PAPER Special Section on Parallel and Distributed Computing and Networking

# An Improved Rete Algorithm Based on Double Hash Filter and Node Indexing for Distributed Rule Engine

**Tianyang DONG**[†a)], **Jianwei SHI**[†], **Jing FAN**[†], *Nonmembers*, *and* **Ling ZHANG**[††], *Student Member*

**SUMMARY** Rule engine technologies have been widely used in the development of enterprise information systems. However, these rule-based systems may suffer the problem of low performance, when there is a large amount of facts data to be matched with the rules. The way of cluster or grid to construct rule engines can flexibly expand system processing capability by increasing cluster scale, and acquire shorter response time. In order to speed up pattern matching in rule engine, a double hash filter approach for alpha network, combined with beta node indexing, is proposed to improve Rete algorithm in this paper. By using fact type node in Rete network, a hash map about 'fact type - fact type node' is built in root node, and hash maps about 'attribute constraint - alpha node' are constructed in fact type nodes. This kind of double hash mechanism can speed up the filtration of facts in alpha network. Meanwhile, hash tables with the indexes calculated through fact objects, are built in memories of beta nodes, to avoid unnecessary iteration in the join operations of beta nodes. In addition, rule engine based on this improved Rete algorithm is applied in the enterprise information systems. The experimental results show that this method can effectively speed up the pattern matching, and significantly decrease the response time of the application systems.

***key words:*** *Rete algorithm, double hash filter, beta node indexing, rule engine*

## 1. Introduction

Rule engine, also known as production system, is a common way to build expert systems. It has been widely used in many fields such as business, science, engineering, manufacturing, and medicine. The approach to distributed rule matching and multiple firing based on MapReduce can promote the performance of rule engine for massive rules reasoning, and keep the consistency of working memory in the process of distributed rule matching.

Business rule in rule engine usually defines or restricts some aspects of enterprise business. It is used to identify the business structure, control or influence business behaviors. There are two existing forms of business rule in enterprises. One form is coded in all kinds of enterprise information systems (EISs), that makes these EISs hard to adapt to the changes of business requirements, and causes the difficulty in system maintenance and updation. The other form not only contains the knowledge and business experiences grasped by the employees, but also exists in all kinds of non-formal documents, such as program operation manuals, enterprise agreements and contracts. The second form lacks of effective unified management, and will disappear with the turnover of employees and the damage of documents, resulting in the invisible loss of the enterprise soft power.

In order to solve these problems, more and more enterprise systems adopts Rule Engine [1] to realize the separation of business rules and business logics, and decouple the frequent changes of business rules from information systems, which greatly enhances the intelligence, robustness and adaptability of systems, including expert systems and EISs. The corresponding Business Rules Management System (BRMS) [2] can uniformly define and manage all kinds of business rules with the form of production rules, which makes the business knowledge effective to maintain and reuse. However, these rule-based systems may have a performance bottleneck, when there is a large amount of facts data to be matching with rules. So it is urgent to present more efficient pattern matching algorithm to handle this performance problem.

The Rete algorithm is so far considered as the most efficient pattern matching algorithm, and was first proposed by Dr. Charles L. Forgy of Carnegie Mellon University in 1974. And he gave the more detail description of the Rete algorithm in his Ph.D. thesis, and a paper in 1982 [3]. Rete has been applied in numerous rules engines such as OPS5, CLIPS, Jess and ILOG JRules, to realize efficient pattern matching.

In order to meet some special requirements and application scenarios, many researchers have made some adaptable improvement for Rete algorithm [4]–[11]. Doorenbos [4] added left unlinking and right unlinking to Rete, which made the improved Rete more suitable to large scale of rules. Liping Yan and Zhenyun Pan [5] speeded up processing efficiency when the system deleted facts, through storing parent records and children records in the items of alpha memories and beta memories. Zhijun Ren and Ding Xiao et al. Ref. [6], [7] made Rete support the 'or' connection of patterns, and use shadow proxy mechanism to update fact data. For the problem of Rete lacking of support for time-sensitive patterns, Ref. [8]–[11] have used timestamp technology to extend Rete to deal with facts and events simultaneously, support relative temporal constraints and temporal reasoning.

All the above improvement researches of Rete algorithm haven't solved the problem of slow propagation, when large amounts of facts enter Rete network. This paper introduces fact type node to Rete algorithm, and constructs hash maps for successor nodes in root node and fact type nodes, then speeds up the propagation of facts in alpha net-

Manuscript received January 20, 2013.
Manuscript revised May 28, 2013.
[†]The authors are with ZJUT, Hangzhou, 310023 China.
[††]The author is with ZJU, Hangzhou, 310027 China.
a) E-mail: dty@zjut.edu.cn
DOI: 10.1587/transinf.E96.D.2635

work. Meanwhile, it constructs indexes for facts in beta node memories, and decreases unnecessary join tests. Finally, this improved Rete algorithm is deployed in the rule engine based on MapReduce, and applied in several enterprise information systems. The experimental results show that this algorithm can improve the speed of pattern matching, and reduce system execution time.

## 2. Inferring Cycle of Rule Engine

A typical rule engine contains three parts: Production Memory (PM), Working Memory (WM) and Inference Engine. The most important Inference Engine consists of three components: Pattern Matcher, Agenda and Execution Engine.

As shown in Fig. 1, the rule engine will execute the following "Match-Resolve-Act" cycle. The rule engine loads the facts input by users as the initial working memory elements (WMEs) into WM, which will match with the rules of PM in pattern matcher. Those rules whose condition parts match the contents of current WM are instantiated and added into the Agenda. There may be more than one rule instantiation, which form a conflict set. The Agenda selects just one appropriate instantiation, according to some conflict resolution strategies. The execution engine is responsible for executing the action part of selected instantiation. The involved operations include: adding new WMEs to WM, deleting existing WMEs from WM, and modifying existing WMEs in WM. Afterwards, these changes of WM will match with the rules. The rule engine will repeat the above inferring cycle, until there is no instantiation in the Agenda. The WM at this moment represents the terminal state of rule engine, and contains the final processing results.

