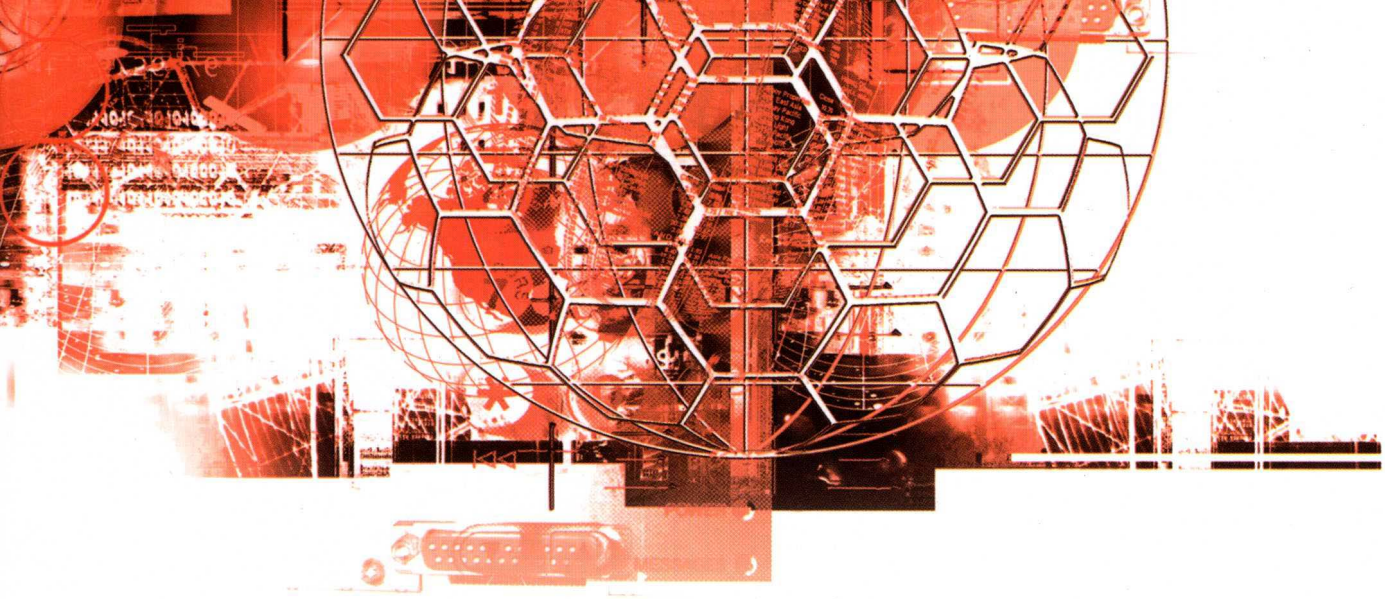

System Simulation Technology & Application (Vol. 17)

陈宗海 主编

系统仿真技术及其应用

·第17卷·

中国科学技术大学出版社

系统仿真技术及其应用

·第17卷·

System Simulation Technology & Application

(Vol. 17)

陈宗海　主编

中国科学技术大学出版社

2016·合肥

内 容 简 介

本书为中国自动化学会系统仿真专业委员会和中国系统仿真学会仿真技术应用专业委员会的学术会议的论文选编。

本书收录了会议论文70篇，是近年来系统仿真科学与技术在自然科学和社会科学各领域以及航空、航天、石油、化工、能源、国防、轻工等行业中应用的最新成果，以及建模与仿真学、复杂系统新领域等的最新进展。

本书可供科研、设计部门和厂矿企业中系统仿真科学与技术的研究和应用人员以及高等学校相关专业师生参考。

图书在版编目(CIP)数据

系统仿真技术及其应用. 第17卷/陈宗海主编. —合肥：中国科学技术大学出版社，2016. 8
ISBN 978-7-312-4059-7

Ⅰ. 系… Ⅱ. 陈… Ⅲ. 系统仿真—文集 Ⅳ. TP391. 9-53

中国版本图书馆CIP数据核字(2016)第204138号

出版 中国科学技术大学出版社
安徽省合肥市金寨路96号，230026
http://press.ustc.edu.cn
印刷 安徽联众印刷有限公司
发行 中国科学技术大学出版社
经销 全国新华书店
开本 880 mm×1230 mm 1/16
印张 22.25
字数 890千
版次 2016年8月第1版
印次 2016年8月第1次印刷
定价 186.00元

《系统仿真技术及其应用》编委会

写在卷首

“建模与仿真技术同高性能计算一起，正成为继理论研究和实验研究之后，第三种认识改造客观世界的重要手段。”(李伯虎院士)随着计算机、通信、控制等高科技的不断发展，大数据、云计算、物联网、车联网等新技术的不断涌现，人工智能、虚拟现实等技术获得突破性发展，第三次产业革命的大幕正逐渐拉开，“仿真”这个古老而又年轻的学科更加朝气蓬勃地散发着无限的青春活力。

