Speculative Approximations for Terascale Distributed Gradient Descent Optimization

Chengjie Qin
UC Merced
5200 N Lake Road
Merced, CA 95343
cqin3@ucmerced.edu

Florin Rusu
UC Merced
5200 N Lake Road
Merced, CA 95343
frusu@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 incapacity to evaluate multiple parameter configurations simultaneously and the lack of support to quickly identify sub-optimal configurations are the principal causes.

In this paper, we develop two database-inspired techniques for efficient model calibration. *Speculative parameter testing* applies advanced parallel multi-query processing methods to evaluate several configurations concurrently. *Online aggregation* is applied 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 apply the proposed techniques to *distributed gradient descent optimization* – batch and incremental – for support vector machines and logistic regression models. We implement the resulting solutions in GLADE PF-OLA – a state-of-the-art Big Data analytics system – and evaluate their performance over terascale-size synthetic and real datasets. The results confirm that as many as 32 configurations can be evaluated concurrently almost as fast as one, while sub-optimal configurations are detected accurately in as little as a $1/20^{\text{th}}$ fraction of the time.

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 [38, 39, 24], SystemML [3], SimSQL [41], Google Brain [22], GLADE [11] and libraries, e.g., MADlib [23], Bismarck [14], MLlib [12], 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.

Motivating example. *Gradient descent optimization* [5] is a fundamental method for model calibration due to its generality and simplicity. It can be applied virtually to any analytics model [14] – 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.

Problem statement. 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 train-

ing. 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. Therefore, we investigate *speculative processing* and *intra-iteration approximations* for the optimization of distributed model calibration. Speculative iteration processing allows concurrent evaluation of multiple parameter configurations in a single pass over the training data—without the proportional increase in running time. Intra-iteration approximation allows for faster convergence detection—and corresponding reduction in iteration time. When put together, these techniques have the potential to reduce model calibration time significantly.

Contributions. In this paper, we develop two database-inspired techniques for efficient model calibration. *Speculative parameter testing* applies advanced parallel multi-query processing methods to evaluate several configurations concurrently. *Online aggregation* is applied to identify sub-optimal configurations early in the processing by incrementally sampling the training dataset and estimating the objective function corresponding to each configuration.

Our major contribution is the conceptual integration of parallel multi-query processing and approximation for efficient and effective large-scale model calibration with gradient descent methods. Specific contributions include:

- We design speculative parameter testing algorithms that evaluate multiple configurations simultaneously. The number of configurations is determined adaptively and dynamically at runtime. The configurations are drawn from a parametric distribution that is continuously updated using a Bayesian statistics [17] approach.
- 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 speculative estimators and of their corresponding confidence bounds that guarantee fast convergence.
- We apply these techniques to distributed gradient descent optimization for SVM and logistic regression models. As a prerequisite, we formalize gradient descent optimization as a database aggregation problem.
- We provide an extensive comparison between batch and incremental gradient descent that reveals that – contrary to the generally accepted opinion – batch gradient descent methods are better suited for distributed processing over massive datasets due to their linearity properties. With the proposed techniques integrated, batch gradient descent outperforms the incremental method both in convergence speed and execution time in our experiments.

Outline. The model calibration problem is presented in Section 2. Gradient descent solutions are introduced in Section 3. Speculative parameter testing is presented in Section 4, while intra-iteration approximation with online aggregation is detailed in Section 5. Experimental results that evaluate thoroughly the proposed methods are presented in Section 6. Section 7 discusses relevant related work, while Section 8 concludes the paper.

2. PROBLEM FORMULATION

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

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

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 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. For example, the objective function in SVM classification with -1/+1 labels and L_1-norm regularization is given by $\sum_i \left(1 - y_i \vec{w}^T \cdot \vec{x_i}\right)_+ + \mu ||\vec{w}||_1$.

Gradient descent represents, by far, the most popular method to solve the class of optimization problems given in Eq. (1). 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. (1) 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 $\alpha^{(k)}$. With these, we can write the recursive equation characterizing any gradient descent method:

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha^{(k)} \nabla\Lambda\left(\vec{w}^{(k)}\right) \qquad (2)$$

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. This implies that both the loss f and the regularizer R are convex.

The specific problem we consider in this paper is how to solve the optimization formulation given in Eq. (1) using generic gradient descent methods when the training dataset consisting of N (vector, label) pairs $\{(\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)\}$ is 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. Gradient and loss computation can take a prohibitive amount of time in this case. Moreover, gradient descent methods are highly sensitive to a series of parameters that require careful tuning, i.e., repeated execution with different configurations, for every dataset. Thus, novel techniques are required in order to scale gradient descent to the largest Big Data models.

3. GRADIENT DESCENT METHODS

In this section, we introduce the basic gradient descent optimization algorithms. We start with the standard batch gradient descent algorithm which is a direct implementation of the theory. Then, we present incremental or stochastic gradient descent, a popular alternative tailored for datasets containing a large number of examples. In addition to these primitive methods, we also discuss two derived algorithms—coordinate descent and L-BFGS. We conclude with a comparison of the two standard gradient descent approaches.

3.1 Batch Gradient Descent

The pseudo-code for gradient descent is given in Algorithm 1. This is also known as batch gradient descent (BGD). The algorithm takes as input the data examples and their labels, the loss function f and the gradient of the objective $\nabla\Lambda$, and initial values for the model and step size. The optimal model is returned. The main stages are gradient computation and model and step size update. They are executed until convergence is achieved. Convergence can be specified as a fixed number of iterations or based on the loss, e.g., the loss difference across consecutive iterations decreases below a given threshold. In the later case, the loss has to be computed after every iteration, which incurs an additional pass over the data.

