Chapter 8

Analysis and Learning Frameworks for Large-Scale Data Mining

Kohsuke Yanai and Toshihiko Yanase

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/51713

1. Introduction

Recently, lots of companies and organizations try to analyze large amount of business data and leverage extracted knowledge to improve their operations. This chapter discusses techniques for processing large-scale data. In this chapter, we propose two computing frameworks for large-scale data mining:

1. Tree structured data analysis framework, and
2. Parallel machine learning framework.

The first framework is for analysis phase, in which we find out how to utilize business data through trial and error. The proposed framework stores tree-structured data using vertical partitioning technique, and uses Hadoop MapReduce for distributed computing. These methods enable to reduce disk I/O load, and to avoid computationally-intensive processing, such as grouping and combining of records.

The second framework is for model learning phase, in which we create predictive models using machine learning algorithms. The proposed framework is another implementation of MapReduce. The framework is designed to ease parallelization of machine learning algorithms and reduce calculation overheads for iterative procedures. The framework minimizes frequency of thread generation and termination, and keeps feature vectors in local memory and local disk during iteration.

We start with discussion on process of data utilization in enterprise and organization described in Figure 1. We suppose the data utilization process consists of the following phases.

1. Pre-processing phase
2. Analysis phase

3. Model learning phase
4. Model application phase

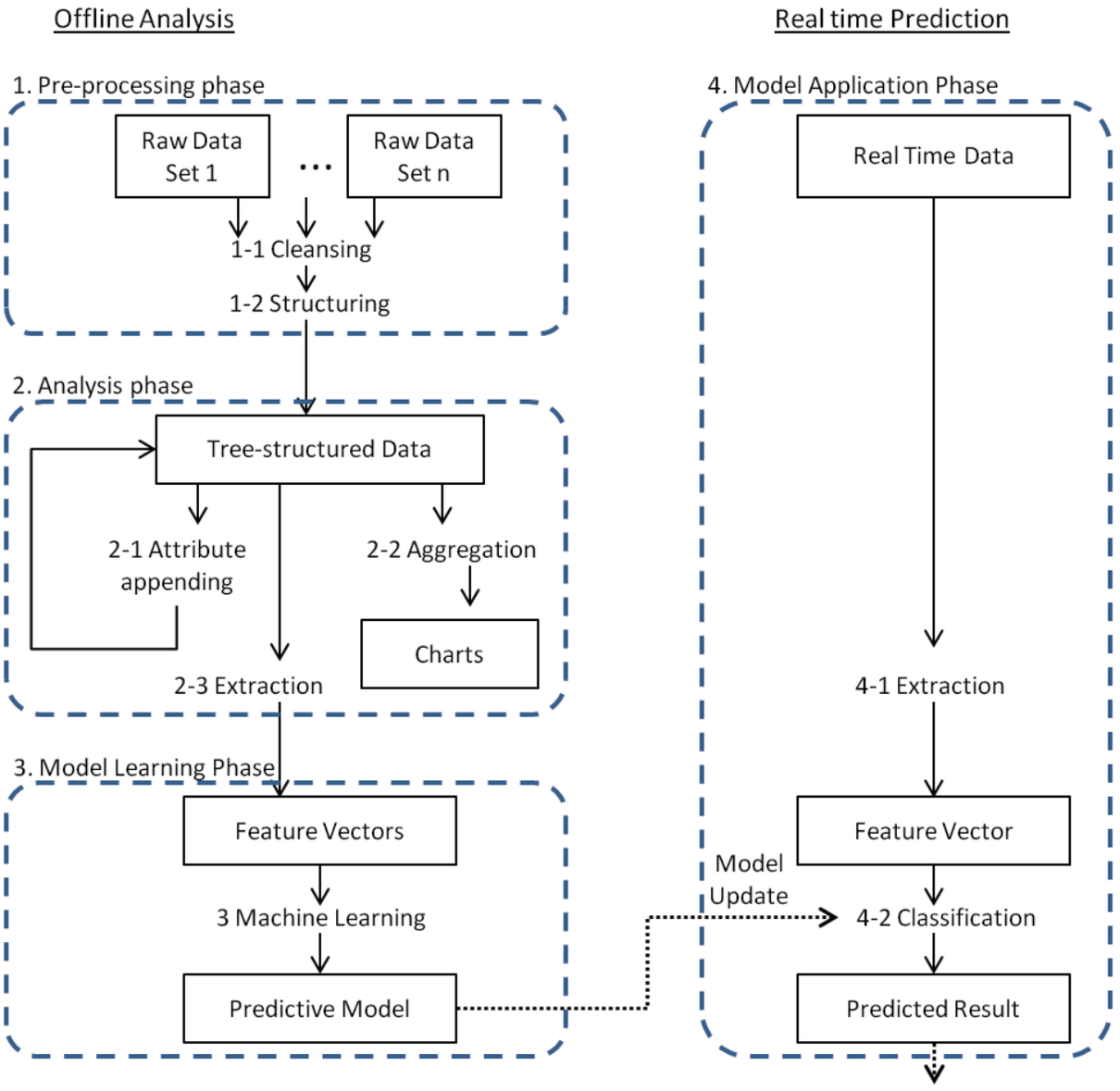

Figure 1. Process of data utilization.

1.1. Pre-processing phase

Pre-processing phase consists of 2 steps:

Step 1-1 Cleansing

Step 1-2 Structuring

Firstly Step 1-1 removes incorrect values and secondly Step 1-2 transforms table-format data into tree-structured data. This pre-processing phase combines raw data from multiple data

sources and creates tree-structured data in which records from multiple data sources are "joined".

