ChinaRxiv

AI translation · View original & related papers at
chinarxiv.org/items/chinaxiv-201810.00041

Storage-Improved Partitioned Parallel Association Rule Mining Algorithm (Postprint)

Authors: Wang Yonggui, Xie Nan, Qu Haicheng

Date: 2018-10-11T00:00:00+00:00

Abstract

Association rule mining is attracting widespread attention in the field of big data mining, where the key focus and challenge of algorithms lies in mining frequent itemsets. Addressing the situation where existing algorithms suffer from simple storage structures, generation of numerous redundant candidate sets, high time and space complexity, and unsatisfactory mining efficiency, this paper proposes an association rule mining algorithm based on improved memory structures. The algorithm leverages the Spark distributed framework to mine frequent itemsets through partitioned parallel processing. It introduces the use of Bloom filters for item storage during the mining process and performs simplification operations on both transaction sets and candidate sets, thereby achieving the objectives of optimizing frequent itemset mining speed and conserving computational resources. Under conditions of reduced memory consumption, the algorithm demonstrates significant improvements in frequent itemset mining efficiency compared to YAFIM and MRApriori algorithms. The algorithm not only effectively enhances mining speed and alleviates memory pressure, but also exhibits excellent scalability, enabling its application to larger-scale datasets and clusters, thus achieving the goal of optimizing algorithmic performance.

Full Text

Partitioned Parallel Association Rule Mining Algorithm Based on Storage Improvement

Wang Yonggui, Xie Nan†, Qu Haicheng
(School of Software, Liaoning Technical University, Huludao, Liaoning 125105, China)

Abstract: Association rule mining is attracting widespread attention in the field of big data mining, where the key challenge lies in mining frequent itemsets. Existing algorithms suffer from simple storage structures, generation of

numerous redundant candidate sets, high time and space complexity, and suboptimal mining efficiency. To further improve the speed of frequent itemset mining and optimize algorithm performance, this paper proposes an association rule mining algorithm based on improved memory structures. Built upon the Spark distributed framework, the algorithm partitions and mines frequent itemsets in parallel, utilizing Bloom filters for item storage during the mining process while performing simplification operations on both transaction sets and candidate sets to optimize mining speed and conserve computational resources. Compared with YAFIM and MRApriori algorithms, our approach demonstrates significant improvements in frequent itemset mining efficiency while consuming less memory. The algorithm not only enhances mining speed and reduces memory pressure but also exhibits excellent scalability, enabling its application to larger-scale datasets and clusters, thereby achieving the goal of optimized algorithm performance.

Keywords: association rule; big data; candidate set; Bloom filter; Spark

0 Introduction

Association rules were originally developed to discover interesting relationships among products in market basket analysis and have gradually become one of the most important methods in data mining [1]. The objective of association rule mining is to uncover hidden relationships among items in large datasets, with the primary focus being the identification of frequent itemsets—collections of items that frequently co-occur. The most classical and influential algorithm in association rule mining is the Apriori algorithm proposed by Agrawal et al. [2], which employs a level-wise search iterative approach to efficiently and accurately mine association rules.

Numerous improved Apriori algorithms have since emerged, but most adopt serial execution methods that only perform well when the dataset is small. With the advent of the big data era, as the number of items in datasets increases, the required storage space grows larger and retrieval speeds become slower. Various standalone Apriori implementations can no longer satisfy demands for time and efficiency.

Researchers recognized the high parallelism potential of the Apriori algorithm, leading to the introduction of parallel algorithms [3–5] for more efficient frequent itemset mining. While cluster-based association rule algorithms can improve the efficiency of processing transaction datasets, they suffer from complex algorithmic structures and issues such as synchronization and data replication. Studies have found that implementing association rule mining based on MapReduce yields better frequent itemset mining efficiency, and parallel algorithms have subsequently been adapted to the MapReduce framework. MapReduce-based Apriori implementations [6–8] have demonstrated significant performance gains over traditional Apriori. Apache Hadoop [9] represents one of the best platforms for the MapReduce model. Literature [10,11] presents Apriori algorithm

implementations based on the Hadoop platform, which parallelize the Apriori algorithm using MapReduce, store datasets in HDFS, and discover frequent itemsets in massive data, with experiments showing clear advantages over conventional Apriori algorithms.