信息社会化的进程，使得仿真科学与技术面对丰富多彩的客观世界。信息化和信息社会化，使人类处理的系统规模与复杂性日益增长，人类对系统的认识和研究逐步深化，可利用的信息资源的影响已具有全球的性质，同时对知识性工作自动化的需求也逐渐迫切起来。这个信息社会化和知识自动化迅猛发展的背景，推动了系统仿真方法学的革新、发展与进步。

近年来，建模与仿真方法学致力于更自然地抽取事物的特征、属性和实现其更直观映射描述，寻求使模型研究者更自然地参与仿真活动的方法。现阶段依托包括网络、多媒体等在内的计算机技术、通信技术等科技手段，通过友好的人机界面构造完整的计算机仿真系统，提供强有力的、具有丰富功能的软硬件营造的仿真环境，使开放复杂巨系统的模型研究，从单纯处理数学符号映射的计算机辅助仿真(CAS)，到强化包括研究主体(人)在内的具有多维信息空间的映射与处理能力，逐步创建人、信息、计算机融合的智能化、集成化、协调化高度一体的仿真环境，构建信息和物理深度融合的系统(CPS)。可见，信息时代的到来正在孕育着系统仿真科学和技术某些新的突破。正待开发的系统仿真方法和仿真技术广阔无垠，需要我们从事系统仿真的科技工作者付出艰辛的劳动，使仿真这门迄今为止最有效、最经济的综合方法和推动技术进步的战略技术在现代化进程中发挥更大的促进作用。

人类社会已进入21世纪的第二个十年，随着云计算、大数据、高速无线通信等信息技术的兴起，信息革命正以“数据化”这一崭新还富有冲击性的形式影响人类生活的方方面面。随着数据获取成本的降低，数据采集精度、数据存储设备的性能和容量的提高，人类社会正在经历一个全面“数据化”的过程，现实的物理世界在数字化的“赛博空间”(Cyberspace)中的投影越来越清晰、越来越丰富，数据正在创造一个新的世界。继物理世界、人的精神世界之后，由计算机、通信、控制等数字化技术构建的“赛博空间”正在成为人类生活中不可或缺的一部分。数据不仅是企业的重要宝藏，也是赛博空间的氧气，离开了数据，人们在未来世界中将无法生存。在赛博空间中，人们不仅可以进行各种社交活动、游戏，也可以进行教育、科研、实验等各种具有创造性的社会活动，赛博空间不应该只有消耗，没有产出。但这个世界和物理世界不是割裂的，它是现实世界的部分投影以及人类心灵世界的部分投影。同时，赛博空间也是物理空间与人类精神世界之间的桥梁。仿真科学与技术应该在这一股数据洪流席卷世界的大潮中扮演什么样的角色，是值得所有从事仿真科学工作的研究者共同思考和探讨的问题。

由中国自动化学会系统仿真专业委员会联合中国系统仿真学会仿真技术应用专业委员会主办的“第17届中国系统仿真技术及其应用学术年会(17th CCSSTA 2016)”,共收到论文100篇,录用70篇。其中,大会特邀报告2篇;建模与仿真技术17篇;系统仿真15篇;航天与装备仿真9篇;控制与决策及其他22篇。另有两个专题:灰色定性仿真1篇;大数据与云计算4篇。收录的论文涉及广泛的领域,内容丰富多彩,反映了当前学科发展的方向和技术应用的水平。这次学术交流,无疑将对我国系统仿真科学与技术的发展起到积极的推进作用。

编 委 会

2016年6月于中国科学技术大学

目　录

第一部分　大会报告

第二部分　建模与仿真方法

第三部分　系统仿真

第四部分　航天与装备仿真

第五部分　控制与决策

第六部分　灰色定性仿真

第七部分　大数据与云计算

第一部分

大会报告

Keyword Search over Semantic Internet of Things Using MapReduce

Huang Hai, Chen Zong-hai
(Department of Automation, University of Science and Technology of China, Hefei, Anhui, 230026)

Abstract: The Internet of Things (IoT) has recently received considerable interest from both academia and industry that are working on technologies to develop the future Internet. Semantic technologies based on machine-interpretable representation formalism (such as RDF) have shown promise for describing objects, sharing and integrating information. The addition of semantics has also helped create machine-interpretable and self-descriptive data in the IoT domain. However, the increasing scale of RDF data brings challenges to search over Internet of things, making this problem even more time consuming to solve. Existing solutions, developed to run on a single machine, only work well on small graphs. In this paper, we target on answering keyword queries over semantic internet of things on multiple computing nodes. The scalability issue was solved by employing MapReduce jobs to compute answers through iteratively joining the building blocks generated by each computing node. The experimental results on synthetic and real-world datasets verify the efficacy of our methods.
Key words: Semantic Web of Things; RDF; Cloud Computing; MapReduce

