Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 236084, 12 pages
http://dx.doi.org/10.1155/2015/236084

Research Article

Efficient ELM-Based Two Stages Query Processing Optimization for Big Data

Linlin Ding,[1] Yu Liu,[1] Baoyan Song,[1] and Junchang Xin[2]

[1] *School of Information, Liaoning University, Shenyang, Liaoning 110036, China*
[2] *College of Information Science & Engineering, Northeastern University, Shenyang, Liaoning 110819, China*

Correspondence should be addressed to Baoyan Song; bysong@lnu.edu.cn

Received 22 August 2014; Accepted 27 November 2014

Academic Editor: Yi Jin

MapReduce and its variants have emerged as viable competitors for big data analysis with a commodity cluster of machines. As an extension of MapReduce, ComMapReduce realizes the lightweight communication mechanisms to enhance the performance of query processing applications for big data. However, different communication strategies of ComMapReduce can substantially affect the executions of query processing applications. Although there is already the research work that can identify the communication strategies of ComMapReduce according to the characteristics of the query processing applications, some drawbacks still exist, such as relative simple model, too much user participation, and relative simple query processing execution. Therefore, an efficient ELM-based two stages query processing optimization model is proposed in this paper, named ELM to ELM (*E2E*) model. Then, we develop an efficient sample training strategy to train our *E2E* model. Furthermore, two query processing executions based on the *E2E* model, respectively, Just-in-Time execution and Queue execution, are presented. Finally, extensive experiments are conducted to verify the effectiveness and efficiency of the *E2E* model.

1. Introduction

Nowadays, MapReduce [1] has become a widespread programming framework for big data analysis. Hadoop (http://hadoop.apache.org/) is a popular open-source implementation of MapReduce that many academies and industrial organizations adopt it in research and production deployments. The success of MapReduce stems from hiding the details of parallelization, fault-tolerance, and load balancing in a simple programming framework. MapReduce and its variants are used to process the big data applications, such as Web indexing, data mining, machine learning, financial analysis, scientific simulation, and bioinformatics research [2–10].

As one of the successful extensions of MapReduce, ComMapRedcue [3, 4] adds simple lightweight communication mechanisms to generate the certain *shared information* and implements the query processing applications for big data. In ComMapReduce framework, three basic and two optimization communication strategies are presented to solve the problem of how to communicate and obtain the *shared information* of different applications. After further analyzing the ComMapReduce execution course and the abundant experiments, we find out that different communication strategies of ComMapReduce can substantially impact the performance of query processing applications.

To identify the communication strategy, the existing research work [11] proposes a query optimization model, named *ELM_CMR*, based on ELM [12] which has the classification performance at an excellent standard. According to the characteristics of query processing programs, *ELM_CMR* can identify the communication strategy of ComMapReduce. However, *ELM_CMR* still has the following drawbacks. First, regardless of the MapReduce or ComMapReduce framework, a program only consists of black_box Map and Reduce functions, without knowing the distributed details about the framework. But the configuration parameters of the framework can fully influence the performance of query processing. Finding the most suitable configuration parameters setting itself is difficult. The burden falls on the user who submits the MapReduce or ComMapReduce job to specify settings for all configuration parameters, which highlights the challenges the user faces. How to identify the proper configuration parameters according to the user program is a difficult problem. Second, the implementation algorithm of multiple queries in *ELM_CMR* only processes the queries based on the classification results and predicted execution time, but there is no consideration of the characteristics among different queries. Different queries can share computation and data so as to enhance the performance further. Third, *ELM_CMR* only adopts simple training method to gain the classification model. If more efficient training method is adopted to obtain the training data, the accuracy of the classification model can be improved.

Therefore, in this paper, for resolving the above problems and drawbacks, we propose an efficient ELM-based two stages query processing optimization model, named ELM to ELM (*E2E*) model. According to the characteristics of the users programs, the first stage can gain the *feature parameters* using ELM algorithm. After that, according to the results of the first stage, the second stage can identify the final classification result by ELM algorithm too. Furthermore, an efficient sample training strategy is presented to train the *E2E* model. We also develop two query processing executions based on the *E2E* model, Just-in-Time execution and Queue execution. The contributions of this paper can be summarized as follows.

(i) We propose an efficient ELM-based two stages query processing optimization model, named *E2E* model, which can realize the most optimal executions of query processing applications in MapReduce or ComMapReduce framework.

(ii) We develop a sample training strategy to train our *E2E* model and two query executions based on *E2E* model, Just-in-Time execution, and Queue execution.

(iii) The experimental studies using synthetic data show the effectiveness and efficiency of the *E2E* model.

The remainder of this paper is organized as follows. Section 2 briefly introduces the background, containing the ELM and *ELM_CMR*. Our *E2E* model is proposed in Section 3. The two executions for query processing applications based on *E2E* are presented in Section 4. The experimental results to show the performance of *E2E* model are reported in Section 5. Finally, we conclude this paper in Section 6.

2. Background

In this section, we describe the background of our work, which includes a brief overview of the traditional ELM and the detailed descriptions of the *ELM_CMR*.

2.1. Review of ELM. Nowadays, extreme learning machine (ELM) [12] and its variants [13–28] have the characteristics of excellent generalization performance, rapid training speed, and little human intervene, which have attracted extensive attention from more and more researchers. ELM is originally designed for the single hidden-layer feedforward neural networks (SLFNs [29]) and is then extended to the "generalized" SLFNs. ELM algorithm first randomly allocates the input weights and hidden layer biases and then analytically computes the output weights of SLFNs. Contrary to the other conventional learning algorithms, ELM reaches the optimal generalization performance at a very fast learning speed. ELM is less sensitive to the user defined parameters, so it can be deployed fast and convenient. That is the reason we choose ELM as our basic method to construct the query processing optimization model.

For N arbitrary distinct samples $(\mathbf{x}_j, \mathbf{t}_j)$, where $\mathbf{x}_j = [x_{j1}, x_{j2}, \ldots, x_{jn}]^T \in \mathbb{R}^n$ and $\mathbf{t}_j = [t_{j1}, t_{j2}, \ldots, t_{jm}]^T \in \mathbb{R}^m$, standard SLFNs with hidden nodes L and activation function $g(x)$ are mathematically modeled as

$$\sum_{i=1}^{L} \boldsymbol{\beta}_i g_i(\mathbf{x}_j) = \sum_{i=1}^{L} \boldsymbol{\beta}_i g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i) = \mathbf{o}_j \quad (j = 1, 2, \ldots, N), \tag{1}$$

where L is the number of hidden layer nodes, $\mathbf{w}_i = [w_{i1}, w_{i2}, \ldots, w_{in}]^T$ is the weight vector between the ith hidden node and the input nodes, $\boldsymbol{\beta}_i = [\beta_{i1}, \beta_{i2}, \ldots, \beta_{im}]^T$ is the weight vector connecting the ith hidden node and the output nodes, b_i is the threshold of the ith hidden node, and $\mathbf{o}_j = [o_{j1}, o_{j2}, \ldots, o_{jm}]^T$ is the jth output vector of the SLFNs.