Because most rule engines consume about 90% of execution time in match phase of the inferring cycle [3]. When the number of facts dramatically increases, the rule engine will spend more time in matching WMEs with rules to acquire firing rule instantiations. So the match phase becomes the performance bottleneck of rule engine, and it needs efficient pattern matching algorithm to solve this problem.

Rete algorithm is the most widely used pattern matching algorithm, and applied in the pattern matcher to build a pattern recognition network—Rete network, thus realizing the efficient match of facts and rules. Rete uses the temporal redundancy and structure similarity to match the rules with the changes of WMEs, greatly decreasing the execution time in the match phase of rule engine.

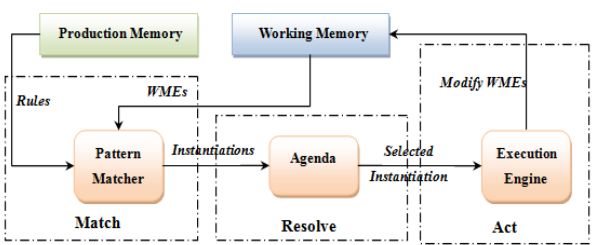



**Fig. 1** The inferring cycle of rule engine.

## 3. A Novel Rete Algorithm Based on Double Hash Filter and Node Indexing

### 3.1 Overview of Rete Algorithm

The Rete algorithm compiles pattern elements in the condition parts of all production rules in rule base into a dataflow recognition network, called Rete network, which acts as the pattern matcher of the inference engine. Rete uses the temporal redundancy to continuously store the matching states of the match phase in the inferring process, and only deals with the changing parts of WM. Rete also uses the structure similarity of rule-based systems, extracts similar patterns or pattern set from several rules as the public components to share with other distinct components, which greatly decreases the computation in public parts.

The input of Rete network is a serial of tokens representing changes of WM, which includes the initial facts inserted into WM by the user, and the modification made by the act phase of the inferring cycle. The output of this network includes the changes to the conflict set, the addition and removal of rule instantiations in the conflict set made by the matching of facts and rules.

The original Rete network [3] contains five kinds of nodes: root node, constant-test nodes, memory nodes, two input nodes and terminal nodes. As shown in Fig. 2, the root node is the input node of network. The constant-test nodes, also called alpha nodes, lie in the first layer of Rete network. They store these attributes having constant values, and perform the intra-condition tests, which judge whether WMEs satisfies the constant value of the corresponding condition elements or not. The tokens which pass all the constant tests of a condition element are stored in the alpha memory. The two-input nodes are also named beta nodes or join nodes. The two inputs of these nodes respectively connect with a beta memory and an alpha memory. The token set from the beta memory are compared the consistency of the variable bindings with the tokens from the alpha memory, and the results are stored in the subsequent beta memory. There is at least one terminal node corresponding to each

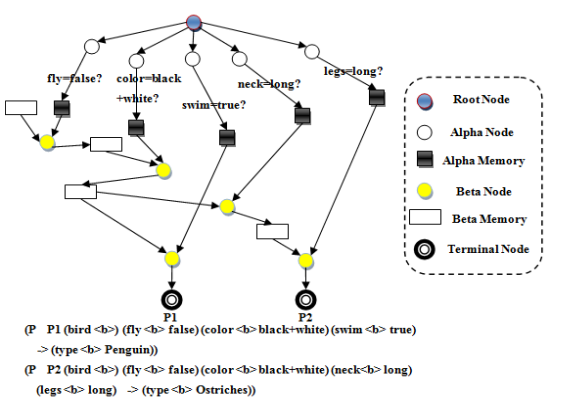



**Fig. 2** The Rete network based two sample rules.

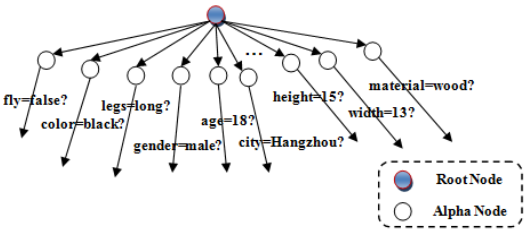



**Fig. 3** The sample alpha network constructed by basic Rete algorithm.

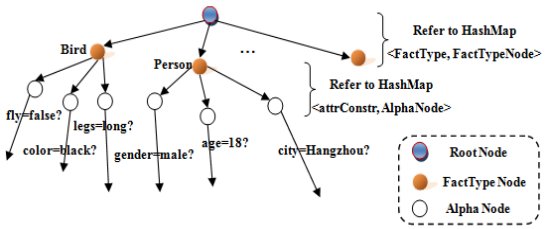



**Fig. 4** The alpha network with the double hash filter mechanism.

rule in the rule base. When the condition part of a rule is fully matched, the corresponding tokens will arrive at the terminal node, representing that an instantiation of this rule should be added into or removed from the conflict set. The parts from root node to alpha memories in Fig. 2 constitute the alpha network, which mainly does the intra-condition pattern matching. The rest parts in Fig. 2 form the beta network that primarily processes the inter-condition join operations. The alpha network plus the beta network is the integral Rete network.

### 3.2 Rete Algorithm Based on Double Hash Filter and Node Indexing

In the original Rete, the root node is attached directly with several alpha nodes, which are used to do the constant value tests for attributes of the facts. When there are large amounts of fact types respectively containing many attributes, this kind of one-to-many relation between the root node and alpha nodes will result in slow propagation of tokens in the Rete network.