However, Hadoop-based Apriori implementations still face limitations. On the Hadoop platform, each iteration stores results to HDFS and retrieves input from HDFS for the next iteration, causing performance degradation. Literature [12] also notes that MapReduce in Hadoop suffers from excessive task startup and disk I/O overhead during iterative computations, reducing execution efficiency. Researchers subsequently proposed the Spark platform, which effectively addresses these challenges. Spark [13] solves these problems through its Resilient Distributed Dataset (RDD) architecture, which stores results in local cache at the end of each iteration and makes them available to the next iteration. This motivates our proposal to improve the Apriori algorithm based on the Spark platform. Spark performs in-memory computing, overcoming the aforementioned Hadoop platform issues and making it more suitable for online, iterative, and data stream algorithms.

Built on Spark, Zhang et al. [14] proposed a distributed frequent itemset mining algorithm for big data analytics that significantly outperforms Hadoop-based Apriori implementations. However, these algorithms rely on data structures such as linear lists, linked lists, and trees, where the relative positions of data records in the structure are random, requiring extensive keyword comparisons during lookups. Consequently, lookup efficiency heavily depends on the number of comparisons performed, and efficiency declines as data volume grows.

Qiu et al. [15] proposed the YAFIM algorithm, which partitions and mines frequent itemsets in parallel based on the Spark framework. This algorithm stores candidate sets in a hash tree and is considered one of the better Apriori implementations. Experimental results show YAFIM is many times faster than Hadoop-based algorithms. However, the hash tree-based storage improvement for Apriori requires first generating candidate sets through self-joining before storing them in the hash tree. When datasets are large, hash functions are prone to address collisions, requiring substantial memory for each iteration, with memory consumption becoming particularly severe during the second iteration when the maximum number of candidate sets is generated, significantly impacting algorithm efficiency.

Existing association rule algorithms employ simple data structures, resulting in low efficiency when processing large transaction datasets. All transactions always reside in the file system or database, requiring the transaction database to be scanned during each iteration. When mining frequent itemsets, the Apriori algorithm generates numerous item collections, leading to excessive I/O overhead and system resource consumption, resulting in poor temporal performance and efficiency. During the second iteration, if there are n frequent 1-itemsets, self-joining generates $2n$ candidate 2-itemset collections. The number of candidate sets and computational resources consumed during this phase are the highest

throughout the entire mining process, often causing the second iteration to take excessively long or even fail to complete.

With large data volumes, frequent addition and deletion of itemsets in data structures can cause structural degradation into linked list form. During iterative processes, storing massive data in memory places enormous pressure on servers, inevitably degrading frequent itemset mining performance. Therefore, efficiently determining whether an item exists in a dataset is key to improving algorithm execution speed. Bloom filters use only a few bits at a low level to represent an element, and both item insertion and query times are constant, offering tremendous advantages in space and time. This makes Bloom filters highly suitable for parallel Apriori implementation. Based on this, we propose applying Bloom filters to the Apriori algorithm to improve its storage structure for efficient item existence checking. Our algorithm, built on the Spark distributed framework, partitions large datasets into similarly-sized, non-overlapping partition datasets for parallel frequent itemset acquisition. This approach significantly saves memory space while substantially improving data processing efficiency.

1 Related Knowledge

1.1 Apriori Algorithm

The Apriori algorithm is a method for mining Boolean association rule frequent pattern sets [16]. It first scans the transaction database once to count all 1-itemsets, then filters out those below the minimum support threshold to obtain frequent 1-itemsets L_1. The algorithm then iterates: frequent (k-1)-itemsets L_{1} undergo join and pruning operations to generate candidate k-itemsets C, which are then filtered by minimum support to obtain frequent k-itemsets L. This process continues until L becomes empty, at which point iteration stops and rules are output. The core idea is to obtain frequent itemsets through level-wise search iterations using Theorem 1.

Theorem 1. If a pattern is not frequent, then all supersets of that pattern are also not frequent.

1.2 Spark Platform

Spark is a big data processing platform built around speed, ease of use, and sophisticated analytics, providing a comprehensive and unified framework. Developed for interactive queries and iterative algorithms, Spark optimizes processing steps in big data workflows through delayed computation of queries. It supports in-memory storage and efficient fault tolerance mechanisms, storing intermediate results in memory rather than writing to disk, elevating MapReduce to a higher level. Under the Spark platform, machine learning algorithms requiring multiple iterations can be parallelized, reducing time and space complexity while achieving more efficient results on standard datasets, making Spark well-suited for implementing the Apriori algorithm.