Figure 2 illustrates an example of tree-structured server logs, in which the log data are grouped by each site at the top level. Site information consists of site ID (e.g. "site001") and a list of server information. Server information consists of server ID (e.g. "serv001"), average CPU usage (e.g. "ave-cpu:84.0%") and a list of detail records. Furthermore, a detail record consists of date (e.g. "02/05"), time (e.g. "10:20"), CPU usage (e.g. "cpu:92%") and memory usage (e.g. "mem:532MB").

```
[(site001
  [(serv001 ave-cpu:84.0%
    [(02/05 10:10 cpu:92% mem:532MB)
     (02/05 10:20 cpu:76% mem:235MB)])
   (serv002 ave-cpu:12.6%
    [(02/05 15:30 cpu:13% mem:121MB)
     (02/05 15:40 cpu:15% mem:142MB)
     (02/05 15:50 cpu:10% mem:140MB)])])
 (site021
  [(serv001 ave-cpu:50.0%
    [(02/05;11:40 cpu:88% mem:889MB)
     (02/05;11:50 cpu:12% mem:254MB)])])]
```

Figure 2. Example of tree-structured data.

If we store the data in table format, data grouping and data combining are repeatedly computed in analysis phase which comes after the pre-processing phase. Data grouping and data combining correspond "group-by" and "join" in SQL respectively. Note that the tree structure keeps the data be grouped and joined. In general when data size is large, the computation cost of data grouping and data combining becomes intensive. Therefore, we store data in tree structure format so that we avoid repetition of these computationally-intensive processing.

1.2. Analysis phase

Analysis phase finds out how to utilize the data through trial-and-error. In most situations the purpose of data analysis is not clear at an early stage of the data utilizatoin process. This is the reason why this early phase needs trial-and-error processes.

As described in Figure 1, the analysis phase consists of 3 independent steps:

Step 2-1 Attribute appending

Step 2-1 Aggregation

Step 2-1 Extraction

This phase iterates Step 2-1 and Step 2-2. The purpose of the iterative process is

- To obtain statistical information and trend,
- To decide what kind of predictive model should be generated, and

- To decide which attributes should be used to calculate feature vectors of the predictive model.

Step 2-1 appends new attributes to tree-structured data by combining existing attributes. We suppose the iteration of attribute appending increases data size by 5-20 times. On the other hand, Step 2-2 calculates statistics of attributes and generates charts that help to grasp characteristics of the data. The calculations of Step 2-2 include mean, variance, histogram, cross tabulation, and so on.

An instance of the iterative process consisting attribute appending and aggregation is the following.

1. Calculate frequencies of CPU usage (Step 2-2)
2. Append a new attribute "average memory usage for each server" (Step 2-1)
3. Calculate standard deviation of a new attribute "average memory usage" (Step 2-2)
4. Append a new attribute "difference of memory usage from its average" (Step 2-1)
5. ...

We usually append more than 10 new attributes into the raw data. Attribute appending increases value and visibility of data, and eases trial-and-error process for finding how to utilize the data.

After the iterative process of attribute appending and aggregation, Step 2-3 extracts feature vectors from tree-structured data, which are used in model learning phase.

1.3. Model learning phase

Model learning phase generates predictive models which are used in real-world operations of enterprises and organizations. The model learning phase uses machine learning techniques, such as SVM (support vector machine) [1] and K-Means clustering [2].

For instance, this phase generates a model that predicts when hardware troubles will happen in IT system. The input of the model is history of CPU usage and memory usage. The output is date and time.

1.4. Model application phase

Model application phase applies the predictive models obtained from the model learning phase into actual business operations. We emphasize the input data is "real time".

As described in Figure 1, the model application phase consists of 2 steps:

Step 4-1 Extraction

Step 4-2 Classification

Step 4-1 extracts a feature vector from real time data. Usually computation of this step is similar to that of Step 2-3. Step 4-2 attaches a predictive label to the input data by using predictive models. For example, this label represents date and time of hardware trouble. The label is used in business operations as an event.

2. Architecture

We propose architecture for large-scale data mining. Figure 3 illustrates our architecture.

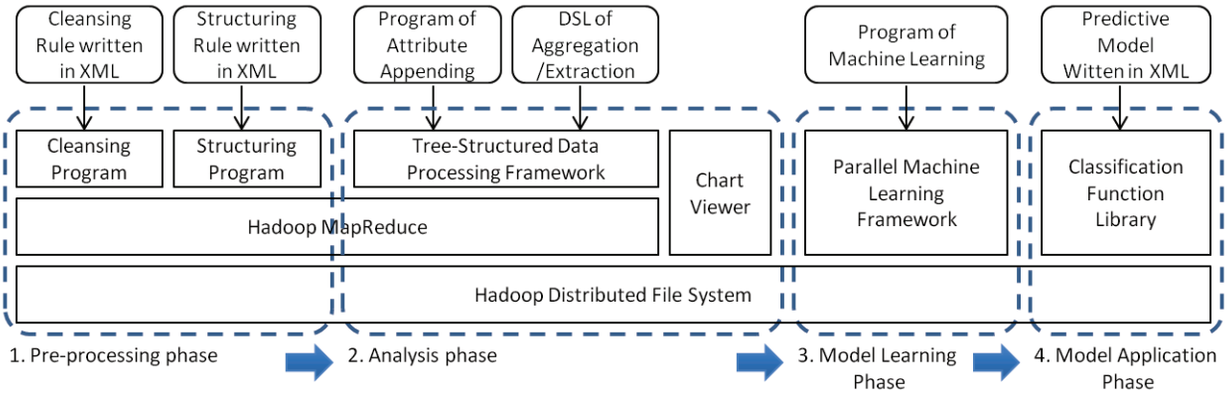

Figure 3. Architecture.