For example, there are a fact type of Bird with attributes of 'fly', 'color' and 'legs', a fact type of Person with attributes of 'gender', 'age' and 'city', and a fact type of Table with attributes of 'height', 'width' and 'material'. The alpha network is constructed according to the basic Rete shown as Fig. 3, in which the root node is attached with alpha nodes of these nine attributes. When there are numerous fact instances with the type of Bird, Person and Table, the Rete network may test the value of 'city' for the instances of Bird type, test the value of 'fly' for Person instances, or test the 'gender' value for Table instances. When the Rete network in Fig. 3 involves more fact types with more attributes, the value tests of these unrelated attributes will spend more time, which makes facts propagation slowly in the alpha network.

In response to the above issue, this paper introduces fact type node to the Rete network, and classifies the alpha nodes according to the involved fact type. A hash map *factType2NodeMap* is constructed in root node, and uses the fact type as key and the fact type node as value, to realize quick searching and locating from the root node to the target fact type node. Similarly, a hash map *attConstr2NodeMap* is built in each fact type node, using the involved attribute constraint of the alpha node as key and the corresponding alpha node as value. As presented in Fig. 4, when large amounts of tokens propagate from root node to alpha nodes, the alpha

```
1.Create a Root Node rootNode;
2.For each rule r in rule set {
3.   For each pattern p in current rule r {
4. If(! rootNode.hasFactType(p.getFactType()) {
5.Create a Fact Type Node factTypeNode;
6.rootNode.getFactType2NodeMap().put(p.getFactType(),factTypeNode)
7.}
8. fTypeNode = rootNode.getFactType2NodeMap.get(p.getFactType());
9.   For each constraint c in current pattern p {
10.   attrName = c.getAttriName();
11.   oper = c.getOperator();
12.   attrValue = c.getValue();
13.   hashKey = getHashKey(attrName, oper, attrValue);
14.   If(! fTypeNode.hasConstraint(hashKey)) {
15.   Create a Alpha Node alphaNode;
16.   alphaNode.setConstraint(c);
17.   fTypeNode.getAttConstr2NodeMap().put(hashKey, alphaNode);
18.    }// end of if '! fTypeNode.has...'
19.   }// end of for each 'each constraint c ...'
20   }// end of for each 'For each pattern p ...'
21.}// end of for each 'For each rule r ...'
```

**Fig. 5** The pseudo-code of the alpha network building process adding double hash filter mechanism.

network adopting this kind of double hash filter mechanism, firstly uses some tokens of the fact type as key to find the corresponding fact type node in root node. The token then enters the alpha nodes corresponding to this fact type for further processing. This filter mechanism avoids iteration tests of large number of unrelated alpha nodes, and dramatically reduces the propagation time of tokens in the alpha network.

The implementation of the hash map used in this improved Rete algorithm is the ConcurrentHashMap in java.util.concurrent. The *factType2NodeMap* used in root node is with type of <Class<?>, FactTypeNode>, and the *attConstr2NodeMap* used in fact type Node is with type of <String, AlphaNode>. No matter what kind of domain knowledge used by BRMS, the alpha network building process with double hash filter mechanism is the same, and the corresponding of the pseudo-code is presented in Fig. 5.

The construction of the Rete network begins with creating root node. Each rule in the production memory is acquired its pattern elements. As for the pattern element

p of the current rule r, if the corresponding fact type node doesn't exist in the *factType2NodeMap* of root node, a fact type node *factTypeNode* will be created, and a map item will be added into the *factType2NodeMap* using the corresponding fact type as key and *factTypeNode* as value. Then as for each attribute constraint c of the current pattern p, an alpha node *alphaNode* is created, and a hash value *hashKey* is calculated with the parameters of attribute name, operator and attribute value of c. Then an item is added into the *attConstr2NodeMap* of *fTypeNode*, using *hashKey* as key and *alphaNode* as value. Pattern elements of all the rules will be processed respectively, until the construction of the alpha network is completed.

A beta node has two inputs: one accepts tuples (lists of facts), and the other accepts facts. The former is placed in the left beta memory of the beta node, and the latter is held in the right alpha memory of the beta node. When tuples enter into the left memory of the current beta node, they need to do join tests with all the facts in the right memory. Similarly, when facts enter the right memory, they will join with all the tuples in the left memory. The join operations in the beta node are consistency test of some variable bindings. A new tuple will be created and propagated to the successor node after the success of the join operation. When the number of facts entering the two sides of the beta node increases dramatically, it may cause the situation that each fact in the alpha memory will compare with each tuple in the beta memory. It is a time-consuming task, and forms another performance bottleneck of the Rete network, which can be effectively solved by using beta node indexing technology [12].

Figure 6 presents the process of constructing and using indexes for facts in the input memories of the beta node. This paper adopts the element list to store specific facts or tuples. Each element list corresponds with a unique index value, and all the references of the first items in these element lists form a hash table. When a fact element $fact_k$ enters into the right alpha memory of the current beta node, a hash value *hashCode* will be acquired with the parameter of this fact element, and then an index number *index* will be calculated corresponding to *hashCode*. If an element matching *hashCode* in the element list related to *index* is found, it will return the corresponding element list if found, and use the containing facts to join with the tuples in the left memory. If the element list related to *index* is not found, it will create a new one adding this fact, and return this element list.

By this method, when new facts enter the current beta node, it can use the facts of the element list satisfying some specific constraints to join with the tuples in the left input memory, which avoids unnecessary iteration tests by the way of narrowing the data extent.