Our algorithm improves the Apriori algorithm based on a distributed platform built on the Spark big data framework. While ensuring result accuracy, it solves the memory resource constraints of standalone systems and significantly improves time performance. In the big data context, the algorithm can perform association rule mining quickly and accurately.

1.3 Bloom Filter

A Bloom filter [17] is a probabilistic data structure consisting of a long binary vector and a series of random hash functions, used to determine whether an element is in a set. We construct a Bloom filter as an M-bit array initialized to 0 and a set of hash functions $H(x) = \{h_0(x),\ h_1(x),\ h_2(x)\cdots\}$. For a transaction $T = \{A,\ B,\ C\}$, each item in the transaction is hashed by K different hash functions to K random positions in the array, which are set to 1.

To determine whether an item w is in transaction T, we hash w into points in the bit array using the hash functions. If all these points are 1, the element is in the set; otherwise, it is not. For example, to check if item w belongs to the set, we hash w using three hash functions. If any address value is 0, we can conclude that w is not in the set. An example is shown in Figure 1 [Figure 1: see original paper].

Bloom filters may produce false positives when determining item membership—an item not belonging to the transaction may have all hash positions equal to 1. The probability p of false positives after mapping is:

$$p = \left(1 - e^{-Kn/M}\right)^K$$

Let q be defined as:

$$q = e^{-Kn/M}$$

Then p can be rewritten as:

$$p = (1 - q)^K$$

According to symmetry principles, p reaches its minimum when $q = 0.5$, i.e., when $Kn/M = \ln 2$. Our algorithm adapts to the dataset based on these conditions, adaptively selecting the number of hash functions K and array size M according to the number of frequent 1-itemsets, thereby reducing unnecessary hashing operations and minimizing the false positive rate while avoiding wasted time and space.

2 Algorithm Implementation

2.1 Candidate Set Simplification

Our algorithm generates candidate sets based on the Spark framework. First, we compress the transaction set. Since C must consist of k items, transactions with fewer than k items cannot generate candidate set C , so we delete transactions containing fewer than k items. We then compress items within transactions, keeping only frequent single items stored in the Bloom filter according to Theorem 2, and compose all possible k-itemsets from these frequent single items through a map() function. Finally, we compress k-itemsets based on Corollary 1, filtering out k-itemsets that do not appear exactly $k(k\text{-}1)/2$ times.

Theorem 2. All non-empty subsets of a frequent itemset are frequent.

Corollary 1. A frequent k-itemset L consists of k (k-1)-itemsets and appears exactly $k(k\text{-}1)/2$ times.

Proof. Assume a frequent 3-itemset $\{A, B, C\}$ could be formed by only two frequent 2-itemsets $\{A, B\}$ and $\{A, C\}$. Then $\{A, B, C\}$ would appear only once, implying $\{B, C\}$ is not frequent, which contradicts Theorem 2. Therefore, $\{A, B, C\}$ must be formed by $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$, appearing three times. This holds for frequent k-itemsets ($k > 3$) as well, proving the corollary.

Our algorithm no longer generates redundant candidate sets through join and prune steps, instead simplifying both transaction sets and candidate sets. This reduces the number of transactions, items, and candidate sets that need to be scanned during each iteration, thereby improving mining efficiency and saving computational resources.

2.2 Algorithm Implementation

Our algorithm operates based on Spark' s internal mechanisms. It reads transaction datasets from HDFS and loads them into Spark RDDs, leveraging cluster memory for parallel computation on resilient datasets. Datasets are stored as RDDs, using the textFile operator to scan specified datasets and set partition counts, dividing them into similarly-sized, non-overlapping data blocks assigned to each worker node for processing. The entire Spark programming framework is based on RDD operations, combining flatMap, map, reduceByKey, and filter operators for frequent itemset mining, implemented in Scala. During frequent itemset mining, each item in frequent itemsets is stored in a Bloom filter. The frequent itemset mining process is shown in Figure 2 [Figure 2: see original paper].

The mining process consists of two phases: acquiring frequent 1-itemsets and iteratively generating frequent k-itemsets.