# Phase	Data	Computation	Approach
1 Pre-processing	Table format	I/O-intensive	Hadoop MR
2 Analysis	Tree-structured, large number of attributes	I/O-intensive	Hadoop MR + vertical partitioning + data store in tree-structured format
3 Model learning	Vectors	CPU-intensive	Iterative MR + Hadoop DFS
4 Model application	Steream	Real time	Event driven software

Table 1. Approach of each phase. MR: MapReduce, DFS: Distributed File System.

As discussed in Section 1, we suppose four phases: pre-processing, analysis, model learning and model application. In pre-processing phase, data is in table format and the computation is I/O-intensive. Hadoop MapReduce [3] is appropriate for the pre-processing from the view point of data format and I/O load reduction. Hadoop MapReduce is distributed computing platform based on MapReduce computation model [4, 5]. Hadoop MapReduce consists of three computation phases: Map, Combine and Reduce. Hadoop MapReduce parallelizes disk I/O by reading and writing data in parallel on Hadoop DFS (Distributed File System). Regarding details of Hadoop, refer to the literature [4, 5].

We develop cleansing program and structuring program which run on Hadoop MapReduce. The cleansing program and the structuring program are general-purpose, which means we can use the same programs for all cases. The cleansing program and the structuring program read cleansing rule and structuring rule respectively, and programs run by following the rules written by users as XML files.

In analysis phase, data is tree-structured and the computation is I/O-intensive. In addition, the number of attributes is large since this phase repeatedly appends new attributes. Therefore, key approach is also reduction of I/O load. We propose a method combining Hadoop MapReduce, vertical partitioning and data store in tree-structured format in Section 3. This phase also needs chart viewer that displays result of aggregation of Step 2-2.

On the other hand, I/O load in model learning phase is permissive. Because input of machine learning algorithms is feature vectors whose size is much smaller than that of raw data. The

computation in model learning phase is CPU-intensive since machine learning algorithms include iterative calculation for optimization. Section 4 proposes another MapReduce framework for parallel machine learning, in which iterative algorithms are easily parallelized.

In model application phase, data is stream and the computation should be performed in real time. Therefore, we develop event-driven software that runs at the timing of input data coming. The software includes a library of classification function. It reads a predictive model written in PMML [6] that is XML-based language for model description.

We summarize our approaches in Table 1. The rest of this paper focuses on frameworks for analysis phase and model learning phase. Because while new techniques are necessary for efficient computation in the two phases, system for pre-processing and model application is easily implemented by combining existing technologies.

3. Tree-structured data analysis framework

3.1. Mathod

This section proposes a computing framework that performs data analysis on a large amount of tree-structured data. As discussed in Section 1, an early stage of the data utilization process needs trial-and-error processes, in which we repeatedly append new attributes and calculate statistics of attributes. As a result of repetition of attribute appending, the number of attributes increases. Therefore, not only scalability to the number of records but also scalability to the number of attributes is important.

The key approaches of the proposed framework are:

1. To partition tree-structured data in column-wise and store the partitioned data in separated files corresponding to each attribute, and
2. To use Hadoop MapReduce framework for distributed computing.

The method (1) is referred to as "vertical partitioning." It is well known that vertical partitioning of **table format data** is efficient [7]. We propose vertical partitioning of tree-structured data. Figure 4 illustrates the vertical partitioning method. The proposed framework partitions tree-structured data into multiple lists so that each list includes values belonging to the same attribute. Then the framework stores the lists of each attribute in correspoinding files. Note that the file of "Average CPU usage" in Figure 4 includes only values belonging to "Average CPU usage" attribute, and does not include values of any other attributes.

The framework reads only 1-3 attributes required in data analysis out of 10-30 attributes, and restores tree-structured data that consists of only required attributes. In addition, when appending a new attribute, the framework writes only the newly created attribute into files. If we do not use the vertical partitioning technique, it should write all of existing attributes into files. Thus the proposed method reduces amount of input data as well as amount of output data.