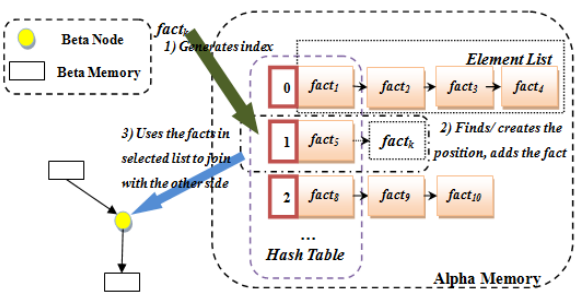



**Fig. 6** The application process sample of the beta node indexing.

## 4. Application of Improved Rete Algorithm

### 4.1 Distributed Rule Engine Based on MapReduce

The rule engine based on the improved Rete algorithm proposed in this paper can integrate with all kinds of enterprise information systems as the form shown in Fig. 7. Originally, the frequent changing business rules are coded in the application systems, documented in the files with the paper or electronic versions, or grasped by the business employees as experiences. These forms of business rules lacking of unified management and maintenance are easily lost and damaged. The situation changes when introducing the business rule management platform including the rule engine. Business rules and business knowledge are stored in rule base with the form of production rules. The expression and editing process of this knowledge are done in the business rule modeling tool. The Web-based business rule management system uniformly manages and maintains the rule resources in rule base. All kinds of enterprise information systems interact with the rule engine through the engine APIs. The rule engine receives the business data from the outside, uses rules in the rule base to match and process it, and returns the final results.

MapReduce [13] is a programming model used to process and generate massive dataset in the circumstance of distributed clusters in large scale. The programs based on MapReduce have intrinsic parallelism, and can run in the cluster of moderate commercial machines. These MapReduce-based programs have two phases: Map and Reduce. In map phase, the master server gets inputs, splits them into smaller subtasks and assigns them to the map workers; then map workers receive and process the given inputs, produce intermediate key/value pairs. Then into reduce phase, the intermediate results with the same key are acquired by a specific set of reduce workers, and merge into

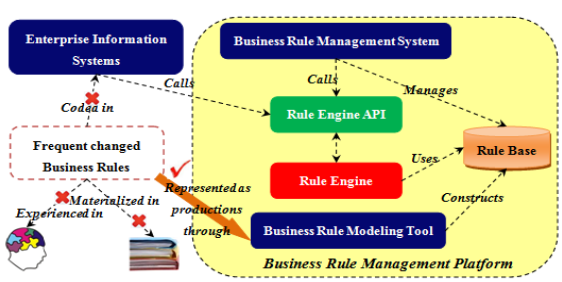



**Fig. 7** The integration way of the EISs and the rule engine.

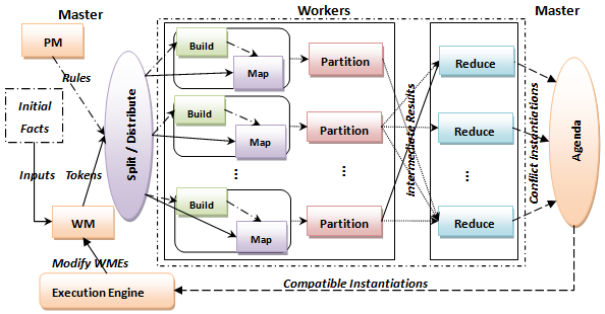



**Fig. 8** The distributed matching and multiple firing rule engine based on MapReduce.

smaller set of key/value pairs; these outputs are sent to and processed by the master server into the final results.

This paper aims at combining the multiple rule firing method based on access control with the MapReduce-based distributed rule matching, and developing a rule engine with functions of distributed matching and multiple firing. This rule engine adopts the master-salve pattern to build, consisting of a master server and several workers. The workers can be divided into two categories: map worker and reduce worker. The architecture of the overall distributed system is presented in Fig. 8. The master server needs not only to monitor and manage the workers in the runtime and do the pre-process of the rule set, but also to dispatch the match tasks to the cluster, select and execute the compatible rule instantiations generated by the multiple rule firing method in this paper after the completion of the resolve phase. Map workers are in charge of building the local Rete network, and executing the specific distributed matching task. Reduce workers are responsible for resolving the intermediate matching results, and sending the final results to the master server.

The distributed rule engine will go through the following phases:

- Build Phase: After the cluster starts up, the master server firstly acquires some state information of the workers in the distributed environment; secondly, it decomposes the rules and reconstructs them into sub-rule sets, and keeps related information preparing for merging the intermediate matching results, selecting and executing rules; then it selects and dispatches sub-rule sets according to the running state of the map workers. Map workers receive the rule sets from the master server, and build them into the Rete network with double hash filter mechanism and beta node indexing.
- Map Phase: Not only for the loading of initial fact sets into the WM, but also for the adding, deleting and modifying of WMEs in the inferring cycle, the master server encapsulates all these WM changes as token sets. Then according to some strategy, it dispatches them to the appropriate map workers for matching. Map worker receives the tokens and inserts them into the local Rete network to generate the intermediate matching results with the form of <key, value>, which indicates some specific sub-rule has been matched successfully.
- Reduce Phase: Reduce workers receive the location information of the intermediate results from the master server, and read these results from the corresponding map workers by remote procedure call. Then the intermediate results with the same key are merged, and matching results with the form of <key′, value′> are generated and sent to the master server.
- Select Phase: The results sent by reduce workers corresponds with a serial of rule instantiations, which are put into the agenda of master server. Master server selects the compatible rule instantiation set from the conflict rule set according to the multiple rule firing method proposed in Sect. 3.
- Act Phase: Then master server executes the RHSs of these compatible instantiations, which will add, delete and modify the WMEs in the WM.

