# A Database-Hadoop Hybrid Approach to Scalable Machine Learning

Makoto YUI* and Isao KOJIMA*
*Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
Tsukuba, Ibaraki 305-8568, Japan
m.yui@aist.go.jp, isao.kojima@aist.go.jp

***Abstract*—There are two popular schools of thought for performing large-scale machine learning that does not fit into memory. One is to run machine learning within a relational database management system, and the other is to push analytical functions into MapReduce. As each approach has its own set of pros and cons, we propose a database-Hadoop hybrid approach to scalable machine learning where batch-learning is performed on the Hadoop platform, while incremental-learning is performed on PostgreSQL. We propose a purely relational approach that removes the scalability limitation of previous approaches based on user-defined aggregates and also discuss issues and resolutions in applying the proposed approach to Hadoop/Hive. Experimental evaluations of classification performance and training speed were conducted using a commercial advertisement dataset provided in the KDD Cup 2012, Track 2. The experimental results show that our scheme has competitive classification performance and superior training speed compared with state-of-the-art scalable machine learning frameworks; 5 and 7.65 times faster than Vowpal Wabbit and Bismarck, respectively, for a regression task.**

***Keywords*-stochastic gradient descent; logistic regression; iterative parameter mixture; in-database analytics; MapReduce; Hadoop**

## I. Introduction

Along with increasing demand for statistical data analysis in enterprise applications, several database vendors are trying to offer more sophisticated data analysis (i.e., machine learning) in relational databases, so that the costs of moving data can be eliminated. EMC Greenplum started the MADlib project [1], which provides machine learning and statistical functionality within a relational database management system (DBMS). Sybase integrated analytical software [2] into Sybase IQ, so that the analytical functions can be run within the DBMS. Teradata has partnered with SAS to implement in-database analytics, and consequently, a number of SAS functions are being implemented within Teradata and run in parallel on Teradata's massively parallel processing database architecture. The rationale behind in-database analytics is that carrying out the analysis in the database where the data resides, eliminates the cost of moving data. Keeping the data in database is especially critical when the size of the dataset shifts from terabytes to petabytes and beyond.

The other school of thought for large-scale data analysis suggests pushing analytical functions into Hadoop (a popular open-source implementation of MapReduce [3]) owing to its efficient parallel processing; it should be noted that fault-tolerant and straggler node handling properties are missing in relational databases. Because machine learning typically comes with time-consuming complex functions, such properties are indispensable in performing complex analytical functions on shared-nothing commodity servers.

Although Hadoop is good at batch-oriented processing where throughput is the primary factor, it is not suitable for online accesses where low latency is critical. Hadoop is known to have high latency bottlenecks in the job submission process (since it needs to distribute a job (i.e., jar) file to all workers) as well as its pull-based task scheduling process [4]. Online transaction processing databases are, on the other hand, specifically designed for low-latency requirements to host operational systems in which many concurrent users are retrieving, adding, updating, and deleting a small fraction of the data at a time on an unscheduled basis.

As each approach has its own set of pros and cons, we adopt a database-MapReduce hybrid approach to scalable machine learning. We perform batch-learning, where restarting the entire task from scratch is critical, on Hadoop, while incremental-learning is performed on PostgreSQL. The incremental-learning is implemented as a database stored procedure, which is periodically invoked upon changes in the dataset. The stored procedure incrementally updates the prediction model created by batch learning on Hadoop. To integrate the two different systems, we propose a purely relational approach to large scale machine learning that removes the interoperability barrier and scalability limitation in previous approaches. To evaluate the classification performance as well as the training speed of the proposed approach, we conducted experiments using a commercial advertisement dataset provided for the KDD Cup 2012, Track 2 [5]. The experimental results show that our scheme offers competitive classification performance and superior training speed and scalability compared with state-of-the-art scalable machine learning frameworks, Vowpal Wabbit [6] and Bismarck [7], for a regression task.

The rest of this paper is organized as follows. We begin with a brief description of the machine learning problem discussed in this paper, with related work discussed in Section II. In Section III, we describe the overall architecture of the proposed hybrid scheme as well as a detailed algorithm

for each incremental[1]/batch learning scheme. We present an evaluation of the performance of the proposed scheme based on experiments conducted in Section IV, and conclude the paper in Section V.