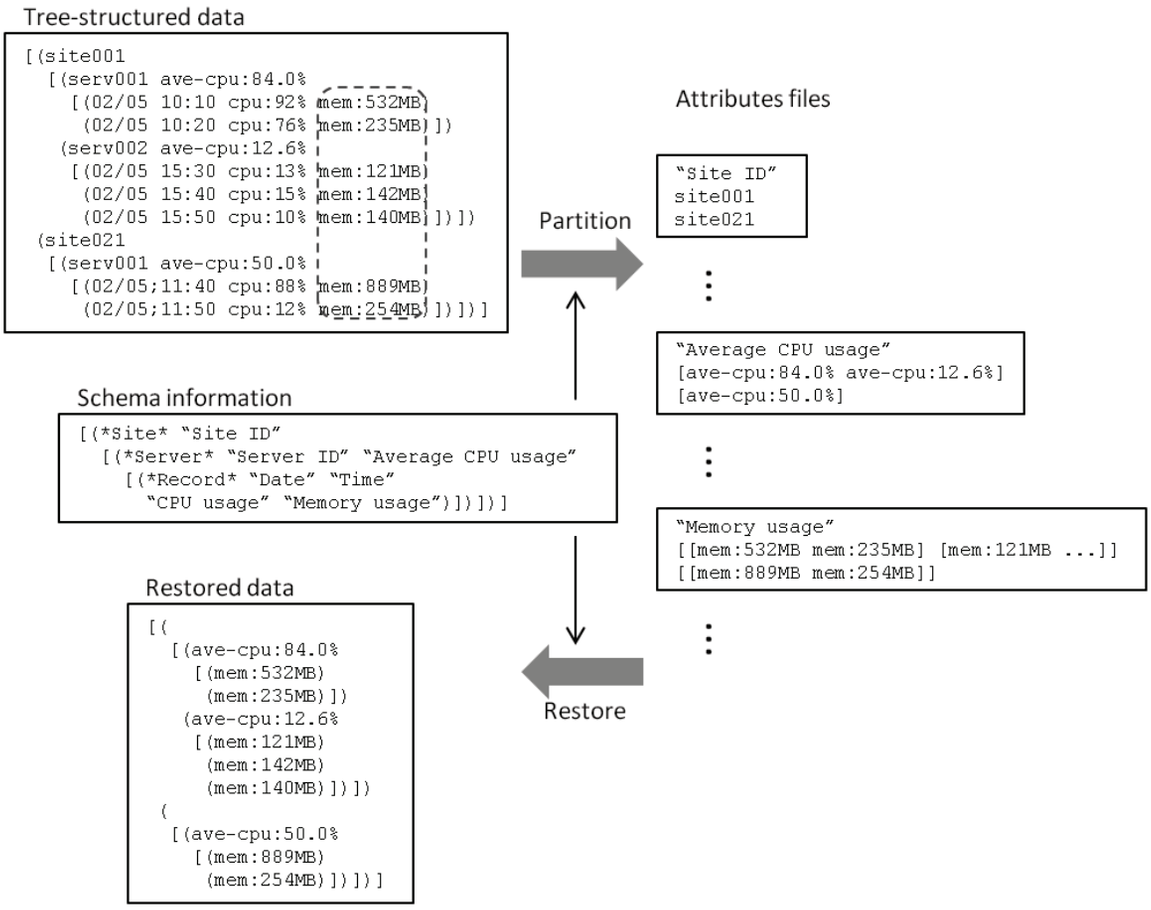

Figure 4. Vertical partitioning of tree-structured data.

3.2. Implementation

The data model of the proposed framework is a recursively-defined tuple.

Tuple: Combination of lists and values. e.g. `("serv002", 13, [(15, 10)])`.

List: Sequence of tuples whose types are the same. e.g. `[("serv001" 4.0) ("serv002" 2.6)]`.

Value: Sscalar, vector, matrix or string. e.g. `"532MB"`.

A round bracket `()` represents a tuple while a square bracket `[]` represents a list. In this paper, elements of a tuple and a list are separated by white spaces.

Figure 5 describes pseudo code of partitioning algorithm. The algorithm partitions tree-structured data into recursive lists by running the function "Partition" recursively. Each list to be generated by the algorithm consists of values belonging to the same attribute.

Similarly Figure 6 describes pseudo code of restoring algorithm. The algorithm restores tree-structured data from divided attribute data. An example of input for the algorithm is shown in Figure 7. S is trimmed schema which excludes attributes unused in analysis computation. D is generated by replacing attribute names in trimmed schema with recursive lists stored in attribute files.

```
Partition(S, D) {
  if S is atom then return [D]
  if S is list then {
    L = []
    foreach di in D:
      append Partition(SOE(S), di) to L
    return transpose of L
  }
  if S is tuple then {
    L = []
    foreach (si, di) in (S, D):
      L = L + Partition(si, di)
  return L
}
```

Figure 5. Pseudo code of partitioning algorithm. S is schema information, D is tree-structured data, The function SOE returns schema of an element of a list.

```
Restore(S, D, d=0) {
  if S is atom then return D
  if S is list then {
    L = []
    foreach (di) in D:
      append Restore(SOE(S), di, d+1) to L
    return transpose L with depth d
  }
  if S is tuple then {
  L = []
  foreach (si, di) in (S, D):
    append Restore(si, di, d) to L
  return L
}
```

Figure 6. Pseudo code of restoring algorithm. An example of the input is shown in Figure 7. The function SOE returns schema of an element of a list.

```
S: [(
     [("Average CPU uage"
      [("Memoery Usage")])])]
D: [(
    [([[ave-cpu:84.0% ave-cpu:12.6%] [ave-cpu:50.0%]])
     [([[[mem:532MB mem:235MB] [mem:121MB ...]] [[mem:889MB mem:254MB]])])])]
```

Figure 7. Example of input of the restoring alogorithm.

We implemented the partitioning algorithm and the restoring algorithm in Gauche. Gauche is an implementation of computer language Scheme. Users implement programs for attribute appending and aggregation using Gauche. The proposed framework combines user programs with partitioning and restoring programs. Then the combined program runs in parallel on Hadoop Streaming of Hadoop MapReduce 0.20.0. Table 2 summarizes key Hadoop components for implementation of the framework.

Item	Description
Hadoop aggregation package	Used for Combine and Reduce calculation.
CompositeInputFormat class	Used for multiple file input and "Map-side join" [3].
MultipleTextOutputFormat class	Used for multiple file output.

Table 2. Hadoop configuration.