Phase 1 identifies all frequent 1-itemsets from the input dataset in HDFS. First, we apply the flatMap() function to each partition to obtain individual transactions, then apply flatMap() to each transaction to extract individual items. We

then apply the map() function to generate <key, value> pairs where the key represents each item and the value is set to 1. Using reduceByKey(), we obtain item counts <item, count>, and finally filter() to select items exceeding the minimum support threshold. Results are stored in Spark RDD, and frequent 1-itemsets are stored in a Bloom filter. Algorithm 1 shows the frequent 1-itemset acquisition process, with an example illustrated in Figure 3 [Figure 3: see original paper].

Algorithm 1. Frequent 1-Itemset Acquisition

Input: Dataset D, minimum support min_{sup}
Output: Frequent 1-itemsets L_1

1. For each transaction T D
2. flatMap(T.split(" "))
3. For each item i T
4. `yield (*i*, 1)`
5. End flatMap
6. Store at RDD_1
7. $RDD_2 = RDD_1$.reduceByKey(_ + _)
8. For each tuple t RDD_2
9. flatMap(t, count)
10. If t.count $\geq$ min_{sup}
11. yield (t, count)
12. End flatMap()
13. Store at RDD_3

Phase 2 begins iteration to generate k-itemsets from frequent (k-1)-itemsets. First, we apply flatMap() to the dataset in RDD to obtain each transaction, then map() to prune transactions by deleting those with fewer than k+1 items and retaining only itemsets present in the Bloom filter. For pruned transactions, map() generates all possible k-itemset pairings, creating <key, value> pairs where the key represents k-item candidate patterns and the value is the set of all k-item candidate patterns in the transaction (set to integer 1), keeping only those appearing $k(k+1)/2$ times. Using reduceByKey(), we combine all key-value pairs and obtain counts for each candidate pattern in <itemset, count> format. Finally, filter() selects sets exceeding the minimum support threshold, storing frequent K+1 itemsets in Spark RDD and Bloom filter. Iteration ends when no more frequent itemsets are generated, and all frequent itemsets are merged. Algorithm 2 shows the frequent k-itemset acquisition process, with an example for frequent 2-itemset generation illustrated in Figure 4 [Figure 4: see original paper].

Algorithm 2. Frequent *k*-Itemset Acquisition

Input: Transaction database D, frequent (k-1)-itemsets L_{1}
Output: Frequent k-itemsets L

1. L_{1}.storeInBloomFilter
2. For each transaction $T \in D$
3. flatMap(T.split(" "))
4. item = intersect(T, L_{1})
5. C = map(item => (itemSet, 1))
6. For each $s \in C$
7. `*C* = intersect(*C*, *s*.count = *k*(*k*-1)/2))`
8. End flatMap
9. Store at RDD_1
10. $RDD_2 = RDD_1$.reduceByKey(_ + _)
11. For each itemSet $\in RDD_2$
12. flatMap(itemSet, count)
13. If itemSet.count $\geq$ min_{sup}
14. `yield (itemSet, count)`

2.3 Algorithm Complexity Analysis

Let f be the number of frequent (K-1)-itemsets L_{1}, x be the number of transactions, m be the number of Mappers, g be the average number of items per transaction, n be the number of candidate sets generated in the K-th iteration, t_1 be the time to search an element in a hash tree, t_2 be the time to search an element in a Bloom filter, t_3 be the time for transaction and item compression, b_1 be the bytes per candidate set, and b_2 be the bytes per single item.

2.3.1 Time Complexity For the K-th iteration generating frequent K-itemsets from frequent (K-1)-itemsets, the time components are:

- Candidate set generation time (join and prune): $T_1 = f \times n$
- Time to store candidate sets in hash tree: $T_2 = f \times n$
- Key-value pair generation time: $T_3 = t_1 \times x \times g \,/\, m$

The total time complexity for the hash tree-based improved algorithm in the K-th iteration is:

$$O(T_a) = O\left(f \times n + t_1 \times \frac{x \times g}{m}\right)$$

For our Bloom filter-based algorithm, the time components are:

- Time to store frequent single items in Bloom filter: $T_4 = t_2$
- Transaction pruning time: $T_5 = g \times t_2 \ / \ m + g \times t_3$
- Key-value pair generation time: $T_6 = n \times g \times x \ / \ m$