It will repeat the process from the map phase, until the agenda is empty or there is an execution of some explicit shutdown rules. Then the inferring cycle of this distributed rule engine completes. The elements in the WM of master server are just the final processing results, corresponding to the terminal state of the distributed rule engine.

Comparing with the naive MapReduce-based programs, the MapReduce-based distributed rule engine firstly has more phases in the processing cycle. Secondly, except for the rule files are as one kind of the inputs and built into Rete network in memories of map workers, the facts or business data are as the other kind of inputs and processed and stored in memories. The inputs and computations of native M/R programs are based on files in most occasions, but the data and computation of the proposed distributed rule engine are based on memories, in view of the characteristics of Rete algorithm and the process phases of the rule engine.

The rule engine illustrated in Fig. 8 can meet the processing challenge of massive rules in certain extent. But when the rule scale reaches to TB or even exceeds, the system execution time will become much longer and the processing stress of master server will dramatically increase. So the system architecture with single master server will result in potential security risks. In order to avoid the system performance bottleneck caused by the problem described above, it needs to do further functional expansion for this architecture.

The rules can be divided into many application categories according to the application domains, and each category can be further divided into several more specific levels with tree structures. This tree hierarchy existing in domain rules determines that the usage of domain rules also has certain hierarchy. Therefore, we can keep the management mechanism of single master server corresponding to single cluster, and process the domain rules in current category of the same application domain. It needs to expand the system architecture by using the way presented in Fig. 8. That will make the distributed rule engine better respond to

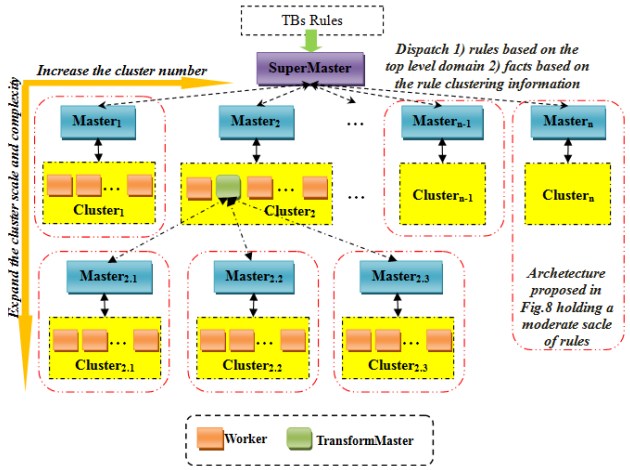



**Fig. 9** The rule engine architecture for the processing requirements of massive rules.

the processing requirements of massive rules.

As shown in Fig. 9, we can construct rule engine for massive rules according to the steps described below. Firstly, it needs to construct several clusters in accordance with the system architecture with single master server, and build a supermaster server above the multiple master servers. Secondly, in the supermaster, the rules with the scale of TBs are divided according to the top application domain, and rule sets with a moderate scale are dispatched to the clusters managed by each master server. Thirdly, the map workers in each cluster build the received rules into the Rete network. Fourthly, the supermaster sends the fact set to the appropriate master server according to the involved application domain, the cluster managed by the latter will process the specific rule reasoning. Finally, after some clusters finish their reasoning, the belonging master server sends the final results to the supermaster server. In the process described above, the reasoning between the clusters managed by the supermaster server is in parallel; that means the distributed matching in each cluster is also in parallel.

Additionally, the clusters in Fig. 9 can be classified into two kinds. One is the cluster having the system architecture with single master server, the workers in which do the distributed rule matching based on MapReduce. The other is the expanded cluster with the transform-master server, which acting as forward node manages the belonging master servers and dispatches the corresponding fact sets. This kind of cluster with tree structure constructed through the transform-master server is to meet the hierarchical application requirements of the massive domain rules.

The expanded rule engine architecture can be adjusted through the horizontal and vertical ways, enlarging the scale and complexity of the clusters. Accordingly, the throughput of the overall system will be increased, and the system execution time will be shortened. The rule engine architecture illustrated in Fig. 9 accords with the hierarchical application requirements of domain rules, and can effectively meet the processing challenge of massive rules.

### 4.2 Application Result

In real business environments we involve and develop such as the Automatic Generating Test Paper System (AGTPS) [14], the Wage Tax Calculation System (WTCS) and the Mobile Fee Calculation System (MFCS), the BRMS is used as a basic midware component. It manages the business rules applied in those EISs, and provides the rule engine to support business decisions. The present and manageable forms of business rules are all kinds of rule files, but the usage of these rules in EISs are compiled into Rete network to match domain data. In the application fields of the EISs mentioned above, domain data or business data is entered by user through web pages and stored in database. The Rete network must exist in the memories access the processing nodes of the distributed rule engine, so does the process of rule computation. The I/O may happen just under these circumstances: a) the rules are read by the distributed rule engine, b) the domain data (facts) are sent into or returned from the distributed rule engine, c) intermediate results are sent between master and workers in the distributed rule engine. The memories of expandable clusters can hold the rule match network, business data as well as the whole process of rule computation.

AGTPS is one of the research achievements of Excellent Course Construction Project of the Zhejiang University of Technology, contains two subsystems for the common question base management and [14] generating test paper management. The AGTPS uses 11 attributes such as id, type, difficulty, cognitive level, subject, knowledge point, content, key, score, answer time and exposure to describe a question in the system, which correspond to an 11-dimensional vector (a1, a2, a3, . . . , a11). Generating a test paper with $n$ questions actually determines an $n * 11$ objective matrix. The sum of the particular column elements in this matrix should meet some constraints involved in total score, total time, and score distributions about question type, cognitive level, difficulty and knowledge point.

