

A Distributed Architecture for Rule Engine to Deal with Big Data

Siyuan Zhu, Hai Huang, Lei Zhang

School of Computer Science and Technology
Beijing University and Telecommunications ,China
thinkingfar@163.com, {hhuang,zlei}@bupt.edu.cn

Abstract—Rule engine, which acknowledges facts and draws conclusions by repeatedly matching facts with rules, is a good way of knowledge representation and inference. However, because of its low computational efficiency and the limitation of single machine's capacity, it cannot deal well with big data. As traditional MapReduce architecture can only address this problem in certain conditions, we have made some improvements and therefore proposed a distributed implementation of the rule engine using MapReduce-based architecture. It is designed to deal with a large amount of data in a parallel and distributed way by using a computing cluster that consists of multiple machines, on which certain part of the Rete algorithm would be operated. In the phase of splitting rules and the Rete-net, Apriori algorithm is also improved and adopted so as to gain a better system performance. This paper not only describes details of the design and its implementation, but also shows its high performance through several experiments.

Keywords—rule engine, big data, rete algorithm, map-reduce, apriori algorithm.

I. INTRODUCTION

A rule engine (or production system) is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behaviour. These rules, termed productions, are a basic representation found useful in automated planning, expert systems and action selections. A production system provides the mechanism necessary to execute productions in order to achieve some goal for the system.

However, rule engines are computationally expensive and slow. When the size of problems continues to grow, the efficiency becomes much lower. In order to address these issues, lots of research works have been done in past few decades. One of the most important contribution was the creation of Rete by Forgy in 1982[1], which hereafter inspired lots of its improvements and modifications including Treat [2], Rete/UL [3], Rete [4]. But these could still not deal well with the situation where the number of rules and facts become too large under the limitation of one single computer's capability. Reference [5] fixed on rule firing instead of rule matching. The author of [6] did not consider the situations of mass rules matching. In [7]-[8], new approaches to implement rule engines were also proposed, using special hardware architectures or message-passing model, thus lack of flexibility.

Nowadays, the development of cloud computing offers some new potential methods to speed up rule engines when solving big data, and MapReduce technology is a popular way of dealing with big data through batch processing in cloud computing. One recent work [9]-[10] implemented a rule engine on a MapReduce-based architecture. It could not be compatible well when the rules are complex and have strong relevance, so we make use of apriori algorithm to data mining the relevance of rules, and according to it we partition the Rete-net into sub-net, and then we can perform Rete concurrently in different computers on which deployed a sub-Rete.

In this paper, we first present the background of our paper in Section II. Secondly, we discuss how we design our rule engine in Section III. Then, we discuss about the experimental evaluation in Section IV. Section V concludes the paper and points out our future work.

II. BACKGROUND

A. Rule Engine

The Rule Engine originated from rule-based expert systems, belongs to the category of artificial intelligence. It can use heuristic reasoning methods to get conclusions, mimicking human. The main task of the rule engine rule is to pattern match the facts submitted to the system and rules already in the system, and to activate those corresponding business rules. Generally, the rule engine consists of three parts: Rule Base(knowledge base), Working Memory(fact base) and Inference Engine which has three components: Pattern Matcher, Agenda and Execution Engines, as show in Figure 1.

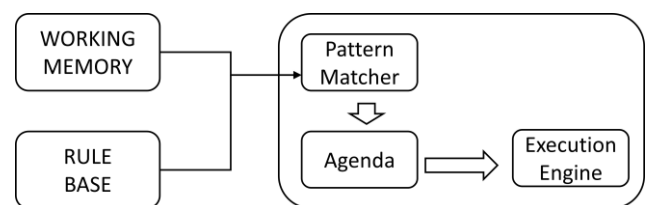


Figure 1. architecture of rule engine

B. The Rete Algorithm

The Rete algorithm describes how the Rules in the Production Memory are processed to generate an efficient discrimination network. In non-technical terms, a

discrimination network is used to filter data as it propagates through the network. The nodes at the top of the network would have many matches, and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. Drools is one of the most popular business rule engine implementation and is implemented in Java. Drools implement Rete algorithm as show in Figure 2.

