


## Article

# Aggregated Boolean Query Processing for Document Retrieval in Edge Computing

Tao Qiu <sup>1,\*</sup> , Peiliang Xie <sup>1</sup>, Xiufeng Xia <sup>1</sup>, Chuanyu Zong <sup>1</sup> and Xiaoxu Song <sup>2</sup>

<sup>1</sup> School of Computer Science, Shenyang Aerospace University, Shenyang 110136, China; 192306057421@email.sau.edu.cn (P.X.); xiaxf@sau.edu.cn (X.X.); zongcy@sau.edu.cn (C.Z.)

<sup>2</sup> School of Software, Shenyang University of Technology, Shenyang 110870, China; songxiaoxu@sut.edu.cn

\* Correspondence: qiutao@sau.edu.cn

**Abstract:** Search engines use significant hardware and energy resources to process billions of user queries per day, where Boolean query processing for document retrieval is an essential ingredient. Considering the huge number of users and large scale of the network, traditional query processing mechanisms may not be applicable since they mostly depend on a centralized retrieval method. To remedy this issue, this paper proposes a processing technique for aggregated Boolean queries in the context of edge computing, where each sub-region of the network corresponds to an edge network regulated by an edge server, and the Boolean queries are evaluated in a distributed fashion on the edge servers. This decentralized query processing technique has demonstrated its efficiency and applicability for the document retrieval problem. Experimental results on two real-world datasets show that this technique achieves high query performance and outperforms the traditional centralized methods by 2–3 times.

**Keywords:** aggregated query processing; Boolean query; edge computing; document retrieval



**Citation:** Qiu, T.; Xie, P.; Xia, X.; Zong, C.; Song, X. Aggregated Boolean Query Processing for Document Retrieval in Edge Computing. *Electronics* **2022**, *11*, 1908. <https://doi.org/10.3390/electronics11121908>

Academic Editor: Stefanos Kollias

Received: 12 April 2022

Accepted: 16 June 2022

Published: 19 June 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Large search engines use significant hardware and energy resources to process billions of document requests per day. This has motivated a large body of research that aims to reduce the cost of processing queries. Keyword-based Boolean query is the typical one in search engines, which utilizes the logical **and/or** expression to represent the set of keywords contained by the target documents. Generally, search engines use inverted index structures to quickly obtain the ordered sets (document IDs) for the keywords, then retrieve the target documents by evaluating the corresponding set expression [1]. For example, the keyword-based Boolean query “(data **or** knowledge) **and** (mining **or** science)” retrieves the documents containing at least one of the keywords in {data mining, data science, knowledge mining, knowledge science}, the set expression  $(S(\text{data}) \cup S(\text{knowledge})) \cap (S(\text{mining}) \cup S(\text{science}))$  is evaluated to identify the target documents, where  $S(w)$  is the ordered set of IDs for the documents containing the keyword  $w$ .

With the rapid growth of the number of internet users, traditional centralized document retrieval mechanisms may not be an appropriate strategy, when documents are distributed on the local caching servers and each local caching server corresponds to a certain sub-region of the network. Instead, document retrieval requests should be processed in a localized and distributed fashion, while only the intermediate results computed by the local caching servers should be aggregated to the cloud server. We argue that this decentralized strategy is appropriate in the context of distributed documents, especially when documents are large in volume. In recent years, edge computing has been widely adopted in many applications, which aims to decentralize the procedure of query processing and reduce the system energy consumption [2,3]. To this end, we explore the Boolean query processing for document retrieval in edge computing in this paper, which aims to improve

the query efficiency by processing the user queries in a distributed and localized fashion as much as possible.

Traditional techniques process the Boolean queries for document retrieval in a centralized fashion. These methods can be divided into comparison-based and non-comparison-based algorithms. The comparison based algorithms generally compile the Boolean query to a parse tree, then utilize the parse tree to guide the comparisons for the elements in the ordered sets of  $Q$  [4–6]. The algorithms introduced in [4,5] obtain a worst-case optimal complexity on the sizes of ordered document sets in the query, they utilize a bottom-up fashion to compute the intermediate results for each internal node of the parse tree, and the final evaluation results are obtained as the intermediate results for the root of parse tree are computed. An adaptive algorithm is proposed in [6], which preferentially computes the individual evaluation result. Additionally, the comparison-based algorithms for multi-list intersection can be used for the Boolean query problem by only considering the intersection [1,7–12]. Bille P. et al. proposed the non-comparison-based techniques on the Boolean query problem [13]. They adopted the approximate set representation for the ordered sets by computing the hash function values for the original ordered sets. Although the non-comparison based algorithm achieves efficient performance on the evaluation time, it only obtains the approximate results.

To the best of our knowledge, a distributed and localized method has not been investigated to support the Boolean queries for document retrieval. To address this challenge, this paper proposes an aggregated Boolean query processing technique in edge computing. In this context, the network is divided into sub-regions, where each sub-region corresponds to an edge network regulated by an edge server (i.e., the local caching server). Additionally, all edge servers are connected to a center node (cloud server) of the network. Boolean queries are initially processed at the cloud server of the network, then the following processing tasks are assigned to the edge servers, where documents are retrieved locally, and finally the results of the document retrieval are aggregated at the cloud server. To summarize, we make the following contributions in this paper.

- *Aggregated Boolean query processing in contiguous edge networks.* We design a marginal edge network query mechanism for the document retrieval request, which distributes the converted user query to the edge servers. This query mechanism computes the intermediate query results in each edge server with limited data transmission between edge servers, and the accurate query results are finally aggregated in the cloud server.
- *Processing Boolean queries in single edge networks.* For the query processing in the single edge network, we propose the evaluation tree-based plan to compute results for the document request query in the edge server, which avoids unnecessary checks for the document sets. We also design optimization techniques for skipping invalid element comparisons to further accelerate the query evaluation.

The rest of the paper is organized as follows: Section 3 gives the problem definition and introduces the necessary background. We introduce the marginal edge network query mechanism in Section 4, and present the query processing method for single-edge networks in Section 5. Experimental results are presented and analyzed in Section 6. Section 2 surveys the related work. At last, we conclude this paper in Section 7.

## 2. Literature Review

The existing work related with our problem can be broadly classified into two streams. One stream is concerned with the research in edge computing, while the other stream addresses the problem of Boolean query processing.

### 2.1. Edge Computing

Along with the popularity of IoT applications, edge computing has been proposed in recent years as the complement of cloud computing [14–16], which shifts the computational and storage resources toward the edge of the network. We review the following three directions in the field of edge computing that relate to our studied problem.

### 2.1.1. Decentralized Query Processing in Edge Computing

Query processing as an important ingredient of the IoT applications in edge computing has been studied recently, where the applications are processed in a decentralized fashion as much as possible [17–20]. A typical work is presented in [18], which processes the multi-attribute aggregation query in a decentralized fashion by constructing the energy-aware IR-tree in single-edge networks. A blockchain-based decentralized platform CoopEdge is proposed to support cor-operative edge computing [21], where an edge server can safely publish a computation task for others, and reliable edge servers are selected. In addition, the research field of serverless-enabled edge computing, which aims to bring computational resources closer to the data source, can also realize the decentralized query processing in edge computing [22–25]. A decentralized framework for serverless edge computing is proposed to minimize the query processing time by dispatching the stateless tasks to executors of the network [26]. In [27], a simulation tool is designed for the testing and evaluation for the serverless edge computing applications.

Although these techniques propose some solutions for decentralized query processing in edge computing, the query processing mechanisms cannot be reused for different types of queries, and these techniques cannot be directly utilized for the Boolean query processing problem.

### 2.1.2. Computation Offloading in Edge Computing

Computation offloading is a critical direction in edge computing, in which the computation tasks are offloaded from the resource-limited edge devices to the powerful network edges so that the applications obtain low latency [28–31]. Existing computation offloading techniques can be divided into two categories: traditional heuristic rule-based offloading schemes and learning-based intelligent offloading schemes. Traditional offloading schemes utilize heuristic algorithms to solve different optimization objectives, including network delay and energy consumption [32,33]. Intelligent offloading schemes solve the network delay and energy consumption problem, using the technique of online learning [34–36].

Computation offloading is a different problem than decentralized Boolean query processing. Therefore, the techniques for computation offloading cannot be used for our studied problem.

### 2.1.3. Privacy Preserving in Edge Computing

Some works have studied the privacy-preserving problem in edge computing [37–40]. The application in edge computing achieves efficient data/query processing by sinking parts of the cloud server business to the edge of the device and enabling the data/query to be processed in the edge devices. The security protocols/frameworks are then proposed to protect the privacy of user data in the edge devices under the edge computing environment. A security framework for big data analytics in VANETs equipped with edge computing nodes was proposed in [41]. A method integrating edge computing, cloud computing and differential privacy was devised to provide efficient privacy preservation for the location-based data stream processing [42]. Additionally, some techniques solve the data privacy issue in the edge devices using homomorphic encryption [43–45], which enables the protection of the data privacy for the edge devices with good computing ability.