After inputting the constraint values of a test paper, the user selects “Computer Network” as the subject of the target test paper, and enters the score distribution of the involving knowledge points: physical layer (15), data link layer (15), network layer (30), transport layer (20), application layer (10), network security (5), and introductions (5). Then the “The Fifth Test Paper for Computer Network” is entered as the test paper name, the test time is 120 minutes, the total score is 100, and the expected average score is 75. The score distribution of question type is: selection (20), fill in black (22), judgment (8), correction (0), calculation (30), programming (0) and explanation (20). Additionally, the entered score distribution of the four cognitive levels is: memorizing (30), understanding (30), application (20) and synthesis (20). After pressing the “Generating” button, the system will send these constraint values containing in this web page to the background rule engine for processing.

The verification procedures of these constraint values

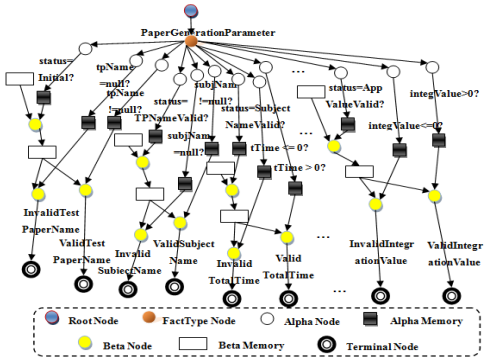



**Fig. 10** The sample Rete network constructed by the rule file.

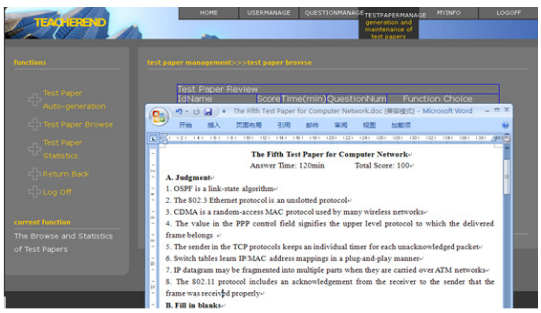



**Fig. 11** A test paper generated according to constraint values.

for generating the test paper are expressed by production rule and stored in the rule file. When it needs to add a new verification procedure or modify some existing ones, the modification is only made in the rule file. The AGTPS can adapt to the changes of the application requirements with a few modification or none, which makes the system have a better robustness and maintainability.

The AGTPS uses rule engine to compile the rule file to build the Rete network presented in Fig. 10, and adopts the improved Rete algorithm to verify the validation of the parameters for generating test papers. It then combines with the auto-generating test paper algorithm based on the improved genetic algorithm to generate the objective test paper, within an acceptable response time. Figure 11 shows an example of generated test paper.

The personal income tax calculation module of the WTCS accepts the pre-tax wage entered by users, and displays the tax and after-tax wage. The package fee calculation module of the MFCS receives the consuming information of a mobile package, processes it according certain mobile fee calculation rules, and represents the corresponding billing information. The two modules of these application system use rule engine to perform the corresponding business processing.

Considering the application scenarios of improved Rete algorithms mentioned in [4]–[11] are quite different from the involved existing EISs mentioned above, and the comparison between them with the proposed one is unnecessary, so this paper just compares the improved Rete described in Sect. 3.2 with the basic Rete which already exists

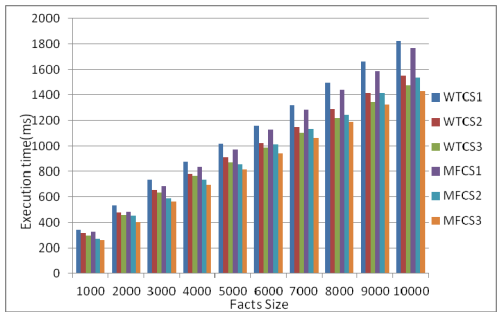



**Fig. 12** The execution time comparison of Rete algorithms in the form of basic, only with double hash and with all the two improved approaches.

**Table 1** The system configuration of the experiment.

| CPU | Genuine Intel® CPU T2080@1.73GHz |
|---|---|
| Memory | 2.5 GB |
| Hard Disk | SAMSUNG 40 GB |
| LAN | 100Mbps |
| OS | Windows XP SP2 |
| JRE | Jdk_1.6.21 |
| Web Server | Tomcat 6.0 |

in those EISs.

Figure 12 represents the total execution time spent by the corresponding business modules of the WTCS and MFCS, adopting the basic Rete and the improved Rete. The system configuration of the experiment is shown in Table 1. The system abbreviations ending with "1" are the systems using the basic Rete algorithm, such as WTCS1 and MFCS1. The system abbreviations ending with "2" are the one adopting the improved Rete algorithm only with the double hashes, such as WTCS2 and MFCS2. The system abbreviations ending with "3" are the one adopting the Rete algorithm with the two improved approaches, such as WTCS3 and MFCS3.

Overall, as for the WTCS and MFCS, under the same scale of facts, the execution time of the systems using Rete algorithm with the two improved approaches (serials '3') is generally lower than the original systems adopting the basic Rete (serials '1'). For the fact scale of 1000~4000, the former reduces slightly compared to the latter. But when the fact scale expands from 5000 to 10000, the reducing extent becomes more obvious. Under the condition of 10000 initial facts, the execution time of the WTCS drops from 1822.6ms to 1474ms, and the execution time of the MFCS decreases from 1766.6ms to 1427.3ms. The overall performance of the WTCS promotes about 14%-23%, and the one of the MFCS increases about 19%-25%.

