Neural Comput & Applic (2016) 27:35–44
DOI 10.1007/s00521-013-1543-3

An efficient query processing optimization based on ELM in the cloud

Linlin Ding · Junchang Xin · Guoren Wang

Received: 25 October 2013 / Accepted: 24 December 2013 / Published online: 18 January 2014

Abstract Nowadays, MapReduce has emerged as a facto programming model for parallel processing of large-scale datasets with a commodity cluster of machines. MapReduce and its variants have been widely researched in the industry and academic communities. ComMapReduce further extends MapReduce by adding lightweight communication mechanisms and also enhances the efficiency of query processing applications. However, we find that the performance of query processing applications changes a lot in different communication strategies of ComMapReduce framework. It is necessary to identify the most optimal communication strategies of the query processing applications. Extreme learning machine (ELM) can exactly provide classification performance with an extremely fast training speed. Therefore, in this paper, first, we propose an efficient query processing optimization approach based on ELM in ComMapReduce framework, named *ELM_CMR*. Then, we design two implementations of our *ELM_CMR* approach to further optimize the performance of query processing applications. Finally, extensive experiments are conducted to verify the effectiveness and efficiency of our proposed *ELM_CMR*.

L. Ding (✉) · J. Xin · G. Wang
College of Information Science and Engineering, Northeastern University, Shenyang, China
e-mail: linlin.neu@gmail.com

J. Xin
e-mail: xinjunchang@ise.neu.edu.cn

G. Wang
e-mail: wanggr@mail.neu.edu.cn

L. Ding
College of Information Science and Technology, Liaoning University, Shenyang, China

Keywords Extreme learning machine · MapReduce · Query processing applications · ComMapReduce

1 Introduction

Nowadays, MapReduce [1] and its public available implementation, Hadoop,[1] have emerged as the de facto standard programming framework for performing large scalable and parallel tasks with a community cluster of machines. This programming framework is scalable, fault tolerant, cost-effective and easy to use. The successes of MapReduce and its variants have resulted in their deployments in the industry [2–6] and academic communities [7–15]. As one of the improvements of MapReduce, ComMapRedcue [16, 17] adds simple lightweight communication mechanisms to generate the certain *shared information* and then enhances the performance of query processing applications with large-scale datasets in the cloud. In addition, three basic and two optimization communication strategies of ComMapReduce framework are proposed to illustrate how to communicate and obtain the *shared information* of different applications.

ComMapReduce is a successful improvement of the original MapReduce framework. Numerous query processing applications can largely enhance the performance with the communication strategies of ComMapReduce. However, through the abundant experiments and further analysis of the execution course of ComMapReduce framework, the characteristics of ComMapReduce are further summarized as follows. First, not all the query processing applications are appropriate for ComMapReduce framework. In other words, the performance of

[1] http://hadoop.apache.org/.

certain queries in MapReduce framework is optimal to the performance in ComMapReduce framework. Second, different communication strategies of ComMapReduce can substantially affect the performance of query processing applications. In ComMapReduce framework, the performance of one query processing application is different with the different communication strategies of ComMapReduce framework. Third, in MapReduce programming, the configuration parameters can fully specify how the job should execute, such as the number of Map and Reduce tasks, the size of block, whether adopting Combiner, and so on. ComMapReduce is the improvement of MapReduce and inherits the basic programming framework of MapReduce, so these configuration parameters also have a sharp impact on the performance of ComMapReduce jobs.

Therefore, for a query programm, whether processing in ComMapReduce and adopting which communication strategies of ComMapReduce framework are urgent problems to be resolved. If we can adopt efficient classification algorithm to optimize the implementations of query processing applications, the whole ComMapReduce framework can reach an excellent performance. Extreme learning machine (ELM) [18] proposed by Huang et.al is exactly developed for generalized single hidden-layer feedforward networks (SLPNs) with a wide variety of hidden nodes. ELM can provide classification performance at an extremely fast training speed. Therefore, in this paper, we propose an efficient query processing optimization approach based on ELM in ComMapReduce framework, named *ELM_CMR* approach. Our *ELM_CMR* approach can effectively analyze the query processing applications and obtain the most optimal solution. First, after analyzing the overview of our *ELM_CMR* approach, we choose the adaptive *feature parameters* to train the ELM model for query processing optimization. Then, we propose two implementations of our *ELM_CMR* approach, one query implementation and multiple queries implementation. The contributions of this paper can be summarized as follows.

- We propose an efficient query processing optimization approach in ComMapReduce framework based on ELM and select the adaptive *feature parameters* to generate our ELM Classifier.
- Two implementations of *ELM_CMR* approach, one query and multiple queries, are proposed to optimize the performance of query processing applications.
- Our experimental studies using synthetic data show the effectiveness and efficiency of our *ELM_CMR* approach.

