment which are taken care of transparently by GLADE. Thus, the user can focus only on estimation modeling.

3.2 GLADE Implementation

Adding online aggregation to GLADE requires the extraction of a snapshot of the system state that can be used for estimation. Our solution overlaps the process of taking the snapshot with the actual GLA processing in order to have minimum impact on the overall execution time. This process consists of multiple stages. In the first stage, data are stored in random order on disk. This is an offline process executed at load time. Once data are available, we can start executing aggregate queries. The user application submits a query to the coordinator—a designated node managing the execution. It is the job of the coordinator to send the query further to the worker nodes, coordinate the execution, and return the result to the user application once the query is done. The result of a query is always a GLA containing the final state of the aggregate. When executing the query in non-interactive mode, the user application blocks until the final GLA arrives from the coordinator. When running in interactive mode with online aggregation enabled, the user application emits requests to the coordinator asking for the current state of the computation. A query always starts executing in non-interactive mode. Partial results are extracted asynchronously based exclusively on user application requests. The reason for this design choice is our goal to allow maximum asynchrony between the nodes in the system and to minimize the number of communication messages. It is clear that generating partial results interferes with the actual computation of the query result. However, in the case of aggregate computation, this is equivalent to early aggregation which is nonetheless executed as part of the final aggregation.

At a high level, enhancing GLADE with online aggregation is just a matter of providing support for the enhanced GLA interface in Table 1. The main challenge is how to overlap online estimation and actual query processing at all levels of the system. Abstractly, this corresponds to executing two simultaneous GLA computations. However, rather than treating actual computation and estimation as two separate GLAs, we group everything into a single enhanced GLA. This simplifies the control logic depicted in Figure 2. A partial result request triggers the merging of all existent GLAs. The resulting GLA is the partial result. Unlike the final result which is extracted from the waypoint and passed for further merging across the nodes, a copy of the partial result needs to be kept inside the waypoint and used for the final result computation. The newly arriving chunks are processed as before. The result is a completely new list of GLA states. The local GLA resulted through merging is added to this new list once the merging process ends.

4 Analytics Model Calibration

Consider the following model calibration problem with a linearly separable objective function:

$$\Lambda(\vec{w}) = min_{w \in \mathbb{R}^d} \sum_{i=1}^{N} f\left(\vec{w}, \vec{x_i}; y_i\right) + \mu R(\vec{w}) \tag{8}$$

in which a d-dimensional vector $\vec{w}$, $d \geq 1$, known as the *model*, has to be found such that the objective function is minimized. The training dataset consists of N (vector, label) pairs $\{(\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)\}$ partitioned into M subsets. Each subset is assigned to a different processing node for execution. We focus on the case when N is extremely large and each of the M subsets are disk-resident. The constants $\vec{x_i}$ and y_i, $1 \leq i \leq N$, correspond to the feature vector of the i$^{\text{th}}$ data example and its scalar label, respectively. Function f is known as the loss while R is a regularization term to prevent overfitting. μ is a constant.

4.1 Gradient Descent Optimization

Gradient descent represents, by far, the most popular method to solve the class of optimization problems given in Eq. (8). Gradient descent is an iterative optimization algorithm that starts from an arbitrary point $\vec{w}^{(0)}$ and computes new points $\vec{w}^{(k+1)}$ such that the loss decreases at every step, i.e., $f(w^{(k+1)}) < f(w^{(k)})$. The new

points $\vec{w}^{(k+1)}$ are determined by moving along the opposite Λ gradient direction. Formally, the Λ gradient is a vector consisting of entries given by the partial derivative with respect to each dimension, i.e., $\nabla\Lambda(\vec{w}) = \left[\frac{\partial\Lambda(\vec{w})}{\partial w_1}, \ldots, \frac{\partial\Lambda(\vec{w})}{\partial w_d}\right]$. Computing the gradient for the formulation in Eq. (8) reduces to the gradient computation for the loss f and the regularizer R, respectively. The length of the move at a given iteration is known as the step size, denoted by $\alpha^{(k)}$. Gradient descent is highly sensitive to the choice of the *hyper-parameter* $\alpha^{(k)}$ which requires careful tuning for every dataset. With these, we can write the recursive equation characterizing any gradient descent method:

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha^{(k)}\nabla\Lambda\left(\vec{w}^{(k)}\right) \tag{9}$$

In order to check for convergence, the objective function Λ has to be evaluated at $\vec{w}^{(k+1)}$ after each iteration. Convergence to the global minimum is guaranteed only when Λ is convex. The process is depicted in Figure 3(a).