The standard SLFNs can approximate these N samples with zero error. The error of ELM is $\sum_{j=1}^{L} \|\mathbf{o}_j - \mathbf{t}_j\| = 0$ and there exist $\boldsymbol{\beta}_i$, $\mathbf{w}_i$, and b_i such that

$$\sum_{i=1}^{L} \boldsymbol{\beta}_i g(\mathbf{w}_i \cdot \mathbf{x}_j + b_i) = \mathbf{t}_j \quad (j = 1, 2, \ldots, N). \tag{2}$$

Equation (2) can be expressed compactly as follows:

$$\mathbf{H}\boldsymbol{\beta} = \mathbf{T}, \tag{3}$$

where

$$\begin{aligned} &\mathbf{H}(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_L, b_1, b_2, \ldots, b_L, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L) \\ &= \begin{bmatrix} h(x_1) \\ h(x_2) \\ \vdots \\ h(x_N) \end{bmatrix} \\ &= \begin{bmatrix} g(\mathbf{w}_1 \cdot \mathbf{x}_1 + b_1) & g(\mathbf{w}_2 \cdot \mathbf{x}_1 + b_2) & \ldots & g(\mathbf{w}_L \cdot \mathbf{x}_1 + b_L) \\ g(\mathbf{w}_1 \cdot \mathbf{x}_2 + b_1) & g(\mathbf{w}_2 \cdot \mathbf{x}_2 + b_2) & \ldots & g(\mathbf{w}_L \cdot \mathbf{x}_2 + b_L) \\ \vdots & \vdots & \vdots & \vdots \\ g(\mathbf{w}_1 \cdot \mathbf{x}_N + b_1) & g(\mathbf{w}_2 \cdot \mathbf{x}_N + b_2) & \ldots & g(\mathbf{w}_L \cdot \mathbf{x}_N + b_L) \end{bmatrix}_{N\times L}, \end{aligned} \tag{4}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_1^T \\ \beta_2^T \\ \vdots \\ \beta_L^T \end{bmatrix}_{L\times m}, \qquad \mathbf{T} = \begin{bmatrix} t_1^T \\ t_2^T \\ \vdots \\ t_N^T \end{bmatrix}_{N\times m}. \tag{5}$$

$\mathbf{H}$ is set as the hidden layer output matrix of the neural network. The ith column of $\mathbf{H}$ is called the ith hidden node output with respect to inputs $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$. The smallest norm least-squares solution of the above multiple regression system is shown as follows:

$$\widehat{\boldsymbol{\beta}} = \mathbf{H}^{\dagger}\mathbf{T}, \tag{6}$$

where $\mathbf{H}^{\dagger}$ is the Moore-Penrose generalized inverse of matrix $\mathbf{H}$. Then the output function of ELM can be modeled as follows:

$$f(\mathbf{x}) = \mathbf{h}(\mathbf{x})\boldsymbol{\beta} = \mathbf{h}(\mathbf{x})\mathbf{H}^{\dagger}\mathbf{T}. \tag{7}$$

The computational course for ELM is shown in Algorithm 1. Only after properly setting the related parameters, ELM can start the training process. The first step is to generate L pairs of hidden node parameters $(\mathbf{w}_i, b_i)$ (Lines 1–3). The second step actually calculates the hidden layer output matrix $\mathbf{H}$ by using (4) (Line 4). The third step mainly computes the corresponding output weight vector $\boldsymbol{\beta}$ (Line 5). After completing the above training process, the output of the new dataset can be predicted by ELM according to (7).

2.2. ELM_CMR Model. ComMapReduce [3, 4] is an improved MapReduce framework with lightweight communication mechanisms. A new node, named the Coordinator node, is added to store and generate the certain *shared information* of different applications. In ComMapReduce, three basic communication strategies, LCS, ECS, and HCS, and two optimization communication strategies, PreOS and PostOS, are proposed to identify how to receive and generate the *shared information*. In short, without affecting the existing characteristics of the original MapReduce framework, ComMapReduce is a successful parallel programming framework with global *shared information* to filter the unpromising data of query processing programs.

The existing *ELM_CMR* [11] is an efficient query processing optimization model based on ELM. It can identify the communication strategies of query processing applications in ComMapReduce according to the features of queries. Figure 1 shows the architecture of *ELM_CMR* model. The four components of *ELM_CMR* are, respectively, the *Feature Selector*, the *ELM Classifier*, the *Query Optimizer,* and the *Execution Fabric*.

The *Feature Selector* mainly examines the training query processing programs and selects the configuration parameters that can wholly affect the query performance by the job profiles. Naturally, the parameters of program p can be divided into three types, parameters that predominantly affect Map task execution; parameters that predominantly affect Reduce task execution; and the cluster parameters. Then, in each cluster, we adopt the minimum-redundancy-maximum-relevance (mRMR) [30] feature selection to find the optimal parameters sharply affecting the performance. And then, we generate the globally optimal configuration parameter settings by combining the results of each subspace. Therefore, the near-optimal configuration parameter setting can be generated.

After selecting the features of training data, the *Feature Selector* sends the extracted training data to the *ELM Classifier*. It uses the training data to construct the ELM model by the traditional ELM algorithm. After that, when there are one or multiple queries to be processed, the *ELM Classifier* can rapidly obtain the classification results of the queries and then sends them to the *Query Optimizer*.

The *Query Optimizer* applies the classification results of the *ELM Classifier* and combines the implementation patterns to choose an optimized *execution order*. After gaining the *execution order*, the query is sent to the *Execution Fabric*.

The *Execution Fabric* implements the program in ComMapReduce framework. When there is one query to be processed, the *Execution Fabric* implements the query according to the classification result of the *Query Optimizer* in *ELM_CMR*. When there are multiple queries to be processed, the multiple queries can be classified by *ELM Classifier* and gain the best communication strategy of each program. Then, a Task Scheduler Simulator is used to simulate the execution time of queries. According to the execution time and the classification results of the queries, the *Query Optimizer* designs an *execution order* following the common principle of Shortest Job First (*SJF*) to implement multiple queries.

