A Distributed Rule Engine for Streaming Big Data

Debo Cai[1(✉)], Di Hou[1], Yong Qi[1], Jinpei Yan[1], and Yu Lu[2]

[1] Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China
yushisx@163.com
[2] Troops 69064 of PLA, Xinjiang, China

Abstract. The rules engine has been widely used in industry and academia, because it can separate the rules from the execution logic and incorporate the features of expert knowledge. With the advent of big data era, the amount of data has grown at an unprecedented rate. However, traditional rule engines based on PCs or servers are hard to handle streaming big data owing to limitation of hardware performance. The structured streaming computing framework can provide new solutions for these challenges. In this paper, we design a distributed rule engine based on Kafka and Structured Streaming (KSSRE), and propose a rule-fact matching strategy using the Spark SQL engine to support a large number of event stream inferences. KSSRE uses DataFrame to store data and inherits the load balancing, scalability and fault-tolerance mechanisms of Spark2.x. In addition, in order to remove the possible repetitive rules and optimize the matching process, we use the ternary grid model [1] for representing rules and design a scheduling model to improve the memory sharing in the matching process. The evaluation shows that KSSRE has a better performance, scalability and fault tolerance based on DBLP data sets.

Keywords: Rule engine · Spark2.x · Event stream

1 Introduction

The rules engine simulates the decision process of a human expert and handles events and triggers corresponding actions based on prior knowledge in the pre-set rule base. Because the rules engine separates the rules from the execution logic, and the interface with expert experience is friendly, it has been successfully applied in insurance and insurance claims, bank credit and many other areas. With the development of information technology, big data has become one of the main themes of the information age. For example, Mobike, which is based on the Internet of Things (IoT), officially announces that the average amount of data generated per minute is close to 1G. How to perform multi-dimensional analysis and processing of a large number of data streams in real time and accurately will be a serious challenge for the rule engine to adapt to development.

In order to solve the above problems, many researchers have designed a distributed rule engine based on big data processing frameworks such as Hadoop and Spark to

X. Meng et al. (Eds.): WISA 2018, LNCS 11242, pp. 123–130, 2018.
https://doi.org/10.1007/978-3-030-02934-0_12

improve the matching efficiency. However, these solutions also have their own imperfections. Referring to these scenarios, based on the Kafka and Structured Streaming computing framework, we designed and implemented a distributed rules engine (KSSRE) to support a large number of event flow inference. The purpose is to improve the matching efficiency of the rules engine and achieve better load balancing and fault tolerance. Using the Kafka clustering feature to decouple the event flow, a relatively efficient rule-fact matching strategy is designed and implemented on the Spark SQL engine. At the same time, in order to improve the calculation rate, use DataFrame/DataSet which is better than RDD in both time and space to store data. In order to remove the possible repetitive rules and optimize the matching process, we improved the ternary grid model for representing rules, and designed a scheduling model to improve the memory sharing in the matching process. In addition, because KSSRE is based on Structured Streaming, it inherits the load balancing, scalability, and fault-tolerance mechanisms of Spark 2.x.

The rest of the paper is organized as follows. Section 2 provides some background information and explains related work. Section 3 elaborates on the design and implementation of KSSRE. In Sect. 4, we use the DBLP data set to conduct an experimental analysis of the KSSRE. Section 5 concludes the paper and discusses future work.

2 Background and Related Work

2.1 Rule Engine

The rule engine usually consists of three parts, namely rule base, fact collection, and inference engine. The fact is that there is a multiple relationship between objects and their attributes. Rules are inferential sentences that consist of conditions and conclusions. When facts meet the conditions, the corresponding conclusions are activated. The general form of the rule is as follows:

```
Rule_1:  /* Rule Name*/
   Attributes /* Rule-Attributes*/
   LHS /* conditions*/ => RHS /* actions*/
```

The LHS is a condition and consists of several conditions. It is a generalized form of known facts and a fact that it has not been instantiated. The RHS is a conclusion and consists of several actions.

2.2 Apache Kafka and Structured Streaming

Apache Kafka [2] is a distributed streaming platform, which consists of Producer, Kafka cluster and consumer. Producer publishes the message to the specified topic according to the set policy. After receiving the message from the Producer, the Kafka cluster stores it on the hard disk. The Consumer pulls the data from the Kafka cluster and uses the offset to record the location of the consumption. Kafka guarantees high

processing speed while guaranteeing low latency and zero loss in data processing. Even with terabytes of data, it can guarantee stable performance.