Model and step size update. The standard approach to compute the updated model $\vec{w}^{(k+1)}$, once the direction of the gradient is determined, is to use line search methods [6]. $\vec{w}^{(k+1)}$ is found by iteratively trying different step sizes along the opposite gradient direction until the decrease in loss is above a user-defined threshold, i.e., the Wolfe conditions [6]. Line search achieves a tradeoff between optimality and runtime by choosing only an approximation to the optimal step size, but in shorter time. Nonetheless, line search is still iterative in nature and requires 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, i.e., $\alpha^{(k)} \to 0$ as $k \to \infty$. The initial step size $\alpha^{(0)}$ and the decay are highly sensitive parameters, specific to each dataset, that require intensive tuning. By fixing the step size, the burden is essentially moved from runtime evaluation to offline tuning.

Algorithm 1 Batch Gradient Descent (BGD)

Input: $\{(\vec{x}_j, y_j)\}_{1 \le j \le N}$, f, $\nabla\Lambda$, $\vec{w}^{(0)}$, $\alpha^{(0)}$
Output: $\vec{w}^{(k-1)}$

1. Initialize $\vec{w}^{(0)}$ and $\alpha^{(0)}$ with random values if not provided
2. Let $k = 1$
3. **while** (**true**) **do**
4. **if** model_convergence($\{f(\vec{w}^{(l)})\}_{0 \le l < k}$) **then break**
5. Compute gradient: $\nabla\Lambda(\vec{w}^{(k-1)})\{(\vec{x}_j, y_j)\}_{1 \le j \le N}$
6. Update model: $\vec{w}^{(k)} = \vec{w}^{(k-1)} - \alpha^{(k-1)}\nabla\Lambda(\vec{w}^{(k-1)})$
7. Update step size $\alpha^{(k)}$
8. Let $k = k + 1$
9. **end while**
10. **return** $\vec{w}^{(k-1)}$

SQL representation. We formulate the batch gradient solution to the objective function in Eq. (1) as SQL aggregate queries. This allows us to identify database-specific optimizations. There are two dataset-wide operations in Algorithm 1—loss evaluation and gradient computation. Both of them can be expressed as SQL queries over a relation $T(\vec{x}, y)$ in which each tuple represents a training example. The loss at a given point $\vec{w}$ is evaluated by the query:

```
SELECT SUM(f(w, x, y)) FROM T
```

in which the vector operations involving the example feature vector $\vec{x}$ and the multi-dimensional point $\vec{w}$ can either be expressed as array functions or be mapped explicitly into arithmetic operators. Gradient computation at a point $\vec{w}$ requires one aggregate for every dimension, as shown in the following SQL query:

```
SELECT SUM(∂f/∂w1 (w, x, y)), ..., SUM(∂f/∂wd (w, x, y))
FROM T
```

It is important to emphasize that both the point $\vec{w}$ and the function f and its gradient ∇f are constants at a given iteration in the two queries given above. To make this point clear, we show the actual queries corresponding to SVM classification with -1/+1 labels:

```
SELECT SUM(Σ_{i=1}^{d} 1 − y x_i w_i) FROM T
WHERE (Σ_{i=1}^{d} 1 − y x_i w_i) > 0
SELECT SUM(−y x_1), ..., SUM(−y x_d) FROM T
WHERE (Σ_{i=1}^{d} 1 − y x_i w_i) > 0
```

For other model types, only the formula of the loss function and its gradient change. The rest stays the same. Most importantly, the query shape is unchanged.

Based on these examples, we formalize the BGD computations with the following abstract SQL query:

```
SELECT SUM(f1(t)), ..., SUM(fp(t)) FROM T        (3)
```

in which p different aggregates are computed over the training tuples in relation T. At least two instances of this query have to be executed per iteration—one for the gradient and one for the loss. The exact number is typically much larger and is determined by the number of times the loss is computed in line search.

3.2 Incremental Gradient Descent

The problem with BGD is that one pass – or many more if line search is executed to find the optimal step size – over the entire data is required in order to move a single step. This is a big issue in our target scenario – massive number of examples N – since the time per iteration is linear in N and many iterations are typically required in order to achieve convergence. Incremental or stochastic gradient descent (IGD) [5] addresses this issue by taking N steps per iteration. The direction of each step is given by the gradient corresponding to a single data example $\vec{x}_i$. Essentially, the entire gradient is approximated with a single (or a few) term(s) in the summation, i.e., $\nabla\Lambda(\vec{w}) \approx \nabla f(\vec{w}, \vec{x}_i; y_i)$. If we ignore the regularizer R the step recurrence becomes:

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha^{(k)}\nabla f\left(\vec{w}^{(k)}, \vec{x}_{\eta^{(k)}}; y_{\eta^{(k)}}\right) \quad (4)$$

where $\eta^{(k)}$ returns the k$^{\text{th}}$ example in a random permutation of the data. The permutation is necessary to guarantee that progress towards convergence is made inside an iteration. Moreover, a different permutation should be used at each iteration to avoid stalling at a non-minimum point and increase the convergence rate. Given the large number of steps taken inside a single iteration, the step size $\alpha^{(k)}$ has to be carefully tuned to minimize extreme oscillations. Executing line search after every step is out of the question. Algorithm 2 summarizes the IGD differences compared to BGD. Lines 5 and 6 in Algorithm 1 are replaced with the `for` loop given in Algorithm 2, where $\vec{w}^{(k)}_{(0)} = \vec{w}^{(k-1)}$ and $\vec{w}^{(k)} = \vec{w}^{(k)}_{(N)}$, respectively.