3. *E2E* Model

In this section, the overview of our *E2E* model is introduced first in Section 3.1. Then, training the *E2E* model is shown in Section 3.2. Finally, predicting the *E2E* model is proposed in Section 3.3.

3.1. Overview of E2E Model. Our *E2E* model can identify the optimal communication strategies of query processing programs in MapReduce or ComMapReduce, which contains three main phases, respectively, the *training phase*, the *prediction phase,* and the *execution phase*. Figure 2 shows the whole workflow of query processing in the *E2E* model. The main workflow is as follows.

First, the *training phase* is responsible for extracting the training query processing programs that have a large effect on the *E2E* model. In the *training phase*, we use sample-based training strategies to run our workload, in isolation, pairwise, and at several higher multiprogramming levels. After gaining the training samples, they can be used to generate the *E2E* model in the *prediction phase*. The details of training phase will be introduced in Section 3.2.

Second, in the *prediction phase*, by using the training samples from the *training phase*, the two stages *E2E* model can be generated based on the traditional ELM algorithm. According to the user's programs, the first stage can obtain the most optimal *feature parameters* of the programs by ELM. And then, using the *feature parameters* of the first stage, the second stage can identify the classification results of the query processing programs by ELM. The details of the *prediction phase* will be shown in Section 3.3.

Third, after gaining the *E2E* model, for the query processing programs submitted to MapReduce or ComMapReduce,

```
(1) for $i = 1$ to $L$ do
(2)     Randomly generate the hidden node parameters $(\mathbf{w}_i, b_i)$;
(3) end for
(4) Calculate the hidden layer output matrix $\mathbf{H}$;
(5) Calculate the output weight vector $\boldsymbol{\beta} = \mathbf{H}^{\dagger}\mathbf{T}$;
```

Algorithm 1: ELM.

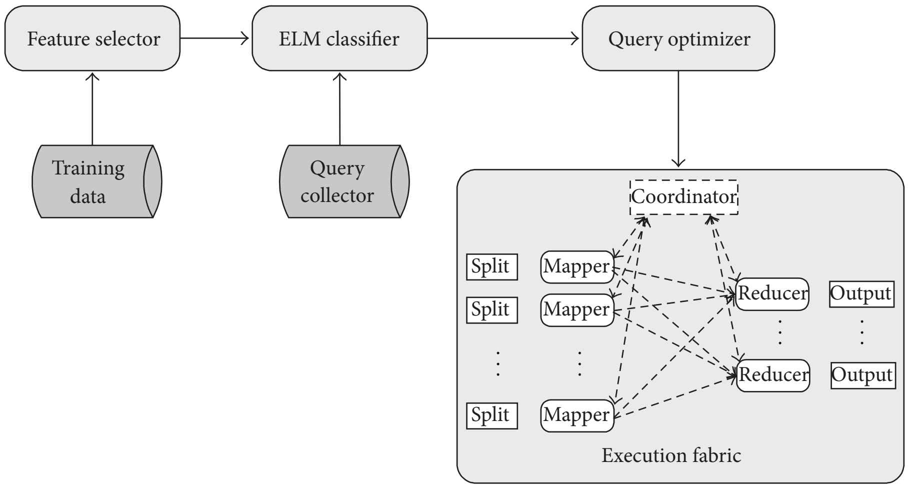

Figure 1: Architecture of *ELM_CMR* model.

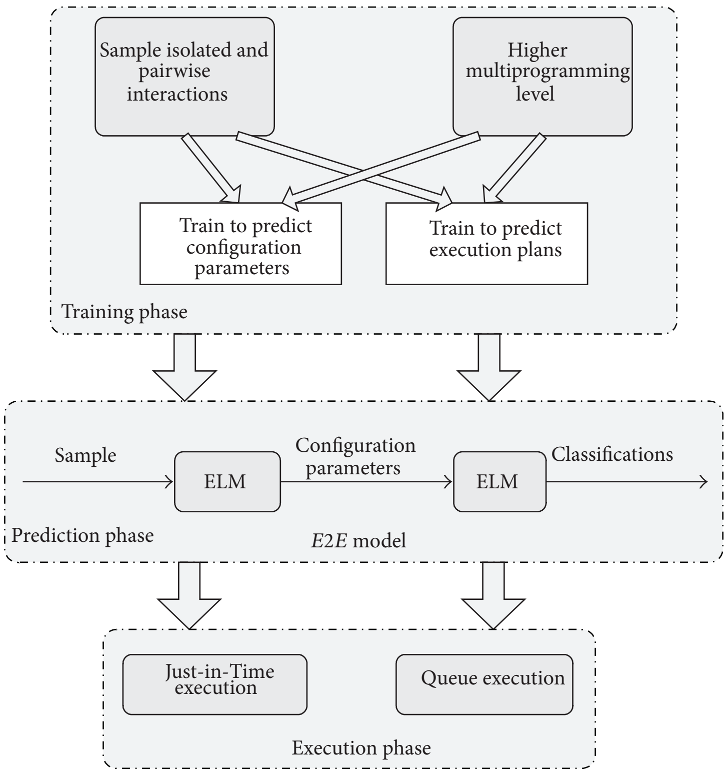
Sample isolated and pairwise interactions
Higher multiprogramming level
Train to predict configuration parameters
Train to predict execution plans
Training phase
Sample
ELM
Configuration parameters
ELM
Classifications
Prediction phase
E2E model
Just-in-Time execution
Queue execution
Execution phase

Figure 2: Workflow of *E2E* model.

we can predict their optimal communication strategies in the *execution phase*. We propose two query executions based on *E2E* model, the Just-in-Time execution which tackles one query processing program, and the Queue execution which tackles multiple concurrent queries. The details of the execution phase will be illustrated in Section 4.

3.2. Training E2E Model. To obtain the prediction model more accurately, we need to train the *E2E* model. Different from the simple training course of *ELM_CMR* model, the *training phase* of our *E2E* model consists of running the queries in isolation, pairwise as well as at several higher multiprogramming levels. In addition, in order to gain the measurements that capture the interactions, we omit the first few samples in our experiments. This approach makes us ignore the overhead of the initial query setup and caching the initial supporting structures.