The remainder of this paper is organized as follows. Section 2 briefly introduces the ELM and ComMapReduce framework. Our *ELM_CMR* approach and two implementations for query processing applications are proposed in Sect. 3. The experimental results to show the performance of *ELM_CMR* are reported in Sect. 4. Finally, we conclude this paper in Sect. 5.

2 Background

In this section, we describe the background for our work, which includes a brief overview of the traditional ELM and a detailed description of ComMapReduce framework.

2.1 Review of ELM

Recently, with the characteristics of excellent generalization performance, rapid training speed and little human intervene, extreme learning machine (ELM) [18] and its variants [19–34] have attracted increasing attention from more and more researchers. ELM is originally developed for single hidden-layer feedforward neural networks (SLFNs) and is then extended to the "generalized" SLFNs. ELM first randomly assigns the input weights and hidden-layer biases and then analytically determines the output weights of SLFNs. Contrast to the other conventional learning algorithms, ELM reaches the optimal generalization performance with a sharply fast learning speed. ELM is less sensitive to the user-defined parameters, so that it can be deployed faster and more conveniently than the others.

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 [34].

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

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

The equation above can be expressed compactly as follows:

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

where $\mathbf{H}(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_L, b_1, b_2, \ldots, b_L, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_L)$

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

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

$\mathbf{H}$ is named 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:

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

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 process for ELM training is given in Algorithm 1. Only after properly setting the related parameters, ELM can start the training process. Step one is to generate L pairs of hidden node parameters $(\mathbf{w}_i, b_i)$ (Lines 1–3). Step two actually calculates the hidden-layer output matrix $\mathbf{H}$ by using Eq. (4) (Line 4). Step three 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 Eq. (7).

Algorithm 1 ELM Training

```
1: for i = 1 to L do
2:     Randomly generate hidden node parameters (w_i, b_i)
3: end for
4: Calculate the hidden layer output matrix H
5: Calculate the output weight vector β = H†T
```

2.2 ComMapReduce framework

MapReduce is a parallel programming framework processing of the large-scale datasets on clusters with numerous commodity machines. An overview of the execution course of a MapReduce application is shown in Fig. 1. When a MapReduce job is processed in the cluster, as the brain of the whole framework, the Master node schedules a number of parallel tasks to run on the Slave

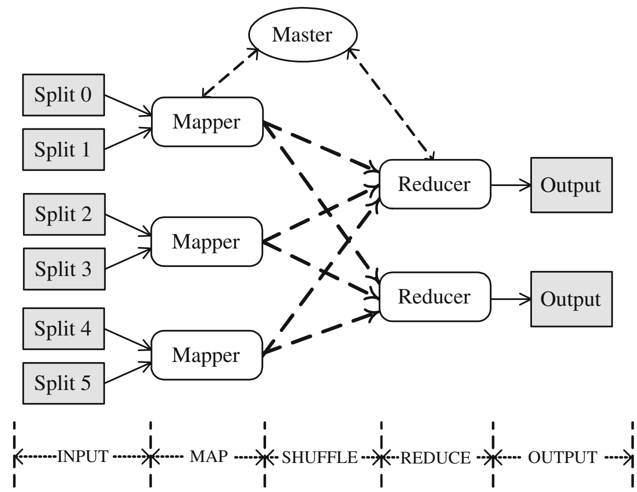

Fig. 1 Framework of MapReduce

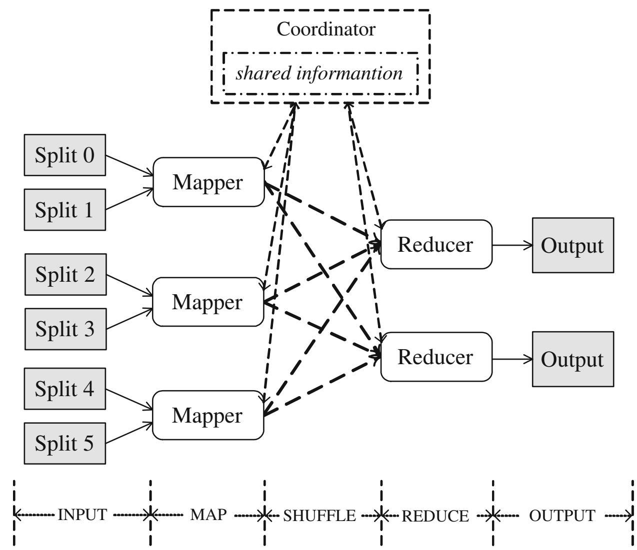

Fig. 2 Framework of ComMapReduce

nodes. First, in Map phase, each Map task independently operates a non-overlapping split of the input file and calls the user-defined Map function to emit its intermediate *<key,value>* tuples in parallel. Second, once a Map task completes, each Reduce task fetches all the particular intermediate data remotely. This course is called the shuffle phase in MapReduce.