1 Introduction

It is estimated that there will be around 25 billion devices connected to the Internet by 2015 and 50 billion by 2020. Such a huge number of highly distributed and heterogeneous devices will need to be interconnected and communicate in different scenarios autonomously. This implies that providing interoperability among the things on the IoT is one of the most fundamental requirements to support object addressing, tracking, and discovery as well as information representation, storage, and exchange. Using semantic annotations in the IoT domain provides machine-readable and machine-interpretable metadata to describe the IoT resources and data. The semantic Web technologies include well-defined standards and description frameworks (e. g. RDF, OWL, SPARQL) and a variety of open-source and commercial tools for creating, managing, querying, and accessing semantic data.

In this paper, we focus on implementing efficient keyword query processing over semantic web of things which can be regarded as a large RDF data graph. Keyword query enables inexperienced users to easily search the data graph with no specific knowledge of complex structured query languages (such as SPARQL).

Searching or analysing a graph even with millions of nodes would be time-consuming at best and totally impossible at worst, especially when the graph cannot be stored in memory on a single computer. Thus, we hope to solve this problem using parallel data processing techniques on a cluster of computing nodes. We believe that the most crucial problem for real-world keyword query systems is to return users answers as quickly as they can (say in a few seconds), even these answers are not the complete or best. We believe that the most crucial problem for real-world keyword query systems is to return users answers as quickly as they can (say in a few seconds), even these answers are not complete or best. Intuitively, keyword query over a large RDF graph can be solved in a Divide-and-Conquer strategy.

However, given a RDF data partitioned and distributed across computing nodes, the communication cost is unavoidable during keyword search. Since each computing node holds only a partial graph, information has to be exchanged among computing nodes to find the global optimal answers. The potential multiple rounds of communication over the network would lead to high query latencies. Thus, there is a trade-off between the

Huang Hai (1980—), male, Anhui Hefei, associate researcher, Ph. D. Research interest: semantic internet of things; **Chen Zong-hai** (1963—), male, Anhui Tongcheng, professor, doctoral supervisor. Research interest: robotics, intelligent system, new energy technology.

quality of query results and the response time. Our target is to maximize the quality of local query answers, and minimize the response time in both local search and global boosting process.

To achieve the above mentioned goal, we use MapReduce jobs to iteratively joining the building blocks generated by each computing node. The intermediate results exchanged on the network are minimized to reduce the communication overhead. The iterations stop when no new building block appears.

Our proposed method has the following novelties and advantages:

- We parallelize the process of keyword search over large RDF graphs on a cluster of computing nodes. MapReduce is introduced to compute the global optimal results based on building blocks of subgraphs for the first time.
- The evaluation on both synthetic and real-world datasets shows the efficacy of our method, which can return results in a short time.

2 Problem Definition

Definition 1 *Given a set of RDF triples D, an RDF graph $G_D(V, E)$ is a directed graph where V is the set of nodes (RDF subjects (s) or objects (o) in D) labelled with s or o, and E is the set of edges in G_D labelled with predicates.*

There are many definitions for keyword query answers that are returned to users such as Steiner tree, distinct rooted tree, r-radius subgraph[1]. In this paper, we define keyword query answers as distinct rooted trees.

Definition 2 (Keyword Query Answers) *Given an RDF data graph G and a keyword query $Q(k_1, k_2, k_3, \cdots, k_l)$ where each k_i is a keyword, S_i is the set of nodes in G that contains keyword k_i. An answer $\langle r, (n_1, n_2, \cdots, n_l)\rangle$ of Q is a tree with a root r such that for each $n_i \in S_i$, there exists a directed path from r to n_i.*

Definition 3 *Given a keyword query Q on graph G, the score of an answer $T\langle r, (n_1, n_2, \cdots, n_l)\rangle$ is defined as follows:*

$$Score(T) = \sum_{i=1}^{l} dist(r, n_i) \quad (1)$$

where dist (r, n_i) is the shortest distance from r to keyword node n_i in G.

Note that the score definition above adopts "match-distributive semantics". In this semantics, all root-match paths contribute independently to the final score, even if these paths may share some common edges.

Similar to [2], given a keyword query Q on RDF graph G, we are concerned with finding the top-k best answers, which have distinct roots.