## II. Background and Related Work

### A. Gradient Methods

A machine learning task can be considered an optimization problem of feature weight vector $w$. Given a training set $D = \{x_i, y_i\}_i^n$, where $x_i$ is the feature vector and $y_i$ is the label of the $i$-th example, and letting $\ell$ represent the objective loss function, a supervised classifier is expected to induce $f : X \to Y$ such that the empirical loss is minimized. Here $f$ takes feature weight $w$ as the model parameter and the empirical loss $L$ takes the form of (II.1).

$$L = \frac{1}{n} \sum_{i=0}^{n} \ell(f(x_i; w), y_i) \qquad \text{(II.1)}$$

For each epoch $t$, the batch gradient descent (GD) computes gradients using all training instances and updates the weight vector as follows:

$$w^{(t+1)} = w^{(t)} - \gamma^{(t)} \frac{1}{n} \sum_{i=0}^{n} \nabla \ell(f(x_i; w^{(t)}), y_i) \qquad \text{(II.2)}$$

On the other hand, stochastic gradient descent (SGD) [8] computes gradients for each instance:

$$w^{(t+1)} = w^{(t)} - \gamma^{(t)} \nabla \ell \left( f(x; w^{(t)}), y \right) \qquad \text{(II.3)}$$

The major advantage of SGD is its scalability, which is basically linear to the number of training instances. SGD updates the feature weights for each training instance, whereas batch gradient descent requires all training instances to update the feature weights as shown in (II.2). Promising approaches for large scale learning therefore, use SGD, a representative of online learners that scales well with the number of training examples [9], [10]. Though SGD is an inherently sequential algorithm, several recent papers have proposed schemes to parallelize SGD on multi-core processors [11], [12].

SGD allows us to implement a number of convex optimization tasks (e.g., Logistic Regression, Support Vector Machine, Low-rank Matrix Factorization, and Conditional Random Fields) by changing the objective loss function [7].

### B. Machine Learning using MapReduce

Chu et al. [13] showed that a large class of machine learning algorithms is expressible in the statistical query model, and that these algorithms can be represented as MapReduce programs. For example, Apache Mahout [14] implements a number of machine learning algorithms including those given in [13]. As the statistical query model can be expressed in a specific summation form, it can easily be parallelized by performing calculations over subgroups of data on mappers, and then aggregating the mapper outputs on reducers. In other words, the approach [13] considers a machine learning algorithm to be an aggregate function, and parallelizes the aggregation with a partial aggregation technique [15] that has been widely adopted in relational databases.

While MapReduce lacks built-in support for iterative processes that are mandatory for machine learning tasks, there are a number of studies on improving MapReduce for iterative processing [16], [17], [18].

### C. Parameter Mixing

One option for distributed training is to partition the data into non-overlapping sets and to assign a trainer for each partition. The Distributed Gradient Descent algorithm [10], [13] simply parallelizes the gradient computation of the standard batch gradient descent. The key insight is that the gradient is the sum of the gradients and thus, it is easy to distribute the computation. While the distributed gradient approach can yield substantial improvements in wallclock speed over non-distributed approaches, it cannot take advantage of the fast convergence offered by online stochastic gradient updates.

Another group of studies [10], [19] perform stochastic parameter updates in each of the parallel computations. Each worker operates independently and shares parameters when its local model is fully optimized. This technique is called (iterative) parameter mixing in [10]. Algorithm 1 gives the pseudo-code for an iterative parameter mixing algorithm, where $\gamma$ is the learning-rate parameter decaying over time and $\nabla F$ is a function that computes the gradient for each training example as in (II.3).

As commented in Algorithm 1, parameter mixing is easy to parallelize using MapReduce. First, it distributes the training data S into K partitions. Note that S is already partitioned into blocks if the training data is stored on a Hadoop distributed file system (HDFS). Then, each worker in a map task independently applies stochastic gradient updates to the local model. Finally, to aggregate the K learned models at the end of an epoch, the reduce phase averages the updates from each of the map phase distributed optimizations. When distributing previously learned parameters to each mapper for each epoch, the algorithm is called iterative parameter mixing. This model averaging technique is well suited to the computational model for aggregates.

### D. Machine Learning on a Relational DBMS

Madlib [1] and Bismarck [7] provide in-database analytical functionality through user-defined functions (UDFs). They consider a machine learning algorithm as an aggregate function and implement machine learning algorithms as user-defined aggregate functions (UDAFs). Bismarck uses

[1]Because online machine learning represents a training method that learns one instance at a time in the literature, we call our online processing scheme incremental learning.