In the actual applications of MapReduce, when the final results are much smaller than the original data, such as a top-k query, there are a large number of unpromising intermediate data to be transferred in the shuffle phase, leading to the waste of disk access, CPU resources and network bandwidth. ComMapReduce [16, 17] is an optimized MapReduce framework with lightweight communication mechanisms. In ComMapReduce framework as shown in Fig. 2, a new node, named the Coordinator node, is added to store and generate the certain *shared information* of different applications. The Coordinator node can communicate with the Mappers and Reducers with simple lightweight communication mechanisms.

In Map phase, after each Mapper completes, it computes its local *shared information* according to the features of the application and sends it to the Coordinator node. After that, the Coordinator node gains the most optimal one as the global *shared information* from the local *shared information* it receives according to the features of the application too. Simultaneously, the Mappers receive the global *shared information* to filter out their unpromising intermediate data to be transferred in the shuffle phase. The amounts of the intermediate data can be decreased, so as to shorten the latency time and improve the utility of bandwidth and CPU resources. Three basic communication strategies are designed to illustrate how to communicate with the Coordinator node to obtain the global *shared information*, respectively, lazy communication strategy (LCS), eager communication strategy (ECS) and hybrid communication strategy (HCS). Two optimization communication strategies are proposed to enlarge the ways of receiving and generating the *shared information*, respectively, prepositive optimization strategy (PreOS) and postpositive optimization strategy (PostOS).

In summary, without affecting the existing characteristics of the original MapReduce framework, ComMapReduce is an efficient parallel programming framework with global *shared information* to filter out the unpromising data. It can not only process the one pass massive data applications, but also implements the iterative massive data analysis applications.

3 ELM-based query processing optimization

In this section, the overview of our *ELM_CMR* approach is introduced first in Sect. 3.1, and then, we propose an efficient feature subset selection method to train the ELM model in Sect. 3.2. In Sect. 3.3, two implementations of *ELM_CMR* are presented, one query and multiple queries.

3.1 Overview of ELM_CMR approach

There are four main components of Our *ELM_CMR* approach that can optimize the query processing programs effectively in ComMapReduce framework. Figure 3 shows the flow of information through the approach. The four main components are, respectively, the *Feature Selector*, the *ELM Classifier*, the *Query Optimizer* and the *Execution Fabric*. The *Feature Selector* examines the training data and selects the features that can wholly affect the query performance. There are many features that can be used to describe a ComMapReduce job, but not all of them can drastically affect the performance. Therefore, it is important to select the main features. How to select the main features is to be illustrated in Sect. 3.2 in detail. After selecting the features of training data, the *Feature Selector* sends the extracted training data to the *ELM Classifier*. The *ELM Classifier* 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*. How to choose the *execution order* will be presented in Sect. 3.3. After gaining the *execution order*, the *Query Optimizer* sends it to the *Execution Fabric*. The *Execution Fabric* implements the program in ComMapReduce framework.

For the query processing applications, we can identify the most optimal communication strategies of ComMapReduce framework by using our *ELM_CMR* approach. With the optimal communication strategy, the processing cost of the shuffle phase can be reduced drastically. Although the computation of *ELM_CMR* approach adds the whole processing cost, the computation cost of *ELM_CMR* approach is relatively cost-effective and time-efficient contrast to the processing course with the other communication strategies of ComMapReduce framework. So, we can realize the optimized query processing implementation so as to further enhance the performance of ComMapReduce framework by *ELM_CMR* approach.

3.2 Feature subset selection

A query processing program q of MapReduce or ComMapReduce is regarded as job $j = <q, d, r, c>$, where d is the original input data; r is the cluster resource; and c is the configuration parameter setting of q. In this situation, because d, r and c can have different configurations, a number of selections can be made to fully specify how the job should execute. For example, d contains the data size and distribution of the input data; r contains the number of Slave nodes and the network configuration. Moreover, c in $j = <q, d, r, c>$ comes from a high dimensional space of configuration parameters settings that contain (but are not limited to):

- The number of Map and Reduce tasks.
- The size of the memory buffer to use while sorting mapout.
- Whether adopting Combiner function to aggregate map outputs.

We call these parameters the *feature parameters* of a query program q. Figure 4 shows the impact of execution time of a skyline [35] query in ComMapReduce framework