In our problem setting, user documents are distributively stored in the edge devices. Although the data privacy issue also exists in this setting, we only focus on the method of Boolean query processing in this paper. These techniques protecting data privacy in edge computing are orthogonal to our proposed method.

## 2.2. Boolean Query Processing

Traditional techniques have been developed to support Boolean query processing, which can be divided into comparison-based and non-comparison-based algorithms. The basic idea of comparison-based algorithms is to compare the elements in the ordered sets of the query and find the results satisfying the Boolean logic [4–6]. They generally compile

the query to a parse tree, which can be used to guide the comparison order for the elements. An algorithm obtaining a worst-case optimal complexity on the sizes of ordered sets is introduced in [4], it computes the evaluation results using the parse tree in a bottom-up fashion. For each internal node  $V$  of the parse tree, the intermediate evaluation results for  $V$  are computed by applying multiple ordered sets intersection (respectively, union) if  $V$  is an intersection (respectively, union) node. Finally, the evaluation results for the root node of the parse tree are the evaluation results of the query. An adaptive algorithm was proposed in [6], which also utilizes the parse tree to compute the evaluation results. Its basic idea is iteratively checking the picked candidate elements to obtain the evaluation results. In each iteration, the algorithm first computes a candidate element from the ordered sets of the query that could be the evaluation result, then checks it by the parse tree. A non-comparison based algorithm for processing Boolean queries was proposed in [13]. The approximate set representations for the ordered sets are computed through the hash values of the original ordered sets. In order to store the hash values into a word so that the word-level parallelism can be used, they also utilized a compressed representation for the hash values. Afterwards, the evaluation results are computed by performing the set operations on the mapping compressed hash values, rather than the original ordered sets.

Additionally, the multi-list intersection algorithms can also be used to process the query that only contains intersection operations [1,7–12]. A straightforward algorithm is widely used, called *SvS*, which iteratively applies the two-set intersection in increasing order by size, i.e., starting with the two smallest ordered sets. *Adaptive* can be viewed as a variant of *SvS*, which selects the candidate element from the set with the least remaining elements, rather than the initial smallest set [11,12]. If a mismatch occurs before all sets have been checked, the sets are reordered based on the number of remaining elements in each set, and the new candidate element is picked from the smallest remaining subset. *Max* is proposed in [1], which does not need to reorder the sets when picking a candidate element. As a mismatch occurs, the element that leads to the previous candidate element to be mismatched is used to skip the invalid elements on the smallest set, choosing the successor on the smallest set as the new candidate element.

**Discussion:** In the context of edge computing, these traditional methods can be adopted for processing Boolean queries in a centralized fashion, that is, collecting all data in the cloud server and then evaluating the queries by these methods. Compared to the traditional centralized methods, our proposed approach improves the query efficiency from two aspects. At first, we design a decentralized query mechanism for Boolean queries which assigns the decomposed Boolean queries to the edge servers (Section 4). Another aspect is that an efficient evaluation algorithm for the decomposed Boolean queries in single networks is proposed (Section 5).

### 3. Preliminaries

The inverted index is widely used for document retrieval [1]. For each word  $w$  in the document collections, an ordered set with respect to  $w$  is built that records the ID of documents containing  $w$ . An inverted index for document collections consists of the ordered sets for all words. Given a keyword  $w_i$ , we can quickly obtain the ordered set  $S_i$  for  $w_i$  through the pre-built inverted index.

In general, users specify a keyword-based expression to retrieval the target documents. The processing system obtains the document sets of the keywords from the pre-built inverted index, then performs the corresponding Boolean set expression using these document sets.

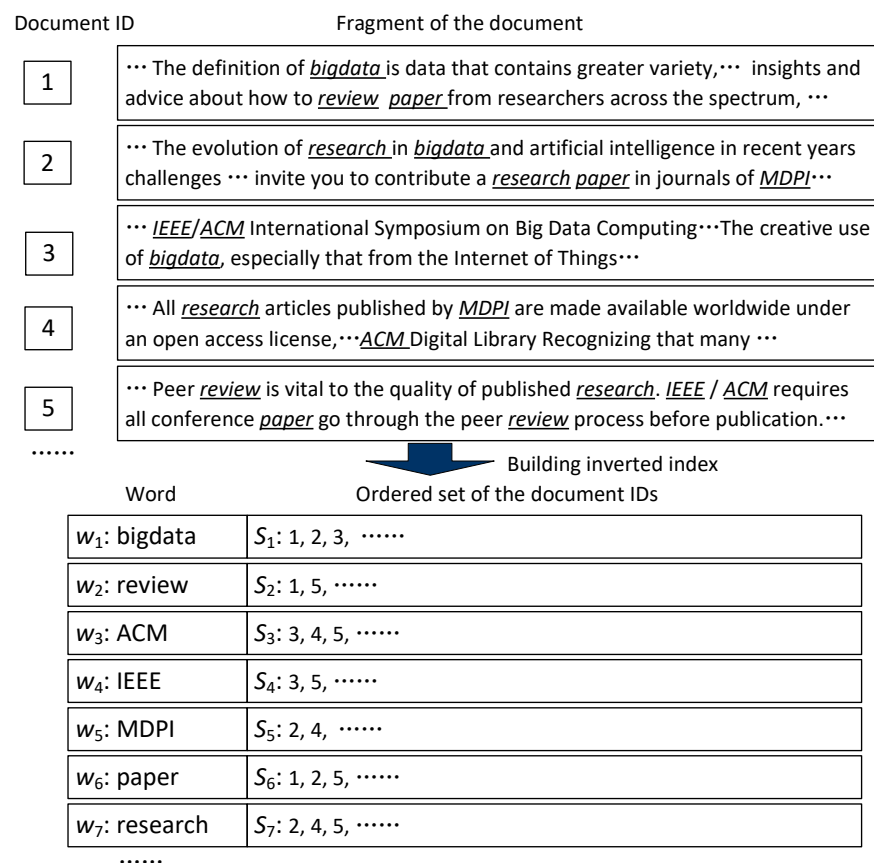
In this paper, a *Boolean set expression*  $Q$  is defined recursively as follows. (i)  $Q = S_i$  is a Boolean set expression which represents a document set  $S_i$ ; (ii)  $Q = Q_i \cap Q_j$  is a Boolean set expression which denotes a set intersected by the document sets of  $Q_i$  and  $Q_j$ , where  $Q_i$  and  $Q_j$  are the Boolean set expressions; and (iii)  $Q = Q_i \cup Q_j$  is a Boolean set expression which denotes the union for the document sets of  $Q_i$  and  $Q_j$ . Since the query processing system evaluates the Boolean set expression to retrieve the documents

describing by an user-specified keyword expression, we consider the Boolean query as a Boolean set expression directly in the remainder of this paper.

**Example 1.** For the user-specified keyword expression  $bigdata \wedge ((review \wedge (ACM \vee IEEE)) \vee (MDPI \wedge paper)) \wedge research$ , it is used to retrieve the documents, where each result document must contain a set of keywords (e.g.,  $\{bigdata, MDPI, paper, research\}$ ) described by the expression. Let  $w_1, w_2, \dots, w_7$  be the keywords  $bigdata, review, \dots, research$ , and  $S_1, S_2, \dots, S_7$  be the corresponding ordered sets storing the IDs of documents that contain the keywords  $w_1, w_2, \dots, w_7$ , respectively. The user query is answered by evaluating the following ordered set-based Boolean query.

$$Q = S_1 \cap ((S_2 \cap (S_3 \cup S_4)) \cup (S_5 \cap S_6)) \cap S_7$$

Figure 1 shows the fragments of five document samples, which contain the keywords in above example. Additionally, this figure shows the ordered sets for the keywords, e.g.,  $S_1 = \{1, 2, 3, \dots\}$  since the keyword  $w_1$ : *bigdata* exists in the first three samples. By evaluating the Boolean query  $Q$ , we know that the document with id = 2 is a result for the user query since this document contains words *bigdata*, *MDPI*, *paper*, *research*.



**Figure 1.** The fragments of sample documents and the inverted index.

In this paper, the region of a network consists of several disjoint edge networks in edge computing, and each edge network is managed by an edge server which stores the documents and the corresponding inverted index in this sub-region, i.e., the documents are distributed and stored on different edge servers. A marginal edge network is formally defined as follows.

**Definition 1. (Marginal edge network)** A marginal edge network is defined as a tuple  $G = (C_N, s_c)$ , where  $C_N$  is the collection of edge servers contained in marginal edge networks and all edge servers in  $C_N$  are connected to each other, and  $s_c$  is the cloud server connecting to all edge servers in  $C_N$ .

**Aggregated Boolean Query Processing.** The user sends a document request by a Boolean query  $Q$  to the cloud server  $s_c$ . Then, the request is distributed to all edge servers by  $s_c$ , and each edge server in  $C_N$  processes the document request locally and obtains the intermediate results. The results of  $Q$  are finally aggregated at the cloud server  $s_c$  by the intermediate results from edge servers.