Figure 8 shows an example of user program. The program appends a new attribute "Average memory usage". The variable "new-schema" represents a location of the newly appended attribute in tree structure. The function mapper generates a new tree-structured data including only the attribute to be appended. The framework provides accessors to attributes and tuples, such as "`ref-Server-tuples`" and "`ref-Memory-usage`".

```
(define new-schema
  '(
    [("Average memory usage")])
)

(define (mapper site)
   (tuple
    [foreach (ref-Server-tuples site) (lambda (server)
     (tuple
       (mean (map ref-Memory-usage (ref-Record-tuples server)))
     )])))
```

Figure 8. Example of user program.

3.3. Evaluation

We evaluated the proposed framework on 6 benchmark tasks.

Task A Calculates average CPU usage for each server and append it as a new attribute into the corresponding tuple of server information. The SQL for the calculation includes "group-by" and "update" if relational database is used instead of the proposed framework.

Task B Calculates difference between CPU usage and average CPU usage for each server. The SQL of the calculation includes "join".

Task C Calculates frequency distribution of CPU usage with interval of 10. The SQL of the calculation includes "group-by".

Task D Calculates difference between CPU usages of two successive detail records and append it as a new attribute into the corresponding tuple of a detail record. It is impossible to express the calculation with SQL.

Task E Searches detail records in which both of CPU usage and memory usage is 100%.

Figure 9 shows the result of evaluation on 90 GB data. We used 19 servers as slave machines for Hadoop: 9 servers with 2-core 1.86 GHz CPU and 3 GB memory, and 10 servers with two of 4-core 2.66 GHz CPU and 8 GB memory. Thus the Hadoop cluster has 98 CPU cores in total. The vertical axis of Figure 9 represents average execution time over 5 runs. The result indicates

that the vertical partitioning accelerates the calculations by 17.5 times on the task A and by 12.7 times on the task D. The task A and D require the processing of attribute appending, in which a large amount of tree-structured data is not only read from files, but also written into files. That is the reason why the acceleration on the task A and D is more than that on the other tasks.

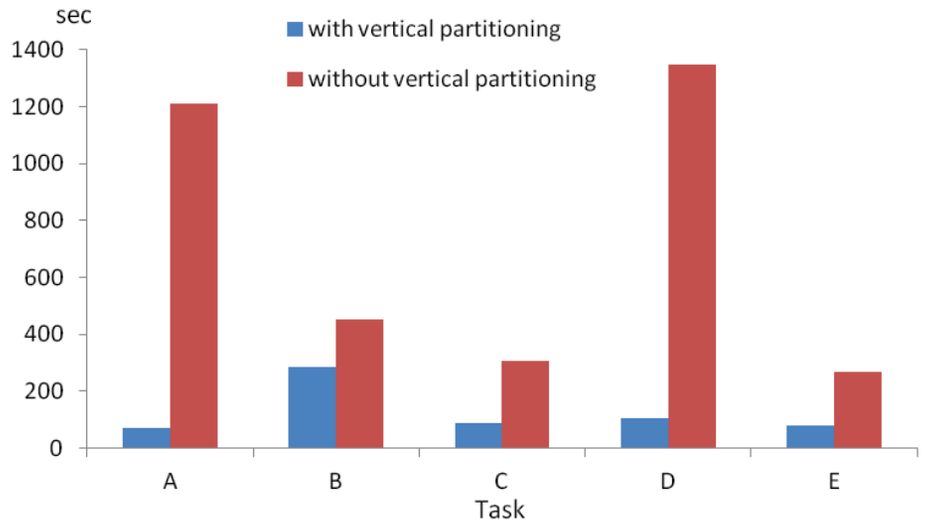

Figure 9. Evalution of the tree-structured data analysis framework.

Table 3 compares the proposed method with MySQL. Both of the proposed framework and MySQL run on a single server, and the size of benchmark data is 891 MB. Note that parallelization is not used in this experiment so that we investigate the effect of vertical partitioning and data store in tree-structured format without the disturbing factor due to parallel computation. We created indexes on columns of primary id, CPU usage and memory usage in MySQL tables. Table 3 shows average and standard deviation of execution times over 5 runs. The performance of the proposed method is comparative or superior to that of MySQL on the task A, B, C and D despite the proposed method is mainly implemented in Gauche. On the other hand, the performance of the proposed method on the task E is inferior to that of MySQL. This is because MySQL finds records that match the condition by using indexes while the proposed framework scans whole data linearly to find out the records. However, the actual execution time of the proposed framework on the task E is permissible since it is not long compared to that on the other tasks.

Task	Proposed method [sec]	MySQL [sec]
A	10.67 ± 0.08	402.72 ± 5.55
B	76.67 ± 0.36	445.48 ± 3.42
C	13.21 ± 0.18	12.89 ± 0.05
D	36.36 ± 0.20	-
E	16.87 ± 0.14	1.34 ± 2.66

Table 3. Comparison of the tree-structured data analysis framework and MySQL using a single server.

As a result of the experiments, we conclude that the proposed framework is efficient for data analysis of a large amount of tree-structured data. The performance can be improved by implementing it using Java, instead of Gauche.

4. Parallel machine learning framework

4.1. Mathod

This section proposes a computing framework for parallel machine learning. The proposed framework is designed to ease parallelization of machine learning algorithms and reduce calculation overheads of iterative procedures.