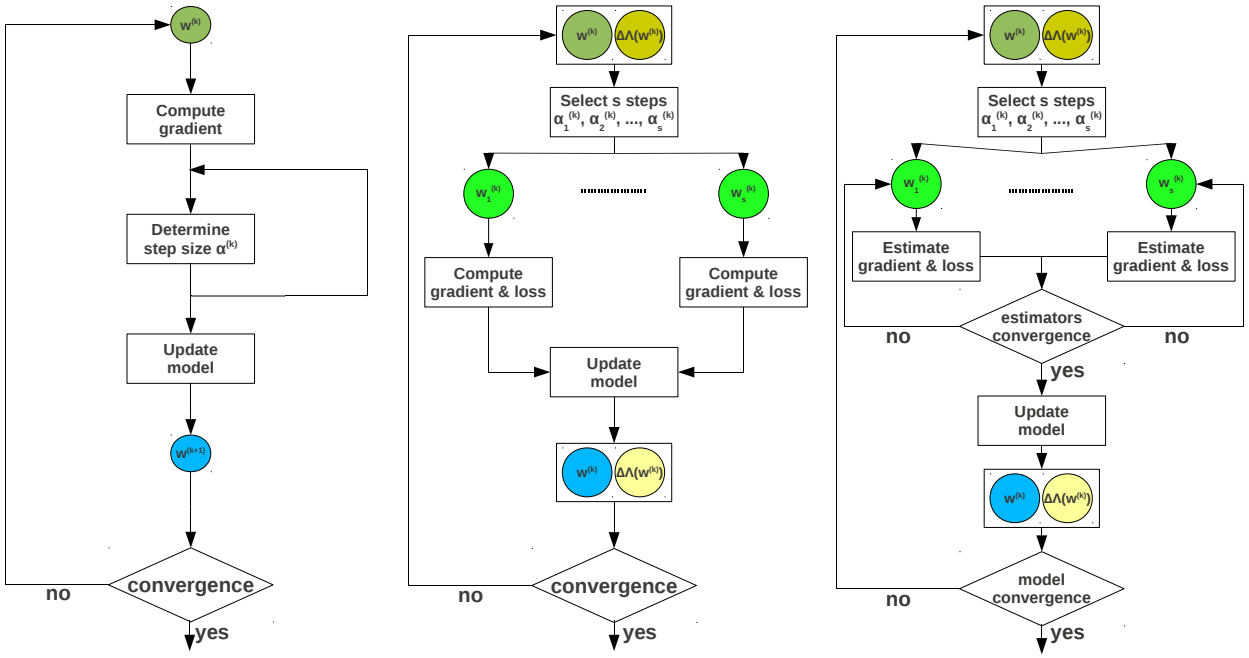
Batch Gradient Descent (BGD) The direct implementation of gradient descent is known as Batch Gradient Descent (BGD). We adopt it in this work due to its applicability to online aggregation. A detailed discussion of other alternatives is available in [11, 36]. The standard approach to compute the updated model $\vec{w}^{(k+1)}$ in BGD – once the direction of the gradient is determined – is to use line search methods which require objective and gradient loss evaluation. These involve multiple passes over the entire data. A widely used alternative is to fix the step size to some arbitrary value and then decrease it as more iterations are executed. The burden is essentially moved from runtime evaluation to offline tuning.