The *E2E* model realizes the *training phase* by sampling approach, containing isolation, pairwise, and higher degree of concurrency, which allows us to approximate the running of queries in MapReduce or ComMapReduce. The course of the *training phase* is displayed in Figure 2. First, we sample the workload in isolation to gain how each query behaves, which can be seen as the baseline of training. Second, according to the query types in our experiments, we build a matrix of interaction by running all unique pairwise combinations. Pairwise sample can help us to simply estimate the degree of concurrency. Here, Latin hypercube sampling approach (LHS) can uniformly distribute our samples throughout our prediction space, which can be realized by creating a hypercube with the same dimension as the multiprogramming level. We adopt LHS to sample at pairwise or several higher multiprogramming levels. We then select the samples that every value on every plane gets inter selected exactly one. A simple example of two-dimensional LHS is shown in Table 1. In Table 1, Query-A, Query-B, Query-C, and Query-D stand for four queries to be processed by pairwise combinations. After combining using LHS, the four queries can obtain the optimal combinations.

Furthermore, the above strategy shows how to run the samples, in isolation, pairwise, and at several higher multiprogramming levels using LHS. As we know, the number of queries to be processed is large, and in order to enhance the training accuracy, we need to choose the queries that is the representative queries in each query type and then construct the samples to be trained. The corresponding cluster algorithms based on DBSCAN [31] can be also adopted to gain the most representative query in each query type. In practice, although the training time of *E2E* may be long, it is worth mentioning that *training phase* allows our model to be extremely lightweight once it reaches the *prediction phase* and enhance the accuracy.

3.3. Predicting E2E Model. Then, we introduce the details of the *prediction phase*. A MapReduce job j can be expressed by a MapReduce program p running on input data d and cluster r, which can be expressed as $j = \langle p, d, r, c\rangle$ in short. We call the d, r, and c the *feature parameters* of p. The users only submit their jobs to MapReduce or ComMapReduce without knowing the internal configuration details of the system.

Table 1: An example of LHS.

Query	Query-A	Query-B	Query-C	Query-D
Query-A		▲		
Query-B			▲	
Query-C	▲			
Query-D				▲

Table 2: *Feature parameters* in the experiments.

Property name	Type	Default value
io.sort.mb	int	100
io.sort.factor	int	10
min.num.spills.for.combine	int	3
mapred.compress.map.output	boolean	false
mapred.reduce.parallel.copies	int	5
mapred.reduce.copy.backoff	int	300
dfs.heartbeat.interval	int	3
dfs.block.size (M)	int	64
mapred.map.task	int	4
mapred.reduce.task	int	4
mapred.tasktracker.map.task.maximum	int	4
mapred.tasktracker.reduce.task.maximum	int	4
Data size (G)	int	10
Data distribution	boolean	uniform
Number of slave nods	int	8

However, a number of choices can be made in order to fully specify how the job should be executed. These choices, represented by c in $\langle q, d, r, c\rangle$, stand for a high dimensional space of configuration parameter settings, such as the number of Map and Reduce task, the block size, and the amount of memory. The performance changes a lot in different configuration parameters. For any parameter, its value is not specified explicitly during job submission, the default values either shipped with the system or specified by the system administrator. However, the normal users do not understand the running details of MapReduce. That is to say, finding good configuration settings for MapReduce job is time consuming and requires extensive knowledge of system internals. Because the users have little information of the parallelization details of MapReduce, it is necessary to gain the suitable configuration parameters. The first stage of our *E2E* model is to generate the model for identifying the suitable configuration parameters of query processing programs by ELM. The main goal of the first stage is to construct a black_box *feature parameters* of queries, containing $\langle d, r, c\rangle$. The configuration parameters wholly affecting the performance adopted by *E2E* are the same as our *ELM_CMR* approach as shown in Table 2. The corresponding information of the job can be obtained from sampling a few tasks of the job. After the first stage, the *feature parameters* setting can be obtained by ELM algorithm.

```
(1) Generate the feature parameters of $j$ by the first stage of E2E model;
(2) Generate the execution plan of $j$ by the second stage of E2E model;
(3) Execute $j$ with its communication strategy;
```

Algorithm 2: Just-in-Time execution.

After gaining the configuration parameters, the second stage is to use them to predict the communication strategy of ComMapReduce or MapReduce by ELM algorithm too and then generates the classification results. This stage is similar to the prediction of *ELM_CMR*, so we do not illustrate in detail in this section.

4. Executions of *E2E* Model

After generating the *E2E* model, the pending queries can be implemented under our *E2E* model. In this section, two scenarios are proposed for executing query processing programs using the *E2E* model. In the first scenario, one new query is being submitted for immediate execution, named as Just-in-Time execution. In the second scenario, a Queue-based multiple concurrent queries execution is proposed, where an ordered list of queries to run is gained.

4.1. Just-in-Time Execution. When there is one new query being submitted, the *E2E* model generates its classification result as soon as possible. According to the characteristics of the coming query, the first stage of *E2E* model can generate the *feature parameters* of this query based on ELM. After that, the user can make a decision that whether adopting ComMapReduce or which communication strategy can be adopted by the second stage of *E2E* model, and then implements the query processing application.

The course of Just-in-Time execution is shown in Algorithm 2. First, the most optimal *feature parameters* of query processing job j are extracted in the first stage of *E2E* model (Line 1). Second, after obtaining the *feature parameters* of job j, the *E2E* generates the classification result of j (Line 2). Third, according to the classification of j, the *E2E* ensures how to implement the query and sends it to MapReduce or ComMapReduce framework. The framework uses the optimization result to execute the query program (Line 3).

For example, for a submitted skyline query, the first stage of *E2E* can obtain the most optimal *feature parameters* of this skyline query. After abstracting its *feature parameters*, the *E2E* can generate its classification and then identifies the communication strategy of this skyline query, such as PostOS. After that, the skyline query will be implemented in ComMapReduce with PostOS.

4.2. Queue Execution. When there are multiple concurrent queries to be processed, we design an efficient *execution order* of queries based on *E2E* model. Under the *execution order*, the performance of multiple concurrent queries can reach a nice status. Of course, we can execute these queries one by one with their classification results of *E2E*, but a lot of calculations and execution time would be wasted. Therefore, the *execution order* is important to enhance the performance of multiple concurrent queries.

As we know, different queries usually contain the overlap parts, such as the same query type and the same processing data. Therefore, when processing multiple concurrent queries, if we can identify the correlations of query type and processing data of the multiple queries, then the sharing computation and data can be used to prevent the repeated processing and scanning.

The main processing course of Queue execution is shown in Figure 3. There are three main components in the Queue execution, respectively, the *Analyzer*, the *Optimizer*, and the *Executor*.

First, for the multiple concurrent queries to be executed, after obtaining the classification results by *E2E* model, the *Analyzer* analyzes the queries and divides the queries into some subsets by considering the share of computation. This dividing approach can divide the queries with the same query type into the same subset, then the queries in a subset can be computed together.