**Algorithm 1:** Iterative Parameter Mixing

$w = \emptyset$
$t = 0$
**repeat**
  $S = \{D^1, D^2, ..., D^K\}$ — *shuffle training data S into K partitions*
  **for** $j = 1 \ldots K$ **do** {run on mappers.}
    $w_0^j = w_{t-1}$ — *mix parameters for each epoch*
    **for** $i = 1 \ldots |D^j|$ **do** {for each instance, update weights based on gradient}
      $w_i^j = w_{i-1}^j - \gamma^t \nabla F_{D_i^j}\left(w_{i-1}^j\right)$
  **for** $f = 1 \ldots |w|$ **do** {for each feature in the model, aggregate on reducers}
    $w^{t+1}(f) = \frac{1}{K}\sum_{j=1}^{k} w_{|D^j|}^j(f)$ — *calculate average of feature weights*
  $t = t + 1$
**until** *converged*;

a model averaging technique like the one in Section II-C for its parallel gradient computation. However, these UDAF-based approaches are not suitable for parallel computation on MapReduce and have scalability limitations. This issue is discussed further in Section III.

## III. Hybrid Approach to Scalable Machine Learning

As mentioned in Section I, there are two popular schools of thought for performing large-scale machine learning, with each approach having its own set of pros and cons. Instead of choosing one of these, we propose a database-Hadoop hybrid approach to scalable machine learning. We use Hadoop for batch-learning and a relational database for incremental-learning. We assume that the batch-learning on the Hadoop platform is processed on a daily (or hourly) basis, while the incremental-learning on the relational database continuously updates the prediction model at very small intervals, possibly after changes have been made.

Figure 1 presents an overview architecture of our hybrid approach. As shown in this figure, we use a PostgreSQL open-source DBMS for incremental learning. The incremental learning is implemented as a database stored procedure, which is invoked upon changes in the training dataset through insert/update triggers or on a periodic basis. This means that the model being stored in the relational database is incrementally updated through transactional updates and the up-to-date model can be used for prediction in subsequent transactions. The incremental and low latency model updates are particularly important, for example, in adapting machine learning to online advertising since online advertising involves rapid changes in user interests over

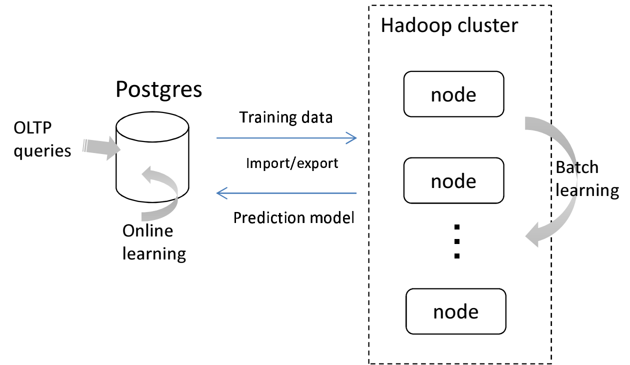


Figure 1. Architecture of the proposed system.

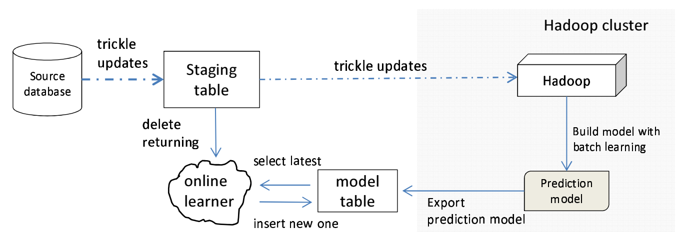


Figure 2. Data flow in the proposed architecture.

time, non-stationary bids from advertisers, and frequent updates to ad campaigns. We demonstrate in Section IV, that incremental updates to the prediction model can, in fact, improve classification performance.

Figure 2 shows the detailed data flow in the proposed architecture. As seen in this figure, changes in the training data are continuously fed into staging tables. The changes are optionally transferred to the HDFS using a data change capturing system (e.g., Databus [20]) or other specialized systems. In this study we used Apache Sqoop [21] to migrate the prediction model between two systems. Apache Sqoop is a tool designed to transfer bulk data efficiently between Apache Hadoop and structured data stores such as relational databases. When performing batch learning on Hadoop, we import the training data into the HDFS from PostgreSQL. When the batch learning process finishes, the new prediction model is returned to PostgreSQL, replacing the old prediction model. We assume here that the initial prediction model on PostgreSQL is created with a batch learning process on Hadoop.

*Technical Challenges in Database-Hadoop Hybrid Learning*

Integrating machine learning methods on the two platforms, namely, Hadoop and PostgreSQL, is not straightforward and has the following difficulties:

- First, previously proposed in-database analytical approaches that implement machine learning algorithms in user-defined aggregates [1], [7], are not suitable for

shared-nothing settings, and particularly for MapReduce execution.