Parallel BGD The straightforward strategy to parallelize BGD is to overlap gradient computation across the processing nodes storing the M data subsets [7]. The partial gradients are subsequently aggregated at a coordinator node holding the current model $\vec{w}^{(k)}$, where a single global update step is performed in order to generate the new model $\vec{w}^{(k+1)}$ based on Eq. (9). The update step requires completion of partial gradient computation across all the nodes, i.e., update model is a synchronization barrier. Once the new model $\vec{w}^{(k+1)}$ is computed – using line search or a fixed step size – it is disseminated to the processing nodes for a new iteration over the data. Parallel BGD works because the objective function Λ is linearly separable, i.e., the gradient of a sum is the sum of the gradient applied to each term. It provides linear processing speedup and logarithmic communication in the number of nodes. In terms of convergence, though, there is no improvement with respect to the sequential algorithm—only faster iterations for gradient computation.

4.2 Speculative Gradient Descent

Speculative gradient descent addresses two fundamental problems—hyper-parameter tuning and convergence detection. Gradient descent depends on a series of hyper-parameters—the most important of which is the step size. Finding the optimal value for the hyper-parameters typically requires many trials. While convergence detection requires loss evaluation at every iteration, the standard practice, e.g., Vowpal Wabbit [1], MLLib [15], is to discard detection altogether and execute the algorithm for a fixed number of iterations. The reason is simple: loss computation requires a complete pass over the data. Discarding loss computation increases both the number of trials in hyper-parameter tuning as well as the duration of each trial.

The main idea in speculative gradient descent is to *overlap gradient and loss computation for multiple hyper-parameter configurations across every data traversal*. The number of hyper-parameter configurations used at each iteration is determined adaptively and dynamically at runtime. Their values are drawn from parametric distributions that are continuously updated using a Bayesian statistics approach. Only the model with the minimum loss survives each iteration, while the others are discarded. Speculative gradient descent allows for early bad configuration identification since many trials are executed simultaneously and timely convergence detection. Overall, faster model calibration.



(a) (b) (c)

Figure 3: Gradient descent optimization: (a) batch; (b) speculative; (c) online approximate.

Figure 3(b) depicts the speculative BGD algorithm. It starts with a set of possible step sizes $\{\alpha_1, \ldots, \alpha_s\}$, $s \geq 1$. An updated model $\vec{w}_i^{(k+1)}$, $1 \leq i \leq s$ is generated for each of these step sizes and the corresponding loss functions are computed concurrently. The model $\vec{w}_i^{(k+1)}$ with the minimum loss is chosen as the new model. The possible values of the step sizes are updated according to the computed loss function values. The procedure is repeated until convergence.

4.3 Online Approximate Gradient Descent

Speculative gradient descent allows for a more effective exploration of the parameter space. However, it still requires complete passes over the entire data at each iteration. *Online approximate gradient descent applies online aggregation sampling to avoid this whenever possible and identify the sub-optimal hyper-parameter configurations earlier.* This works because both the gradient and the loss can be expressed as aggregates in the case of linearly separable objective functions. The most important challenges we have to address are how to compute multiple concurrent sampling estimators corresponding to speculative gradient and loss computation (Figure 4) and how to design halting mechanisms that allow for the speculative query execution to be stopped as early as a user-defined accuracy threshold is achieved. A discussion on concurrent estimators for gradient and loss is available in [11]. In the following, we present the halting mechanisms.

When to stop gradient computation? In the case of gradient computation, there are d aggregates—one for every dimension in the feature vector. With online aggregation active, there is an estimator and corresponding confidence bounds for every aggregate. Given a desired level of accuracy, we have to determine when to stop the computation and move to a new iteration. We measure the accuracy of an estimator by the *relative error*, defined as the ratio between the confidence bounds width and the estimate, i.e., $\frac{high-low}{estimate}$. For example, if we aim for