Structured Streaming [3] is a real-time computational framework for Spark 2.x. It uses DataFrame to abstract data. DataFrame is a collection of Row objects (each Row object represents a row of records) and contains detailed structural information (patterns). Spark clearly knows the structure and boundaries of the dataset, so that it is easy to implement the exactly-once of the data at the framework level. In particular, Structured Streaming re-uses its Catalyst engine to optimize SQL operations, which improves computational efficiency.

2.3 Related Work

With the rise of big data and the IoT, some researchers based on the Hadoop framework to decompose the rules and map the matching tasks into the Map and Reduce processes in the cluster and obtain the matching results [4, 5]. Zhou and other researchers use the message passing model to transform the matching process of rules into messages between processes, and implement parallel and distributed reasoning [6]. Researchers such as Chen and others used Spark 1.x's stream data calculation framework to map rules and facts to Dstream operations for event stream processing [7]. Researchers such as Zhang and others used Spark's process and relational API to map the matching process of rules and facts to the operation of an enhanced RDD, which is DataFrame, and achieved parallel distribution rule matching [8]. Referring to these scenarios, we have designed and implemented a distributed rule engine based on Kafka and Structured Streaming for reasoning on a large number of event streams.

3 Implementation and Optimization of the KSSRE

3.1 Overall Design

The overall design of the KSSRE is shown in Fig. 1 and consists of three parts.

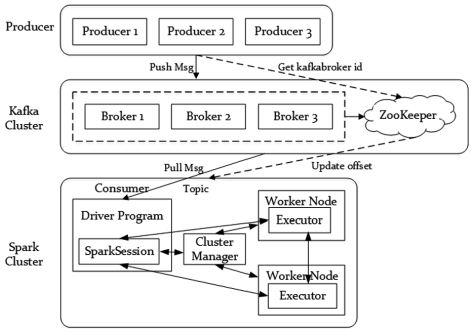

Fig. 1. KSSRE architecture

Producer is the source of real-time data generation. The Kafka cluster receives real-time events from Producer, which decouples these real-time data and performs pre-processing. The Spark cluster pulls data from the Kafka cluster and processes it to generate inference results.

3.2 Inference Process

KSSRE decoupled and preprocess event flows through Kafka clusters. Based on the Structured Streaming real-time computing framework, we use DataFrame/DataSet which is better than RDD in both time and space to store data. We designed and implemented an effective rule-fact matching strategy, converting the rules to SQL operations and using the Catalyst engine to optimize the SQL operations, ultimately achieving inference.

As shown in Fig. 2, KSSRE divides the inference process into four stages of "Hash-Filter-Trigger-Select", and implements inference by periodically cycling through four stages.

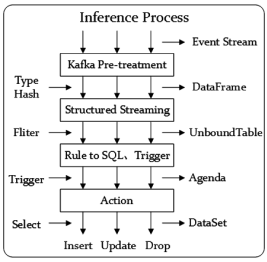

Fig. 2. Inference process.

- The first layer is the Kafka data preprocessing layer that implements asynchronous processing of data producers and consumers.
- The second layer is the Structured Streaming data filter layer, which implements the matching of the LHS part of the rules and the facts.
- The third layer is the rule preprocessing layer, which implements conversion query and conflict resolution from rules to SQL statements.
- The fourth layer is the SQL execution layer, which executes all SQL statements in Agenda and produces a Result Table.

3.3 Optimization

In the actual experiment, the efficiency of the above algorithm is low. There are two main reasons for this: First, there may be duplication of rules in the rule base. In addition, in the Filter stage, all rules need to be filtered in turn, and for rules that have mutually exclusive conditions, there will be a lot of redundancy. Second, the use frequencies of conditions in the LHS part of the rules are different. The degree of

memory sharing corresponding to different execution orders has a great influence on the matching efficiency. For these two reasons, we directionally optimize the algorithm.

On the one hand, the rules are handled in advance. As shown in Fig. 3, we have designed a new rule storage style from the ternary grid [1]:

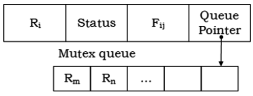

Fig. 3. The representation of a rule.