- Second, the expected frequency for applying model updates is not obvious. It is possible to update the prediction model by an insert/update trigger for each update; however, issuing model updates (e.g., UPDATE model SET weight = ? WHERE feature = ?) for each update can cause performance degradation of the operational systems because it involves disk writes for ACID compliance.

In most relational databases, a user-defined aggregate consists of a well-known pattern of transition, merge, and final steps [1], [7]. Figure 3 shows a SGD classifier implemented in Bismarck[2] [7]. In the *Transition* step, each trainer takes part of the training examples as input (note that this step is thus a data-parallel operation) and produces a partial model. The subsequent *Merge* step takes the partial models as inputs and produces a pair comprising the sum of the weights and the count for each feature. Finally, the final merge operator takes each partial result and calculates the average weight for each feature.

A flaw in the UDAF-based approach is that scalability is often limited by the maximum fan-out of the final merge operator. Moreover, there are heavy memory requirements when there is a large number of inputs to an operator. Although each operator in Figure 3 only needs to hold the output and not all the inputs, the final merge operator tends to exhaust the allocatable memory if the number of trainers is large. We initially implemented the UDAF-based classifier in Hive/Hadoop, but found that the UDAF approach is not suited to the Hadoop/Hive environment because the allocatable memory in each map/reduce task is limited by the following formula: $\frac{Total\ memory\ reserved\ for\ mapreduce}{\#mapslots + C * \#reduceslots}$, where C is typically set between 1.0 and 1.3. In our Hadoop setup for a machine with two quad-core processors (hyper-threading enabled) with 24 GB memory, the allocated memory for each map/reduce task is set to just 2 GB with #mapslots set to 7 and #reduceslots to 3. There is also another issue with the current Hive/Hadoop implementation in that each tuple is stored as text in memory upon writing the job output to the HDFS, and multi-level aggregation is not supported in Hadoop/MapReduce, although a query with a group-by clause is optimized by map-side partial aggregations. Hence, scalar aggregates computing a large single result are not suitable for shared-nothing settings, particularly on Hive/Hadoop. We address this issue in Section III-A and propose a relational approach using user-defined table generating functions (UDTFs).

[2]Bismarck has an optimized execution mode for shared-memory (i.e., a single node and multiple CPU cores) settings in which each trainer shares a global prediction model allocated in shared memory. In the shared-memory mode, the execution is the same as that in [12] and the merge and final steps merely return the final result in memory.

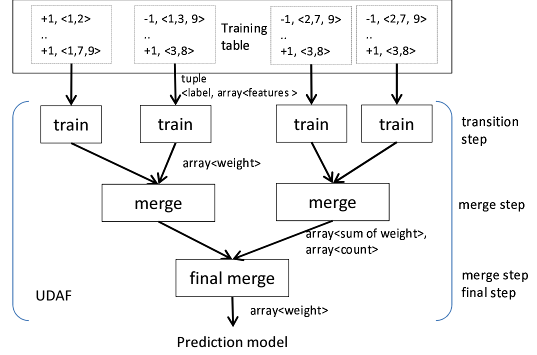


Figure 3. SGD classifier implemented as a UDAF.

Regarding the second issue, we consider that the expected frequency for applying model updates depends on the latency and throughput of both the batch and incremental learning. The expected time to transfer a prediction model from Hadoop to PostgreSQL is another important factor in hybrid learning. We discuss this issue in Section IV and provide an insight into how our system should be configured.

### A. UDTF-based Approach for Large Scale Machine Learning on Hive/Hadoop

As discussed in the previous section, a UDAF-based approach is not suitable for running machine learning on Hadoop. Instead of UDAFs, we propose a relational approach using UDTFs for machine learning on Hadoop. A UDTF is a function that takes a single input row and possibly returns more than one row. A UDTF function needs to implement three methods invoked by the system: initialize, process, and close. Hive supplies tuples to the UDTF through the process method for each row. To emit output, a UDTF calls the forward method for each tuple.

Figure 4 illustrates the basic idea behind the UDTF-based approach compared with the UDAF-based approach. In our UDTF-based approach, the classifier runs as a map-only job of Hadoop. The steps for computing a prediction model are as follows. For each *process* method call from Hive, a trainer computes the gradients and locally updates the prediction model. At the end of the input, the *close* method is called by Hive and then our UDTF-based classifier produces the final outputs by calling the *forward* method for each feature.