We start with discussion on a model of machine learning algorithms. Let $D = (x_n, y_n)$ be training data, where x_n is a feature vector with d dimension, y_n is a label. Machine learning algorithm estimates a model M describing D well. In this paper we discuss machine learning algorithms that are describable as an iteration of the following steps:

$$z_n = f(x_n, y_n, M) \tag{1}$$

$$M = r(g([z_0, z_1, ..., z_{N-1}])), \tag{2}$$

where M represents a model to be trained, g is a function which satisfies the following constraint.

$$\forall i < j < ... < k < N, g([z_0, ..., z_{N-1}]) = g([g[(z_0, ..., z_{i-1})], g[(z_i, ..., z_{j-1})], ..., g[(z_k, ..., z_{N-1})]]) \tag{3}$$

For instance, a function that summates elements in an array satisfies the constraints mentioned above. By using the characteristic of g, we re-formulate the steps of machine learning algorithms as follows.

$$M_{i..j} = g([f(x_i, y_i, M), f(x_{i+1}, y_{i+1}, M), ..., f(x_{j-1}, y_{j-1}, M)]) \tag{4}$$

$$M = r(g([M_{0..i}, M_{i..j}, ..., M_{k..N}])) \tag{5}$$

Note that parallelization of the calculation of $M_{i..j}$ is possible since the calculation is independent of other (x_n, y_n).

Consider we use MapReduce for parallelization; Map phase calculates $M_{i..j}$ and Reduce phase calculates M. Although MapReduce fits parallelization of machine learning algorithms described with the above formula, use of Hadoop Mapreduce, that is, the most popular implementation of MapReduce, is unreasonable. Because the implementation of Hadoop MapReduce is optimized so that it performs non-iterative algorithms efficiently. The problems with repeatedly using Hadoop MapReduce are following.

- Hadoop MapReduce does not keep feature vectors in memory devices during iterations.
- Hadoop MapReduce restarts threads of Map and Reduce at every iteration. Initialization overheads of these threads are large compared to computation time of machine learning algorithms.

Consequently, the proposed framework provides another MapReduce implementation for iterative algorithms of machine learning. The key approaches of the framework are follows.

1. It keeps feature vectors in memory devices during iterations. In case data size of feature vectors is larger than memory size, it uses local disk as a cache.
2. It does not terminate threads of Map and Reduce and uses the same threads repeatedly.
3. It controls iterations, read/write and data communication.
4. Users implement only 4 functions: initialization of M, calculation of $M_{i..j}$, update of M and termination condition.
5. It utilizes Hadoop DFS as its file system.

A few MapReduce frameworks for iterative computation have been proposed. Haloop [8] adds the functions of loop control, caching and indexing into Hadoop. However, it restarts threads of Map and Reduce at every iteration like Hadoop. Therefore, the initialization overheads still remain. Twister [10, 11] and Spark [9] reduce the initialization overheads and keep feature vectors in memory devices during iterations. These frameworks perform similarly to the proposed framework if input data size is smaller than total memory size of a computing cluster. However, in case the data size is larger than total memory size, the performance of the proposed framework is superior to that of Twister and Spark since the proposed framework uses local disk as a cache.

4.2. Implementation

We implemented the proposed framework using Java. The framework reads feature vectors and configuration parameters from Hadoop DFS with version of 0.20.2. Figure 10 illustrates the sequence diagram of the proposed framework. The framework consists of a master thread, a Reduce thread and multiple Map threads. The master thread controls the Reduce thread and the Map threads. The Reduce thread controls iterations. The Map threads parallelize calculations of $M_{i..j}$.

Firstly the master thread starts multiple Map threads, which read feature vectors from Hadoop DFS and keep the data in memory and HDD in a local machine during an iteration. Secondly the master thread starts a Rreduce thread. The Map threads and the Reduce thread are not terminated until the iteration ends. Next the Reduce thread initializes M, and then the Map threads calculate $M_{i..j}$ in parallel. The Reduce thread updates M by collecting the calculation results from the Map threads and continue the iteration.

Figure 11 and Figure 12 shows implementation of parallel K-Means algorithms using the proposed framework. We omit initialization of M and termination condition since these implementations are obvious. As shown in Figure 11 and Figure 12, parallelization of the algorithm is easily implemented, and the source code is short. The rest procedures are implemented inside the framework, and users do not have to write codes of data transfer and data read. Thus users are able to focus on core logics of machine learning algorithms.

4.3. Evaluation

We compared the proposed framework with Hadoop. We used Mahout library as implementations of machine learning algorithms on Hadoop [16]. We used 6 servers as slave machines for both of the proposed framework and Hadoop: 4 servers with 4-core 2.8 GHz CPU and 4 GB memory, and 2 servers with two of 4-core 2.53 GHz CPU and 2 GB memory. In