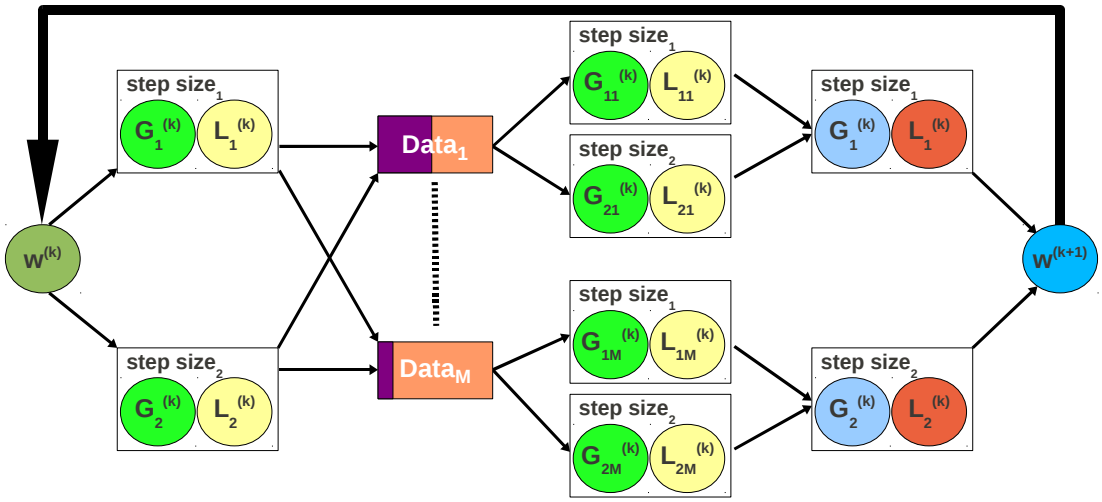


Figure 4: Parallel GLA evaluation for online approximate gradient descent.

95% accuracy, the relative error has to be below the user-defined threshold $\epsilon = 0.05$. What makes our problem difficult is that we have d independent relative errors, one for each estimator. The simple solution is to wait until all the errors drop below the threshold ϵ. This might require processing the entire dataset even when only a few estimators do not satisfy the accuracy requirement, thus defeating the purpose of online aggregation. An alternative that eliminates this problem is to require that only a percentage of the estimators, e.g., 90%, achieve the desired accuracy. Another alternative is to define a single convergence threshold across the d estimators. For example, we can define $\epsilon' = d \cdot \epsilon$ and require that the sum of the relative errors across estimators is below ϵ'.

When to stop loss computation? While the estimators in gradient computation are independent – in the sense that there is no interaction between their confidence bounds with respect to the stopping criterion – the loss estimators corresponding to different step sizes are dependent. Our goal is to choose only the estimator generating the minimum loss. Whenever we determine this estimator with high accuracy, we can stop the execution and start a new iteration. Notice that finding the actual loss – or an accurate approximation of it – is not required if gradient descent is executed for a fixed number of iterations. We design the following algorithm for stopping loss computation. The idea is to prune as many estimators as possible early in the execution. It is important to emphasize that pruning impacts only the convergence rate—not the correctness of the proposed method. Pruning conditions are exact and approximate. All the estimators for which there exists an estimator with confidence bounds completely below their confidence bounds, can be pruned. The remaining estimators overlap. In this situation, we resort to approximate pruning. We consider three cases. First, if the overlap between the upper bound of one estimator and the lower bound of another is below a user-specified threshold, the upper estimator can be discarded with high accuracy. Second, if an estimator is contained inside another at the upper-end, the contained estimator can be discarded. The third case is symmetric, with the inner estimator contained at the lower-end. The encompassing estimator can be discarded in this case. The algorithm is executed every time a new series of estimators are generated (Figure 3(c)). Execution can be stopped when a single estimator survives the pruning process. If the process is executed until the estimators achieve the desired accuracy, we choose the estimator with the lowest expected value.