On the other hand, under the same scale of facts and comparing with basic Rete (serial '1'), the execution time of the WTCS and MFCS is firstly dramatically decreased by the Rete only with double hashes (serial '2'), and then further decreased by the Rete with the two improved approaches (serial '3'). As for the WTCS with 9000 initial facts, the execution time of the three serials drops from 1661ms to 1412ms and 1348ms.Comparing to the basic

**Table 2** The system configuration of the cluster.

| | Configuration of Master | Configuration of Worker |
|---|---|---|
| CPU | Intel Pentium Duo E7400@2.8GHz | Intel Pentium Duo E7200@2.53GHz |
| Memory | 4GB | 2GB |
| Hard Disk | SATA 250GB | SATA 250GB |
| LAN | 100Mbps | 100Mbps |
| VM | VMware-workstation-7 | VMware-workstation-7 |
| OS of VM | Ubuntu 10.10 | Ubuntu 10.10 |
| JRE | Jdk_1.6.21 | Jdk_1.6.21 |

Rete, the Rete only with double hashes provides about 11%-18% performance promotion for WTCS, and the one of MFCS is approximately 15.4%-21.6%.

The experimental results show that the Rete algorithm with the two improved approaches can speed up the propagation of facts in the Rete network, and thus reduce the time consumed in pattern matching for the rule engine; the double hashes can promote the system performance dramatically. Guaranteeing the systems have better flexibility and maintainability, the proposed improved Rete can reduce the execution time of the enterprise information systems.

### 4.3 Performance Evaluation

Hadoop [15] is the most popular and widely used implementation of the MapReduce programming model, but we implement our prototype of the MapReduce-based distributed rule engine without Hadoop based on the following reasons: 1) the inputs are always files in scale of TBs in Hadoop-based programs, while the one in the proposed distributed rule engine are the small rule files, which cannot make full use of the advantage of HDFS; 2) the MapReduce implementation of Hadoop is based on HDFS and the computation of it is based on files, while the Rete algorithm and processing cycle of the rule engine is based on Rete network and facts in memories.

Because the value of MapReduce programming model is that it can make best use of moderate commercial machines to operate distributed parallel computation, as well as considering the available options in reality, this paper uses the cluster consisting of several PCs to construct the prototype system presented in Fig. 8, which contains master server, map workers and reduce workers. The system configuration of master server and workers is shown in Table 2. The scale of the clusters with these configurations which are obviously common and universal can be easily expanded with relatively low cost.

Most of the available PCs are already installed Windows series OS such as Win XP/Win 7, the performance evaluation needs to operate in Linux. So usage of VMs is the best way to construct a large amount of Linux environment without making incommodity to the users of those PCs.

In order to guarantee the validation of the data, the statistic of the experimental data is acquired by calculating the average values of several tests. The experiments are divided into two groups. The first group is under the condition

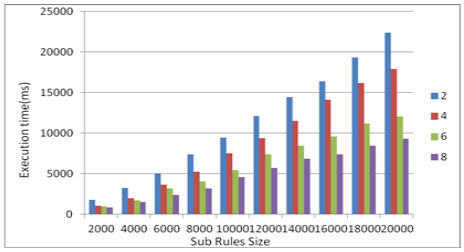



**Fig. 13** The execution time of the system with the MapReduce-based distributed rule matching.

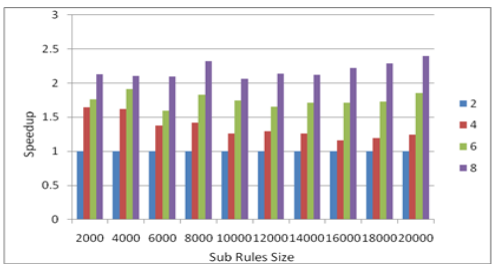



**Fig. 14** The speedup comparison of the distributed rule matching with different cluster scale.

of giving the initial 600 facts, and evaluates the execution of MapReduce-based distributed rule engine. In the first test, the system uses some conflict resolution strategy to select just one rule instantiation to execute, and does the operations similar to those described above. The second group of test evaluates the execution of the system adding the multiple rule firing method proposed in this paper, under the same conditions.

After several tests, we find that each map worker of the distributed rule engine in this paper only can compile at most 10000 rules. Under the condition of the 600 initial fact instances, the number of the sub-rules is increased from 2000 to 20000, the cluster is successively set as 2, 4, 6 and 8 map workers with 1 reduce worker, to evaluate the execution of the distributed rule engine. That means the distributed rule engine adopts the improved Rete algorithm proposed in this paper. The corresponding experimental results are shown in Fig. 13 and Fig. 14. Considering the available scale of the cluster and the processing ability threshold of each map worker, the scale of testing rules is enough to validate the effectiveness and performance of the prototype system.

Under the condition of different number of map workers and different size of subrules, the system execution time without the proposed multiple rule firing method is shown in Fig. 13, and the serial with the same color represents a scale of cluster. It can be seen from Fig. 13 that, with the growth in the number of map workers, the execution time of the system drops notably; when the number of the sub-rules increases continuously, this trend becomes more obvious. With the growth in the number of subrules, the execution time of the cluster with 2 map workers increases from 1733ms to 22315ms; while the one with 8 map workers increases from 814ms to 8317ms. The execution time of the

latter increases slowly because there are more map workers. The distributed parallel matching offsets the execution time brought by the growth of more rules.

Based on the comparison data of execution time in Fig. 13, under the same scale of the sub-rules (ranging from 2000 to 20000), the speedup presented in Fig. 14 is derived from dividing the system execution time with 2 map workers by with other scale of map workers. There is a trend that more map workers bring more speedup to the system, under the condition of processing the same scale of sub-rules. However, when the number of sub-rules reaches some scale, the speedup of the system will slow or even decline in local. There is more communication cost due to more map workers in the cluster, which gradually offsets some speedup brought by more map workers. Combining with Fig. 13, it can be seen that this drop in speedup is not so important. After all, the overall execution time of the system decreases obviously with the increase of workers. Under the condition of inputting moderate scale of initial facts, the system can meet the processing challenge brought by the dramatically increase of rules through enlarging the number of map workers.