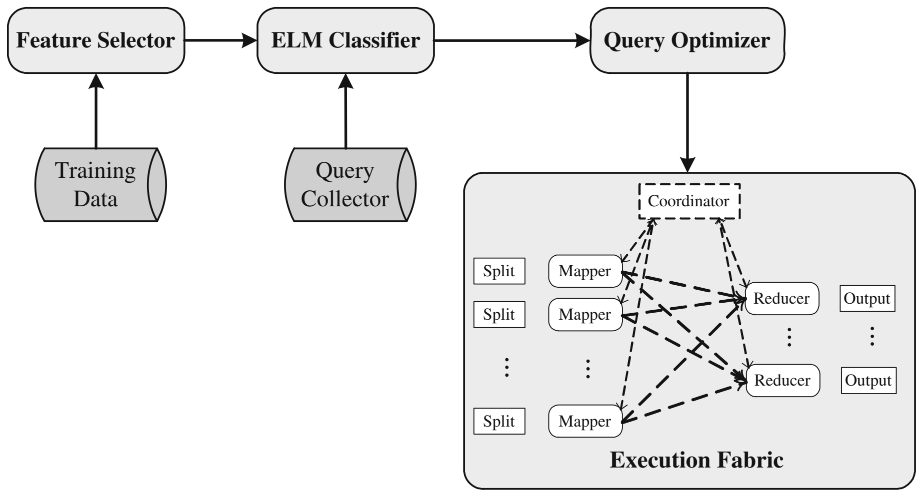

Fig. 3 Architecture of *ELM_CMR* approach

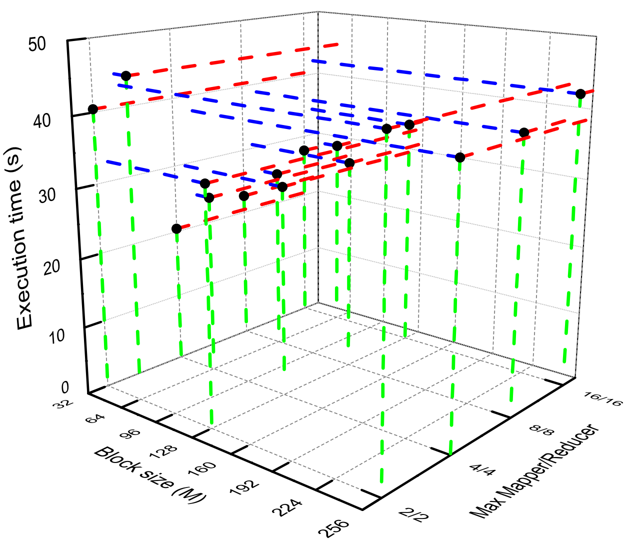

Fig. 4 Execution time in different *feature parameters*

by changing two *feature parameter*s. We can see that the execution time changes a lot with different *feature parameters*. Therefore, it is important to specify the proper settings of *feature parameters* for the submitted job j. Due to the high dimensional property of c in j, we should identify the configuration parameters that can largely affect the performance of q. For any parameter whose value is not specified explicitly during job submission, either is shipped with the system or specified by the system administrator. Finding the proper configuration parameter setting is a time-consuming course, which requires extensive knowledge of the whole framework. In this paper, we adopt the execution time as the performance metric, but is not limited this metric.

The first problem is to obtain the proper configuration parameters of program q by dynamically generating the concise statistical summaries of MapReduce job execution. In this paper, we use the job profiles to obtain the configuration parameter settings. The job profile is a vector where each field captures some unique features during the job execution. We use task-level sampling to generate the appropriate job profiles while keeping the run-time overhead low. In order to collect a job profile for j, the profile can be gained by only selecting small samples of j's tasks. For example, for a job containing 50 Map tasks, it is only to run 5 tasks of them to generate the profile.

The second problem is to minimize the number of parameters in the near-optimal configuration parameter settings. All configuration parameters form a space of parameter settings S. There are so many parameters in S that the high dimensionality space of S affects the scalability of our approach. If the individual parameters in S can be grouped into clusters, S_i, the globally optimal setting in S can be computed from the optimal settings of the clusters S_i as shown in Algorithm 2. Step one divides the high dimensional space S into the lower dimensional subspaces S_i (Line 1). Step two considers an independent optimization problem in each smaller subspace (Lines 2–4). Step three combines the optimal parameter settings found in per subspace S_i (Line 5).

Algorithm 2 Optimal Parameters

```
1: Divide S into subspace S_i (i=1,2,...l);
2: for i = 1 to l do
3:     Generate the optimal parameters in S_i by mRMR;
4: end for
5: Compose the optimal parameters in each S_i;
```

Naturally, the parameters of program q can be divided into three clusters, parameters that predominantly affect Map task execution; parameters that predominantly affect Reduce task execution and the cluster parameters. For example, Hadoop's *io.sort.record.percent* parameter affects the storing record boundaries of the Map outputs,

Table 1 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(top-*k*, *k*NN), 1(skyline), 2(join)
Data distribution	char	Uniform
Number of slave nods	int	8

while *mapred.job.shuffle.merge.percent* only affects the shuffle phase in Reduce tasks. The *dfs.hearbeate.interval* determines the interval of sending the heartbeat information of the whole system, and so on. In this paper, we adopt the minimum-redundancy-maximum-relevance (mRMR) [36] feature selection to find the optimal parameters sharply affecting the performance in each cluster. The mRMR is a first-order incremental feature selection to select a compact set of superior features at very low cost. And then, we generate the globally optimal configuration parameter settings by combining the results of the each subspace.