The training task is embarrassingly parallel and the number of map tasks is configurable through an input split size setting in Hadoop. The outputs of trainers are shuffled by feature values and each reducer computes the final model for the corresponding partition. Unlike the UDAF-based approach, the number of reducers is configurable in our UDTF-based approach and thus there is no scalability bottleneck. The training task is invoked through HiveQL as shown in Figure 5. Model averaging is expressed as a standard group-by aggregation query and executed in parallel by MapRe-

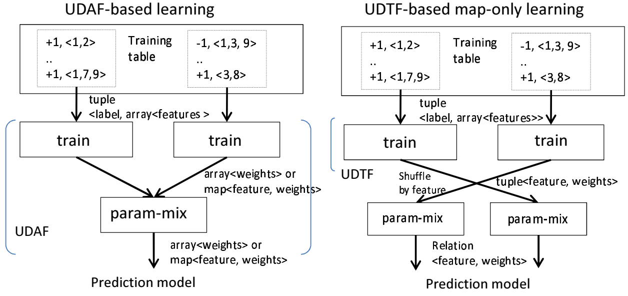


Figure 4. Comparison of the UDAF- and UDTF-based approaches.

```
1:  CREATE TABLE model AS
2:  SELECT
3:   feature, -- reducers perform model averaging in parallel
4:   avg(weight) as weight
5:  FROM
6:   (SELECT
7:     trainLogistic(features,label,..) as (feature,weight)
8:    FROM train
9:   ) t -- map-only task
10: GROUP BY feature; -- shuffled to reducers
```

Figure 5. HiveQL query to generate the prediction model on Hive.

duce. The output prediction model is in a sparse format, that is, a relation consisting of two columns: feature::int and weight::float. Because a dense model format with collection types (e.g., weights::<float[]> or weights::map<int,float>) has interoperability barriers when migrating the model from Hive to relational databases, using a relation with standard types is suitable not only for MapReduce, but also for enabling interoperability.

### B. Periodic Learning using a Database Stored Procedure

As seen in Figure 2, our online classifier periodically runs its training process bringing the prediction model up-to-date. Figure 6 shows the body of our online classifier for implementing logistic regression using a SGD method. Our online classifier is implemented as a database stored procedure that takes an old prediction model as input and returns a new prediction model as output.

Because the number of training examples is infinite in incremental learning, we use a relatively small, fixed learning-rate of 0.03. We use non-aggressive decay for the gradient updates because the model created by a batch learning process is considered to be stable if enough training examples are used. This is the reason why the batch learning process is needed as well as the incremental model updates.

The following query gives an example of how to invoke our incremental learning. An application can use the newest prediction model (or older ones) depending on its requirements.

```
1: INSERT INTO model
2: SELECT
3:  'newmodelname' as name,
4:   now() as created,
5:   updatemodel(weights) as weights
6: FROM
7:  (SELECT weights FROM model WHERE name = 'oldmodelname')
```

```
1:  FOR tr IN
2:   DELETE FROM tr_staging RETURNING label, features
3:  LOOP
4:   FOREACH f IN tr.features LOOP
5:     weightsum += weightvec[f+1];
6:   END LOOP;
7:   gradient := case when (-100.0 < weightsum) then tr.label
        - sigmoid(weightsum) else tr.label end; -- compute
        a gradient with logistic loss function
8:   delta := gradient * learningrate;
9:   FOREACH f IN tr.features LOOP -- update gradients for
       each feature
10:    weightvec[f+1] += delta;
11:  END LOOP;
12: END LOOP;
13: RETURN weightvec;
```

Figure 6. Body of the online classifier.

## IV. Experimental Evaluation

In this section, we discuss a series of experiments conducted to evaluate the classification performance as well as scalability of the proposed scheme compared with other state-of-the-art techniques [6], [7]. For all the experiments, we used a commercial advertisement dataset provided for KDD Cup 2012, Track 2 [5] as the evaluation dataset. Tencent Corporation provided a large real dataset for the KDD Cup competition, which was obtained from the logs of Tencent's commercial search engine, SOSO.com. The dataset is one of the largest publically available datasets for machine learning.

We used our in-house cluster consisting of 33 nodes for the experimental evaluation. The hardware specification of each node is given in Table I. For Hadoop, we used 32 nodes as worker nodes hosting *tasktracker* and *datanode*, and a single node as the master node hosting *jobtracker* and *namenode*. We used JBOD partitions for the HDFS as recommended in [22] and a software RAID 0 partition for the other standalone software.