#### 4. Marginal Edge Network Query Processing

In this section, we discuss how to process the query in the marginal edge network. A straightforward method is to process the query in each edge server, then aggregate the results of the edge servers in the cloud server. Although this method avoids the data transmission between edge networks, the final results computed in the cloud server may not be accurate, since each edge server only considers the documents in the related edge network when processing a query.

In order to compute the accurate results (documents) for the query  $Q$ , we can transfer the ordered sets of all related documents of  $Q$  from the edge servers to the cloud server and process the query on it. However, this way will lead to massive data transmission from the edge server to the cloud server. In this section, we introduce a method that enables the intermediate results to be computed in the edge server with limited data transmission between edge servers, and the accurate result is aggregated in the cloud server by the intermediate results from edge servers.

##### 4.1. Marginal Edge Network Query Mechanism

Given a query  $Q$ , let  $R$  be the accurate result set for  $Q$  over all documents in the cloud. Assuming that there are  $M$  edge networks,  $R^{(i)}$  is the result set computed from the  $i$ -th edge server  $N_i$ , where  $1 \leq i \leq M$ . Our target is to aggregate the results  $R^{(i)}$  in the cloud server to compose  $R$ , that is,  $R$  can be computed by  $R^{(1)} \cup R^{(2)} \dots R^{(M)}$ , so that only the results  $R^{(i)}$  are transferred from the edge servers to the cloud server, and the results are computed in a distributed way.

Let  $S_x^{all}$  be the document set containing the keyword  $k_x$  over the whole edge servers (named as *global set*) and  $S_x^{(i)}$  be the document set in the edge server  $N_i$  (named as *local set*), i.e.,  $S_x^{all} = S_x^{(1)} \cup S_x^{(2)} \dots S_x^{(M)}$ . Although  $S_x^{all} \cup S_y^{all} = \bigcup_{i=1}^M (S_x^{(i)} \cup S_y^{(i)})$ , i.e., the union of sets can be computed separately in each edge server, the intersection operation does not satisfy this property since  $S_x^{all} \cap S_y^{all} \neq \bigcup_{i=1}^M (S_x^{(i)} \cap S_y^{(i)})$ , so  $R^{(i)}$  cannot be simply computed in an edge server by replacing  $S_x^{all}$  with  $S_x^{(i)}$  for the query  $Q$ . To this end, we develop another way for the intersection operations so that the set intersections can be performed separately in each edge server.

Since  $S_x^{all} = S_x^{(1)} \cup S_x^{(2)} \dots S_x^{(M)}$ , we can obtain  $S_x^{all} \cap S_y^{all} = (S_x^{(1)} \cup S_x^{(2)} \dots S_x^{(M)}) \cap S_y^{all}$ . According to the distributive law of set operations, this equation can be expanded to  $S_x^{all} \cap S_y^{all} = (S_x^{(1)} \cap S_y^{all}) \cup (S_x^{(2)} \cap S_y^{all}) \dots \cup (S_x^{(M)} \cap S_y^{all})$ . Therefore, when  $S_x^{(i)} \cap S_y^{all}$  is computed in the edge server  $N_i$ , the results of  $S_x^{all} \cap S_y^{all}$  can be obtained in the cloud server by union of the intermediate results sent from the edge servers.  $S_x^{(i)}$  is the set of documents stored in  $N_i$  and can be accessed directly, while  $S_y^{all}$  is the global document set over all edge networks. In order to access  $S_y^{all}$  in an edge server, we need to broadcast  $S_y^{(i)}$  to other edge servers for each  $N_i$ . Although this operation leads to the data transmission between edge servers, the transferred data size is limited, and the query can be processed in a distributed fashion.



The above property gives the opportunity of processing the query  $Q$  in edge servers. Next, we introduce how to decompose  $Q$  to  $Q^{(i)}$  for an edge server  $N_i$ . Since  $Q$  is recursively defined and consists of two cases, we separately discuss them. (i)  $Q = Q_1 \cup Q_2$ , where  $Q_1, Q_2$  represent the set expression (sub-query) or the set of document. In this case,  $Q^{(i)}$  is easily obtained by  $Q_1^{(i)} \cup Q_2^{(i)}$ . (ii)  $Q = Q_1 \cap Q_2$ . According to the above introduced method, either  $Q_1$  or  $Q_2$  is converted to the local set, and another one is still the global set, then  $Q^{(i)}$  is computed by  $Q_1^{(i)} \cap Q_2^{all}$  or  $Q_1^{all} \cap Q_2^{(i)}$ . In this way, for any edge server  $N_i$ , the corresponding query  $Q^{(i)}$  can be computed by recursively applying the above two conversions on  $Q$ .

**Theorem 1.** Give a Boolean query  $Q$  for document sets,  $R$  is the result of  $Q$  over all edge networks.  $Q^{(i)}$  is the decomposed query from  $Q$  for the edge server  $N_i$ ,  $R^{(i)}$  is the result of  $Q^{(i)}$  computed in  $N_i$ .  $R$  can be computed by  $R^{(1)} \cup R^{(2)} \dots R^{(M)}$ , where  $M$  is the number of edge networks.

#### 4.2. Query Processing

The procedure of query processing is presented at Algorithm 1. In each edge server  $N_i$ ,  $Q^{(i)}$  is computed from  $Q$  by the aforementioned method. Then, all global sets in  $Q^{(i)}$  are broadcast between the edge servers so that each edge server  $N_i$  has the corresponding document sets for processing  $Q^{(i)}$  locally. Next,  $Q^{(i)}$  is processed in  $N_i$  (line 6), and the query processing in single edge networks is detailed in Section 5. At last, the intermediate results  $R^{(i)}$  computed by edge servers are aggregated in the cloud server, and the final result  $R$  is obtained by taking union for these intermediate results  $R^{(i)}$ .

---

##### Algorithm 1: MarginalBoolQuery

---

**Input** : A Boolean query  $Q$  for document sets, the collection  $\mathcal{C}_N$  of edge servers.

**Output** : The evaluation results  $R$ .

---

```

1  $R \leftarrow \emptyset$ ;
2 for each edge server  $N_i$  in  $\mathcal{C}_N$  do
3   Convert  $Q$  to  $Q^{(i)}$ ;
4   for each global set  $S_y$  in  $Q^{(i)}$  do
5     Broadcast  $S_y$  between edge servers;
6    $R^{(i)} \leftarrow N_i.\text{TreePlanMatch}(Q^{(i)})$ ;
7 Wait all results  $R^{(i)}$  are sent from edge servers;
8  $R \leftarrow R^{(1)} \cup R^{(2)} \dots \cup R^{(M)}$ ;
9 return  $R$ 

```

---

When computing  $Q^{(i)}$  from  $Q$ , one of the operands for the set intersection operation should be selected as the global set, e.g.,  $Q_1$  or  $Q_2$  should be the global set for the set intersection  $Q_1 \cap Q_2$ . Since the global sets are needed to be broadcast between the edge servers, the selected global sets affect the size of the transferred data. In order to minimize the broadcast data, we need to select the document sets with a smaller size as the global sets. To this end, we store the size of all document sets in the cloud server and utilize this size information to determine the global sets of  $Q$  in the cloud server, so each edge server can directly apply the same strategy to select global sets when computing  $Q^{(i)}$  in the edge server.

Considering the set intersection  $Q_1 \cap Q_2$  and letting  $|S_1|$  and  $|S_2|$  be the size of result sets  $S_1$  and  $S_2$  for  $Q_1$  and  $Q_2$ , the size of  $Q_1 \cap Q_2$  is estimated by  $\min(|S_1|, |S_2|)$  since the result set size is bounded by the minimum of  $|S_1|$  and  $|S_2|$ . On the contrary, the size of  $Q_1 \cup Q_2$  is estimated by the size upper bound  $|S_1| + |S_2|$ . For the recursively defined query  $Q$ , we can use these two rules to estimate the result size for the operands of set intersection, then select the one with a smaller size as the global set to minimize the transferred data.

For example, consider the running example, suppose that the size of document sets  $S_1, S_2, \dots, S_7$  is recorded in the cloud server and  $|S_1| < |S_2| < |S_3| < |S_4| < |S_7| < |S_5| < |S_6|$ . At first, consider the set intersection between  $S_1$  and  $((S_2 \cap (S_3 \cup S_4)) \cup (S_5 \cap S_6)) \cap S_7$ ;

$S_1$  is selected as the global set since it has the smaller size. Then, the query conversion is applied to the later sub-query. The size of  $((S_2 \cap (S_3 \cup S_4)) \cup (S_5 \cap S_6))$  is estimated by  $\min(|S_2|, |S_3| + |S_4|) + \min(|S_5|, |S_6|)$ , which is greater than  $|S_7|$ ;  $S_7$  is selected as the next global set of the sub-query, and the query conversion is applied to  $((S_2 \cap (S_3 \cup S_4)) \cup (S_5 \cap S_6))$ . Because both of the operands for the set union should be applied to the query conversion, we can obtain the converted sub-query  $((S_2^{all} \cap (S_3^{(i)} \cup S_4^{(i)})) \cup (S_5^{all} \cap S_6^{(i)}))$  by using the above rules. Eventually,  $Q^{(i)} = S_1^{all} \cap ((S_2^{all} \cap (S_3^{(i)} \cup S_4^{(i)})) \cup (S_5^{all} \cap S_6^{(i)})) \cap S_7^{all}$  is computed for the edge server  $N_i$  in which  $S_3^{(i)}$ ,  $S_4^{(i)}$  and  $S_5^{(i)}$  represent the local document sets in  $N_i$ . In the cloud server, the result  $R$  of  $Q$  is computed by performing  $R^{(1)} \cup R^{(2)} \dots R^{(M)}$ .