Algorithm 2 Incremental Gradient Descent (IGD)

1. **for** $i = 1$ **to** N **do**
2. Approximate gradient: $\nabla f\left(\vec{w}^{(k)}_{(i-1)}, \vec{x}_{\eta^{(i)}}; y_{\eta^{(i)}}\right)$
3. $\vec{w}^{(k)}_{(i)} = \vec{w}^{(k)}_{(i-1)} - \alpha^{(k)}\nabla f\left(\vec{w}^{(k)}_{(i-1)}, \vec{x}_{\eta^{(i)}}; y_{\eta^{(i)}}\right)$
4. **end for**

3.3 Beyond First-Order Gradient Descent

Apart from the first-order gradient descent methods presented in this section, coordinate descent [38] and L-BFGS [1, 22] are two alternatives that build upon BGD and IGD. *Coordinate Descent (CD)* optimizes a multi-dimensional function by minimizing it along one dimension at a time. Instead of moving along the overall gradient direction, CD takes steps along each coordinate direction sequentially. The search along each coordinate is done by line search, which requires a step size. *L-BFGS* is a quasi-Newton method that searches the model space by approximating the inverse Hessian matrix of the objective function. Different from gradient descent, L-BFGS maintains a history of the last m updates to model w and gradient $\nabla\Lambda$. The direction $\vec{d}$ at iteration k is given by $\vec{d}^{(k)} = -H^{(k)}\nabla\Lambda^{(k)}$, where $H^{(k)}$ is the approximation to the Hessian. Without going into details on how $H^{(k)}$ is computed from historical models and gradients, we point out that, after computing the search direction $\vec{d}^{(k)}$, a line search is performed.

The techniques proposed in this paper can be applied to any of the gradient-based optimization methods – first- and second-order – used for large scale model calibration. We emphasize that the number of tunable hyper-parameters, i.e., parameters of the optimization method, is small in all these methods. In the case of coordinate descent, only the step size has to be tuned. In L-BFGS, only the size of the history and the step size used in line search have to be calibrated.

3.4 Discussion

Considering the alternative gradient descent methods discussed in this section, a comparison from the perspective of our specific problem, i.e., Eq. (1) with the assumption that N is extremely large, is required to clarify the merits of each approach. According to the thorough survey by Bertsekas [5], two cases have to be considered. Far from the minimum, IGD can have a convergence rate as much as N times faster for identical loss functions f and large N. Close to the minimum, IGD requires diminishing step sizes in order to converge. This results in sub-linear convergence rate—slower than the linear convergence of the batch method. Based on these observations, a hybrid approach [1], that first executes incremental gradient followed by batch gradient when no progress is made anymore, is likely to be the optimal solution in practice.

We do not include the extension of the gradient descent algorithms to a distributed setting in this version of the paper due to lack of space. For a complete treatment, we invite the reader to consult our extended report [9] available online.

4. SPECULATIVE ITERATIONS

In this section, we address two fundamental problems specific to model calibration—convergence detection and parameter tuning. As with any iterative method, gradient descent convergence is achieved when there is no more decrease in the objective function, i.e., the loss, across consecutive iterations. While it is obvious that convergence detection requires loss evaluation at every iteration, the standard practice, e.g., Vowpal Wabbit [1], MLLib [12], 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, which doubles the execution time. This approach suffers from at least two problems. First, it is impossible to detect convergence before the specified number of iterations finishes. And second, it is impossible to identify bad parameter configurations, i.e., configurations that do not lead to model convergence. Recall that both BGD and IGD depend on a series of parameters, the most important of which is the step size. Finding the optimal step size typically requires many trials. Discarding loss computation increases both the number of trials as well as the duration of each trial.

High-level approach. We propose a unified solution for convergence detection and parameter tuning based on speculative processing. The main idea is to *overlap gradient and loss computation for multiple parameter configurations across every data traversal*. This allows for timely convergence detection and early bad configuration identification since many trials are executed simultaneously. Overall, faster model training. With the right mix of tasks and judicious scheduling, the degree of parallelism supported in hardware can be fully utilized by executing multiple tasks concurrently. Moreover, the overall execution time is similar to the time it takes to execute each task separately.

Our contribution is to *design speculative gradient descent algorithms that test multiple step sizes simultaneously and overlap gradient and loss computation*. The number of step sizes used at each iteration is determined adaptively and dynamically at runtime. The step sizes are drawn from a parametric distribution that is continuously updated using a Bayesian statistics [17] approach. Only the model with the minimum loss survives each iteration, while the others are discarded (Figure 1). As long as the execution time does not dramatically increase when multiple step sizes are tested, speculative execution leads to faster model training. To the best of our knowledge, this is the first solution that uses speculative execution for gradient descent parameter tuning and loss computation. Speculative execution is applied in Vowpal Wabbit [1] and MLlib [12], but only to deal with the problem of slow nodes. Not to test multiple step sizes simultaneously. Vowpal Wabbit also supports progressive loss estimation which overlaps gradient and loss computation. Progressive loss provides only local estimates. It never computes the exact loss at the end of an iteration. This can be done only with an additional pass over the data. Our solution is exact – not approximate – and works in a distributed setting.

4.1 Speculative BGD

Speculative BGD works as follows. Define a set of possible step sizes $\{\alpha_1, \ldots, \alpha_s\}$, $s \geq 1$. Generate an updated model $\vec{w}_i^{(k+1)}$, $1 \leq i \leq s$, for each of these step sizes and compute the corresponding objective function value concurrently. Choose the model $\vec{w}_i^{(k+1)}$ with the minimum loss as the new model. Repeat the procedure for every iteration until convergence. The pseudo-code is given in Algorithm 3. The statements contained inside an *in parallel* loop are executed concurrently across iterations. Moreover, the two statements inside the loop (lines 8 and 9) are executed in parallel even for the same iteration. In the following, we present the main components of the algorithm. For a complete discussion, please consult [9].

Gradient and loss computation overlapping. The model that minimizes the loss across the s possible step sizes is taken as input by the subsequent iteration. The gradient has to be computed for this model. Remember though that we have already used this model to compute the loss in the previous iteration. Unfortunately, we did not know if this model is selected or not. Under the assumption that data access dominates the execution time, we argue for bundling loss and gradient computation together in order to save one pass over the data. Overall, the number of data traversals is determined entirely by the number of gradient computations while loss evaluation piggybacks on the data access. We have to apply the overlapping strategy for all the s models though, since we do not know the one minimizing the loss. Although we might expect this to increase the execution time of an iteration, the aggressive use of SIMD pipeline parallelism across the s step sizes minimizes the impact of additional computation and higher memory usage, as the experimental results in Section 6 show.