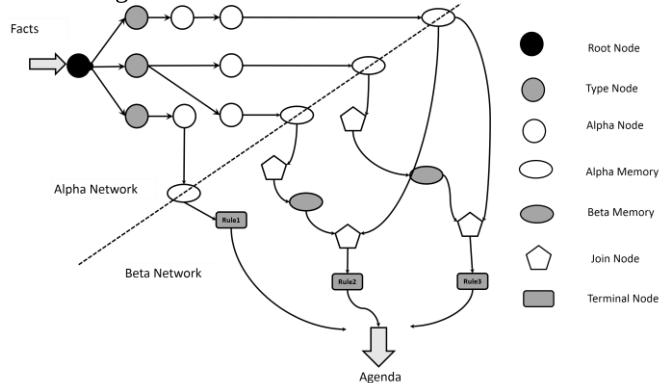


Figure 2. Rete - Network

Root Node: where all objects enter the network. From there, fact immediately goes to the Type Node.

Type Node: the purpose of the Type Node is to make sure the engine does not do more work than needed. To make things efficient, the engine should only pass the object to the nodes that match the object type.

Alpha Node: it used to evaluate literal conditions. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an Account object, it must first satisfy the first literal condition before it can proceed to the next Alpha Node.

Alpha Memory: remembers all incoming objects.

Join Node: these appear in the low layer and test for consistency satisfaction of distinct condition elements in two conditions. In more details, it tests whether the identical variable appearing in these two conditions is bound to the same value. Join Nodes also have memory. The left input is called the Beta Memory and remembers all incoming tuples.

Terminal Node: Each one appears at the bottom of the network and means the end of a rule, and actions in the RHS of the rule are stored there waiting for being triggered.

C. The Apriori Algorithm

The Apriori algorithm was proposed by Agarwal and Srikant in 1994[11]. Apriori is designed to operate on databases containing transactions (for example, collections of items bought by customers, or details of a website frequentation). Each transaction is seen as a set of items (an itemset). Given a threshold C, the Apriori Algorithm identifies the item sets which are subsets of at least C transactions in the database.

Apriori uses a "bottom up" approach, where frequent subsets are extended to one item at a time (a step known as candidate generation), and groups of candidates are tested

against the data. The algorithm terminates when no further successful extensions are found.

III.DESIGN

A. The Overall Architecture

In traditional MapReduce based architectures, each workers will deployed a whole Rete-net and the facts will dispatch to each workers randomly. But when two facts which can match a rule is dispatched to two different workers, it cannot work.as shown in Figer3 left. In our architecture we spilt the Rete-net to some sub-Retenet according to types of facts, and then deploy them on each workers, each workers only subscribe facts Involved and one type of fact can only be send to on worker as shown in Figer3 right.

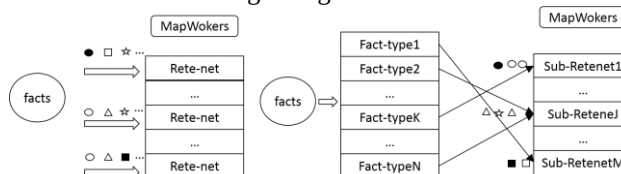


Figure 3. Traditional architecture and our architecture

As shown in figure 4, the architecture we proposed for rule matching in production system also adopted the Master/Worker model. Except for managing and monitoring all workers in the distributed environment, master node is also responsible for spilt the rules and dispatch facts. It spilt the Rete-net by spilt rules (see in **B.Rule Spilt**), and dispatch them(see in **C.Rule Dispatch**) on workers and master2 which to reduce the interim data produced by workers.

Each worker is an independent rule engine .It parses its own sub-rules set to a Rete-net (Figure.2) .Then it notices the master when it is over and subscribe facts involved .Next, rule matching begins. The master passes the facts in a queue from clients to different workers on demand. Workers will not perform matching until facts are arrived. Each map task matches assigned facts with its own all the sub rules by returning an interim data which indicates whether it is completed. If completed, the triggered rules are placed onto the agenda of this worker and executed immediately. The agenda is responsible for scheduling the execution of these triggered rules using a conflict resolution strategy. If not, send the interim data to the master. Finally, master node allocates the results to corresponding workers for the execution of reduce tasks.