Table I
Hardware specifications of the experimental environment

| | Server specifications |
|---|---|
| CPU | E5520 @ 2.27 GHz |
| Sockets | 2 |
| Cores | 4 |
| Hyper threading | 2 |
| Total threads | 16 |
| Memory | 24 GB |
| Disk | SATA 7200 rpm x3 (Software RAID 0 / JBOD) |
| Ethernet | 1 Gbps |

### KDD Cup 2012, Track 2 Task

The task in this competition was to predict the click through rate of an advertisement; in other words, we need

to predict the probability of each ad being clicked. In the competition, the Area Under Curve (AUC) [23] measure was used to evaluate the accuracy of the prediction models.

The training data consisted of 149,639,105 records derived from log messages of search sessions, a total of 10 GB of data. Because multiple sessions with the same properties were rolled into a single record in the training data, we expanded these to individual sessions. After converting categorical variables in the dataset to quantitative data for the purpose of analyzing the data, the size of the feature vector was 54,686,452. We prepared a feature vector using a hashing trick [24] for evaluation. This process was required to evaluate the task under fairer conditions because Bismarck requires each feature to be number-encoded. Using this hashing technique, the size of the feature vector was reduced from 54,686,452 to 16,777,216 (2^24). We confirmed that the effect of hash collisions was very small for both Vowpal Wabbit and our system. These feature engineering steps produced 235 million (235,582,879) training records, about 23 GB of training data.

The challenge of the KDD Cup 2012, Track 2 was not only about developing the most accurate predictor, but also the processing of relatively big data with respect to the number of training examples and the size of the feature vectors. The big data challenge in this task was, given 235 million instances as training data, to predict the click-through rates for a test set with 20,297,594 instances.

### A. Performance Comparison with Competitive Systems

To evaluate the classification performance as well as training speed of our batch learner, we compared our approach with state-of-the-art classifiers using the KDD Cup 2012, Track 2 dataset. The implementations used in the experiments were: Vowpal Wabbit (standalone version), Vowpal Wabbit (parallel version running on Hadoop), and Bismarck [7]. Both the above systems support logistic regression using SGD. We used a Hadoop cluster consisting of 32 worker nodes and a master node for evaluation. We used a worker node only for implementations executing on a single node.

Vowpal Wabbit (VW) [6] is a fast state-of-the-art learning system sponsored by Microsoft Research and previously by Yahoo! Research. VW combines online learning and a brute force gradient-descent optimization. First, VW runs a single pass SGD over the whole dataset to find a rough solution quickly. Then, a limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method is used to converge to the optimal solution. The cost function and its gradient are computed locally and AllReduce is utilized to collect the global function values and gradients for the update. Using *mapscript.sh* provided in the distribution, VW runs in parallel on Hadoop as a map-only job. In the experiments we denote the standalone version of VW that runs a single pass SGD as VW, and the parallel version running on Hadoop as VW-MR. The default setting in *mapscript.sh* runs a single

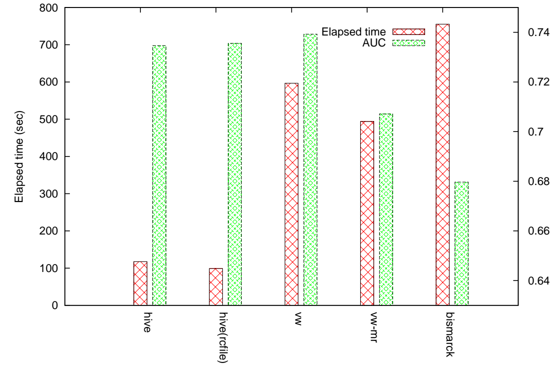


Figure 7. Performance evaluation of the KDD CUP 2012, Track 2 task.

SGD and subsequently 20 L-BFGS processes as described in [6]. We used this default setting for VW-MR, together with version 7.1 of VW.

Bismarck is an in-database analytical framework based on SGD. We ran Bismarck on PostgreSQL 9.2. On PostgreSQL, Bismarck (at least the publically available one) runs in a single thread only. We performed both ten iterations and one iteration of training. However, the resulting AUC shows no clear differences (although a worse AUC value was achieved with ten iterations) and thus we only show the results for one iteration using Bismarck.

For the other implementation, we evaluated SGD-based logistic regression implemented in Mahout [14]. However, the SGD trainer was so slow in training (it did not finish in over 3 hours), because the SGD runs only in a main thread reading data from the HDFS and does not incorporate any map-reduce. When running the same task on Liblinear [25], known as a fast classifier, more than 50 GB memory was required and it took 242 min 39.4 s for training.