How to choose the step sizes? The simple solution to choosing the step sizes at a given iteration is to have s constants that cover a large range of values, some of which are very small. The constants can be kept the same across iterations or they can be decreased at some rate, i.e., the decay in IGD. The problem with this approach is that we discard the knowledge we gain from previous iterations. Ideally, we want to set the current s values based on the previous best-performing step sizes. Bayesian statistics [17] provides a principled framework to accomplish this. We start with a prior parametric distribution for the step size. This can be a single distribution or a mixture. The prior can be learned from the historical workload or it can be set to some default distribution, e.g., normal. The s step sizes used at each iteration are sampled randomly from the current distribution. The resulting losses are normalized and converted to probabilities. The s pairs (step size, loss) are combined together with the prior in order to compute the

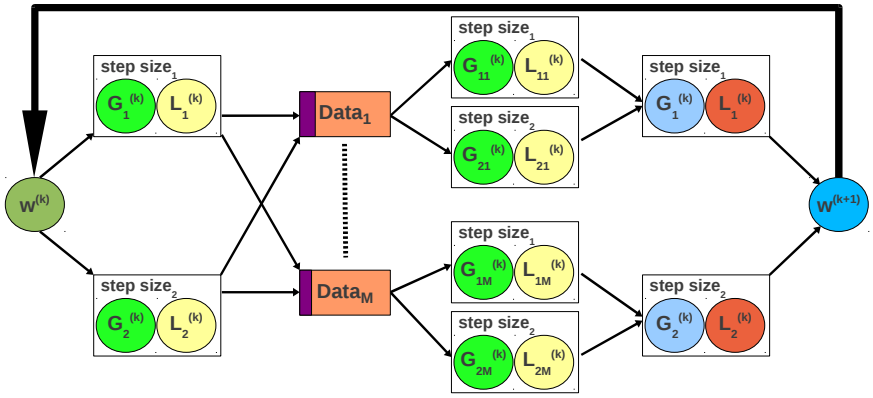


Figure 1: Speculative parameter testing with online aggregation.

updated step size distribution—known as the posterior. This consists in determining the posterior distribution parameters and it can be done with a straightforward Expectation-Maximization (EM) algorithm that identifies the Maximum Likelihood Estimator (MLE) based on the available samples. We do not provide the details of the EM algorithm since it is dependent on the shape of the prior. The posterior distribution becomes the prior in the subsequent iteration and the entire procedure is repeated.

Algorithm 3 Speculative BGD

Input: $\{(\vec{x}_j, y_j)\}_{1 \le j \le N}$, f, $\nabla\Lambda$, $\vec{w}^{(0)}$, $\nabla\Lambda(\vec{w}^{(0)})$, s, Φ
Output: $\vec{w}^{(k)}$

1. Initialize $\vec{w}^{(0)}$ and $\nabla\Lambda(\vec{w}^{(0)})$ randomly if not provided
2. Let $k = 1$
3. **while** (**true**) **do**
4. Draw s step sizes $\{\alpha_1, \ldots, \alpha_s\}$ from distribution Φ
5. Let $\vec{w}_i = \vec{w}^{(k-1)} - \alpha_i \nabla\Lambda(\vec{w}^{(k-1)})$, $1 \le i \le s$
6. **for each example** $(\vec{x}_j, y_j)$ **do**
7. **for each model** $\vec{w}_i$ **do** ***in parallel***
8. Compute gradient: $\nabla\Lambda(\vec{w}_i)\{(\vec{x}_j, y_j)\}$
9. Compute loss: $f(\vec{w}_i, \vec{x}_j; y_j)$
10. **end for**
11. **end for**
12. Let $\vec{w}^{(k)} = min_{f(\vec{w}_i)}\{\vec{w}_i\}$, $1 \le i \le s$
13. **if** model_convergence($\{f(\vec{w}^{(l)})\}_{0 \le l < k}$) **then break**
14. Update step distrib: $\Phi = Bayes(\Phi, \{\alpha_i\}, \{f(\vec{w}_i)\})_{i \le s}$
15. Update number of steps s based on nested loops (lines 6-11) execution time
16. Let $k = k + 1$
17. **end while**
18. **return** $\vec{w}^{(k)}$

4.2 Speculative IGD

At high level, the speculative techniques proposed for BGD are directly applicable to IGD: compute s models instead of one and overlap model update with loss evaluation. At closer look though, there is a significant difference. While computing the loss for a given model, the model changes continuously since multiple steps are taken during the update phase. As a result, the loss obtained at the end of an iteration corresponds to the starting model, while the final model is the updated model generated from the same starting model. Notice that they are different models though. This creates problems for at least two reasons. First, it is not clear that the resulting model corresponding to the original model having the minimum loss is the optimal model to select. And second, s step sizes for s initial models generate s^2 resulting models after one iteration. Since the number grows exponentially with the number of iterations, a pruning mechanism that selects only s models at every iteration is required. We address both these issues with the following strategy: select the s resulting models corresponding to the initial model having minimum loss. This guarantees that the starting model is optimally chosen. Since we do not know which of the s step sizes is optimal, we keep the models corresponding to all of them. The models generated for the sub-optimal $(s-1)$ initial models are all discarded. This provides the necessary pruning mechanism to support efficient processing. The algorithm can be found in [9].

5. INTRA-ITERATION APPROXIMATION

The speculative processing methods proposed in Section 4 allow for a more effective exploration of the parameter space. However, they still require complete passes over the entire data at each iteration in order to detect the sub-optimal parameter configurations. A complete pass is often not required in the case of massive datasets due to redundancy. It is quite likely that a small random sample summarizes the most representative characteristics of the dataset and allows for the identification of the sub-optimal configurations much earlier. This results in tremendous resource savings, more focused exploration, and faster convergence.

High-level approach. We present a *novel solution for using online aggregation sampling in parallel gradient descent optimization to speed-up the execution of a speculative iteration*. We generate samples with progressively larger sizes dynamically at runtime and execute gradient descent optimization incrementally, until the approximation error drops below a user-defined threshold ϵ. Relative to Figure 1, the entire process is executed multiple times during an iteration, on samples with increasing size.