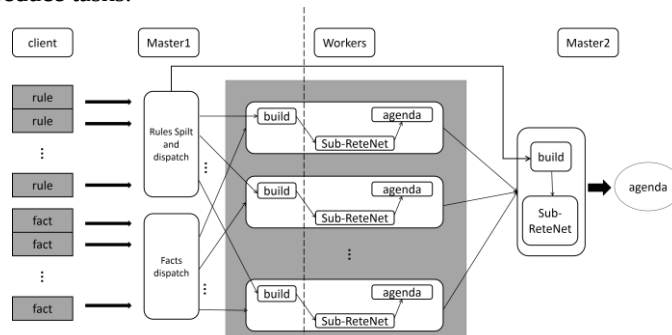


Figure 4. Overall architecture

B. Rule Spilt

Rules in rule engine can be defined as follow:

Definition 1 (Rule): A rule, denoted **R**, is a tuple (LHS, RHS), where:

1. LHS is a finite set of conditions in a rule, called the **left hand side**.

2. RHS is a finite set of actions in a rule, called the **right hand side**.

3. LHS and RHS should satisfy the structure: when LHS then RHS

As Definition 1 describes, a rule specifies that "when" a particular set of conditions occur, specified in the left hand side (LHS), then do this, which specified as a list of actions in the right hand side (RHS).

We can regard that the LHS has been divided into different parts when expressed in a composite way using logical operators. We illustrate this division with 'AND' operator.

A sample rule in Drools is shown in figure 5. the *Temperature* and *Smoke* are two types of fact. They are similar to Java Bean, a fact have many properties such as *id*, *value*...

```
rule "fire"
when
  t:Temperature(value > 40)
  s:Smoke(location = t.location && value > 50)
Then
  System.out.print("fire~")
end
```

Figure 5. A rule in Drools

Splitting the rule to sub-rules by type of fact is as shown in figure 6 left. The fire-sub1 and fire-sub2 will dispatch different workers, and then workers build those sub-rule as Rete-net. It will generate an interim result which will be send to the master as a fact. when match the *Smoke*, the property location of *Temperature* is in need, we say those two sub-rule have relevance, so we must deal with the related part in the master as shown in figure 6 right. As a result, a Complex Rule can be defined as follows:

Definition 2 (Complex Rule): A rule, denoted **R**, is a Complex Rule. Assume that:

$S_0, S_1, \dots, S_k, \dots, S_n$ ($n > 0, 0 < k < n$) are all the sub-rules corresponded to a certain rule **R**.

Rule **R** must satisfy: having a value k , when matching the condition of S_k , others properties from other sub rules are in need;

<pre>Rule "fire-sub1" when t : Temperature(value > 40) then generate(t,rule-sub1); End Rule "fire-sub2" when s : Smoke(value>50) then generate(s,rule-sub2); end</pre>	<pre>Rule "fire-final" When t:rule-sub1 (); s:rule-sub2(location = t.location) then System.out.println("fire~"); end</pre>
---	--

Figure 6. Sub-rule

According to the method mentioned above, we can spilt all rules. But when the rules are complex, there are lots of sub-rule with relevance and the master will deal with a great amount of related parts of sub-rule thus the capability will be the bottleneck of the system.

Considering if there are four types facts, A, B, C and D, 40% number of complex rules are corresponds A and B, 50% number of rules are corresponds A and B, Only 10% number of complex rules are corresponds A, B, C and D. We can treat A and B as a whole when spilt and dispatch rules. This worker only subscribe facts. A and B and when dealing with the rules only corresponds A and B can executed immediately. Similarly we treat A and B as a whole too. Only When matching the 10% number of complex rules, workers will produce a interim data to the master thus Lighten the pressure of the master. But the relevance of rules are always complex, we need to mining it.

C. Mining rule relevance

First we make preparations. For each complex rule, recording all fact types of it and writing them to database in the form of {FactTypeA, FactTypeB,...}. Then We use the Apriori algorithm to data mining the relevance among fact types of the rule, as the basis to group facts.

To select interesting rules from the set of all possible rules, constraints on various measures of significance and interest can be used. The best-known constraints are minimum thresholds on support and confidence.

Support

The support $\text{supp}(X)$ of an itemset X is defined as the proportion of transactions in the data set which contain the itemset.

$$\text{supp}(X) = \frac{\text{the number of transaction which contain the itemset } X}{\text{total number of transactions}}$$

Confidence

The confidence of a rule is defined:

$$\text{conf}(X \rightarrow Y) = \text{supp}(X \cup Y) / \text{supp}(X)$$

The Apriori algorithm is split up into two separate steps:

First, minimum support is applied to find all frequent itemsets in a database.

Second, these frequent itemsets and the minimum confidence constraint are used to form relevance set.