R_i stands for the rule and i is the rule ID. F_{ij} indicates the conditions contained in the rule. Status uses "*−1/1/0/−2/2*" to indicate the current status of the rule. "*0*" indicates unused, "*1*" indicates that F_{ij} is a sub-condition in the LHS part of the rule R_i. "*−1*" indicates that F_{ij} is a negative form of the sub-condition with the number j. "*2*" indicates that F_{ij} is a sub-statement in the RHS part of the rule R_i. "*−2*" indicates that F_{ij} is a negative form of the sub-sequence number j. The Queue Pointer is a pointer to a rule ID queue in which there is an exclusive sub-condition with the rule R_i. The ternary grid is mainly to convert rules into rule matrices to eliminate duplicate rules and meaningless rules. The improved model not only pre-processes the rules, but also reduces the number of rules and facts in the matching process through the Mutex Queue.

Algorithm **1** The Inference Engine Algorithm

Input: Event Stream
Output: DataSet

```
function PRETREATMENT(R1, R2, ...Rn)
    L is the length of all facts
    F[1][L] ← R1.LHS.split
    F[2][L] ← R2.LHS.split
    ...
    F[n][L]R←.Rn.LHS.split
    for i ← 1 to n do
        for j ← (i + 1) to (n − 1) do
            if F[i][L] and F[j][L] are mutually inverse then
                Ri.Mutexqueue ← Ri.Mutexqueue.append(Rj)
                Rj.Mutexqueue ← Rj.Mutexqueue.append(Ri)
            end if
        end for
    end for
end function

procedure INFERENCE(DataSet, Rules)
    rules ← Rules.sortBy(priority)
    for i ← 1 to rules.length do
        if DataSet.filter(rules[i]) is not null then
            rules ← rules.drop(rules[i].inverse)
            ConflictSet ← rules[i]
        end if
    end for
    Agenda ← ConflictSet.sortBy(priority)
    for j ← 1 to Agenda.length do
        DataSet.Select(Agenda[j])
    end for
end procedure
```

The second is that for the problem that the rule LHS partial use frequency influences efficiency differently, we can directly set the priority of the rule through the conditional use frequency. But this is a simple optimization method in the ideal case where the LHS part of the rule contains only one condition. To this end, we have

established a scheduling model. Trigger the execution sequence by changing the rules to maximize the reuse of existing matching results. The scheduling model of the rule is:

$$R_p = \frac{1}{3}\sum_{i=1}^{n}(C_i + C_m + n) \tag{1}$$

Among them, C_i is the frequency of use of each condition. n is the number of conditions contained in the LHS. C_m is the most frequently used condition of all LHS conditions. 1/3 indicates that these three factors each occupy the weight of the scheduling model. R_p reflects the frequency of use of the LHS part of rule i in the rule base. Finally, we optimize the Filter process based on the scheduling model. The pseudo code of the optimization algorithm is shown in Algorithm 1.

4 Experiments

In this section, we refer to the OpenRuleBench [9] and use the DBLP [10] (Digital Bibliography & Library Project) data set to perform an experimental analysis of the KSSRE in terms of both performance and scalability. DBLP is an English literature database in the field of computers. As of January 1, 2018, more than 6.6 million papers and more than 2.1 million scholars were included. The rules use the four parts of the DBLP filter, negative filter, and join operations to generate a total of 20 valid rules.

Query (Id, T, A, Y, M):
- att (Id, title, T), - att (Id, year, Y)
-att (Id, author, A), -att (Id, month, M)

The experiment was based on three servers. The CPU of each server is Inter E3-1231*8 and the memory is 8 G. The servers were interconnected via Gigabit Ethernet and the operating system was Centos 7.2. Each server runs three virtual machines totaling nine nodes. In addition, considering that 90% of the time in the production system is used for matching [11], we use match time as an indicator of performance testing.

4.1 Performance Testing

Performance testing is mainly to compare KSSRE with Drools [12]. Deploying KSSRE on 3 virtual machines on 1 server is compared with Drools 6.5 deployed on another server.

As shown in Fig. 4, Drools has better processing performance than KSSRE when the number of facts is less than 1 million. Drools and KSSRE has similar performance when the amount of data increases to 1 million. When the amount of data increased to 2 million, KSSRE began to provide slightly better performance. When the number of facts continues to increase to 5 million, Drools cannot handle it and KSSRE finishes processing in about 80 s. Although we can make Drools continue to work with large amount of data in a way that improves hardware performance, it is clear that Drools'

memory model is less flexible in terms of garbage collection, which makes it impossible to handle large data sets.