The most important challenge we have to address is how to *design and compute multiple concurrent sampling estimators* corresponding to speculative gradient and loss computation. These estimators arise in the components of multi-dimensional gradients and in the speculative objective function evaluation. Our main contributions can be summarized as follows. We formalize gradient descent optimization as aggregate estimation. We provide efficient parallel solutions for the evaluation of the speculative estimators and of their corresponding confidence bounds that guarantee fast conver-

gence. We design halting mechanisms that allow for the speculative query execution to be stopped as early as possible.

5.1 Approximate BGD

Exact evaluation of query (3) can be a lengthy process when the size of data $|T|$ is large or when the computation of a particular aggregate f_i is complicated even when parallel solutions are used. The only alternative to speed-up computation further is to resort to approximation. Instead of computing the aggregates exactly, they are only estimated. The estimators have to come with sound accuracy guarantees though in order to guarantee verifiable results. Out of the many approximation techniques proposed in the literature, we argue that sampling is best suited for the aggregate estimation in query (3). This is because other methods, e.g., sketches [15], have to be constructed separately for every aggregate. This is infeasible in gradient descent optimization since a different set of aggregates have to be computed at each iteration. Only sampling maintains the identity of the data items and supports the computation of any aggregate as it would be computed over the entire data.

5.1.1 Sampling-Based Estimation

Sampling works as follows. A small dataset T' is randomly sampled from T. Query (3) is executed over the sample T' and the result is scaled-up to compensate for the difference in size between T and T'. It is straightforward to show that $\frac{|T|}{|T'|} \cdot Z_{f_i}$ is an unbiased estimator for the summation corresponding to function f_i, where Z_{f_i} is the result of query (3) executed over T'. Moreover, the accuracy, i.e., confidence bounds, of the estimator can be derived by estimating the variance over the same random sample T'. All this requires is the addition of another summation, i.e., `SUM(f_i^2(t))`, to query (3) for every aggregate function. While this is standard sampling-based single aggregate estimation treated extensively both in statistics as well as in databases [15], what is specific to our problem is the concurrent computation of multiple aggregates. As far as we know, this is a novel problem that has not been considered extensively in the literature.

Algorithm 4 Approximate BGD

1. **for each example** $(\vec{x}_{\eta(j)}, y_{\eta(j)})$ **do**
2. **for each *active* model** $\vec{w}_i$ **do *in parallel***
3. Estimate gradient $G_i[est, std]$: $\nabla\Lambda(\vec{w}_i)\{(\vec{x}_{\eta(j)}, y_{\eta(j)})\}$
4. Estimate loss $L_i[est, std]$: $f(\vec{w}_i, \vec{x}_{\eta(j)}; y_{\eta(j)})$
5. **end for**
6. **if** test_est_convergence(j) **then**
7. Prune out models: *Stop Loss*$(\{L_i[est, std]\}_{i \leq s}, .05)$
8. Let t be the number of remaining models, i.e., *active*
9. **if** $(t = 1)$ **and** *Stop Gradient*$(\{G_{tl}[est, std]\}_{l \leq d}, .05)$ **then break**
10. Let $s = t$
11. **end if**
12. **end for**

5.1.2 Online Aggregation

The main idea in online aggregation (OLA) [21] is to sample at runtime during normal query processing. Sampling and estimation are essentially overlapped with query execution. As more data are processed towards computing the final aggregates, the size of the sample increases and the accuracy of the estimators – as reflected by the width of the confidence bounds – improves accordingly. Whenever the targeted accuracy is achieved, the query, i.e., gradient iteration, can be stopped and a new iteration can be started. In the worst case, the aggregation is executed over the entire data and the exact results are obtained. Algorithm 4 depicts the pseudo-code for BGD with online aggregation. We present only the nested loops portion of the algorithm. They replace the corresponding nested loops in *Speculative BGD* (lines 6-11). `test_est_convergence` can be triggered either based on the number of examples processed, or externally. The pseudo-code for *Stop Gradient* and *Stop Loss* is available in [9].

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.

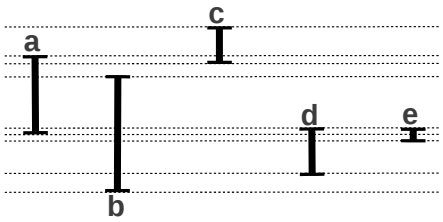


Figure 2: Confidence bounds relative relationships.

Figure 2 depicts a possible confidence bounds configuration for five loss estimators. Although spread horizontally, this configuration is generated at the same time instant during processing. Lower vertical points correspond to lower loss values. It is clear from the figure that estimator `c` has no chance to generate the minimum loss, thus it can be discarded. Estimator `a` overlaps with all of `b`, `d`, and `e`, it is not really sure which one will result in lower loss in the end.

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 [10]. 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 (`a` in the example). This is a straightforward extension of the exact pruning condition. Second, if an estimator is contained inside another at the upper-end, the contained estimator can be discarded (`e` in the example). 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. Execution can be stopped when a single estimator survives the pruning process.

When to stop gradient & loss computation? Remember that gradient and loss computation are overlapped in the speculative solution 4 we propose. When we move from a given point, we compute both the loss and the gradient at all the step sizes considered. Gradient computation is speculative since the only gradient we keep is the one corresponding to the minimum loss. The others are discarded. 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. Thus 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 can continue the estimation for all the step sizes that cannot be discarded based on the exact pruning condition.

5.2 Approximate IGD

Recall that in IGD the loss is computed only for the last $\vec{w}$ generated at the end of an iteration. This makes it impossible to detect if convergence is achieved earlier, when only a sample of the data have been used to update the model $\vec{w}$. Since faster convergence is expected in the case of massive datasets, the question we address is how to detect convergence as early as possible? This allows us to stop the current iteration immediately and start a new iteration from the latest updated model. The pseudo-code is available in [9]. Whenever estimator convergence is triggered, a new snapshot is taken for all the active models. Loss estimation is started for the new snapshot, executed over subsequent examples, and checked for convergence at later snapshots. A minimum number of converged loss estimators that exhibit reduced variance is required for the process to stop. Due to lack of space, we point the readers to [9] for details of how the convergence detection is approximated and overlapped with model updates for IGD.