The time complexity for our Bloom filter-improved algorithm is:

$$O(T_b) = O\left(t_2 + \frac{g \times t_2}{m} + \frac{n \times g \times x}{m}\right)$$

Under big data conditions, k (iteration count), t_1 (hash tree search time), t_2 (Bloom filter search time), and t_3 (compression time) are negligible compared to other values. During iteration, n and f have the relationship:

$$f = \frac{n}{2}$$

Since n is extremely large in big data environments, this relationship can be simplified to $f = n/2$. The difference T - T can be simplified to:

$$T_a - T_b \approx \frac{n \times g \times x}{m}$$

During the second iteration, candidate sets are excessively large (n g), making T T . Therefore, our improved algorithm' s time complexity is far lower than that of the YAFIM algorithm.

2.3.2 Space Complexity The YAFIM algorithm' s space complexity is the sum of memory consumed by generating candidate sets and creating hash trees for storage:

- Space for storing candidate sets: $O(f \times n)$
- Space for creating a hash tree: $O(f^2)$

The overall space complexity is:

$$O(f \times n + f^2)$$

Our algorithm stores single items from frequent itemsets in Bloom filters with space complexity $O(f \times b_2)$. The total space complexity of our algorithm is significantly smaller than that of the Apriori algorithm.

3 Experiments

3.1 Experimental Data

The experiments use standard datasets commonly employed in association rule mining research, as listed in Table 1 .

Table 1 Dataset Information

Dataset	Characteristics
T10I4D100K	
Retail	
Musroom	
Kosarak	
BMSWebView2	
T25I10D10K	

3.2 Experimental Environment

The experimental environment consists of Intel Core i7-6500U processors, 8 GB RAM, and 1 TB hard drives. An 8-node Spark cluster was deployed, with one computer designated as the master node and the other seven as slave nodes. Each node has 8 cores and 64 GB RAM. The software environment includes Hadoop 2.6.0, Scala 2.12.1, Spark 1.6.1, Python 3.6.3, and Anaconda.

3.3 Scalability

Scalability [18] is discussed in conjunction with parallel algorithms and parallel computer architecture. The scalability of an algorithm on a machine reflects whether the algorithm can effectively utilize increasing numbers of cores [19]. Our scalability research aims to enable the algorithm to leverage more processors and predict its performance when ported to large-scale processors.

Experiments were conducted by increasing the number of logical cores and dataset replication factors to observe execution times across different datasets, thereby evaluating the algorithm' s executability and scalability. Using Retail, Musroom, Kosarak, and BMSWebView2 datasets, the results are shown in Figures 5 [Figure 5: see original paper] and 6 [Figure 6: see original paper]. Figure 5 shows that execution time decreases nearly linearly as the number of logical cores increases, while Figure 6 [Figure 6: see original paper] shows that execution time increases nearly linearly as the dataset replication factor increases. These results demonstrate that the algorithm possesses both executability and scalability.

Comparative experiments were conducted against the classic Hadoop-based MRApriori algorithm and the Spark-based YFAIM algorithm under identical datasets and minimum support thresholds. On the Retail dataset with min_{sup} = 0.15%, the time comparison of the three algorithms is shown in

Figure 7 [Figure 7: see original paper]. Our algorithm significantly outperforms both YFAIM and MRApriori, particularly during the second iteration when generating frequent 2-itemsets, achieving nearly 6× speedup over YFAIM and nearly 30× over MRApriori.

On the T10I4D100K dataset with min_{sup} = 0.15%, the time comparison is shown in Figure 8 [Figure 8: see original paper]. Our algorithm demonstrates clear superiority, particularly in the second iteration for frequent 2-itemset generation, achieving nearly 3× speedup over YFAIM and nearly 10× over MRApriori.

On the T25I10D10K dataset with min_{sup} = 0.10%, the time comparison is shown in Figure 9 [Figure 9: see original paper]. Our algorithm again outperforms both counterparts, particularly in the second iteration, achieving nearly 3× speedup over YFAIM and nearly 10× over MRApriori.