## 5. Conclusion

This paper proposes a double hash filter approach for alpha network, combined with beta node indexing to improve Rete algorithm, with the purpose of speeding up pattern matching in rule engine based on MapReduce, and reducing the response time of EISs. This approach introduces the fact type node to Rete network. A hash map about 'fact type - fact type node' is built in root node, and hash maps concerning 'attribute constraint - alpha node' are constructed in fact type nodes. This double hash filter mechanism is used to speed up the propagation of facts in alpha network. Meanwhile, hash tables are built with indexes calculated by the parameters of fact objects entering into the input memories of the beta nodes, to avoid unnecessary iteration of the join operation in the beta nodes. The experimental result shows that the improved Rete algorithm can make the rule engine support efficient pattern matching in face of massive facts, and reduce the overall execution time for the application systems.

## Acknowledgements

This work is supported by National Natural Science Foundation of China (No.61202202), Key Science and Technology Innovation Team Project of Zhejiang Province (No.2009R50009), Major Science and Technology Project of Zhejiang Province in China (No.2009C11027) and Tsinghua - Tencent Joint Laboratory for Internet Innovation Technology.

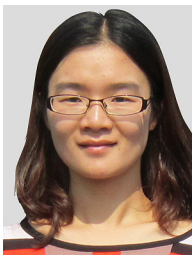
### References

[1] H. Zhang and X. Yang, "Rule engine research and implementation in financial system," Proc. 2009 Fifth International Joint Conference on INC, IMS, IDC, pp.1114–1117, 2009.
[2] M.L. Nelson, R.L. Raiden, and R. Sen, "A lifecycle approach towards business rules management," Proc. 41st Annual Hawaii International Conference on System Sciences, pp.113–123, 2008.
[3] C.L. Forgy, "Rete: A fast algorithm for many pattern/many object pattern match problem," Artificial Intelligence, vol.19, no.1, pp.17–37, 1982.
[4] R.B. Doorenbos, Production Matching for Large Learning Systems, Ph.D. Thesis, Carnegie Mellon University, 1995.
[5] L. Yan and Z. Pan, "The Improvement and Implementation of RETE algorithm," Computer Information, vol.22, no.12, pp.290–292, 2006.
[6] Z. Ren and D. Wang, "The Improvement research on rule matching algorithm rete in electronic commerce application systems," Proc. 2008 International Conference on Wireless Communications, Networking and Mobile Computing, pp.12942–12945, 2008.
[7] D. Xiao, Y. Tong, H. Yang, and M. Cao, "The improvement for Rete algorithm," Proc. 1st International Conference on Information Science and Engineering, pp.5222–5225, 2009.
[8] D. Zhou, Y. Fu, S. Zhong, and R. Zhao, "The Rete algorithms improvement and implementation," Proc. International Conference on Information Management, Innovation Management and Industrial Engineering, pp.426–429, 2008.
[9] B. Berstel, "Extending the RETE algorithm for event management," Proc. Ninth International Symposium on Temporal Representation and Reasoning, pp.49–51, 2002.
[10] K. Walzer, M. Groch, and T. Breddin, "Time to the Rescue - Supporting Temporal Reasoning in the Rete Algorithm for Complex Event Processing, Proc. 19th International Conference on Database and Expert Systems Applications, pp.635–642, 2008.
[11] K. Walzer, T. Breddin, and M. Groch, "Relative temporal constraints in the Rete algorithm for complex event detection," Proc. Second International Conference on Distributed event-based systems, pp.147–155, 2008.
[12] M. Proctor, Chiswick, E. Tirelli, S. Paulo. Beta Node Indexing in a Rule Engine. United States Patent Application Publication, 2009.
[13] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Proc. Sixth Symposium on Operating System Design and Implementation, San Francisco, pp.137–150, CA, 2004.
[14] L. Xiong and J. Shi, "Automatic generating test paper system based on genetic algorithm," Proc. Second International Workshop on Education Technology and Computer Science, vol.3, pp.272–275, 2010.
[15] T. White, Hadoop: The Definitive Guide, 3nd Edition, O' Reilly Media, 2012.

**Tianyang Dong** was born in Dongyang City, Zhejiang Province, China in 1977 and received his M.S. in Mechanical Engineering and Ph.D. in Computer Science from Zhejiang University, Hangzhou, China in 2002 and 2005, respectively. He also went to the University of Queensland, Australia, as a visiting scholar from March to September of 2011. He works for Zhejiang University of Technology since 2005. His research interest includes computer graphics and artificial intelligence.

**Jianwei Shi** was born in Jiaxing City, Zhejiang Province, in 1986 and received his B.S. and M.S. in College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou, China in 2009 and 2012, respectively. His research interests are in distributed computing and software component technology.

**Jing Fan** was born in Hangzhou City, Zhejiang Province, China in 1969 and received her M.S. and Ph.D. in Computer Science from Zhejiang University, Hangzhou, China in 1993 and 2003, respectively. She also went to Aberdeen University, United Kingdom, as a visiting scholar from August to November of 2011. She works for Zhejiang University of Technology since 2001. Her research interest includes virtual reality and software component technology.



**Ling Zhang** was born in Jingzhou City, Hubei Province, in 1980. She received her M.S. in 2004 and is currently completing her Ph.D. coursework in College of Mechanical Engineering, Zhejiang University, Hangzhou, China. She works for Zhejiang University of Science and Technology since 2004. Her research interests are in computer graphics and enterprise information system.