The globally optimal configuration parameter settings, combining with the input data *d* and the cluster resource *r*, form the *feature parameters* of the *ELM Classifier*. Table 1 lists the *feature parameters* in our experiments along with their default values that can impact the performance of jobs, but not all the configuration parameters in the system. We can use the ELM algorithm to generate the ELM model. When a new query or multiple queries come, the ELM model can effectively classify them to identify whether adopting ComMapReduce framework and determine which communication strategies of ComMapReduce to be adopted. After obtaining the feature parameters of the *ELM_CMR* approach, the *ELM Classifier* can generate classification results of the query processing applications. Then, the implementations of our *ELM_CMR* are introduced in Sect. 3.3.

3.3 Implementations of ELM_CMR

After generating the *ELM Classifier*, the pending queries may be one query or multiple queries. In this section, we first propose the implementation of one query, and then the implementation of multiple queries.

3.3.1 Implementation of one query

When there is one query to be processed, the *Feature Selector* abstracts its *feature parameters* of this query, and then, the *ELM Classifier* generates its classification result. After obtaining the classification result, the *Query Optimizer* can make a decision of adopting which communication strategy of ComMapReduce framework is suitable. The *Execution Fabric* then implements the query processing application according to the result of the *Query Optimizer*.

The implementation of one query is shown in Algorithm 3. First, the *feature parameters* of query processing job *j* are extracted using the above feature selection method (Line 1). Second, after obtaining the *feature parameters* of job *j*, the *ELM Classifier* generates the classification of *j* (Line 2). Third, according to the classification result of *j*, the *Query Classifier* ensures how to implement the program and sends it to the *Execution Fabric*. The *Execution Fabric* uses the optimization result to implement the query program (Line 3).

Algorithm 3 One Query

```
1: Extract the feature parameters of j;
2: Generate the classification of j;
3: Implement j with its classification;
```

For example, for a top-*k* query, after abstracting its *feature parameters*, the *ELM Classifier* generates its classification and then identifies the communication strategy of this top-k query, such as ECS. After that, the top-k query is implemented with ECS in ComMapReduce framework.

3.3.2 Implementation of multiple queries

When there are multiple queries to be processed, the *Query Optimizer* can design an *execution order* of the queries without considering the situation of concurrently executing queries. Under the *execution order*, the performance of the multiple queries can reach the most optimal status. So the *execution order* is important to enhance the performance of our *ELM_CMR* approach.

With the same method as one query, the multiple queries can be classified by *ELM Classifier* and gain its best communication strategy of each program. After that, during the course of obtaining the job profile, a Task Scheduler Simulator is used to simulate the scheduling and execution of Map and Reduce tasks of each q. The implementation of the Task Scheduler Simulator is a lightweight discrete event simulation, only requiring a small task of job j. The output from the simulator is a complete description of the execution of job j in the cluster, such as the estimated job completion time, the amount of local I/O or even a visualization of the task execution time. Therefore, according to the classification results, 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), we suppose that the shorter the execution time of a query is, the better priority order the query is. We can generate an *execution order* (O_s) of the multiple queries with the ascending order of the simulated execution time. According to O_s, the multiple queries can be implemented. Algorithm 4 illustrates the complete implementation course of the multiple queries. First, we can obtain the classification and simulate its execution time of each query (Lines 1–4). Then, the *execution order* is generated by Shortest Job First (SJF) principle (Line 5).

Algorithm 4 Multiple Queries

```
1: for each query q do
2:     Generate the classification of q_i;
3:     Simulate the execution time of q_i;
4: end for
5: Calculate the optimized execution order;
```

Figure 5 shows an example of the multiple queries implementation course. Suppose that there are eight queries to be proposed by our *ELM_CMR* approach. We want to confirm the final *execution order*. After classified by the *ELM Classifier*, these queries obtain their classification results as shown in Fig. 5. According to the simulated execution time of the queries and SJF principle, we can get an *execution order* O_s, q_2, q_5, q_3, q_1, q_6, q_4, q_7, q_8.

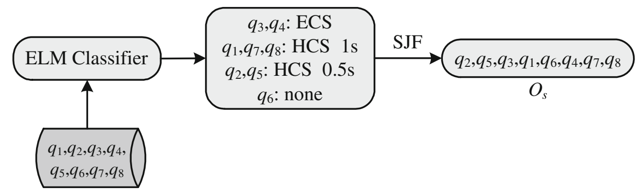

Fig. 5 Implementation of multiple queries

4 Performance evaluation

In this section, the performance of our *ELM_CMR* approach is evaluated in detail with various experimental settings. We first describe the platform used in our experiments in Sect. 4.1. Then, we present and discuss the experimental results in Sect. 4.2.