When to stop gradient & loss computation? Remember that gradient and loss computation are overlapped in speculative gradient descent. When we move from a given point, we compute both the loss and the gradient at all the step sizes considered. In the online aggregation solution, we compute estimators and confidence bounds

for each of these quantities. The goal is to stop the overall computation as early as possible. How do we achieve this? We have to combine the stopping criteria for gradient and loss computation. The driving factor is loss computation. Whenever a step size can be discarded based on the exact pruning condition, the corresponding gradient estimation can be also discarded. Instead of applying the approximate pruning conditions directly, we have to consider the interaction with gradient estimation. While gradient estimation for the minimum loss does not converge, we continue the estimation for all the step sizes that cannot be discarded. This allows us to identify the minimum loss and its corresponding gradient with higher accuracy.

Algorithm 1 *GLA Online Approximate Gradient Descent*

State:

starting model: $\vec{w}$
s candidate models: $\vec{w_1}, \vec{w_2}, \ldots, \vec{w_s}$
s step sizes: $\alpha_1, \alpha_2, \ldots, \alpha_s$
s gradients: $\nabla\Lambda_1, \nabla\Lambda_2, \ldots, \nabla\Lambda_s$
s loss values: $\Lambda_1, \Lambda_2, \ldots, \Lambda_s$
estimation statistics: *gradient_stats*, *loss_stats*

Init ()

1. Initialize *gradient_stats*, *loss_stats*
2. Reset loss values $\Lambda_1, \Lambda_2, \ldots, \Lambda_s$
3. Calculate candidate models $\vec{w_1}, \vec{w_2}, \ldots, \vec{w_s}$ from $\vec{w}$ based on step sizes $\alpha_1, \alpha_2, \ldots, \alpha_s$

Accumulate (Item d)

1. Aggregate d to $\nabla\Lambda_1, \nabla\Lambda_2, \ldots, \nabla\Lambda_s$
2. Aggregate d to $\Lambda_1, \Lambda_2, \ldots, \Lambda_s$
3. Update *gradient_stats*, *loss_stats* with d

Merge (GLA $input_1$, GLA $input_2$, GLA *output*)

1. Merge $input_1.\{\nabla\Lambda_1, \ldots, \nabla\Lambda_s\}$ and $input_2.\{\nabla\Lambda_1, \ldots, \nabla\Lambda_s\}$ into $output.\{\nabla\Lambda_1, \ldots, \nabla\Lambda_s\}$
2. Merge $input_1.\{\Lambda_1, \ldots, \Lambda_s\}$ and $input_2.\{\Lambda_1, \ldots, \Lambda_s\}$ into $output.\{\Lambda_1, \ldots, \Lambda_s\}$

Terminate ()

1. Update $\vec{w}$ with the candidate model selected in *EstimatorTerminate*

EstimatorMerge (GLA $input_1$, GLA $input_2$, GLA *output*)

1. Merge $input_1.\{gradient\_stats\}$ and $input_2.\{gradient\_stats\}$ into $output.\{gradient\_stats\}$
2. Merge $input_1.\{loss\_stats\}$ and $input_2.\{loss\_stats\}$ into $output.\{loss\_stats\}$

EstimatorTerminate ()

1. Select the candidate model $\vec{w_i}$ having the minimum estimated loss according to *loss_stats*

Estimate ($estimator$, $lower$, $upper$, $confidence$)

1. Compute estimate and confidence bounds for gradients $\nabla\Lambda_1, \ldots, \nabla\Lambda_s$ from *gradient_stats*
2. Compute estimate and confidence bounds for loss values $\Lambda_1, \ldots, \Lambda_s$ from *loss_stats*
3. Prune out candidate models with sub-optimal loss estimates
4. Stop estimation for converged candidate models

5 An Example: SVM Classification GLA

We give an illustrative example that shows how SVM classification – a standard analytics model – can be expressed as `SQL` aggregates. The online approximate gradient descent optimization method is applied for

training the model. We provide the corresponding GLA implemented in GLADE and experimental results that confirm the benefits of online aggregation for model calibration.