## 5. Single Edge Network Query Processing

In this section, we introduce the method of processing the decomposed Boolean query in the single edge network. In this scenario, the edge server contains all necessary ordered sets of documents for the decomposed query. At first, we parse the Boolean query into a graph structure in Section 5.1. The method of Boolean query processing in the single-edge network is presented in Section 5.2, which computes an execution plan from the parsed graph.

### 5.1. Parsing Boolean Set Expression

To evaluate the Boolean set expression, a naive way is to perform pair-wise set intersection (or union) starting from the inner set operations. However, this way leads to inefficient query processing since the irrelative elements in ordered sets are compared and the intermediate results are recomputed [1]. For the multi-set intersection problem (which is a simple case for the set expression), it is shown that a holistic element comparison outperforms the pair-wise set comparison; this motivates us to design a holistic element comparison method to compute the results for the set expression.

**Example 2.** The query  $Q^{(i)} = S_1^{all} \cap ((S_2^{all} \cap (S_3^{(i)} \cup S_4^{(i)})) \cup (S_5^{all} \cap S_6^{(i)})) \cap S_7^{all}$  is used as the example (decomposed query) to illustrate query processing in single-edge networks. For simplification, we use  $Q = S_1 \cap ((S_2 \cap (S_3 \cup S_4)) \cup (S_5 \cap S_6)) \cap S_7$  to represent  $Q^{(i)}$  in this section, where  $S_1, S_2, S_5$  and  $S_7$  represent  $S_1^{all}, S_2^{all}, S_5^{all}$  and  $S_7^{all}$ ;  $S_3, S_4$ , and  $S_6$  represent  $S_3^{(i)}, S_4^{(i)}$  and  $S_6^{(i)}$ , respectively. The example of sets in the edge server  $N_i$  are shown in Table 1, and the result of  $Q$  on these sets is  $\{10, 39\}$ .

**Table 1.** An example of ordered sets in the edge server  $N_i$ , where bold elements are the results for the running example.

Ordered Sets	The Set of Document IDs
$S_1(S_1^{all})$	3, <b>10</b> , 14, <b>39</b> , 54, 69, 81, 88, 95
$S_2(S_2^{all})$	3, <b>10</b> , 81, 95
$S_3(S_3^{(i)})$	7, 15, 44, 64, 99
$S_4(S_4^{(i)})$	5, <b>10</b> , 41, 56, 72, 97
$S_5(S_5^{all})$	1, <b>10</b> , 21, <b>39</b> , 56, 65, 77
$S_6(S_6^{(i)})$	<b>10</b> , <b>39</b> , 56, 65
$S_7(S_7^{all})$	5, <b>10</b> , 17, 25, <b>39</b> , 44, 65, 78, 81, 93

In order to evaluate the query  $Q$ , we design a graph structure to parse  $Q$ , named the *union-intersection graph*, as follows.

**Definition 2. (Union-intersection Graph)** A union-intersection graph (UIGraph)  $G = (S_V, S_E, I, F)$  is an undirected graph which represents a set expression  $Q$ , where  $S_V$  is a set of nodes, among which one is the initial node  $I$  and one is the final node  $F$ .  $S_E$  is a set of edges which are the sets from  $Q$ .



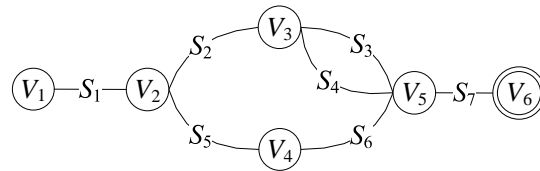
For the UIGraph, the conjunctive and disjunctive structures are used to represent the intersection and union of ordered sets, respectively. Figure 2 shows the two basic structure for  $S_i \cap S_j$  and  $S_i \cup S_j$ . Hence, a query  $Q$  can be represented by an UIGraph  $G$  with nested conjunctive and disjunctive structures.



(a) Conjunctive structure for  $S_i \cap S_j$  (b) Disjunctive structure for  $S_i \cup S_j$

**Figure 2.** Two basic structures of UIGraph for the set query.

We can compute the UIGraph  $G$  for the query  $Q$  by constructing conjunctive and disjunctive structures for intersection and union operations, respectively. Figure 3 shows the built UIGraph  $G$  for the running example.



**Figure 3.** The UIGraph for the query  $Q$ .

For a graph, a path  $P$  starting from the initial node and ending at the final node is called a *simple path* if all of its nodes are distinct [46]. The UIGraph also contains the simple paths, e.g., there are three simple paths for the UIGraph in Figure 3 that are  $S_1$ - $S_2$ - $S_3$ - $S_7$ ,  $S_1$ - $S_2$ - $S_4$ - $S_7$  and  $S_1$ - $S_5$ - $S_6$ - $S_7$ .

**Theorem 2.** For each simple path  $P$  in a UIGraph  $G$ , if and only if there is an element  $e$  such that  $e$  exists in all the order sets of  $P$ , then  $e$  is an evaluation result of the query  $Q$ .

For example, 39 is contained by sets  $S_1$ ,  $S_5$ ,  $S_6$  and  $S_7$  which are the ordered sets of a simple path, so 39 is an evaluation result of  $Q$ . We omit the detailed proof for Theorem 2; it can be easily proved based on the fact that the element exists in the sets of a simple path of  $G$  must satisfy the logical requirements of  $Q$ .

## 5.2. Boolean Query Processing in Single Edge Networks

### 5.2.1. Evaluation Tree

According to Theorem 2, we know that for any result  $e$  of  $Q$ ,  $e$  must be contained by the ordered sets in a simple path of  $G$ . In order to obtain all results, we use the elements in a  $s$ - $t$  cut [46] of  $G$  as *candidate elements*, which may be the final results.

**Definition 3. ( $s$ - $t$  cut)** A  $s$ - $t$  cut is a collection of edges (ordered sets) of UIGraph  $G$  which partitions the nodes of  $G$  into two disjoint subsets, and the initial and final nodes belong to different subsets [46].

We use  $\mathcal{S}_e$  to denote a  $s$ - $t$  cut of  $G$ . For any simple path  $P$  on  $G$ ,  $P$  contains at least a set  $S_i$  in a  $s$ - $t$  cut of  $G$ . Accordingly, any evaluation result of  $Q$  must be contained by the sets in a  $s$ - $t$  cut. Given a  $s$ - $t$  cut  $\mathcal{S}_e$  of  $G$ , we call  $\mathcal{S}_e$  *candidate sets* and use the elements in the sets of  $\mathcal{S}_e$  as candidate elements. For example,  $\{S_1\}$  and  $\{S_2, S_6\}$  are two different  $s$ - $t$  cuts of  $G$ , which can be used as the candidate sets separately.

For a candidate element  $e_b$  from a candidate set  $S_b \in \mathcal{S}_e$ , the verification for  $e_b$  is to check if there exists a simple path  $P$  such that all order sets of  $P$  contain  $e_b$ , which can be represented by a Boolean expression,  $B(e_b, S_b)$ . However, not all simple paths of  $G$  can

produce an evaluation result  $e_b$ . Since  $e_b \in S_b$ , in order to verify  $e_b$ , we only need to consider the ordered sets in the simple paths which contain  $S_b$ . The Boolean expression  $B(e_b, S_b)$  consists of a set of variables  $R(e_b \in S_i)$  on such ordered sets, which indicate whether an element  $e_b$  exists in an ordered set  $S_i$ .

**Example 3.** Consider the running example, assume that  $S_e = \{S_2, S_6\}$  is the set of candidate document sets. The corresponding Boolean expressions of  $S_2$  and  $S_6$  are shown as below. Consider the candidate element  $e_b = 10$  from  $S_6$ , since  $B(10, S_6)$  is true, we can obtain that 10 is a result of  $Q$ .

$$\begin{aligned} B(e_b, S_2) &= R(e_b \in S_1) \wedge R(e_b \in S_5) \wedge R(e_b \in S_6) \wedge R(e_b \in S_7) \\ B(e_b, S_6) &= R(e_b \in S_1) \wedge R(e_b \in S_2) \wedge (R(e_b \in S_3) \vee R(e_b \in S_4)) \wedge R(e_b \in S_7) \end{aligned}$$

So far, to verify a candidate element  $e_b \in S_b$ , we only need to evaluate the corresponding Boolean expression  $B(e_b, S_b)$ . Borrowing the idea of evaluating a Boolean expression by using the binary decision tree [47], we define an *evaluation tree* to guide the checks for the ordered sets in a Boolean expression  $B(e_b, S_b)$ .