4.1 Experimental platform

Our experimental platform is a cluster of 9 commodity PCs in a high-speed Gigabit network, with one PC as the Master node and the Coordinator node, the others as the Slave nodes. Each PC has an Intel Quad Core 2.66GHZ CPU, 4GB 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. The data in our experiments are synthetic data. Table 1 summarizes the parameters used in the experiments including the default values. The *ELM Classifier* divides the communication strategies into 7 classifications, 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 0.5s. We evaluate the performance of *ELM_CMR* in different implementations for one query and multiple queries.

4.2 Experimental results

First, four typical query processing applications are adopted to evaluate the implementations of one query, top-k, kNN, skyline and join. We use the *ELM_CMR* to identify the most optimal communication strategy of each query and then implement the query under different communication strategies to test their performance. Figure 6 shows the performance of a top-k query (k = 1,000). We can see that the performance of this top-k query is different in different communication strategies, and PreOS is the best one. This is same as the classification result of our *ELM_CMR*. When k is much smaller than the original data, the global *shared information* can reach the most optimal one quickly, so the Mappers can retrieve the *shared information* in the initial phase to filter out the unpromising data.

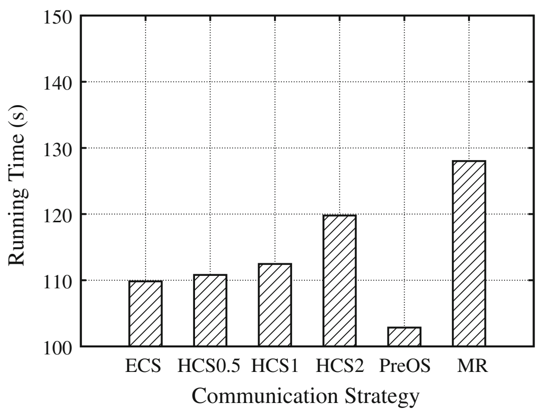

Fig. 6 Performance of top-k query

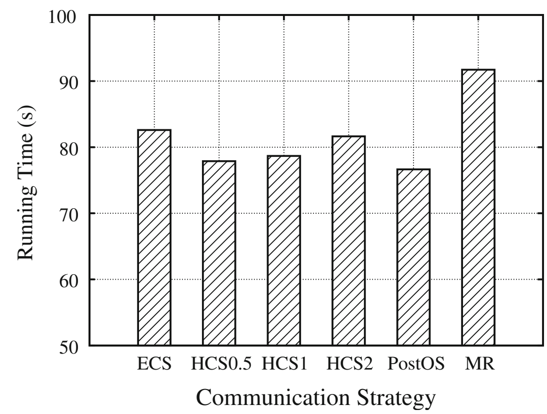

Fig. 8 Performance of skyline query

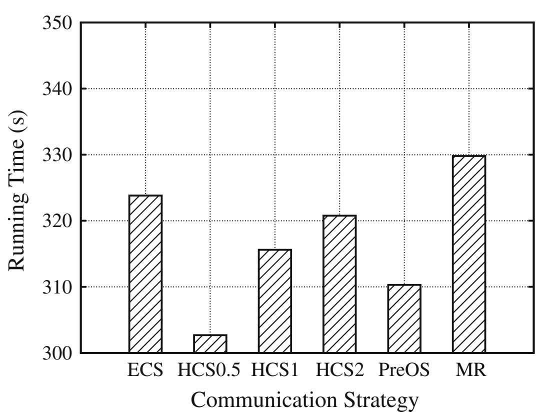

Fig. 7 Performance of kNN query

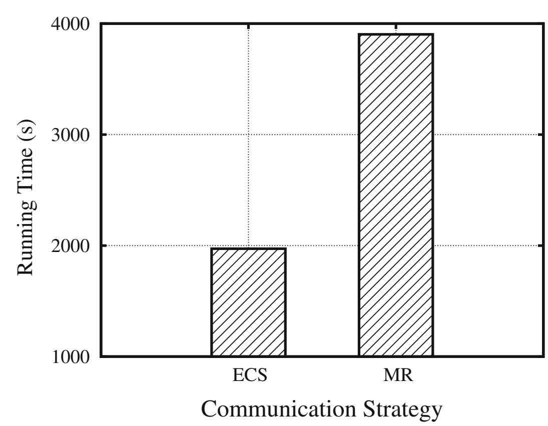

Fig. 9 Performance of join query

Figure 7 shows the performance of a kNN query. The classification of this kNN query is HCS-0.5 and the running time of HCS-0.5 is the shortest in the experiment. HCS can obtain the *shard information* in a preassigned time interval. It does not only have to wait for all Mapper completed wasting extra time, but also receives the *shard information* after each Mapper completes.

Figure 8 shows the performance of a skyline query in anti-correlated distribution. We can see that the performance of different communication strategies is not obviously different, but PostOS is a little better. The classification result of this skyline query is just PostOS. The reason is that the original data are skewed to the final results in anti-correlated distribution. The percentage of filtering is low, so the performance difference is not obvious. In this situation, although *ELM_CMR* can obtain the classification, the query can also choose the other communication strategies.