6. EXPERIMENTAL EVALUATION

The objective of the experimental evaluation is to investigate the efficiency and effectiveness of the speculative parameter testing and intra-iteration approximation techniques across several synthetic and real datasets. We consider model calibration with gradient descent optimization for two standard analytics tasks—SVM and logistic regression (LR). Moreover, we compare the efficiency of our implementation against two other distributed Big Data analytics systems—MLlib [12] and Vowpal Wabbit [1]. The comparison is meant to identify the overhead introduced by the proposed techniques. Specifically, the experiments we design are targeted to answer the following questions:

- How does speculative parameter testing improve the convergence rate of model calibration and what overhead – if there is any – does it incur?
- What is the effect of intra-iteration approximation on convergence rate and execution time? How do speculative parameter testing and intra-iteration approximation interact?
- How do BGD and IGD compare when they integrate the proposed techniques?

6.1 Experimental Setup

Implementation. We implement the speculative and online aggregation versions of distributed gradient descent optimization as GLADE PF-OLA applications. GLADE [31, 11] is a state-of-the-art parallel data processing system that executes tasks specified using the abstract User-Defined Aggregate (UDA) interface. It was previously shown in [14] that UDA is the perfect database abstraction to represent complex analytics tasks—including gradient descent optimization. Our code is thus general enough to be executed by any database supporting UDAs. To be precise, we use the code from Bismarck [14] with slight modifications specific to GLADE. Speculative execution is supported in GLADE through multi-query processing. Multiple instances of the same UDA – with different parameters – are executed against the same example data. Without going into details, we mention only that data access is shared across UDAs over the entire memory hierarchy—from disk, to memory, cache, and CPU registers. Online aggregation requires an extension of the UDA interface with estimation functions and a pre-aggregation mechanism that allows for partial aggregate computation to be triggered during query processing. These are supported by the PF-OLA framework [29] for parallel online aggregation implemented on top of GLADE. Convergence and termination conditions are checked by the driver application. The code contains special function calls to harness detailed profiling data, used to generate the experimental results presented in the paper.

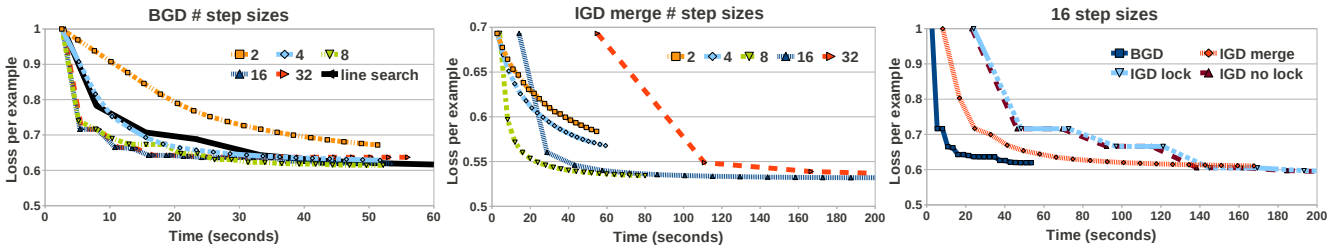
System. We execute the experiments on a 9-node cluster running Ubuntu 12.04.4 SMP 64-bit with Linux kernel 3.2.0-63. One node is configured as coordinator while the other eight are workers. Notice that only the workers are executing data processing tasks. Each worker has 2 AMD Opteron 6128 series 8-core processors – 16 cores – 28 GB of memory, and four 1 TB 7200 RPM SAS hard-drives configured RAID-0 in software. The storage system supports 240, 420 and 1600 MB/second minimum, average, and maximum read rates, respectively—based on the Ubuntu disk utility. There are two file systems running on each node. The storage system containing experimental data is mounted as a local file system, while the code and executables are shared across nodes through an NFS file system instance.

Methodology. We always enforce data to be read from disk in the first iteration by cleaning the file system buffers before execution. Subsequent iterations can access cached data. When the execution time per iteration is reported, the value corresponds to iterations two and above. We execute each algorithm for a fixed number of iterations—typically 20.

Dataset	Dimensions	Examples	Size
`forest` [14]	54	581K	485 MB
`classify50M` [14]	200	50M	136 GB
`splice` [1]	13M	50M	3.2 TB

Table 1: Datasets used in the experiments.

Tasks and datasets. While gradient descent is a general optimization method that can be applied for training a large variety of models, in this paper, we present experiments for convex LR and SVM. We use three datasets – two real and one synthetic, i.e., `classify50M` – with very different characteristics, as depicted in Table 1. It is important to emphasize that, as far as we know, `splice` is the largest dataset available on which gradient descent results have been published in the literature—in machine learning and databases. Since `forest` has small size, we use it to evaluate the performance of speculative processing on a single machine. Only multi-threading parallelism is used in this case. The other two datasets are evenly partitioned across the 8 worker nodes in the cluster using a random hash function. BGD and multiple versions of IGD are implemented for each (model type, dataset) combination. `IGD merge` corresponds to model averaging, in which a separate model is created for every thread in the system. `IGD lock` uses a single model per node and synchronizes access across threads with a light locking mechanism, e.g., `compare&swap`. `IGD no lock` implements the solution proposed in [26] that eliminates synchronization. While we run experiments for all the combinations, we include in the paper only the most representative results.