After finding the frequent itemsets, we compute the confidence of fact types. If $\text{conf}(\text{FactTypeA} \rightarrow \text{FactTypeA})$ and $\text{conf}(\text{FactTypeB} \rightarrow \text{FactTypeA})$ greater than the minimum confidence, we consider the FactTypeA and FactTypeB having strong relevance, and add a list contains those types to the result set.

Finally we can get a set of list, in each there will be fact types which have strong relevance.

D. Rule dispatch

By using Apriori algorithm, we can get a set of fact type list {list1, list2...listk...listN}. Then we compute the load for each list {load(list1), load(list2)...load(listk)...load(listN)} short for L. We spilt rules by the fact type and dispatch the

sub-rules to workers, thus how to group the fact types is the key point.

Assuming that there are M kinds of fact type, denoted $\{f_1, f_2 \dots f_i \dots f_M\}$ and short for F. the load of the fact type can be denoted as $\{w_1, w_2 \dots w_i \dots w_M\}$ and short for W. The worker include a list L which include fact type deployed on it. We present our calculating formula for workload of each map worker as following:

$$Load(L) = \sum_{k=0}^M w(k) \quad f_k \in F$$

By using Apriori algorithm, we can get a set of fact type list $\{list_1, list_2 \dots list_k \dots list_N\}$ short for L. Then computing the load for each list $\{load(list_1), load(list_2) \dots load(list_k) \dots load(list_N)\}$.

Assume the Sum represents the sum of all task complexity in the whole system, that is to say it is the sum of the W. T means the total map workers in the whole system. The produce as shown in figure 7.

```

procedure RuleDispatch(L, sum, T){
  remove the list whose load greater than Sum/T from L.
  n = 0
  while(L is not empty && n < T){
    pick the maximum load $i$ ( $list_i$ );
    for each factType in  $list_i$ {
      add the factType to worker(n)
      n++;
    }
    for each list denoted  $list_j$  in L{
      if ( $list_j \cap list_i \neq \text{null}$ ) remove  $list_j$  from L
    }
  }
}

```

Figure 7. Rule dispatch algorithm pseudocode

IV. EXPERIMENTS

As far as we know, there is no available standard test set of rules which could be used in our evaluation. As a result, we generate the test set of rules automatically through a program designed by ourselves and the corresponding facts are generated as the same way. All the facts we use to match the rules or sub rules are instances of the 30 fact types, and the 300 rules can be divided into different number of sub rules for certain tests. We use several physical machines to build the prototype and on which we conduct our evaluation. All the tests are running on virtual machines which installed on each physical machine. The physical machines is of Intel (R) Core (TM) i5-370 CPU, 4GB memory and Windows 7 operating system. We have 4 workers and 2 masters;

Figure 8 and Figure 9 compares the match durations for different number of facts, each series in figure represents a method: use Drools in one single computer, use map-reduce architecture but without grouping the fact types and splitting the rule randomly, use our architecture mentioned above. In Figure 6 the percentage of complex rules(Definition 3) is about 20% and in Figure 7 the percentage is about 50%. We can see that our method gains its efficiency more and more

apparently when the number of facts become bigger and bigger, especially when the number of complex rules is more.

Then Figure 10 compares the match durations with the same 1500 facts, but the rule set is different from the number of complex rules. With the rising of the percentage of complex rules, our method gains its efficiency more and more apparently.