Figure 9 shows the performance of a join query of small-big tables with its classification ECS. The performance of ECS is much better than MR. By the communication strategy, ECS, the join attributes of the small table can be set as the *shared information* to filter out the unpromising intermediate results.

Second, we evaluate the performance in different *execution order*s of multiple queries. Figure 10 illustrates the performance of four top-k queries in the group. The running time of our optimized *execution order* is shorter than the running time of the original order. Our *ELM_CMR* can identify the proper classifications of the queries to enhance the performance. In the original order, the queries do not have the most optimal classification and implement with random communication strategies.

Figure 11 shows the performance of the multiple queries about different types under different *execution order*s, respectively, top-k, kNN, skyline and join. There are four queries in the multiple queries group. We can see that the running time under our optimized *execution order* is much optimal than the original order. In our *ELM_CMR* approach, according to the classification results of the *ELM Classifier*, the *Query Optimizer* can generate the optimized *execution order* of the queries. Under the optimized *execution order*, the performance is much better than the original one.

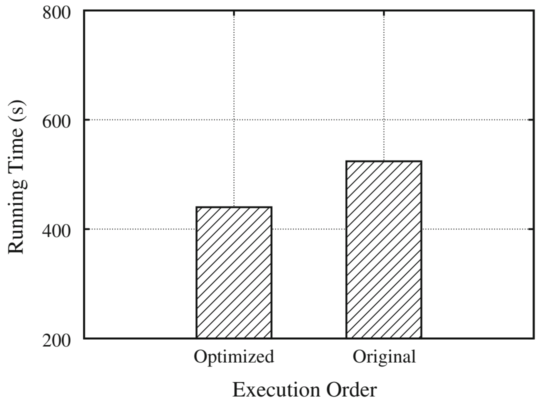

Fig. 10 The same query type

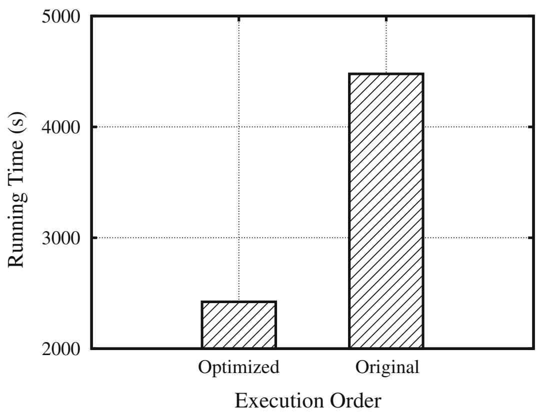

Fig. 11 The different query types

5 Conclusions

In this paper, we propose an efficient query processing optimization based on ELM in ComMapReduce framework. Our *ELM_CMR* approach can effectively analyze the query processing applications, and then generate the most optimized implementations of query processing applications. After analyzing the implementation of ComMapReduce framework, we train the ELM model to classify the query processing applications in ComMapReduce framework. Then, we propose two implementations of our *ELM_CMR*, one query and the multiple queries. The experiments demonstrate that our *ELM_CMR* approach is efficient and the query processing applications can reach an optimal performance.

Acknowledgments This research was partially supported by the National Natural Science Foundation of China under Grant Nos. 60933001, 61025007, and 61100022; the National Basic Research Program of China under Grant No. 2011CB302200-G; the 863 Program under Grant No. 2012AA011004, and the Fundamental Research Funds for the Central Universities under Grant No. N110404009.