Second, in each subset, the *Optimizer* can design the local execution order according to the characteristics of the sharing computation and data. It can analyze the data being processed of the multiple concurrent queries in the same subset, and then the queries with sharing data can share the computation results of the sharing data. The *Optimizer* also can cache the corresponding intermediate results into memory or buffer by the characteristics of the queries so as to reduce the repeated implementations of the same operations and I/O cost. Then, similar to our former *ELM_CMR*, the *Optimizer* also uses a Task Scheduler Simulator to simulate the scheduling and executions of Map and Reduce tasks of each query. The execution time of a job j can be estimated by the Task Schedular Simulator. So, according to the common principle of Shortest Job First (*SJF*) too, the *execution order* of each subset is gained.

Finally, after obtaining the *execution order* of each subset, the *Executor* also uses the *SJF* principle of the subsets by simple merging their execution time of each subset, so the global execution order (O_s) of the multiple queries is the ascending order of the simulated execution time of each subset. Then, the multiple queries can be implemented under the global *execution order*.

Algorithm 3 illustrates the whole execution course of the multiple concurrent queries. First, we can obtain the classification of each query by the *E2E* model (Lines 1–3). Then, the *Analyzer* divides the queries sharing computation into some subsets (Line 4). Then, in each subset, the *Optimizer*

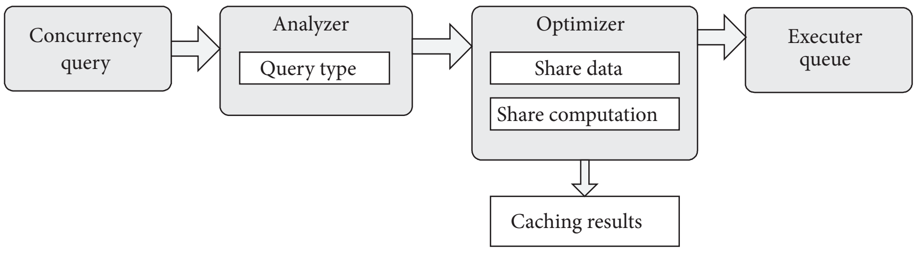

Figure 3: Workflow of Queue execution.

```
(1)  for each query $q_i$ do
(2)      Generate the classification of $q_i$ by E2E;
(3)  end for
(4)  Divide the queries into some subsets by query type;
(5)  for each subset do
(6)      Analyze and divide the sharing data;
(7)      Compute and query of the sharing data;
(8)      Calculate the simulate execution time;
(9)      Generate the local execution order of each subset by SJF;
(10) end for
(11) Generate the global execution order by SJF;
```

Algorithm 3: Multiple queries.

can analyze and divide the sharing data, and then computes the sharing data. After gaining the simulated time of each query in each subset, the local *execution order* is obtained by considering the sharing and features of the queries in each subset by *SJF* (Lines 5–10). Finally, the global *execution order* of the queries is generated by *SJF* too (Line 11).

Figure 4 shows an example of the implementation of the multiple concurrent queries. Suppose that there are eight queries to be processed. First, these queries can obtain their classification results by *E2E*. After that, the *Analyzer* divides these queries into four subsets by the sharing of computation, the same query type, respectively, top-*k*, *k*NN, skyline, and join.

In each subset, the *Optimizer* can make a local *execution order* by the features of the queries. For example, the subset of skyline query contains three skyline queries, q_2, q_5, and q_8, with their processing data sets $s_2 = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $s_5 = \{p_1, p_2, p_3, p_7, p_8, p_9\}$, and $s_8 = \{p_4, p_5, p_6, p_7, p_8, p_9\}$. For these skyline queries, the *Optimizer* divides the sharing data and gains the following sharing subdatasets, $s_a = \{p_1, p_2, p_3\}$, $s_b = \{p_4, p_5, p_6\}$, and $s_c = \{p_7, p_8, p_9\}$. Then, the *Optimizer* can transfer the above skyline queries into some simple skyline queries, $s_2 = s_a \cup s_b$, $s_5 = s_a \cup s_c$, and $s_8 = s_b \cup s_c$. So, the *Optimizer* can make a solution that the skyline queries first compute the results of s_a, s_b, and s_c, and then gain a local *execution order* by *SJF*, q_5, q_8, and q_2. The similar processing course is fit for the top-*k* and *k*NN queries. For our join query of small-big tables, the *Optimizer* can cache the corresponding small table into memory of Map tasks so as to reduce the repeated scan of the small table. Finally, the global *execution order* O_s, q_7, q_1, q_5, q_8, q_2, q_3, q_4, and q_6 can be gained by simple merge of *SJF* of the four subsets.

5. Performance Evaluation

In this section, the performance of our *E2E* model is evaluated in detail with various experimental settings. We first describe the setup used in our experiments in Section 5.1. Then we present and discuss the experimental results in Section 5.2.

5.1. Experimental Setup. The experimental setup is the same as *ELM_CMR* model as follows. The experimental setup is a Hadoop cluster running on 9 nodes in a high speed Gigabit network, with one node as the Master node and the Coordinator node and the others as the Slave nodes. Each node has an Intel Quad Core 2.66 GHz CPU, 4 GB memory, and CentOS Linux 5.6. We use Hadoop 0.20.2 and compile the source codes under JDK 1.6. The ELM algorithm is implemented in MATLABR2009a.

ELM_CMR is an efficient query processing optimization model based on ELM. It can identify the communication strategies of query processing applications in ComMapReduce according to the features of queries. So far, except the *ELM_CMR* model, there is no any other method to study how to choose the optimal strategies of ComMapReduce for query processing applications. So, we evaluate the performance of *E2E* model in different implementations for Just-in-Time execution and Queue execution by comparing with *ELM_CMR*.