Figure 7 shows a performance evaluation of each classifier carrying out the KDD CUP 2012, Track 2 task. In Figure 7, Hive represents our batch learner described in Section III-A. For Hive (rcfile), we transformed the training table in the default (SequenceFile) format to RCFile columnar format [26]. This transformation reduced the table size from 23 GB to 12.85 GB.

The experimental results clearly show that our scheme has competitive classification performance and superior training speed compared with Vowpal Wabbit and Bismarck. Hive (rcfile) finished the training in 98.78 s with acceptable prediction accuracy of 0.736. The training throughput on Hive reached 2,298,010.8 tuples/sec. We consider that this clear advantage comes from the massively parallel execution of our scheme, designed specifically for MapReduce. The parallel version of Vowpal Wabbit (VW-MR) does not show linear scalability to the number of nodes when compared with the single node version (VW). VW achieved only a

Table II
PERFORMANCE OF INCREMENTAL MODEL UPDATES RANGING FROM 80% TO 100% ON POSTGRESQL

| | Training examples | Elapsed time (s) | tuples/sec | AUC |
|---|---|---|---|---|
| Hive (80%) | 199,160,607 | 96.33 | 2067418.3 | 0.7177 |
| +0.1% updates (80.1%) | 180,375 | 4.99 | 36155.4 | 0.7197 |
| +1% updates (81%) | 1,812,531 | 25.96 | 69812.8 | 0.7242 |
| +10% updates (90%) | 18,248,992 | 256.03 | 71278.1 | 0.7291 |
| +20% updates (100%) | 36,422,272 | 499.61 | 72901.4 | 0.7349 |
| Hive (100%) | 235,582,879 | 102.52 | 2298010.8 | 0.7356 |

17.2% speedup using 32 nodes. Moreover, the AUC value decreased because parameter mixing does not work well in our setup, although this could be improved by changing certain parameters. Our scheme, on the other hand, achieves greater scalability to nodes because it was specifically designed for use with MapReduce. Bismarck required 12.6 min to finish the training, which is 7.65 times slower than Hive (rcfile).

### B. Performance Measurements of Periodic Model Updates

In the previous section, we showed that machine learning on Hive performs best for batch processing where training throughput is important. However, as seen in Figure 2, the prediction model built in the batch processes needs to be migrated to a relational database when used for transactional processing. The model migration process poses additional latency for prediction. In this section, we verify this latency for model migration and evaluate how well our Database-Hadoop hybrid approach to machine learning works.

To see the latency and throughput of each approach, we conducted an experiment using a prediction model created with 80% of the training data on Hadoop, and then incrementally updated on a relational database. In the experiments, we used PostgreSQL 9.2 for the relational database running on a server as shown in Table I.

The number of records in the 80% model built on Hadoop is 1,5635,012 (about 323 MB) and Sqoop [21] required 212.4 s to migrate the model to PostgreSQL. Model conversion from a sparse model format (i.e., <int feature, float weight>) to a dense model format (i.e., <weights float[]>) took 57.7 s. These latency numbers are higher than the training time on Hive (see Figure 7) and justify the rationale behind in-database analytics that the cost of moving data is critical for online prediction.

Our database-Hadoop hybrid approach thus performs incremental online-learning on PostgreSQL, while the initial model was built with a batch learning process on Hadoop. Table II shows the performance of the incremental model updates.

As shown in Table II, classification performance improves as the percentage of the total number of training examples used for prediction increases from 80% to 100%. Although there is some initialization cost as seen in 0.1% updates, the online learning on PostgreSQL achieves about 70,000 tuples/sec for training once the initial cost has been amortized. This means that the incremental model update scheme can be used when the rate of updates to the training set is less than 70,000 tuples/sec. Moreover, the periodic model updates in our scheme can be realized with low-latency, less than 5 s, where the difference (i.e., the number of new training examples) is less than 180,375 tuples. Given this result, we consider that the throughput/latency of the incremental model updates is acceptable not only for read-most workloads, but also for a broader range of workloads with a modest number of updates.

## V. CONCLUSIONS

In this paper, we proposed a database-Hadoop hybrid approach to scalable machine learning where batch-learning is performed on Hadoop and incremental-learning is performed on PostgreSQL. While incremental maintenance of prediction models and the software architecture for a hybrid configuration of online/offline processing have rarely been addressed in the research community, Internet service providers are facing increasing demands for hybrid online/offline modeling represented by a Netflix blog entry [27]. We believe that carrying out incremental learning (in-database analytics) on relational databases is a straightforward and efficient approach for dealing with such situations when considering the latency of transferring data between different systems, the low latency requirements of model updates, and the first-class support for transactions and data consistency in relational databases.