The objective function in SVM classification with -1/+1 labels and L_1-norm regularization is given by: $\sum_i \left(1 - y_i \vec{w}^T \cdot \vec{x_i}\right)_+ + \mu||\vec{w}||_1$. This can be written as a standard `SQL` aggregate:

```
SELECT SUM(∑_{i=1}^{d} 1 − yx_i w_i) FROM T
WHERE (∑_{i=1}^{d} 1 − yx_i w_i) > 0
```

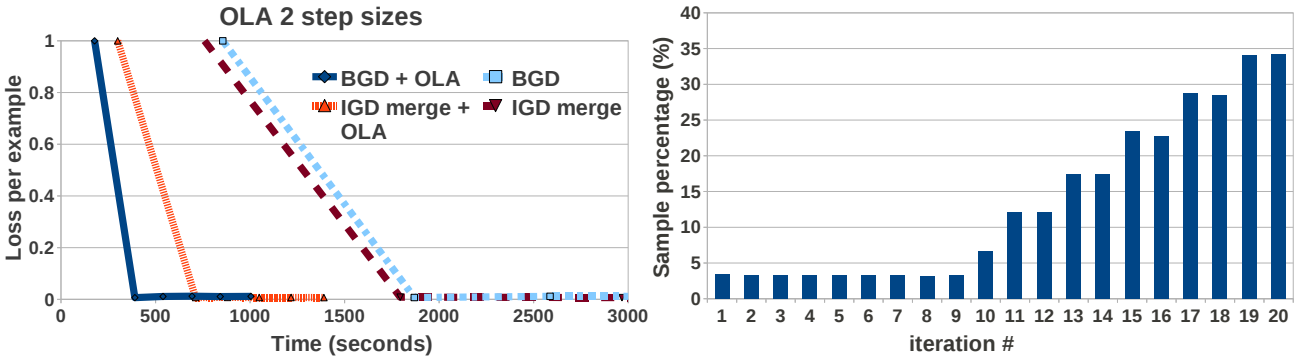
where `T` is a table that stores the training examples and their label. It is important to emphasize that model $\vec{w}$ is constant at any loss function evaluation. Gradient computation at point $\vec{w}$ requires one aggregate for every dimension, as shown in the following `SQL` query:

```
SELECT SUM(−yx_1), ..., SUM(−yx_d) FROM T
WHERE (∑_{i=1}^{d} 1 − yx_i w_i) > 0
```

In general, the BGD computations can be expressed as the following abstract `SQL` aggregate query:

```
SELECT SUM(f_1(t)), ..., SUM(f_p(t)) FROM T
```

in which p different aggregates are computed over the training tuples in relation T. This allows us to map BGD into a GLA and execute it in GLADE. The GLA corresponding to online approximate gradient descent is given in Algorithm 1.



(a) (b)

Figure 5: Experimental results for SVM classification: (a) convergence, (b) sampling ratio.

Figure 5 depicts the results of executing the online approximate gradient descent GLA for SVM classification over a 9-node GLADE cluster. The training dataset in these experiments is `splice` [11]. It consists of 50 million examples over a 13 million feature space. In Figure 5(a) we can observe the improvements in convergence rate online approximate gradient descent provides over standard BGD. Similar benefits translate to stochastic or incremental gradient descent (IGD). Figure 5(b) depicts the sampling ratio used at the time a decision is made about which candidate model is optimal. Far from the optimum, less than 5% sample is enough to identify the optimal candidate model. The ratio increases gradually as we get closer to the optimum. These results prove the benefits online aggregation can provide to analytics model training.

6 Related Work

In this paper, we apply online aggregation estimators to complex analytics, rather than focusing on standard SQL aggregates—the case in our previous work [38, 34]. We are the first to model gradient descent optimization as an aggregation problem. This allows us to design multiple concurrent estimators and to define halting mechanisms that stop the execution when model update and loss computation are overlapped. Moreover, the integration of online aggregation with speculative step evaluation allows for early identification of sub-optimal step sizes and directs the system resources toward the promising configurations. None of the existing systems support concurrent hyper-parameter evaluation or concurrent estimators. Our previous work on gradient descent optimization in GLADE [35, 37] is limited to stochastic gradient descent.