In order to fully evaluate the performance of *E2E* model, four typical query processing applications are adopted to

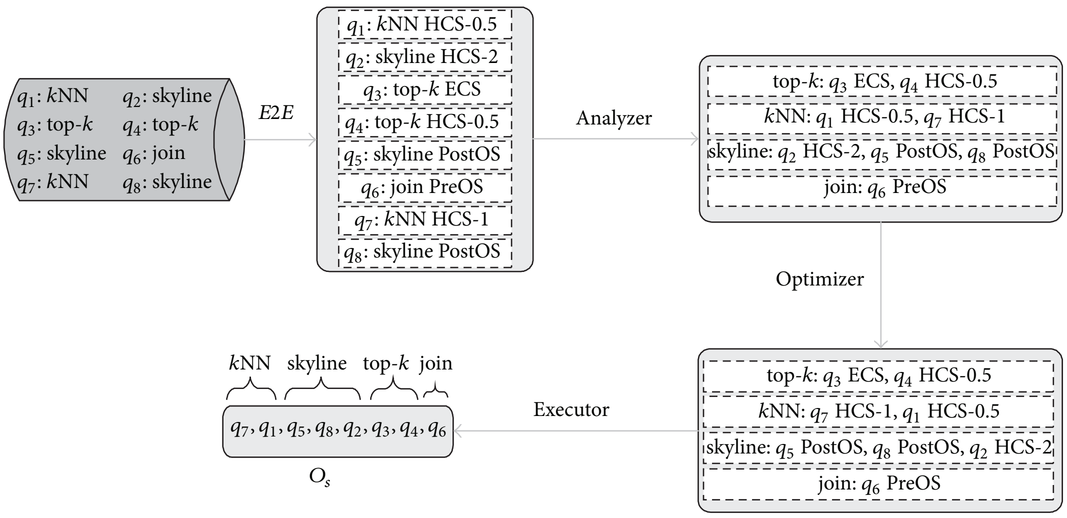
q_1: kNN q_2: skyline
q_3: top-k q_4: top-k
q_5: skyline q_6: join
q_7: kNN q_8: skyline
E2E
q_1: kNN HCS-0.5
q_2: skyline HCS-2
q_3: top-k ECS
q_4: top-k HCS-0.5
q_5: skyline PostOS
q_6: join PreOS
q_7: kNN HCS-1
q_8: skyline PostOS
Analyzer
top-k: q_3 ECS, q_4 HCS-0.5
kNN: q_1 HCS-0.5, q_7 HCS-1
skyline: q_2 HCS-2, q_5 PostOS, q_8 PostOS
join: q_6 PreOS
Optimizer
top-k: q_3 ECS, q_4 HCS-0.5
kNN: q_7 HCS-1, q_1 HCS-0.5
skyline: q_5 PostOS, q_8 PostOS, q_2 HCS-2
join: q_6 PreOS
Executor
kNN skyline top-k join
$q_7, q_1, q_5, q_8, q_2, q_3, q_4, q_6$
O_s

Figure 4: Implementation of multiple queries.

evaluate the implementations of Just-in-Time execution and Queue execution, respectively, top-k, kNN, skyline, and join. In Just-in-Time execution evaluating, we, respectively, test the performance of top-k, kNN, skyline, and join under different data sizes in *E2E* and *ELM_CMR*. In Queue execution evaluating, we design two kinds of experiments. One is that the multiple queries are the same kind of query under different data sizes, taking skyline query as an example. The other one is that the multiple queries are the different kinds of queries under different data sizes, taking top-k (2 G), top-k (8 G), kNN (2 G), and kNN (8 G) as examples.

We use the synthetic data to test the performance. The synthetic datasets include different data sizes and different distributions, such as uniform distribution in top-k and kNN, anticorrelated distribution in skyline, and small-big tables in join. They can reflect the performance under different situations. The classification results of *E2E* model contain 7 types, respectively, ECS, HCS-0.5, HCS-1, HCS-2, PreOS, PostOS, and MapReduce (MR). HCS-0.5 means the preassigned time interval of HCS is 0.5 s. Table 2 summarizes the parameters used in the experiments including the default values.

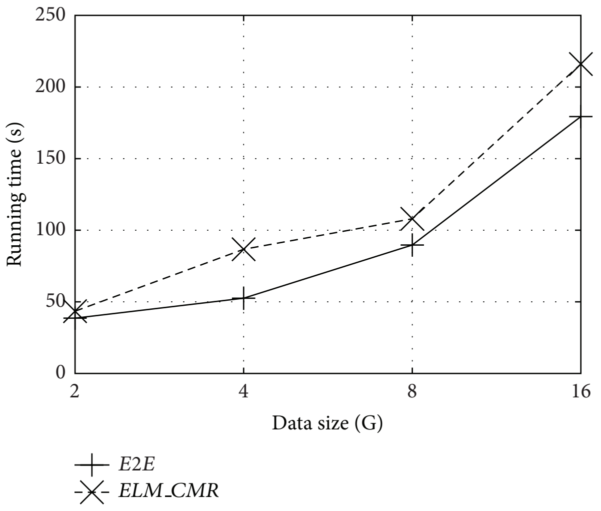

Figure 5: Performance of top-k query.

5.2. Experimental Results. First, we give the classification results of *E2E* model and *ELM_CMR* model by processing the above four query processing applications in different data size shown in Table 3. We can see that for the same query type and data size, the classification results of the queries are different, so the performance of the two models is different too. In the following, we evaluate the performance of the queries according to the classification results of Table 3.

Second, the performance of Just-in-Time execution is shown in the following figures. Figure 5 shows the performance of top-k queries ($k = 1000$), with the data size is 2 G, 4 G, 8 G, and 16 G in uniform distribution. We can see that the performance of top-k queries in the classification results of *E2E* model is better than *ELM_CMR*. The reason is that *E2E* model can effectively evaluate the optimal configuration parameters of the queries than *ELM_CMR* model. When k is much smaller than the original data, the global *shared information* of ComMapReduce can reach the most optimal one quickly, so the Mappers can retrieve the *shared information* in the initial phase to filter the unpromising data.

Figure 6 shows the performance of kNN queries with different data size of uniform distribution. The performance of *E2E* is also optimal to *ELM_CMR* model, where the reason is that *E2E* model can gain the suitable classification results.

Figure 7 shows the performance of skyline queries in anticorrelated distribution with different data size. We can see that the performance of different execution plans is not obviously different, but PostOS is a little better. The reason is that the original data are skewed to the final results in anticorrelated distribution. The percentage of filtering is low, so the performance difference is not obvious. In this situation, although *E2E* and *ELM_CMR* can obtain the classification, it can also choose the other communication strategies.

Table 3: Classifications of *E2E* and *ELM_CMR*.