References

1. Dean Jeffrey, Ghemawat Sanjay (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
2. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data, pp 1099–1110
3. Thusoo A, Sarma Joydeep S, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive: a warehousing solution over a map-reduce framework. Proceed VLDB Endow 2(2):1626–1629
4. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Zhang N, Antony S, Liu H, Murthy R (2010) Hive-a petabyte scale data warehouse using hadoop. In: Data Engineering (ICDE), pp 996–1005
5. Carstoiu D, Lepadatu E, Gaspar M (2010) Hbase-non sql database, performances evaluation. IJACT-AICIT 2(5):42–52
6. Abouzeid A, Bajda-Pawlikowski K, Abadi D, Silberschatz A, Rasin A (2009) HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proceed VLDB Endow 2(1):922–933
7. Yang H-C, Dasdan A, Hsiao R-L, Parker DS (2007) Map-reduce-merge: simplified relational data processing on large clusters. In: Proceedings of the 2007 ACM SIGMOD international conference on management of data, pp 1029–1040
8. Jiang D, Tung Anthony KH, Chen G (2011) Map-join-reduce: toward scalable and efficient data analysis on large clusters. Knowl Data Eng 23(9):1299–1311
9. Blanas S, Patel JM, Ercegovac V, Rao J, Shekita EJ, Tian Y (2010) A comparison of join algorithms for log processing in mapreduce. In: Proceedings of the 2010 ACM SIGMOD international conference on management of data, pp 975–986
10. Vernica R, Carey MJ, Li C (2010) Efficient parallel set-similarity joins using MapReduce. In: Proceedings of the 2010 international conference on management of data, pp 495–506
11. Afrati FN, Borkar V, Carey M, Polyzotis N, Ullman JD (2011) Map-reduce extensions and recursive queries. In: Proceedings of the 14th international conference on extending database technology, pp 1–8
12. Dittrich J, Quiané-Ruiz J-A, Jindal A, Kargin Y, Setty V, Schad J (2010) Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). Proceed VLDB Endow 3(1–2): 518–529
13. Jahani E, Cafarella MJ, Ré C (2011) Automatic optimization for MapReduce programs 4(6):385–396
14. Zhang X, Chen L, Wang M (2012) Efficient multi-way theta-join processing using MapReduce. Proceed VLDB Endow 5(11): 1184–1195
15. Kim Y, Shim K (2012) Parallel top-k similarity join algorithms using MapReduce. In: Data Engineering (ICDE), pp 510–521
16. Ding L, Xin J, Wang G, Huang S (2012) ComMapReduce: an improvement of mapreduce with lightweight communication mechanisms, pp 150–168
17. Ding L, Wang G, Xin J, Wang X, Huang S, Zhang R (2013) ComMapReduce: an improvement of mapreduce with lightweight communication mechanisms. Data Knowl Eng
18. Huang G-B, Zhu Q-Y, Siew C-K (2004) Extreme learning machine: a new learning scheme of feedforward neural networks. In: Proceedings. 2004 IEEE international joint conference Neural Networks, 2004, pp 985–990
19. Chacko BP, Krishnan VRV, Raju G, Anto PB (2012) Handwritten character recognition using wavelet energy and extreme learning machine. Int J Mach Learn Cybern 3(2):149–161
20. Huang G-B, Wang Dian H, Lan Y (2011) Extreme learning machines: a survey. Int J Mach Learn Cybern 2(2):107–122

21. Rong H-J, Huang G-B, Sundararajan N, Saratchandran P (2009) Online sequential fuzzy extreme learning machine for function approximation and classification problems. Systems Man Cybern Part B Cybern 39(4):1067–1072
22. Sun Y, Yuan Y, Wang G (2011) An OS-ELM based distributed ensemble classification framework in p2p networks. Neurocomputing 74(16):2438–2443
23. Wang B, Wang G, Li J, Wang B (2012) Update strategy based on region classification using ELM for mobile object index. Soft Comput 16(9):1607–1615
24. Wang G, Zhao Y, Wang D (2008) A protein secondary structure prediction framework based on the extreme learning machine. Neurocomputing 72(1):262–268
25. Zhang R, Huang G-B, Sundararajan N, Saratchandran P (2007) Multicategory classification using an extreme learning machine for microarray gene expression cancer diagnosis. IEEE/ACM Trans Comput Biol Bioinformatics (TCBB) 4(3):485–495
26. Zhao X-G, Wang G, Bi X, Gong P, Zhao Y (2011) XML document classification based on ELM. Neurocomputing 74(16):2444–2451
27. Jun W, Shitong W, Chung F-l (2011) Positive and negative fuzzy rule system, extreme learning machine and image classification. Int J Mach Learn Cybern 2(4):261–271
28. Wang X-Z, Shao Q-Y, Qing M, Jun-Hai Z (2013) Architecture selection for networks trained with extreme learning machine using localized generalization error model. Neurocomputing 102:3–9
29. Huang G-B, Chen L (2008) Enhanced random search based incremental extreme learning machine. Neurocomputing 71(16):3460–3468
30. Zhai J-h, Xu H-y, Wang X-z (2012) Dynamic ensemble extreme learning machine based on sample entropy. Soft Comput 16(9):1493–1502
31. He Q, Shang T, Zhuang F, Shi Z (2013) Parallel extreme learning machine for regression based on MapReduce. Neurocomputing 102:52–58
32. Huang G-B, Chen L (2007) Convex incremental extreme learning machine. Neurocomputing 70(16):3056–3062
33. Huang G-B, Chen L, Siew C-K (2006) Universal approximation using incremental constructive feedforward networks with random hidden nodes, Neural Networks. IEEE Trans 17(4):879–892
34. Huang G-B, Zhu Q-Y, Siew C-K (2006) Extreme learning machine: theory and applications. Neurocomputing 70(1):489–501
35. Borzsony S, Kossmann D, Stocker K (2001) The skyline operator. In: Proceedings of the 17th international conference on Data Engineering, pp 421–430
36. Peng H, Long F, Ding C (2005) Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. Pattern Anal Mach Intell 27(8):1226–1238