Online Aggregation The database online aggregation literature has its origins in the seminal paper by Hellerstein et al. [24]. We can broadly categorize this body of work into system design [39, 9, 16, 2], online join algorithms [23, 10, 43], online algorithms for estimations other than join [8, 47], and methods to derive confidence bounds [22]. All of this work is targeted at single-node centralized environments. The parallel online aggregation literature is not as rich though. We identified only three lines of research that are closely related to this paper. Luo et al. [19] extend the centralized ripple join algorithm [23] to a parallel setting. A stratified sampling estimator [14] is defined to compute the result estimate while confidence bounds cannot always be derived. Wu et al. [45] extend online aggregation to distributed P2P networks. They introduce a synchronized sampling estimator over partitioned data that requires data movement from storage nodes to processing nodes. In subsequent work, Wu et al. [44] tackle online aggregation over multiple queries. The third piece of relevant work is online aggregation in MapReduce. In [46], standard Hadoop is extended with a mechanism to compute partial aggregates. In subsequent work [31], an estimation framework based on Bayesian statistics is proposed. BlinkDB [42, 41] implements a multi-stage approximation mechanism based on pre-computed sampling synopses of multiple sizes, while EARL [30] and ABS [27] use bootstrapping to produce multiple estimators from the same sample.

Gradient Descent Optimization There is a plethora of work on distributed gradient descent algorithms published in machine learning [33, 25, 52, 21]. All these algorithms are similar in that a certain amount of model updates are performed at each node, followed by the transfer of the partial models across nodes. The differences lie in how the model communication is done [33, 52] and how the model is partitioned for specific tasks, e.g., neural networks [25] and matrix factorization [21]. Many of the distributed solutions are immediately applicable to multi-core shared memory environments. The work of Ré et al. [32, 17, 51] is representative in this sense. Our work is different from all these approaches because we consider concurrent evaluation of multiple step sizes and we use adaptive intra-iteration approximation to detect convergence. Moreover, stochastic gradient descent is taken by default to be the optimal gradient descent method, while BGD is hardly ever considered. Kumar et al. [28] have recently observed the benefits of BGD in training linear models over normalized relations that require join.

7 Conclusions and Future Work

In this paper, we apply parallel online aggregation to identify sub-optimal configurations early in the processing by incrementally sampling the training dataset and estimating the objective function corresponding to each configuration. We design concurrent online aggregation estimators and define halting conditions to accurately and timely stop the execution. The end-result is online approximate gradient descent—a novel optimization method for scalable model calibration. In future work, we plan to extend the proposed techniques to other model calibration methods beyond gradient descent.

Many Big Data analytics systems and frameworks implement gradient descent optimization. Most of them target distributed applications on top of the Hadoop MapReduce framework, e.g., Mahout [5], MLlib [15], while others provide complete stacks, e.g., MADlib [26], Distributed GraphLab [49], and Vowpal Wabbit [1]. With no exception, stochastic gradient descent is the primary method implemented in all these systems. As a first step, the techniques we present in this paper can be incorporated into any of these systems, as long as multi-threading parallelism and partial aggregation are supported. We plan to explore how this can be done in future work. More important, we provide strong evidence that BGD deserves full consideration in any Big Data analytics system.

Acknowledgments The work in this paper was supported in part by a Hellman Faculty Fellowship, a gift from LogicBlox, and a US Department of Energy (DOE) Early Career Award.

References

[1] A. Agarwal et al. A Reliable Effective Terascale Linear Learning System. *JMLR*, 15(1), 2014.

[2] A. Dobra et al. Turbo-Charging Estimate Convergence in DBO. *PVLDB*, 2(1), 2009.

[3] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE 2011*.

[4] A. Sujeeth et al. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML 2011*.

[5] Apache Mahout. `https://mahout.apache.org`. [Online; accessed July 2014].

[6] D. P. Bertsekas. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. MIT 2010.

[7] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS 2007*.

[8] C. Jermaine et al. Online Estimation for Subset-Based SQL Queries. In *VLDB 2005*.

[9] C. Jermaine et al. Scalable Approximate Query Processing with the DBO Engine. In *SIGMOD 2007*.