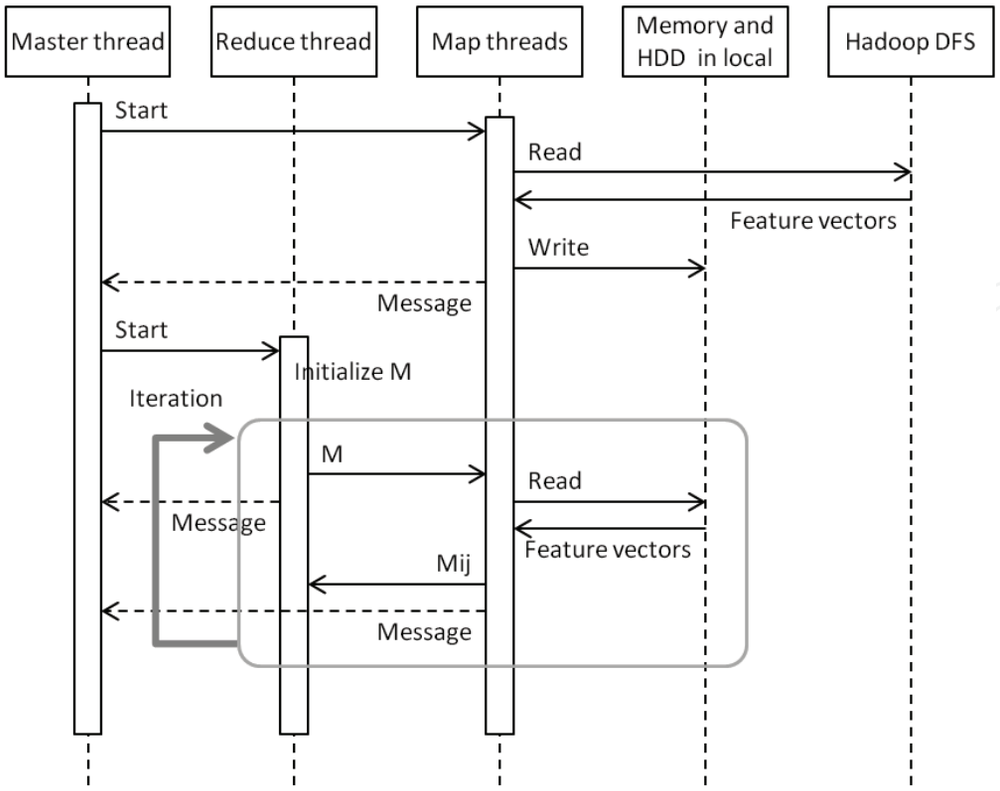

Figure 10. Sequence diagram of parallel machine learning framework

```
class KMeansMapper extends Mapper<KMeansModel> {
  public KMeansModel map(KMeansModel M) {
    KMeansModel Mij = new KMeansModel();
    for (D : x) {
      int cid = argmin_distance(x, M);
      Mij.s[cid].add(x);
      Mij.l[cid] += 1;
    }
    return Mij;
  }
}
```

Figure 11. Implementation of computing $M_{i..j}$ in parallel K-Means algorithm.

the Map phase, 40 Map threads run in parallel. On the other hand, one Reduce thread runs in the Reduce phase. The data size of feature vectors is 1.4 GB. Table 4 shows execution times of one iteration on three machine learning algorithms: K-Means [2], Dirichlet process clustering [12] and IPM perceptron [13, 14]. The values are mean and standard deviation over 10 runs. The result indicates that the proposed framework is 33.8-274.1 times as fast as Mahout.

Figure 13 illustrates scalability of the proposed framework on three machine learning algorithms: K-Means, variational Bayes clustering [15] and linear SVM [1]. The horizontal axes represent the number of Map threads that run in parallel. The vertical axes represent 1 / (execution time), i.e., speed. Figure 13 indicates that the more Map threads in parallel are, the faster the parallelized algorithms run.

```
class KMeansReducer extends Reducer<KMeansModel> {
  public KMeansModel reduce(KMeansModel[] Mijs) {
    KMeansModel M = new KMeansModel();
    for (int cid=0; cid<M.num_of_cluster; cid++) {
      for (Mijs : Mij) {
        M.s[cid].add(Mij.s[cid]);
        M.l[cid] += Mij.l[cid];
      }
      M.centroid[cid] = M.s[cid] / M.l[cid];
    }
    return M;
  }
}
```

Figure 12. Implementation of updating M in parallel K-Means algorithm.

Algorithm	Proposed method [sec]	Mahout [sec]
K-Means	0.93 ± 0.052	31.8 ± 1.49
Dirichlet process clustering	1.14 ± 0.057	67.4 ± 3.87
IPM perceptron	0.11 ± 0.026	30.7 ± 2.00

Table 4. Comparison of the parallel machine learning framework and Mahout on K-Means [2], Dirichlet process clustering [12] and IPM perceptron [13, 14].