**Definition 4. (Evaluation Tree)** An evaluation tree  $T_e$  is a rooted binary tree used to verify a candidate element  $e_b$ , in which each internal node indicates the result of checking if an ordered set  $S_i$  contains  $e_b$ , and leaf nodes are the verification results.

There are two types of leaf nodes, called 1-leaf and 0-leaf, which represent the true and false results of verifying  $e_b$ . Each internal node is a variable in  $B(e_b, S_b)$ , i.e.,  $R(e_b \in S_i)$ , abbreviated as  $R(S_i)$ . If  $R(S_i)$  is true, then there is an element  $e_b$  existing in the set  $S_i$ .

The expected matching cost of the evaluation tree indicates the expected number of checked ordered sets to verify a candidate element  $e_b$ . In order to minimize the matching cost, we compute the evaluation tree with minimal expected cost for each Boolean expression  $B(e_b, S_b)$  where  $S_{b_i} \in S_e$ . The dynamic programming algorithm introduced in [48] is employed in this paper to compute the evaluation tree with minimal expected cost, which has the time complexity  $O(N^2(r+1)^N)$ , where  $r$  is the number of sets, and  $N$  is the number of set operations in  $Q$ .

### 5.2.2. Performing Evaluation Tree-Based Plan

An *execution plan*  $\mathcal{P}$  consists of the candidate sets  $S_e$  and the collection  $S_{T_e}$  of the corresponding evaluation trees which are used to verify the candidate elements in  $S_e$ . We next show how to compute the evaluation results using  $\mathcal{P}$ . The procedure is described in Algorithm 2. There are two issues that need to be solved when performing an execution plan: (1) the scheme of verifying candidate elements in candidate sets; and (2) how to verify a candidate element using evaluation tree.

For the first issue, a basic scheme is to individually verify each candidate set in  $S_e$ . However, this way loses the chance to skip some invalid candidate elements. We observe that if the candidate elements are verified in the ascending order, then some candidate elements that cannot be the evaluation results can be quickly skipped using the intermediate obtained results (we give the details in Section 5.2.3). To achieve this goal, we build a heap  $H$  for the top elements of ordered sets in  $S_e$  which costs time  $\mathcal{O}(|S_e|)$  to obtain an element [49]. For example, as shown in Figure 4, a heap is built for  $S_2$  and  $S_6$ , and the candidate elements are verified in the order  $\{3, 10, 10, 39, 56, 65, 81, 95\}$ .

**Algorithm 2:** TreePlanMatch

---

**Input** : An execution plan  $\mathcal{P}$  which contains the candidate sets  $\mathcal{S}_e$  and corresponding evaluation trees.

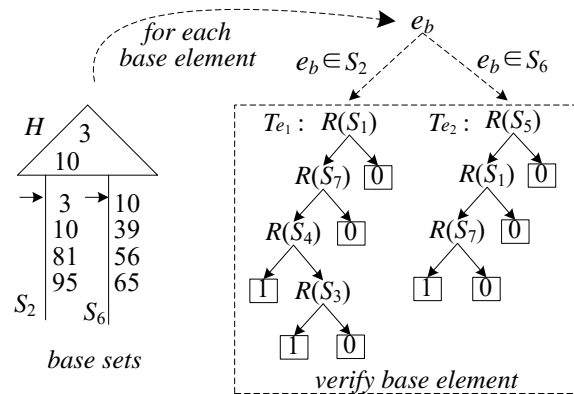
**Output** : The evaluation results  $A$ .

```

1 Build a heap  $H$  for the top elements of ordered sets in  $\mathcal{S}_e$ ;
2 while  $H$  is not empty do
3    $e_b \leftarrow H.\text{pop}()$ ;
4   if  $e_b$  is checked and  $e_b$  is the evaluation result then
5     continue; /* avoid duplicate verification */;
6    $T_e \leftarrow$  the corresponding evaluation tree of  $S_b$  that provides  $e_b$  in  $\mathcal{S}_e$ ;
7   Let  $p_r$  be the pointer on the root of  $T_e$ ;
8   while  $p_r$  is not leaf node do
9     Let  $S_i$  be the set pointed by  $p_r$ ;
10     $R(S_i) \leftarrow \text{Search}(S_i, e_b)$ ; /* gallop search */;
11    if  $R(S_i)$  is true then
12       $p_r \leftarrow p_r.\text{left}$ ;
13    else
14       $p_r \leftarrow p_r.\text{right}$ ;
15  if  $p_r$  is a 1-leaf node then
16    Add  $e_b$  to  $A$ ;
17  Adjust  $H$ ;
18 return  $A$ 

```

---

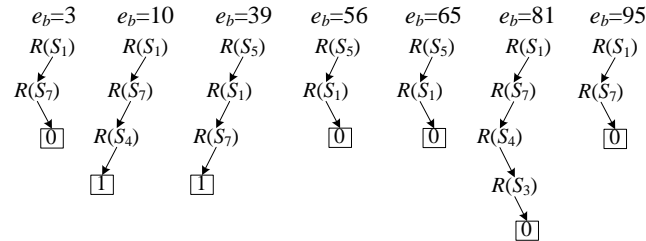


**Figure 4.** An example of evaluation-tree-based execution plan.

For the second issue, for each candidate element  $e_b$  obtained from  $H$ , it is verified by the corresponding evaluation tree  $T_e$ . The algorithm checks the internal nodes of  $T_e$  starting from the root until a leaf node is reached (lines 8–14). In each internal node  $R(S_i)$ , the result is checking if  $S_i$  contains the candidate element  $e_b$ . In this work, we employ the gallop search [50] to search an element from  $S_i$ , which has time complexity  $\mathcal{O}(\log K)$ , where  $K$  is the largest size of the ordered sets in  $Q$ . If a 1-leaf node is reached, then  $e_b$  is an evaluation result of  $Q$  (lines 15–16). Note that the candidate elements obtained from  $H$  could be repeated, e.g., 10 is repeated in the running example, for the repeated element  $e_b$ , if  $e_b$  is already turned out to be an evaluation result, then it is unnecessary to verify it again (lines 4–5).

Let  $L$  be the number of ordered sets in  $Q$ , and let  $N_b$  be the number of candidate elements in  $\mathcal{S}_e$ . The algorithm spends time  $\mathcal{O}(L \log K)$  to verify a candidate element. To obtain candidate elements in the ascending order, time  $\mathcal{O}(L)$  is used to adjust the heap  $H$  after a candidate element is popped. Hence, the time complexity of TreePlanMatch is  $\mathcal{O}(N_b L^2 \log K)$ .

**Example 4.** For the execution plan shown in Figure 4, the procedure of computing evaluation results by the execution plan is shown in Figure 5. TreePlanMatch verifies 7 candidate elements and only checks the ordered sets 18 times in total, in which elements 10 and 39 are turned out to be the results since 1-leaf nodes are reached when verifying them.



**Figure 5.** The procedure of computing evaluation results.

### 5.2.3. Optimizing Execution Plan by Skipping Invalid Candidate Elements

In the previous example, all candidate elements are verified. We observe that some candidate elements that cannot be the evaluation results (called invalid candidate elements) can be safely skipped without verification using the obtained verification results. In this section, we present the technique to skip the invalid candidate elements.

Recall the procedure of verifying a candidate element  $e_b$ ; in order to obtain the result of each internal node  $R(S_i)$  in a root-to-leaf path of  $T_e$ , we search  $e_b$  on  $S_i$  and return the first element  $e'_i$  such that  $e'_i \geq e_b$ . If the returned element  $e'_i > e_b$ , then  $R(S_i)$  is false. Actually, in this case, because the candidate elements are processed in ascending order, the returned failed element  $e'_i$  indicates that the next candidate element should be no less than  $e'_i$  if  $S_i$  contains the next evaluation result. Based on this property, if the result of verifying  $e_b$  is false (i.e., a 0-leaf node is reached), then a lower bound  $\mathcal{L}(e_b)$  for the next candidate element can be computed through the obtained failed elements in the internal nodes of the root-to-leaf path.

For a candidate element  $e_b$ , let  $P_i$  be the root-to-leaf path (i.e., with 0-leaf node) when verifying  $e_b$ , and  $F(e)$  be the set of failed elements returned by the ordered sets in  $P_i$ , then  $\mathcal{L}(e_b)$  is computed as follows.

$$\mathcal{L}(e_b) = \min_{e'_i \in F(e)} \{e'_i\} \quad (1)$$

Apparently, the next evaluation result should be no less than  $\mathcal{L}(e_b)$ . Hence, the candidate elements which are less than  $\mathcal{L}(e_b)$  can be safely skipped. When verifying a candidate element  $e_b$ , the failed elements can be recorded by the matching algorithm (line 11 in Algorithm 2), that is, the minimum  $\mathcal{L}(e_b)$  of the failed elements can be computed in the time complexity  $\mathcal{O}(1)$ . Therefore, skipping invalid candidate elements has the time complexity  $\mathcal{O}(1)$ .