The contributions of this paper are summarized below:

- We proposed an architecture for online prediction in which the prediction model needs to be updated in a low latency process. We described a design principal for achieving scalable machine learning not only for read-most workloads, but also for other workloads with transactional updates.
- We verified through experiments that our batch learning scheme implemented using UDTF has competitive classification performance and superior training speed and scalability compared with state-of-the-art scalable machine learning frameworks, including Vowpal Wabbit and Bismarck.
- We showed that incremental updates to the prediction model can improve classification performance with

an acceptably small latency for updating the model, possibly less than 5 s. Based on the reported results, we consider our approach to be particularly beneficial for real world workloads where frequent model updates are mandatory.

Issues still to be resolved include integrating online A/B testing into our data processing pipeline and developing a scheme to select the best prediction model for each user in each session.

## ACKNOWLEDGMENT

This work was supported by a JSPS grant-in-aid for young scientists (B) #24700111 and a JSPS grant-in-aid for scientific research (A) #24240015.

## REFERENCES

[1] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The MADlib analytics library: or MAD skills, the SQL," *Proc. VLDB*, vol. 5, no. 12, pp. 1700–1711, Aug. 2012.

[2] Fuzzy Logix, LLC., "DB Lytix," http://www.fuzzyl.com/products/in-database-analytics/.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. OSDI*, 2004, pp. 137–150.

[4] J. Yan, X. Yang, R. Gu, C. Yuan, and Y. Huang, "Performance Optimization for Short MapReduce Job Execution in Hadoop," in *Proc. International Conference on Cloud and Green Computing (CGC)*. IEEE, Nov. 2012, pp. 688–694.

[5] KDD Cup 2012, Track 2, " http://www.kddcup2012.org/c/kddcup2012-track2."

[6] A. Agarwal, O. Chapelle, M. Dud'ık, and J. Langford, "A Reliable Effective Terascale Linear Learning System," *CoRR*, vol. abs/1110.4198, 2011.

[7] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-RDBMS analytics," in *Proc. SIGMOD*, 2012, pp. 325–336.

[8] L. Bottou, "Online Algorithms and Stochastic Approximations," in *Online Learning and Neural Networks*, D. Saad, Ed. Cambridge University Press, 1998.

[9] J. Lin and A. Kolcz, "Large-Scale Machine Learning at Twitter," in *Proc. SIGMOD*, 2012, pp. 793–804.

[10] K. B. Hall, S. Gilpin, and G. Mann, "MapReduce/Bigtable for Distributed Optimization," in *Proc. NIPS workshop on Leaning on Cores, Clusters, and Clouds*, 2010.

[11] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized Stochastic Gradient Descent," in *Proc. NIPS*, 2010, pp. 2595–2603.

[12] B. Recht, C. Re, S. J. Wright, and F. Niu, "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *Proc. NIPS*, 2011, pp. 693–701.

[13] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," in *Proc. NIPS*, 2006, pp. 281–288.

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

[15] P.-Å. Larson, "Data Reduction by Partial Preaggregation," in *Proc. ICDE*, 2002, pp. 706–715.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proc. HotCloud*, 2010, p. 10.

[17] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proc. HPDC*, 2010, pp. 810–818.

[18] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *PVLDB*, vol. 3, no. 1, pp. 285–296, 2010.

[19] R. McDonald, K. Hall, and G. Mann, "Distributed Training Strategies for the Structured Perceptron," in *Proc. NAACL*, 2010, pp. 456–464.

[20] S. Das, C. Botev, K. Surlaker, B. Ghosh, B. Varadarajan, S. Nagaraj, D. Zhang, L. Gao, J. Westerman, P. Ganti, B. Shkolnik, S. Topiwala, A. Pachev, N. Somasundaram, and S. Subramaniam, "All aboard the Databus!: Linkedin's scalable consistent change data capture platform," in *Proc. SOCC*, 2012, pp. 18:1–18:14.

[21] Apache Sqoop, "http://sqoop.apache.org/."

[22] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, June 2009.

[23] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, July 1997.

[24] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proc. ICML*, 2009, pp. 1113–1120.

[25] R. E. Fan, K. W. Chang, C. J. Hsieh, X. R. Wang, and C. J. Lin, "LIBLINEAR: A Library for Large Linear Classification," *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, June 2008.

[26] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems," in *Proc. ICDE*. IEEE, 2011, pp. 1199–1208.

[27] The Netflix Tech Blog, "System Architectures for Personalization and Recommendation," http://techblog.netflix.com/2013/03/system-architectures-for.html.