Query	top-*k*				*k*NN				skyline				join			
Data (G)	2	4	8	16	2	4	8	16	0.2	0.4	0.8	1.6	5	10	15	20
E2E	HCS-0.5	HCS-0.5	PreOS	PreOS	ECS	HCS-0.5	PreOS	PreOS	HCS-0.5	HCS-0.5	PostOS	PostOS	ECS	ECS	ECS	PreOS
ELM_CMR	ECS	ECS	HCS-1	HCS-0.5	HCS-0.5	HCS-1	HCS-2	HCS-2	ECS	HCS-1	HCS-2	HCS-2	MR	HCS-1	HCS-2	HCS-2

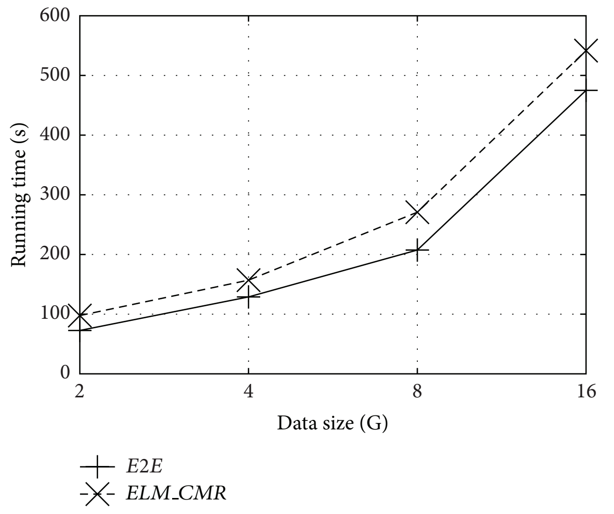

Figure 6: Performance of *k*NN query.

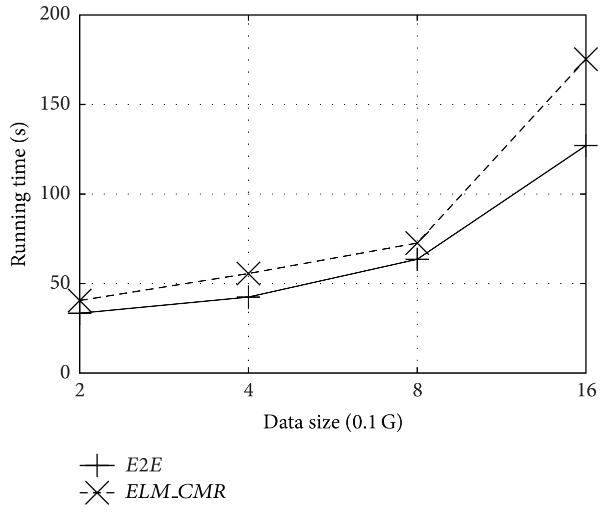

Figure 7: Performance of skyline query.

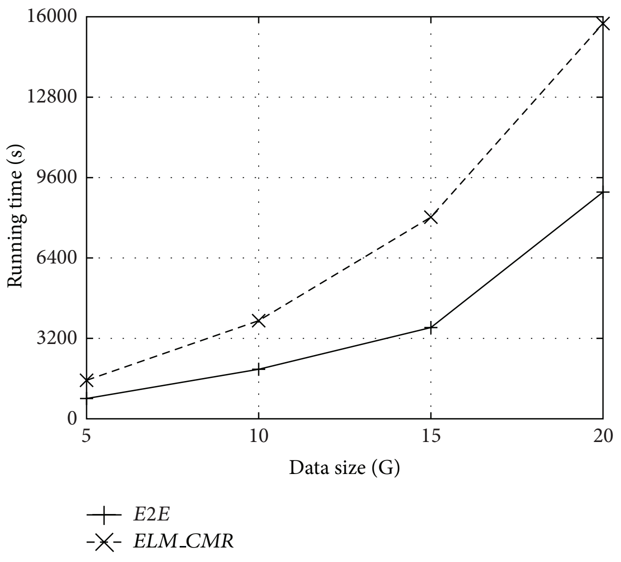

Figure 8: Performance of join query.

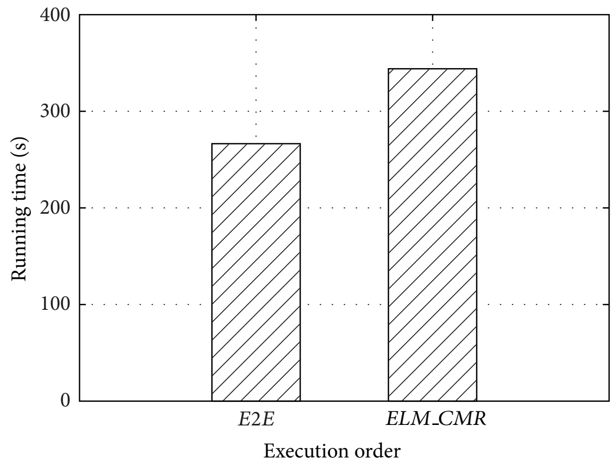

Figure 9: The same query type.

Figure 8 shows the performance of join queries in different data size of small-big tables, with the same data size of the small table 2 G, and the different data sizes of big table are shown in Table 3. The performance of *E2E* is much better than *ELM_CMR*. In ComMapReduce, the join attributes of the small table can be set as the *shared information* to filter the unpromising intermediate results.

Third, we evaluate the performance of the same kind of multiple queries implemented in different *execution orders*. Figure 9 illustrates the performance of two *execution order*s of four skyline queries in different data sizes, 0.2 G, 0.4 G, 0.8 G, and 1.6 G in anticorrelated distribution. The running time of our optimized *execution order* is shorter than the running time of the original order. Our *E2E* can identify the proper classifications and *execution order* of the queries to enhance the performance. In the original order in random, the queries do not have the optimal classifications and implementations with *ELM_CMR* model.

Figure 10 shows the performance of the multiple queries about different types under different *execution orders*, respectively, top-*k* (2 G), top-*k* (8 G), *k*NN (2 G), and *k*NN (8 G) in uniform distribution. We can see that the performance under our *E2E* model is much optimal than the *ELM_CMR* model. The optimized *execution order* of *E2E* model does not only consider the classification results of *E2E* model, but also consider the characteristics of the queries deeply.