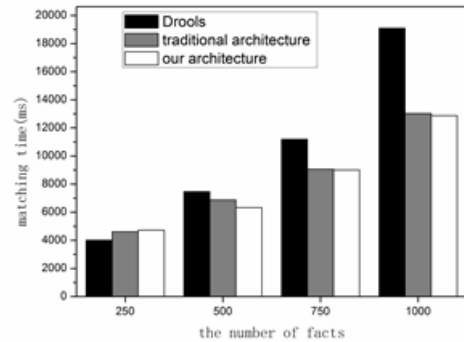


Figure 8. Rule set includes 20% complex rules

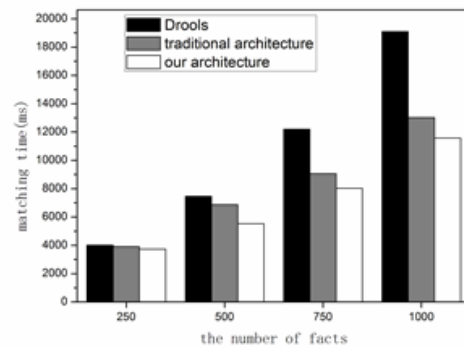


Figure 9. Rule set includes 50% complex rules

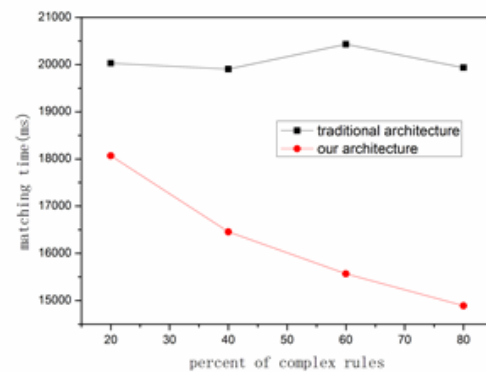


Figure 10. The matching durations in different rule sets

We can conclude that our distributed rule engine gains a less duration than Drools in one computer when dealing with big data, and with the increasing number of complex rules our distributed rule engine shows more efficient. For a larger problem size, we can infer that the execution time would be reduced significantly, and the system would gain a better performance.

V. CONCLUSIONS

In this paper, a MapReduce-based architecture for distributed rule engine and its prototype implementation are presented and studied. Conclusions coming from the relevant simulations confirm the efficiency of our architecture when dealing with big data and complex rules, which owns the properties of parallel and distribution. In order to achieve better performance scalability and adaptive, some complex factors should be considered deliberately, since the architecture is under the circumstance of distribution. As for how to make Adjustment when add some rules to the rule set or add some fact types? It will be another future work for us.

include distributed systems, cloud computing and internet of things. He is the author of a great deal of research studies published at national and international journals, conference proceedings as well as book chapters

REFERENCES

- [1] 17-37, 1982. J. Breckling, Ed., *The Analysis of Directional Time Series: Applications to Wind Speed and Direction*, ser. Lecture Notes in Statistics. Berlin, Germany: Springer, 1989, vol. 61. L. F. Charles, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp.
- [2] D. P. Miranker, *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, San Francisco, USA: Morgan Kaufmann Publishers, 1990.
- [3] B. D. Robert, *Production matching for large learning systems*, Ph.D. dissertation, University of Southern California, 1995.
- [4] W. Ian and J. A. R. Marshall, "The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case," *Int. J. Intell. Games & Simulation*, vol. 2, no. 1, pp. 36-48, 2003.
- [5] Toru Ishida. *Parallel, Distributed and Multi-Agent Production Systems*. Proceedings of the First International Conference on Multiagent Systems. San Francisco, USA, 1995.
- [6] C. Wu, L. Lai, Y. Chang. *Parallelizing CLIPS-based Expert Systems by the Permutation Feature of Pattern Matching*. 2nd International Conference on Computer Engineering and Applications. Bali Island, Indonesia, 2010. pp.214-218.
- [7] Anoop Gupta, Charles L. Forgy, Dirk Kalp, Allen Newell, Milind Tambe., *Parallel OPS5 on the Encore Multimax*. Proceedings of International Conference on Parallel Processing. 1988.
- [8] Wang, Jinghan, et al. "A Distributed Rule Engine Based on Message-Passing Model to Deal with Big Data." *Lecture Notes on Software Engineering* 2.3 (2014): 275-281.
- [9] Cao, Bin, et al. "A MapReduce-based architecture for rule matching in production system." *Cloud Computing Technology and Science (CloudCom)*, 2010 IEEE Second International Conference on. IEEE, 2010.
- [10] Li Y, Liu W, Cao B, et al. An efficient MapReduce-based rule matching method for production system[J]. *Future Generation Computer Systems*, 2015.
- [11] Agrawal R, Srikant R. Fast algorithms for mining association rules[C]//Proc. 20th int. conf. very large data bases, VLDB. 1994, 1215: 487-499.

Siyuan Zhu is a master student in computer science and technology at the Beijing University of Posts and Telecommunications of China (BUPT). He was born in China in 1990. He received his bachelor degree at BUPT in 2013. His research interests include cloud computing, rule-based computing and internet of things.

Hai Huang is a lecturer in the School of Computer Science and Technology at the Beijing University of Posts and Telecommunications of China (BUPT). He was born in China in 1979. He received his P.H.D. degree of engineering in computer science at BUPT. His research interests include internet of things, cloud computing and service software.

Lei Zhang is a professor in the School of Computer Science and Technology at the Beijing University of Posts and Telecommunications of China (BUPT). He was born in China in 1962. He received his Master degree of engineering in computer science at BUPT in 1988. His research interests