**Example 5.** Consider the example shown in Figure 5; when checking  $e_b = 56$ , the failed element  $e'_1 = 69$  is returned by  $S_1$ . Since only  $R(S_1)$  is false in the root-to-leaf path, we can compute  $\mathcal{L}(e_b) = 69$ . Then, the next candidate element  $e_b = 65$  can be skipped since  $65 < \mathcal{L}(e_b)$ . In the next, a 0-leaf node is reached when verifying  $e_b = 81$  and two failed elements  $e'_4 = 97$  and  $e'_3 = 99$  are obtained, so we obtain  $\mathcal{L}(e_b) = 97$ . The following candidate element  $e_b = 95$  is skipped by the lower bound  $\mathcal{L}(e_b) = 97$ . After skipping the invalid candidate elements, only 5 candidate elements are verified and the ordered sets are checked 14 times.

## 6. Experiments

In this section, we present the experimental results of our proposed algorithms with traditional algorithms on two real-world datasets.

### 6.1. Experimental Setup

In order to simulate the context of edge computing, we built a network with 10 nodes in which 9 nodes represent the edge servers and 1 node represents the cloud server of the network. In edge computing, each edge server is used to manage the sub-region (edge network) of the network, that is, each edge server caches the documents for the corresponding sub-region for our studied problem. Therefore, documents are distributed to the edge server nodes in this setting. We used two datasets (documents) with different data sizes in the following experiments, and each dataset is divided into 9 subsets with a random percentage and stored in edge server nodes.

The details of the datasets are shown as follows.

- *Source Files* is a set of source code files that extracted from Github, including Python, Ruby, JavaScript, Java, C and C++ files; it contains 697,485 text files in 3.69 GB of text.
- *Medline* is a bibliographic database of life sciences and biomedical information. We used a late-2008 snapshot of Medline, which consists of 17,104,854 citation entries with abstracts.

In each edge server node, the inverted lists for each word in the subset of dataset are built. The inverted lists are ordered sets that contain the IDs of files. For the source file dataset, we collected the real Boolean queries from the online library and forums (<http://www.regexlib.com>, accessed on 27 December 2021), which aim to find files that contain certain keywords. Among the collected real queries, we used 80 queries whose evaluation results are not empty in this dataset. For the Medline dataset, we used a full-day's query log of PubMed that was obtained from the NLM FTP site (refer to [51] to access the query log). These queries were issued by 626,554 distinct users, in which we also chose 80 queries that contain **and/or** operators as Boolean queries.

In order to study the impact of the complexity of queries on query efficiency, we further classified the Boolean queries into six categories using the number of keywords contained by each query, as shows in Table 2.

**Table 2.** Categories for the Boolean queries.

Categories	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>	C <sub>6</sub>
# of keywords	1–3	4–5	6–7	8–9	9–10	≥11

As far as we know, existing algorithms for document retrieval can only process the Boolean queries in a centralized fashion. We compared our proposed method with these centralized algorithms on the query efficiency. We selected the following algorithms as the representatives of existing algorithms for the Boolean query evaluation.

- WstOpt utilizes the parse tree to recursively compute the evaluation results and obtains the guarantee of the optimal worst case [4].
- Adapt is also a parse-tree-based algorithm proposed in [6] that adopts the adaptive strategy to iteratively compute the evaluation results.
- Max is the representative algorithm that uses the technique of multiple sets intersection [1]. For a Boolean query  $Q$ , we first convert  $Q$  to the form of union on several multiple sets intersections, then use Max to compute the result of each multiple sets intersection, finally using their union to obtain the evaluation results of  $Q$ .
- TreePlan is our proposed algorithm which processes the Boolean query in a distributed fashion, and it utilizes the tree-based execution plan to find evaluation results in each edge server; all optimizations are applied in TreePlan by default.

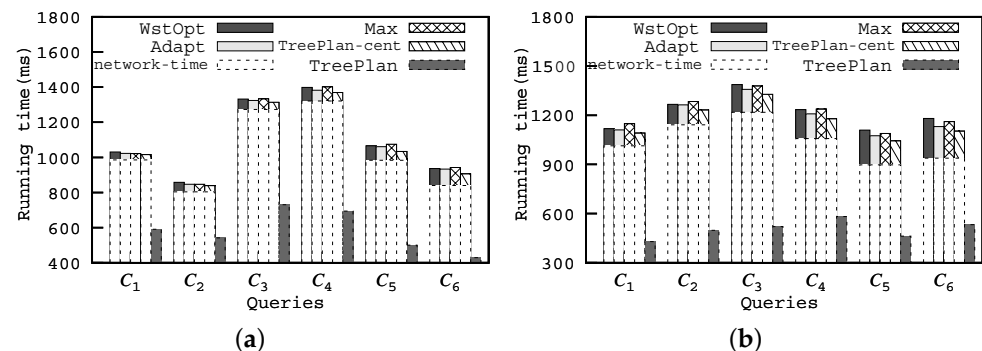
For the first three comparative algorithms, they process the Boolean queries in a centralized fashion, that is, all inverted lists for the relative documents of the query are collected to the cloud server node, then the query is evaluated in the cloud server node.

The experiments were carried out on a PC with an Intel i7-6700 3.4 GHz processor and 8 GB RAM, running Ubuntu 14.04.3. The algorithms were implemented in C++.

### 6.2. Comparison with Traditional Centralized Algorithms

The first experiment compares the running time of our proposed algorithm TreePlan and the comparative algorithms. WstOpt, Adapt and Max are the above introduced comparative algorithms. We also tested our proposed algorithm in the centralized mode (labeled by TreePlan-cent), i.e., collecting all inverted lists of related documents to the cloud server node and using the algorithm in Section 5.2 to process the Boolean query. For the centralized algorithms, we separately recorded the time of data transmission and the elapsed time of algorithms, where the time of data transmission is labeled by network-time. Since TreePlan processes the queries in the edge server nodes, the running time of TreePlan includes the data transmission time and the intermediate results aggregation time.

For each query category  $C_i$ , we averaged the running time of queries in  $C_i$ . We plot the average running time of different algorithms for the two real datasets in Figure 6. We can see that TreePlan obtains a great advantage on the query time over the centralized algorithms on both datasets. For example, for the queries  $C_6$  in the source codes dataset, the running time spent by TreePlan is only 429 ms, while the running time spent by other algorithms is more than  $2\times$  to the running time of TreePlan. The main reason is that the data transmission time for TreePlan is far less than the time for the centralized algorithms. Additionally, from the view of centralized algorithms, our proposed algorithm is also more efficient than the comparative algorithms. TreePlan-cent has the same data transmission time as WstOpt, Adapt and Max, while it spends less time on the phase of query evaluation.



**Figure 6.** Performance comparison of different algorithms. (a) Source Codes. (b) Medline.

### 6.3. Evaluating Tree-Based Query Plan

In this experiment, we tested the tree-based query plan proposed in Section 5.2. To evaluate the algorithm performance accurately, we collected all documents in the cloud server node and queries are processed locally on it, i.e., the running time indicates the elapsed time of the algorithm.

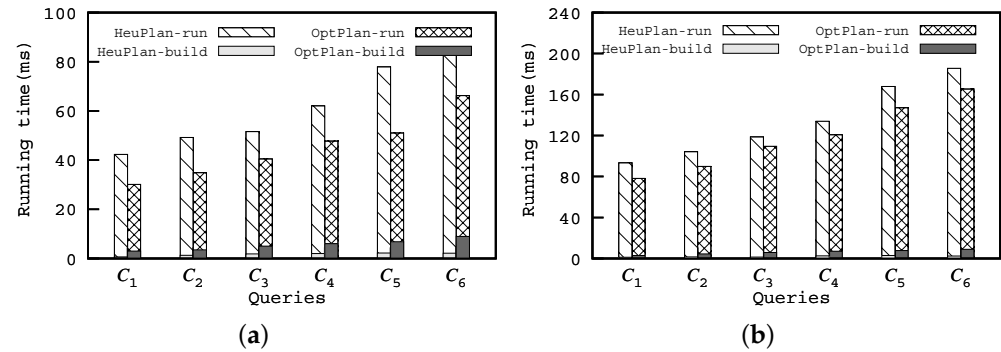
#### 6.3.1. Cost Analysis for Tree-Based Query Plan

In Section 5.2, we computed the optimal execution plan with the minimal matching cost for our proposed algorithm. An alternative way is computing the execution plan using the heuristic rule that utilizes the min-cut [46] of an UIGraph as the candidate sets. We used HeuPlan and OptPlan to represent the algorithms that utilize heuristic and optimal evaluation plan, respectively. Both of HeuPlan and OptPlan are our proposed algorithms which employ different execution plan computing strategies. We separately tested the time of the computing evaluation plan and using plan to find the evaluation results, which are annotated with the suffixes build and run in Figure 7.