[10] C. Jermaine et al. The Sort-Merge-Shrink Join. *TODS*, 31(4), 2006.

[11] C. Qin and F. Rusu. Speculative Approximations for Terascale Analytics. `http://arxiv.org/abs/1501.00255`, 2015.

[12] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.

[13] Y. Cheng and F. Rusu. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. *DAPD*, 2014.

[14] W. G. Cochran. *Sampling Techniques*. Wiley, 1977.

[15] E. Sparks et al. MLI: An API for Distributed Machine Learning. In *ICDM 2013*.

[16] F. Rusu et al. The DBO Database System. In *SIGMOD 2008*.

[17] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD 2012*.

[18] G. Cormode et al. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4, 2012.

[19] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A Scalable Hash Ripple Join Algorithm. In *SIGMOD 2002*.

[20] M. N. Garofalakis and P. B. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *VLDB 2001*.

[21] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *KDD 2011*.

[22] P. J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *SSDBM 1997*.

[23] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD 1999*.

[24] J. Hellerstein, P. Haas, and H. Wang. Online Aggregation. In *SIGMOD 1997*.

[25] J. Dean et al. Large Scale Distributed Deep Networks. In *NIPS 2012*.

[26] J. Hellerstein et al. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 2012.

[27] K. Zeng et al. The Analytical Bootstrap: A New Method for Fast Error Estimation in Approximate Query Processing. In *SIGMOD 2014*.

[28] A. Kumar, J. F. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD 2015*.

[29] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI 2012*.

[30] N. Laptev et al. Early Accurate Results for Advanced Analytics on MapReduce. *PVLDB*, 5(10), 2012.

[31] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11), 2011.

[32] F. Niu, B. Recht, C. Ré, and S. J. Wright. A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS 2011*.

[33] O. Dekel et al. Optimal Distributed Online Prediction Using Mini-Batches. *JMLR*, 13(1), 2012.

[34] C. Qin and F. Rusu. Parallel Online Aggregation in Action. In *SSDBM 2013*.

[35] C. Qin and F. Rusu. Scalable I/O-Bound Parallel Incremental Gradient Descent for Big Data Analytics in GLADE. In *DanaC 2013*.

[36] C. Qin and F. Rusu. Speculative Approximations for Terascale Distributed Gradient Descent Optimization. In *DanaC 2015*.

[37] C. Qin and F. Rusu. Speeding-Up Distributed Low-Rank Matrix Factorization. In *CloudCom-Asia 2013*.

[38] C. Qin and F. Rusu. PF-OLA: A High-Performance Framework for Parallel Online Aggregation. *DAPD*, 32(3), 2014.

[39] R. Avnur et al. CONTROL: Continuous Output and Navigation Technology with Refinement On-Line. In *SIGMOD 1998*.

[40] F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. *OS Review*, 46(1), 2012.

[41] S. Agarwal et al. Knowing When You're Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *SIGMOD 2014*.

[42] S. Agarwal et al. Blink and It's Done: Interactive Queries on Very Large Data. *PVLDB*, 5(12), 2012.

[43] S. Chen et al. PR-Join: A Non-Blocking Join Achieving Higher Early Result Rate with Statistical Guarantees. In *SIGMOD 2010*.

[44] S. Wu et al. Continuous Sampling for Online Aggregation over Multiple Queries. In *SIGMOD 2010*.

[45] S. Wu et al. Distributed Online Aggregation. *PVLDB*, 2(1), 2009.

[46] T. Condie et al. MapReduce Online. In *NSDI 2010*.

[47] M. Wu and C. Jermaine. A Bayesian Method for Guessing the Extreme Values in a Data Set. In *VLDB 2007*.

[48] Y. Low et al. GraphLab: A New Parallel Framework for Machine Learning. In *UAI 2010*.

[49] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8), 2012.

[50] Z. Cai et al. Simulation of Database-Valued Markov Chains using SimSQL. In *SIGMOD 2013*.

[51] C. Zhang and C. Ré. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB*, 7(12), 2014.

[52] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.