6. Conclusions

In this paper, we propose an efficient query processing optimization model based on ELM, *E2E* model. Our *E2E* can effectively analyze the query processing applications and then generates the most optimized executions of query processing applications. After analyzing the problems of the former *ELM_CMR* model, we use two stages model to classify the query processing applications in ComMapReduce

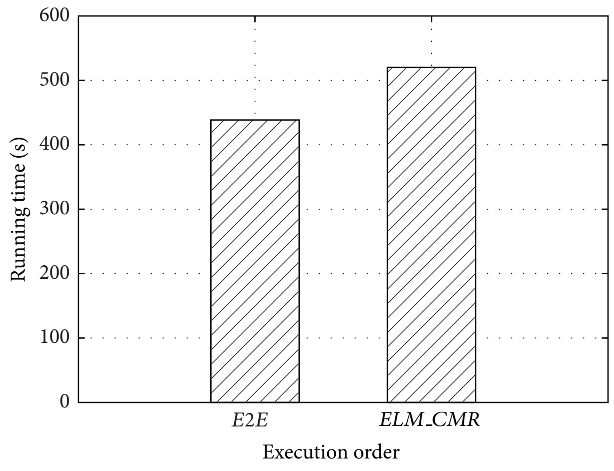

FIGURE 10: The different query types.

framework. Then, we propose an efficient training approach to train our model. We also give the query executions based on *E2E* model in two situations, Just-in-Time execution and Queue execution. The experiments demonstrate that the *E2E* model is efficient and the query processing applications can reach an optimal performance.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This research is supported by National Natural Science Foundation of China (nos. 60873068, 61472169, 61100022, and 61472069) and Talent Projects of the Educational Department of Liaoning Province (no. LR201017).

References

[1] A. Basu, S. Shuo, H. Zhou, M. Hiot Lim, and G.-B. Huang, "Silicon spiking neurons for hardware implementation of extreme learning machines," *Neurocomputing*, vol. 102, pp. 125–134, 2013.

[2] J. W. Cao, T. Chen, and J. Fan, "Fast online learning algorithm for landmark recognition based on BoW framework," in *Proceedings of the 9th IEEE Conference on Industrial Electronics and Applications*, 2014.

[3] J. Cao, Z. Lin, G.-B. Huang, and N. Liu, "Voting based extreme learning machine," *Information Sciences*, vol. 185, pp. 66–77, 2012.

[4] J. Cao and L. Xiong, "Protein sequence classification with improved extreme learning machine algorithms," *BioMed Research International*, vol. 2014, Article ID 103054, 12 pages, 2014.

[5] B. P. Chacko, V. R. V. Krishnan, G. Raju, and P. B. Anto, "Handwritten character recognition using wavelet energy and extreme learning machine," *International Journal of Machine Learning and Cybernetics*, vol. 3, no. 2, pp. 149–161, 2012.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng, "MassJoin: a mapreduce-based method for scalable string similarity joins," in *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE '14)*, pp. 340–351, Chicago, Ill, USA, April 2014.

[8] L. Ding, G. Wang, J. Xin, X. Wang, S. Huang, and R. Zhang, "ComMapReduce: an improvement of MapReduce with lightweight communication mechanisms," *Data and Knowledge Engineering*, vol. 88, pp. 224–247, 2013.

[9] L. Ding, J. Xin, and G. Wang, "An efficient query processing optimization based on ELM in the cloud," *Neural Computing and Applications*, 2014.

[10] L. Ding, J. Xin, G. Wang, and S. Huang, "ComMapReduce: an improvement of mapreduce with lightweight communication mechanisms," in *Database Systems for Advanced Applications*, pp. 150–168, Springer, 2012.

[11] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in MapReduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, 2014.

[12] M. Ester and H. Kriegel, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pp. 226–231, 1996.

[13] S. Fries, B. Boden, G. Stepien, and T. Seidl, "PHiDJ: parallel similarity self-join for high-dimensional vector data with MapReduce," in *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE '14)*, pp. 796–807, Chicago, Ill, USA, April 2014.

[14] J. Gao, J. Zhou, C. Zhou, and J. X. Yu, "GLog: a high level graph analysis system using MapReduce," in *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE '14)*, pp. 544–555, IEEE, April 2014.

[15] Q. He, T. Shang, F. Zhuang, and Z. Shi, "Parallel extreme learning machine for regression based on MapReduce," *Neurocomputing*, vol. 102, pp. 52–58, 2013.

[16] G. B. Huang, "An insight into extreme learning machines: random neurons, random features and kernels," *Cognitive Computation*, vol. 6, no. 3, pp. 376–390, 2014.

[17] G.-B. Huang, D. H. Wang, and Y. Lan, "Extreme learning machines: a survey," *International Journal of Machine Learning and Cybernetics*, vol. 2, no. 2, pp. 107–122, 2011.

[18] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *Proceedings of the IEEE International Joint Conference on Neural Networks*, vol. 2, pp. 985–990, July 2004.

[19] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1–3, pp. 489–501, 2006.

[20] W. Jun, W. Shitong, and F.-L. Chung, "Positive and negative fuzzy rule system, extreme learning machine and image classification," *International Journal of Machine Learning and Cybernetics*, vol. 2, no. 4, pp. 261–271, 2011.

[21] Y. Lan, Z. Hu, Y. C. Soh, and G.-B. Huang, "An extreme learning machine approach for speaker recognition," *Neural Computing and Applications*, vol. 22, no. 3-4, pp. 417–425, 2013.

[22] A. Lendasse, Q. He, Y. Miche, and G.-B. Huang, "Advances in extreme learning machines (ELM2012)," *Neurocomputing*, vol. 128, pp. 1–3, 2014.

[23] W. Lin, X. Xiao, and G. Ghinita, "Large-scale frequent subgraph mining in MapReduce," in *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE '14)*, pp. 844–855, April 2014.

[24] K. Mullesgaard, J. L. Pederseny, H. Lu, and Y. Zhou, "Efficient skyline computation in MapReduce," in *Proceedings of the 17th International Conference on Extending Database Technology (EDBT '14)*, pp. 37–48, 2014.

[25] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 8, pp. 1226–1238, 2005.

[26] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Liu, "Scalable big graph processing in MapReduce," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, pp. 827–838, 2014.

[27] W. Xi-Zhao, S. Qing-Yan, M. Qing, and Z. Jun-Hai, "Architecture selection for networks trained with extreme learning machine using localized generalization error model," *Neurocomputing*, vol. 102, pp. 3–9, 2013.

[28] J.-H. Zhai, H.-Y. Xu, and X.-Z. Wang, "Dynamic ensemble extreme learning machine based on sample entropy," *Soft Computing*, vol. 16, no. 9, pp. 1493–1502, 2012.

[29] R. Zhang, Y. Lan, G.-B. Huang, Z.-B. Xu, and Y. C. Soh, "Dynamic extreme learning machine and its approximation capability," *IEEE Transactions on Cybernetics*, vol. 43, no. 6, pp. 2054–2065, 2013.

[30] X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using MapReduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1184–1195, 2012.

[31] W. Zong and G.-B. Huang, "Learning to rank with extreme learning machine," *Neural Processing Letters*, vol. 39, no. 2, pp. 155–166, 2014.