As we can see from Figure 7, OptPlan achieves better performance than HeuPlan for the different datasets and queries. OptPlan needs more time to build the evaluation plan than HeuPlan, but OptPlan spends less time than HeuPlan in the phase of plan execution. For example, for the queries  $C_5$  in source code dataset, HeuPlan spends only 2.23 ms to build the evaluation plan, versus 6.8 ms used by OptPlan. However, OptPlan only spends 44.22 ms



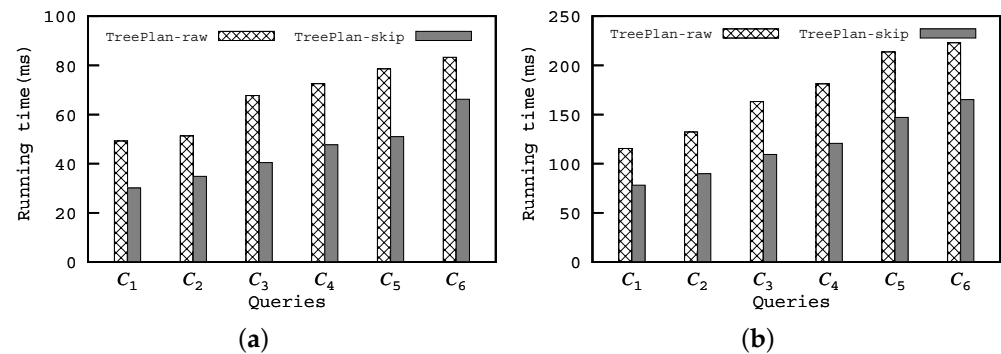
to performing evaluation plan, which is less than the time used by HeuPlan. Meanwhile, we can find that, regardless of the datasets, the evaluation plan can be efficiently built, and the time of building an evaluation plan for any category of queries is less than 10 ms.



**Figure 7.** Cost analysis for tree-based execution plan. (a) Source Codes. (b) Medline.

### 6.3.2. Effect of Skipping Invalid Candidate Elements

In this experiment, we tested the effect of an optimization technique that skips the invalid candidate elements which cannot be the final evaluation results (see Section 5.2.3). We used TreePlan-skip and TreePlan-raw to respectively represent our proposed algorithms that employ the optimization technique and without optimization. The results are shown in Figure 8. In different dataset and queries, TreePlan-skip always spends less running time than TreePlan-raw since TreePlan-skip checks fewer candidate elements. The gap of the running time between the two algorithms is most obvious for the queries in C3 in source file dataset, i.e., the running times of TreePlan-skip against TreePlan-raw are 40.47 ms and 67.05 ms.



**Figure 8.** Effect of skipping invalid candidate elements. (a) Source Codes. (b) Medline.

As for the time cost for skipping invalid candidate elements, we showed that the time complexity for this optimization is  $\mathcal{O}(1)$  in Section 5.2.3. Additionally, to analyze the practical time cost, we tested the algorithm that still computes the minimum  $\mathcal{L}(e_b)$  of the failed elements for each candidate element, but without skipping the invalid candidates. The time for this algorithm is almost the same as the time spent by TreePlan-raw, which means that the cost for skipping invalid candidate elements is minuscule.

## 7. Conclusions

Considering the large scale of the network and the swift growth of users for search engines, the traditional centralized methods for document retrieval may not be efficient and applicable when documents are distributed on the local caching servers, thus edge computing is employed to facilitate the distributed Boolean query processing for document retrieval. In this paper, we propose an aggregated Boolean query mechanism in edge

computing to support efficient document retrieval. Specifically, we design a marginal edge network boolean query mechanism, which enables the query to be processed in the edge servers by a decentralized fashion. To achieve this goal, the original Boolean query is decomposed in the cloud server and assigned to edge servers. The global sets caused by the intersection operations in decomposed Boolean queries are broadcast among edge servers so that each boolean query can be processed locally (Section 4), and the number of intersections positively correlates with the number of global sets. Therefore, our proposed algorithm has a greater advantage in processing the Boolean queries with fewer intersections and more unions.

We also propose an evaluation tree-based method to process the queries in single-edge networks, and design optimization techniques of skipping invalid element comparisons to accelerate the query evaluation. Extensive experiments on real-world datasets were conducted, and the results show that our proposed technique outperforms the traditional centralized methods in query efficiency.

**Author Contributions:** Writing, Literature search and Methodology, T.Q.; Data analysis and Data collection, P.X.; Study design and Data analysis, X.X.; Literature search and Methodology, C.Z.; Data collection and Figures, X.S. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work is partly supported by the National Natural Science Foundation of China (Nos. 62002245 and 61802268) and the Liaoning Provincial Department of Education Science Foundation (No. JYT2020027).

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Culpepper, J.S.; Moffat, A. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.* **2010**, *29*, 1. [\[CrossRef\]](#)
2. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge computing: Vision and challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [\[CrossRef\]](#)
3. Zhu, C.; Zhou, H.; Leung, V.C.; Wang, K.; Zhang, Y.; Yang, L.T. Toward big data in green city. *IEEE Commun. Mag.* **2017**, *55*, 14–18. [\[CrossRef\]](#)
4. Chiniforooshan, E.; Farzan, A.; Mirzazadeh, M. Worst case optimal union-intersection expression evaluation. In Proceedings of the International Colloquium on Automata, Languages, and Programming, Lisbon, Portugal, 11–15 July 2005; Springer: New York, NY, USA, 2005; pp. 179–190.
5. Chiniforooshan, E.; Farzan, A.; Mirzazadeh, M. Evaluation of general set expressions. In Proceedings of the International Symposium on Algorithms and Computation, Gold Coast, Australia, 15–17 December 2008; Springer: New York, NY, USA, 2008; pp. 366–377.
6. Mirzazadeh, M. Adaptive Comparison-Based Algorithms for Evaluating Set Queries. Master's Thesis, University of Waterloo, Waterloo, ON, Canada, 2004.
7. Christopher, D.M.; Prabhakar, R.; Hinrich, S. Introduction to information retrieval. *Introd. Inf. Retr.* **2008**, *151*, 177.
8. Baeza-Yates, R.; Salinger, A. Experimental analysis of a fast intersection algorithm for sorted sequences. In Proceedings of the SPIRE, London, UK, 1–4 September 2015; Springer: New York, NY, USA, 2005; Volume 3772, pp. 13–24.
9. Baeza-Yates, R. A fast set intersection algorithm for sorted sequences. In Proceedings of the Annual Symposium on Combinatorial Pattern Matching, Istanbul, Turkey, 5–7 July 2004; Springer: New York, NY, USA, 2004; pp. 400–408.
10. Barbay, J.; Kenyon, C. Adaptive intersection and t-threshold problems. In Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA, 6–8 January 2002; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002; pp. 390–399.
11. Demaine, E.D.; López-Ortiz, A.; Munro, J.I. Adaptive set intersections, unions, and differences. In Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), San Francisco, CA, USA, 9–11 January 2000; Citeseer: Gaithersburg, MD, USA, 2000.
12. Demaine, E.D.; López-Ortiz, A.; Munro, J.I. Experiments on adaptive set intersections for text retrieval systems. In Proceedings of the ALENEX, Washington, DC, USA, 5–6 January 2001; Springer: New York, NY, USA, 2001; Volume 1, pp. 91–104.
13. Bille, P.; Pagh, A.; Pagh, R. Fast evaluation of union-intersection expressions. In Proceedings of the International Symposium on Algorithms and Computation, Sendai, Japan, 17–19 December 2007; Springer: New York, NY, USA, 2007; pp. 739–750.
14. Hu, M.; Zhuang, L.; Wu, D.; Zhou, Y.; Chen, X.; Xiao, L. Learning driven computation offloading for asymmetrically informed edge computing. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 1802–1815.
15. Hu, M.; Xie, Z.; Wu, D.; Zhou, Y.; Chen, X.; Xiao, L. Heterogeneous edge offloading with incomplete information: A minority game approach. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2139–2154. [\[CrossRef\]](#)