(a) (b) (c)

Figure 3: Convergence as a function of the number of step sizes on `forest`. (a) SVM with BGD. (b) LR with IGD. (c) SVM.

		forest						classify50M						splice	
		1	2	4	8	16	32	1	2	4	8	16	32	1	2
SVM	BGD	2.57	2.58	2.57	2.63	2.71	2.83	95	95	96	96	97	99	625	639
	IGD merge	2.58	2.58	2.74	3.22	8.17	33.2	96	96	99	114	363	1749	631	675
	IGD lock	2.8	2.91	3.12	6.53	24.04	92.78	95	96	101	246	880	3190	840	1064
	IGD no lock	2.5	2.52	3.06	6.39	23.12	86.73	95	96	101	193	726	2834	631	703
LR	BGD	2.65	2.65	2.66	2.74	2.92	3.09	96	96	96	96	97	99	618	640
	IGD merge	2.67	2.67	2.85	4	14.15	55.02	96	96	99	118	432	2213	634	1039
	IGD lock	2.8	2.8	3.1	7.21	27.22	104.24	97	97	102	254	898	3301	2103	2199
	IGD no lock	2.71	2.71	3.06	7.11	26.69	101.04	97	97	101	203	772	2975	636	1023

Table 2: Execution time per iteration for multiple step sizes across all the datasets and all the methods considered in the experiments.

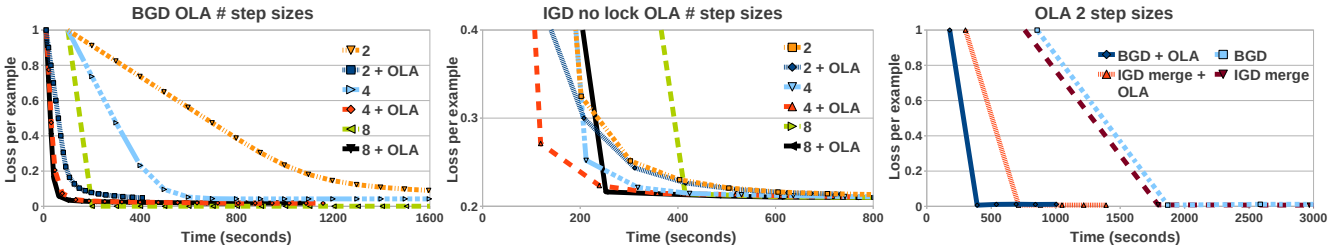
6.2 Speculative Parameter Testing

Convergence rate. To quantify the impact of speculative parameter testing on convergence rate, we execute BGD and IGD with an increasing number of concurrent step sizes. We pick the step size values by mimicking a real parameter tuning procedure. We start with an arbitrary value and then add smaller and larger values. The old values are maintained when increasing the number of steps. This guarantees continuous improvement as the number of steps increases—not always the case for Bayesian inference. Notice though that Bayesian inference can provide better values that result in faster convergence. Figure 3 depicts the effect of increasing the number of step sizes on convergence for the `forest` dataset. The general trend is to achieve faster convergence as the number of step sizes increases. While this is always true for BGD, the IGD behavior is more nuanced. The BGD results are depicted in Figure 3a. They include a comparison with `line search` [6], the standard backtracking step size search method that guarantees a certain loss decrease rate at every iteration. `line search` adjusts the step size at every iteration, thus providing automatic step size tuning. For more than 4 step sizes, speculative processing achieves faster convergence than `line search`. This is because `line search` limits itself to satisfying the imposed loss decrease rate. For a higher decrease rate, `line search` requires more passes over the example data, which results in higher time per iteration, thus slower convergence. Figure 3b depicts the convergence rate for `IGD merge`. While a higher number of step sizes still generates a higher loss decrease per iteration, the overhead incurred is significantly higher in this case. By the time one iteration with 32 steps finishes, 8 steps has already finished execution and achieved convergence. A direct comparison between BGD and the IGD versions for 16 step sizes is depicted in Figure 3c. It is clear that BGD can handle a large number of step sizes better than IGD since the number of speculative models is an order of complexity lower, i.e., linear vs. quadratic. Between the IGD solutions, `IGD merge` clearly outperforms the other two versions, even though the convergence rate per iteration is higher for `IGD lock` and `IGD no lock`, respectively. The hardware cache coherence mechanism coupled with the time to update a large number of models are the reason for the poor behavior of `IGD no lock`.

Overhead. Table 2 contains the execution time per iteration for all the experimental configurations. In the case of the `splice` dataset, for more than two step sizes, the memory required by the model is beyond the physical capacity of the testing machine. The time per iteration changes minimally for speculative BGD when we increase the number of step sizes from 1 to 32. The slightly higher execution time for LR is due to the more complicated gradient computation. IGD incurs a considerable overhead when the number of step sizes increases since the computation is quadratic in the number of step sizes. Between the IGD versions, `IGD merge` is the most efficient, especially for a large number of step sizes. This is due to complete model replication across threads which eliminates contention. Overall, speculative parameter testing is able to boost the convergence rate for BGD up to 32 step sizes, without significantly increasing the execution time per iteration. We are able to achieve this because our implementation takes full advantage of the parallelism available in modern multi-core CPUs, including deep pipelines and vectorized instructions. The main idea is to execute all the processing – across all the models – with minimal data movement, i.e., whenever a data example is brought in the CPU registers, it is used to update all the gradients/models.

6.3 Online Aggregation

The combined effect of online aggregation and speculative parameter testing on convergence rate is depicted in Figure 4. The execution of an iteration is halted as soon as the width of the confidence bounds corresponding to the estimator is below 5% of the estimate, i.e., we are 95% confident on the estimator value. In the



(a) (b) (c)

Figure 4: Convergence as a function of online aggregation on `classify50M`. (a) SVM with BGD. (b) LR with IGD. (c) SVM (`splice`).

case of BGD (Figure 4a), online aggregation provides a considerable boost in convergence rate across all the tested step sizes. The gradient can be estimated accurately from a small sample. The same is true for the loss. This allows for immediate detection of promising step sizes, while the sub-optimal ones can be discarded. In the best case, convergence to the same or a better loss is achieved as much as 20 times faster. The same trend can be observed for IGD (Figure 4b). The benefit of online aggregation is the most obvious for 8 step sizes, since convergence is achieved within one iteration. With standard IGD, convergence can be detected only at the end of a complete pass over the training data. Due to partial model merging, online aggregation detects convergence much earlier. Figure 4c depicts a direct comparison between BGD and IGD with online aggregation enabled for the `splice` dataset. While IGD achieves slightly faster convergence in the standard case, BGD outperforms IGD by more than 50% when online aggregation is enabled. This is because gradient estimation is a considerably easier task than detecting convergence for partial models.