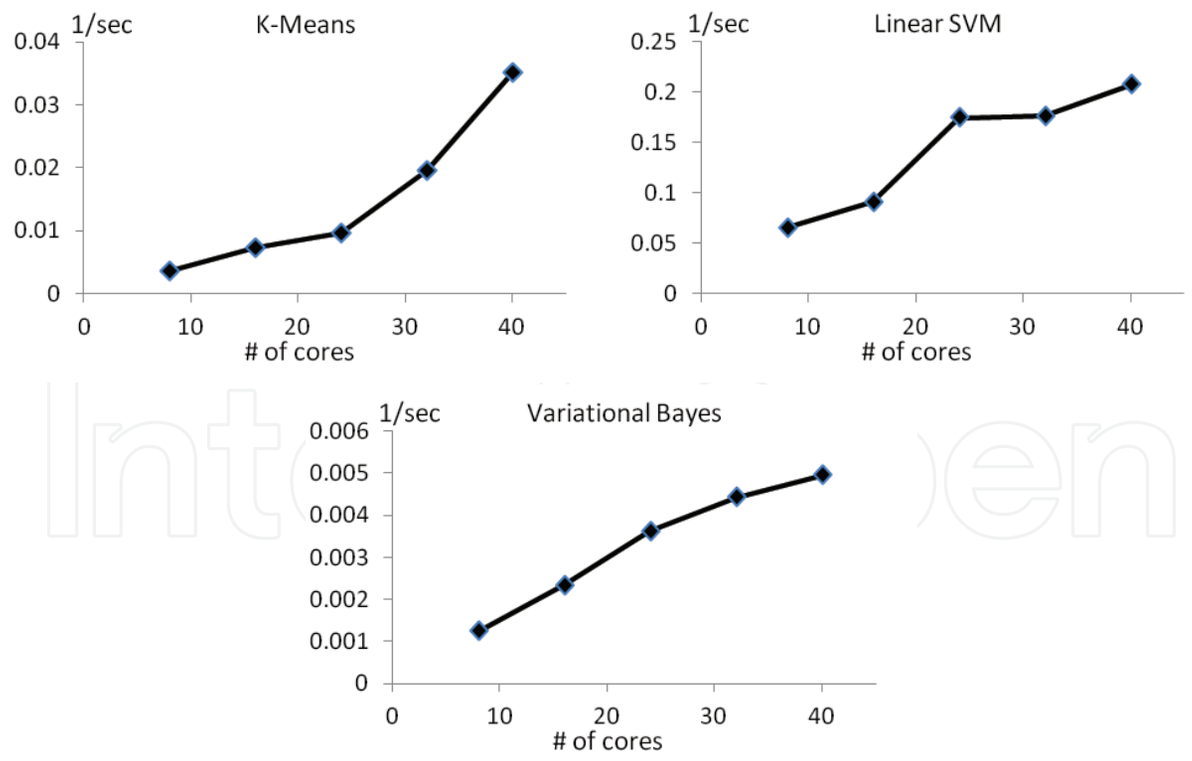

Figure 13. Scalability evaluation of the parallel machine learning framework.

We also applied the framework in order to parallelize a learning algorithm of an acoustic model for speech recognition. The learning algorithm reads voice data and corresponding

text data, and generates a Hidden Markov model by using Forward Backward algorithm. We compared performance of the parallelized algorithm with that of single thread implementation using C language. We used 1.0 GB of feature vectors as a input of these programs. The parallelized algrorithm on the proposed framework with 32 parallel Map threads run 7.15 times faster than the single thread implementation. Considering the difference of speed between Java and C language, the proposed framework performs the parallelization well. Consequently, we conclude that the proposed framework is efficient for parallel machine learning.

5. Conclusion

This chapter discussed techniques for processing large-scale data. Firstly we explained that process of data utilization in enterprises and organizations includes (1) pre-processing phase, (2) analysis phase, (3) model learning phase and (4) model application phase. Secondly we described architecture for the data utilization process. Then We proposed two computing frameworks: tree-structured data analysis framework for analysis phase, and parallel machine learning framework for model learning phase. The experimental results demonstrated that our approaches work well.

Future works are follows:

- To implement tree-structured data analysis framework using Java.
- To design original machine learning algorithms which run on the parallel machine learning framework.
- To formulate a framework for model application phase.

Author details

Kohsuke Yanai
Research & Development Centre, Hitachi India Pvt. Ltd.
Central Research Laboratory, Hitachi Ltd.

Toshihiko Yanase
Central Research Laboratory, Hitachi Ltd.

6. References

[1] Chapelle, O. (2007) Training a Support Vector Machine in the Primal. Neural Computation, Vol. 19, No. 5, pp. 1155-1178.
[2] MacQueen, J. B. (1967) Some Methods for Classification and Analysis of MultiVariate Observations. Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability, Vol. 1, pp. 281-297.
[3] White, T. (2009) Hadoop: The Definitive Guide. Oreilly & Associates Inc.
[4] Dean, J. and Ghemawat, S. (2004) MapReduce: Simplified Data Processing on Large Clusters. Proceedings of Sixth Symposium on Operating System Design and Implementation (OSD2004), pp. 137-150.

[5] Dean, J. and Ghemawat, S. (2008) MapReduce: simplified data processing on large clusters. Communications of the ACM, Vol. 51, No. 1, pp. 107-113.
[6] Data Mining Group. PMML standard, http://www.dmg.org/v4-1/GeneralStructure.html
[7] Manegold, S., Boncz, P. A., and Kersten, M. L. (2000) Optimizing database architecture for the new bottleneck: memory access. The VLDB Journal, Vol. 9, No. 3, pp. 231-246.
[8] Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2010) HaLoop: Efficient Iterative Data Processing on Large Clusters. Proceedings of the VLDB Endowment, Vol. 3, p. 1.
[9] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010) Spark: Cluster Computing withWorking Sets. HotCloud 2010.
[10] Ekanayake, J. and Pallickara, S.: (2008) MapReduce for Data Intensive Scientific Analysis. Forth IEEE International Conference on eScience, pp. 277-284.
[11] Ekanayake, J., Balkir, A. S., Gunarathne, T., Fox, G., Poulain, C., Araujo, N., and Barga, R. (2009) DryadLINQ for Scientific Analyses. Proceedings of Fifth IEEE International Conference on e-Science (eScience2009).
[12] McCullagh, P. and Yang, J. (2008) How Many Clusters. Bayesian Analysis, Vol. 3, No. 1, pp. 101-120.
[13] McDonald, R., Hall, K., and Mann, G. (2010) Distributed Training Strategies for the Structured Perceptron. Proceedings of Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the ACL, pp. 456-464.
[14] Hall, K. B., Gilpin, S., and Mann, G. (2010) MapReduce/Bigtable for Distributed Optimization. Neural Information Processing Systems Workshop on Leaning on Cores, Clusters, and Clouds.
[15] Corduneanu, A. and Bishop, C. M. (2001) Variational Bayesian Model Selection for Mixture Distributions. Artificial Intelligence and Statistics 2001, pp27-34.
[16] Apache Mahout: Scalable machine learning and data mining. http://mahout.apache.org/