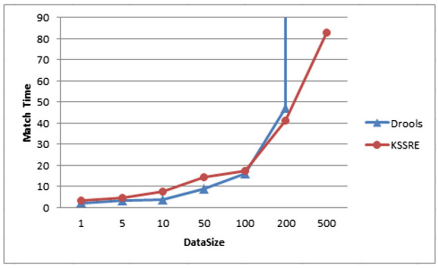

Fig. 4. Performance testing.

4.2 Extensibility Test

Extensibility testing extends the Spark cluster within the same facts, rules, and time intervals to change the number of cluster nodes and record the processing time.

As shown in Fig. 5, matching times for different scales of facts are shown. It is obvious that the matching time decreases as the number of nodes increases. With the same order of magnitude of the fact, the matching time decreases as the number of cluster nodes increases. And on a larger scale of fact, the effect is even more pronounced. Therefore, we can improve the matching efficiency by simply scaling the cluster. In addition, the graph can reflect that when the cluster increases from 3 to 5 nodes, the KSSRE matching time decreases the most. The decrease in the matching time for the change in the number of other nodes is reduced. This is because when the number of nodes of the cluster is 3, the data communication is the internal communication of the server, and when the number of the nodes is more than 3, the cost of the network communication between the nodes needs to be considered.

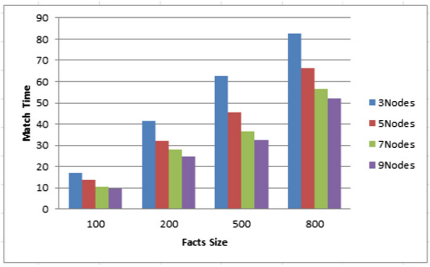

Fig. 5. Comparison of match time.

5 Conclusion

Based on Kafka and Structured Streaming, we extended the general algorithm of the inference engine and implemented a prototype system of distributed rule engine, which is suitable for streaming big data. According to the experimental situation, the rule storage trinomial mesh model is improved and a scheduling model is designed to remove repetitive rules and optimize the matching process. Most importantly, KSSRE is characterized by the reliability, scalability, and fault-tolerance of Structured Streaming. Finally, experimental results show that KSSRE not only supports large-scale factual data inference, but also can achieve better performance by extending the number of cluster nodes.

In the future, we plan to improve the performance of the rules engine from the aspects of optimizing conflict resolution strategies and improving the degree of memory sharing during matching. Moreover, the reasoning model of the current KSSRE is relatively simple and does not consider the time of data generation (Event-time in the data). In addition, we also plan to write inference results to storage or display in real time.

Acknowledgment. This work is partially supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000600.

References

1. Erdani, Y.: Developing algorithms of ternary grid technique for optimizing expert system's knowledge base. In: 2006 Seminar Nasional Aplikasi Teknologi Informasi (2006)
2. Apache Kafka. http://kafka.apache.org/. Accessed May 2018
3. Structured Streaming. http://spark.apache.org. Accessed May 2018
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
5. Cao, B., Yin, J., Zhang, Q., Ye, Y.: A MapReduce-based architecture for rule matching in production system, pp. 790–795. IEEE (2010)
6. Zhou, R., Wang, G., Wang, J., Li, J.: RUNES II: a distributed rule engine based on rete network in cloud computing. Int. J. Grid Distrib. Comput. **7**, 91–110 (2014)
7. Chen, Y., Bordbar, B.: DRESS: a rule engine on spark for event stream processing, pp. 46–51. ACM (2016)
8. Zhang, J., Yang, J., Li, J.: When rule engine meets big data: design and implementation of a distributed rule engine using spark, pp. 41–49. IEEE (2017)
9. Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: an analysis of the performance of rule engines. In: Proceedings of the 18th International Conference on World Wide Web, pp. 601–610. ACM (2009)
10. DBLP: computer science bibliography. http://dblp.uni-trier.de/db/. Accessed May 2018
11. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. Artif. Intell. **19**, 17–37 (1982)
12. Drools. https://www.drools.org/. Accessed May 2018