6.4 Comparison with Existing Systems

We compare our implementation with two state-of-the-art large scale analytics systems – MLlib [12] and Vowpal Wabbit [1] – to validate the efficiency of our solutions. We choose these systems based on the following criteria. They are disk-based distributed systems with native support for gradient descent optimization. In fact, they support only IGD and not BGD. Based on the complete study by Cai et al. [40], no other open-source system satisfies these conditions. Our goal is to prove that the improvement we get from speculative processing and intra-iteration approximation does not come from an inefficient implementation. In fact, as shown in Table 3, our implementation is faster than these systems. The measure we use is the time per iteration for executing a complete gradient update pass and a complete loss computation.

		VW	MLlib	GLADE
`classify50M`	SVM	248	180	96
	LR	446	180	96
`splice`	SVM	1256	70560	631
	LR	3100	70560	634

Table 3: Execution time per iteration (seconds).

MLlib. We run MLlib over Spark on the same 9-node cluster. We set 4 workers per node and 4 cores per worker, which takes-up all the 16 cores in a node. We assign 6 GB of memory per worker, i.e., 24 GB out of the total 28 GB of memory per node. In all the MLlib results, we do not consider the data loading time for Spark, which is considerably larger than the time reported in Table 3. Nonetheless, we do include the data reading time from disk in our solution. For `classify50M`, MLlib is capable to cache the entire dataset in memory, thus it has a decent execution time. This is not the case for `splice` and the results show it.

Vowpal Wabbit (VW). VW is I/O-bound if the input data are pre-loaded in its native binary format. The results reported in Table 3 are for loaded data. VW requires an additional pass over the data to compute the loss exactly. The estimated loss it provides at each node during an iteration is local. The time for a single pass over the data is half of the results shown in Table 3. While closer to the results obtained by our implementation, it is important to remember that we overlap model update and loss computation.

GLADE PF-OLA (GLADE). Notice that we measure the execution time of the naive IGD implementation which does not include speculative processing and intra-iteration approximation. If we consider speculative processing, i.e., multiple concurrent step sizes, the execution time for GLADE PF-OLA stays almost the same (Table 1), while MLlib and VW require n times longer to complete, where n is the number of steps. Likewise, if we consider intra-iteration approximation, the execution time per iteration is even shorter, as shown in Figure 4.

6.5 Discussion

We validate the effectiveness and efficiency of the proposed techniques – speculative processing and intra-iteration approximation – for large scale gradient descent optimization. We show that speculative processing speeds-up the convergence rate of both BGD and IGD, and online aggregation reduces the iteration time further. We find that BGD is better suited to integrate the proposed techniques. In our experiments, BGD always outperforms IGD. By comparing with state-of-the-art systems, we confirm that our solution is able to significantly boost the model calibration time for terascale datasets. For additional experimental results, please consult [9].

7. RELATED WORK

We discuss related work in two main categories: distributed gradient descent optimization and parallel online aggregation. We emphasize the novelty brought by this work when compared to our previous papers on parallel online aggregation [29] and incremental gradient descent in GLADE [28].

Distributed gradient descent optimization. There is a plethora of work on distributed gradient descent algorithms published in machine learning [27, 22, 43, 18]. 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 [27, 43]

and how the model is partitioned for specific tasks, e.g., neural networks [22] and matrix factorization [18]. Many of the distributed solutions are immediately applicable to multi-core shared memory environments. The work of Ré et al. [26, 14, 42] 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, IGD is taken by default to be the optimal gradient descent method, while BGD is hardly ever considered. We provide a thorough comparison between BGD and IGD, and show that – with our optimizations – BGD always outperforms IGD.

Online aggregation. The database online aggregation literature has its origins in the seminal paper by Hellerstein et al. [21]. We can broadly categorize this body of work into system design [30, 7, 13, 2], online join algorithms [20, 8, 34], and methods to derive confidence bounds [19]. All of this work is targeted at single-node centralized environments. The parallel online aggregation literature is not as rich though. We identified only several lines of research that are closely related to this paper [16, 36, 35]. Online aggregation in MapReduce received significant attention recently. In [37], standard Hadoop is extended with a mechanism to compute partial aggregates. In subsequent work [25], an estimation framework based on Bayesian statistics is proposed. BlinkDB [33, 32] implements a multi-stage approximation mechanism based on precomputed sampling synopses of multiple sizes and bootstrapping.

GLADE PF-OLA. The novelty of our work compared to the PF-OLA framework [29] comes from applying online aggregation estimators to complex analytics, rather than focusing on standard SQL aggregates—the case in previous literature. 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, including GLADE PF-OLA, support concurrent hyper-parameter evaluation or concurrent estimators. Our previous work on gradient descent optimization in GLADE [28] is limited to IGD. In this paper, we also consider BGD and propose general methods applicable to distributed gradient descent optimization.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we propose two database techniques for efficient model calibration. Speculative parameter testing allows for several parameter configurations to be evaluated concurrently. Online aggregation identifies sub-optimal configurations early in the processing. We apply the proposed techniques to distributed gradient descent optimization – batch and incremental – and provide an extensive experimental comparison between these two methods. Contrary to the general belief, BGD always outperforms IGD both in convergence speed and execution time. In future work, we plan to extend the proposed techniques to other model calibration methods beyond gradient descent.

Acknowledgments. The work in this paper is supported by a Hellman Faculty Fellowship and a gift from LogicBlox.

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

[6] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

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

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

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

[10] C. Wang et al. On Pruning for Top-K Ranking in Uncertain Databases. *PVLDB*, 4(10), 2011.

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

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

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

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

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

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

[17] A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian Data Analysis*. Chapman & Hall/CRC, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[40] Z. Cai et al. A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms. In *SIGMOD 2014*.

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

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

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