4 Conclusion

This paper proposes a partitioned parallel algorithm based on storage improvement, focusing on enhancing algorithm performance. We leverage the temporal and spatial advantages of Bloom filters for data storage and query operations during mining, avoiding the massive memory pressure caused by storing large amounts of data in memory. Built on the Spark framework, our algorithm compresses transactions during each iteration, filters out items not present in the Bloom filter, composes candidate sets from single items in frequent itemsets, compresses candidate sets, and finally mines frequent itemsets through partitioned parallel processing. Experimental results demonstrate that the algorithm effectively improves frequent itemset mining efficiency, particularly making the second iteration highly efficient, while saving resource space and exhibiting excellent scalability for application to larger-scale datasets and clusters.

References

[1] Cui Yan, Bao Zhiqiang. Summary of association rules mining [J]. Application Research of Computers, 2016, 33(2): 330-334.

[2] Mirjana M, Quintarelli E, Tanca L. Data mining for XML query-answering support [J]. IEEE Trans on Knowledge and Data Engineering, 2012, 24(8): 1393-1407.

[3] Riondato M, DeBrabant J A, Fonseca R, et al. PARMA: a parallel randomized algorithm for approximate association rules mining in MapReduce [C]//Proc of the 21st ACM International Conference on Information and Knowledge Management. New York: ACM Press, 2012: 85-94.

[4] Wang Zuocheng, Xue Lixia. A fast algorithm for mining association rules in image [C]//Proc of the 5th IEEE International Conference on Software Engineering and Service Science. Piscataway, NJ: IEEE Press, 2014: 487-490.

[5] Zhang Zhonglin, Tian Miaofeng, Liu Zongcheng. Research on parallel mining algorithm of association rules in big data environment [J]. Computer Science, 2016, 43(1): 286-289.

[6] Nguyen D, Vo B, Le B. Efficient strategies for parallel mining class association rules [J]. Expert Systems with Applications, 2014, 41(10): 4716-4727.

[7] Lin M, Lee P, Hsueh S. Apriori-based frequent itemset mining algorithms on MapReduce [C]//Proc of the 6th International Conference on Ubiquitous Information Management and Communication. New York: ACM Press, 2012: 1-8.

[8] Lin Xueyan. Mr-apriori: association rules algorithm based on mapreduce [C]//Proc of the 5th IEEE International Conference on Software Engineering and Service Science. Piscataway, NJ: IEEE Press, 2014: 141-144.

[9] Yahya O, Hegazy O, Ezat E. An efficient implementation of A-Priori algorithm based on Hadoop-Mapreduce model [J]. International Sjournal of Reviews in Computing, 2012, 12(7): 59-67.

[10] Lin Zhangfang, Wu Yangyang, Huang Zhongkai, et al. Parallelization of Apriori algorithm based on MapReduce [J]. Journal of Jiangnan University: Natural Science, 2014, 13(4): 411-415.

[11] Saabith A L S, Sundararajan E, Abubakar A. Parallel implementation of apriori algorithms on the Hadoop-mapreduce platform-an evaluation of literature [J]. Journal of Theoretical and Applied Information Technology, 2016, 85(3): 321-351.

[12] Zaharia M, Shixin R, Das T, et al. Apache Spark: a unified engine for big data processing [J]. Communications of the ACM, 2016, 59(11): 56-65.

[13] Shanahan J G, Dai Laing. Large scale distributed data science using apache spark [C]//Proc of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York: ACM Press, 2015: 2323-2324.

[14] Zhang Feng, Liu Min, Gui Feng, et al. A distributed frequent itemset mining algorithm using Spark for Big Data analytics [J]. Cluster Computing, 2015, 18(4): 1493-1501.

[15] Qiu Hongjian, Gu Rong, Yuan Chunfeng, et al. YAFIM: a parallel frequent itemset mining algorithm with Spark [C]//Proc of IEEE IPDPSW. Piscataway, NJ: IEEE Press, 2014: 1664-1671.

[16] Yang Qifang, Ma Guangpin. Improvement of association rules mining apriori algorithm [J]. Electronic Technology and Software Engineering, 2014, 28(19): 199-200.

[17] Holley G, Wittler R, Stoye J. Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage [J]. Algorithms for Molecular Biology Amb, 2016, 11(1): 1-9.

[18] Li Li, Chou Wu, Zhou Wei, et al. Design patterns and extensibility of REST API for networking applications [J]. IEEE Trans on Network & Service Management, 2016, 13(1): 154-167.

Note: Figure translations are in progress. See original paper for figures.

Source: ChinaXiv —Machine translation. Verify with original.