16. Chalapathi, G.S.S.; Chamola, V.; Vaish, A.; Buyya, R. Industrial internet of things (iiot) applications of edge and fog computing: A review and future directions. In *Fog/Edge Computing for Security, Privacy, and Applications*; Advances in Information Security; Springer: New York, NY, USA, 2021; pp. 293–325.
17. Zhao, L.; Zheng, T.; Lin, M.; Hawbani, A.; Shang, J.; Fan, C. SPIDER: A Social Computing Inspired Predictive Routing Scheme for Softwarized Vehicular Networks. *IEEE Trans. Intell. Transp. Syst.* **2021**, 1–12. [\[CrossRef\]](#)
18. Li, X.; Zhou, Z.; Guo, J.; Wang, S.; Zhang, J. Aggregated multi-attribute query processing in edge computing for industrial IoT applications. *Comput. Netw.* **2019**, *151*, 114–123. [\[CrossRef\]](#)
19. Zhao, L.; Li, Z.; Al-Dubai, A.Y.; Min, G.; Li, J.; Hawbani, A.; Zomaya, A.Y. A Novel Prediction-Based Temporal Graph Routing Algorithm for Software-Defined Vehicular Networks. *IEEE Trans. Intell. Transp. Syst.* **2021**, 1–16. [\[CrossRef\]](#)
20. Kaur, K.; Garg, S.; Aujla, G.S.; Kumar, N.; Rodrigues, J.J.; Guizani, M. Edge computing in the industrial internet of things environment: Software-defined-networks-based edge-cloud interplay. *IEEE Commun. Mag.* **2018**, *56*, 44–51. [\[CrossRef\]](#)
21. Yuan, L.; He, Q.; Tan, S.; Li, B.; Yu, J.; Chen, F.; Jin, H.; Yang, Y. Coopedge: A decentralized blockchain-based platform for cooperative edge computing. In *Proceedings of the Web Conference 2021, Ljubljana, Slovenia*, 19–23 April 2021; pp. 2245–2257.
22. Aslanpour, M.S.; Toosi, A.N.; Cicconetti, C.; Javadi, B.; Sbarski, P.; Taibi, D.; Assuncao, M.; Gill, S.S.; Gaire, R.; Dustdar, S. Serverless edge computing: Vision and challenges. In *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, Dunedin, New Zealand, 1–5 February 2021; pp. 1–10.
23. Baldini, I.; Castro, P.; Chang, K.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Slominski, A.; et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*; Springer: New York, NY, USA, 2017; pp. 1–20.
24. Castro, P.; Ishakian, V.; Muthusamy, V.; Slominski, A. The rise of serverless computing. *Commun. ACM* **2019**, *62*, 44–54. [\[CrossRef\]](#)
25. Zhao, L.; Wang, C.; Zhao, K.; Tarchi, D.; Wan, S.; Kumar, N. INTERLINK: A Digital Twin-Assisted Storage Strategy for Satellite-Terrestrial Networks. *IEEE Trans. Aerosp. Electron. Syst.* **2022**. [\[CrossRef\]](#)
26. Cicconetti, C.; Conti, M.; Passarella, A. A decentralized framework for serverless edge computing in the internet of things. *IEEE Trans. Netw. Serv. Manag.* **2020**, *18*, 2166–2180. [\[CrossRef\]](#)
27. Lenarduzzi, V.; Panichella, A. Serverless testing: Tool vendors’ and experts’ points of view. *IEEE Softw.* **2020**, *38*, 54–60. [\[CrossRef\]](#)
28. Rahmani, A.M.; Mohammadi, M.; Mohammed, A.H.; Karim, S.H.T.; Majeed, M.K.; Masdari, M.; Hosseinzadeh, M. Towards data and computation offloading in mobile cloud computing: taxonomy, overview, and future directions. *Wirel. Pers. Commun.* **2021**, *119*, 147–185. [\[CrossRef\]](#)
29. Huda, S.A.; Moh, S. Survey on computation offloading in UAV-Enabled mobile edge computing. *J. Netw. Comput. Appl.* **2022**, *201*, 103341. [\[CrossRef\]](#)
30. Hazra, A.; Amgoth, T. CeCO: Cost-efficient Computation Offloading of IoT Applications in Green Industrial Fog Networks. *IEEE Trans. Ind. Inform.* **2021**, *18*, 6255–6263. [\[CrossRef\]](#)
31. Guo, M.; Li, Q.; Peng, Z.; Liu, X.; Cui, D. Energy harvesting computation offloading game towards minimizing delay for mobile edge computing. *Comput. Netw.* **2022**, *204*, 108678. [\[CrossRef\]](#)
32. Peng, G.; Wu, H.; Wu, H.; Wolter, K. Constrained multiobjective optimization for IoT-enabled computation offloading in collaborative edge and cloud computing. *IEEE Internet Things J.* **2021**, *8*, 13723–13736. [\[CrossRef\]](#)
33. Sarkar, I.; Adhikari, M.; Kumar, N.; Kumar, S. A Collaborative Computational Offloading Strategy for Latency-sensitive Applications in Fog Networks. *IEEE Internet Things J.* **2021**, *9*, 4565–4572. [\[CrossRef\]](#)
34. Seid, A.M.; Boateng, G.O.; Anokye, S.; Kwantwi, T.; Sun, G.; Liu, G. Collaborative Computation Offloading and Resource Allocation in Multi-UAV-Assisted IoT Networks: A Deep Reinforcement Learning Approach. *IEEE Internet Things J.* **2021**, *8*, 12203–12218. [\[CrossRef\]](#)
35. Li, Y.; Xu, S.; Li, D. Deep Reinforcement Learning for Collaborative Computation Offloading on Internet of Vehicles. *Wirel. Commun. Mob. Comput.* **2021**, *2021*, 13. [\[CrossRef\]](#)
36. Wang, T.; Liang, Y.; Zhang, Y.; Zheng, X.; Arif, M.; Wang, J.; Jin, Q. An intelligent dynamic offloading from cloud to edge for smart iot systems with big data. *IEEE Trans. Netw. Sci. Eng.* **2020**, *7*, 2598–2607. [\[CrossRef\]](#)
37. Zhang, L.; Zou, Y.; Wang, W.; Jin, Z.; Su, Y.; Chen, H. Resource allocation and trust computing for blockchain-enabled edge computing system. *Comput. Secur.* **2021**, *105*, 102249. [\[CrossRef\]](#)
38. Ranaweera, P.; Jurecut, A.D.; Liyanage, M. Survey on multi-access edge computing security and privacy. *IEEE Commun. Surv. Tutor.* **2021**, *23*, 1078–1124. [\[CrossRef\]](#)
39. Liu, G.; Wang, C.; Ma, X.; Yang, Y. Keep your data locally: Federated-learning-based data privacy preservation in edge computing. *IEEE Netw.* **2021**, *35*, 60–66. [\[CrossRef\]](#)
40. Ding, X.; Lv, R.; Pang, X.; Hu, J.; Wang, Z.; Yang, X.; Li, X. Privacy-preserving task allocation for edge computing-based mobile crowdsensing. *Comput. Electr. Eng.* **2022**, *97*, 107528. [\[CrossRef\]](#)
41. Garg, S.; Singh, A.; Kaur, K.; Aujla, G.S.; Batra, S.; Kumar, N.; Obaidat, M.S. Edge computing-based security framework for big data analytics in VANETs. *IEEE Netw.* **2019**, *33*, 72–81. [\[CrossRef\]](#)
42. Stephanie, V.; Chamikara, M.; Khalil, I.; Atiquzzaman, M. Privacy-preserving location data stream clustering on mobile edge computing and cloud. *Inf. Syst.* **2022**, *107*, 101728. [\[CrossRef\]](#)
43. Wang, J.; Wu, L.; Zeadally, S.; Khan, M.K.; He, D. Privacy-preserving data aggregation against malicious data mining attack for IoT-enabled smart grid. *ACM Trans. Sens. Netw.* **2021**, *17*, 1–25. [\[CrossRef\]](#)

- 
44. Li, H.; Cheng, Q.; Li, X.; Ma, S.; Ma, J. Lightweight and Fine-Grained Privacy-Preserving Data Aggregation Scheme in Edge Computing. *IEEE Syst. J.* **2021**, *16*, 1832–1841. [[CrossRef](#)]
  45. Liu, Z.; Cao, Z.; Dong, X.; Zhao, X.; Bao, H.; Shen, J. A verifiable privacy-preserving data collection scheme supporting multi-party computation in fog-based smart grid. *Front. Comput. Sci.* **2022**, *16*, 1–11. [[CrossRef](#)]
  46. West, D.B. *Introduction to Graph Theory*; Prentice Hall: Upper Saddle River, NJ, USA, 2001; Volume 2.
  47. Moret, B.M. Decision trees and diagrams. *ACM Comput. Surv.* **1982**, *14*, 593–623. [[CrossRef](#)]
  48. Greiner, R.; Hayward, R.; Jankowska, M.; Molloy, M. Finding optimal satisficing strategies for and-or trees. *Artif. Intell.* **2006**, *170*, 19–58. [[CrossRef](#)]
  49. Li, C.; Lu, J.; Lu, Y. Efficient merging and filtering algorithms for approximate string searches. In Proceedings of the IEEE 24th International Conference on Data Engineering, ICDE 2008, Cancun, Mexico, 7–12 April 2008; pp. 257–266.
  50. Bentley, J.L.; Yao, A.C.C. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.* **1976**, *5*, 82–87. [[CrossRef](#)]
  51. Mosa, A.S.M.; Yoo, I. A study on PubMed search tag usage pattern: Association rule mining of a full-day PubMed query log. *BMC Med Inform. Decis. Mak.* **2013**, *13*, 8. [[CrossRef](#)]