Definition 4 *Given an RDF data graph G and a keyword query $Q(k_1, k_2, k_3, \cdots, k_l)$ where each k_i is a keyword, the best score of answer T with root r is the maximum score over all answers rooted at r. And we represent this best answer with root r as $T\langle r, (k_1, k_2, \cdots, k_l)\rangle$.*

PROBLEM DEFINITION *Given an RDF graph G and a keyword query Q, we are interested in finding the top-k distinct rooted trees in score increasing order.*

3 Partition of RDF Data Graph

Firstly, we manage the large RDF graph by partitioning it into subgraphs and store them on multiple nodes. Obviously, the quality of partition is crucial to both the efficiency and answer quality of keyword search.

A simple way to partition a large RDF graph into subgraphs is by a hash partitioning algorithm. The nodes containing the same keyword are grouped into one subgraph and stored on the same machine. However, a keyword query normally contains multiple keywords. If nodes including different keywords locate on different machines, the communication overhead to find the common root node connecting keyword nodes would be very huge. Thus, this partition method is unwise and we need a better one that can minimize the potential partition-induced communication.

As we know, the process of keyword search is to find tree structures with roots connecting keyword nodes.

Considering that answer trees normally consist of nodes with small distances to each other, we should partition nodes with high connectivities into one subgraph to increase the ability of single computing nodes for answering keyword queries independently. Meanwhile, to reduce the communication cost during keyword search, the number of edges crossing subgraphs should be minimized.

Last but not least, the size of partitioned subgraphs should be balanced as comparable workloads are required to be distributed on the computing nodes by MapReduce.

Graph partition problem has been well-studied and some partition tools have also been developed. In this work, we use the METIS[1] partition tool, it fulfils our aforementioned requirements.

METIS aims to minimize the edge crossings (or the total weight of them) and keep the close nodes in the same subgraph, while balancing the partition size. We first use METIS to divide nodes of a RDF graph into disjoint

1 http://glaros.dtc.umn.edu/gkhome/views/metis/.

partitions. Then we add all outgoing edges to each partition. If node v_i falls in partition G_i, all outgoing edges v_i, v_k of v_i in the original RDF graph are added in partition G_i.

Once the RDF graph partitioned into subgraphs, each subgraph consists of all nodes in a partition as well as all portals incident to the partition. For a subgraph, a portal node can be either in-portal node or out-portal node or both. For an in-portal node, it has at least one incoming edge from another block and at least one outgoing edge in this block. For an out-portal node it has at least one outgoing edge to another block and at least one incoming edge from this block. For example, in Figure 1, node e_1, e_2 are portal nodes. For partition S_1, e_1 is an out-portal node and e_2 is an in-portal node.

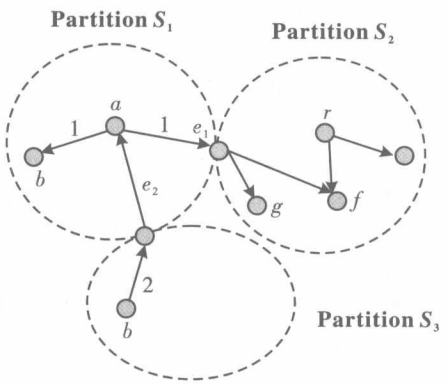

Figure 1 Portal nodes of subgraphs

4 Search Algorithms

In the section, we present our method of computing global top-k answers using MapReduce jobs. We first give some introduction about MapReduce framework.

MapReduce[3] is a programming model for processing large data sets, which cannot be handled on a single machine. Data are partitioned over multiple computing nodes by storing in a distributed file system, e. g., HDFS in Hadoop[2] that is the most popular implementation of MapReduce. Data processing is conducted through Map and Reduce functions on computing nodes.

The Map and Reduce functions of MapReduce are both defined with respect to data structured in (*key*, *value*) pairs.

The Map function takes inputs and produces a set of intermediate *key*/*value* pairs. Through shuffling, all pairs with the same key from all Map functions are grouped together, and then passed to the Reduce function.

The Reduce function accepts an intermediate key and a set of values for that key, and then merges together these values to form a possibly smaller set of values. The data flow of the MapReduce framework is shown in Figure 2.

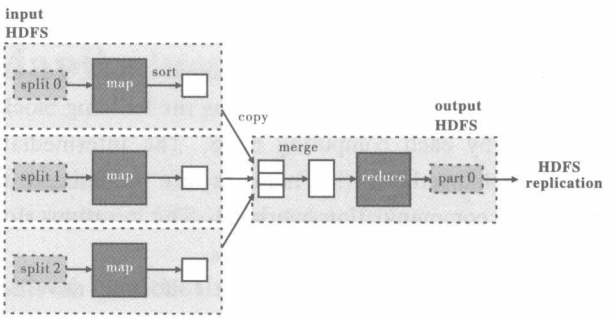

Figure 2 MapReduce Data Flow

4.1 Generating Building Blocks

The core problem of the second search phase is how to obtain the potential answers that span in different computingnodes under MapReduce framework.

As data graph is distributed across multiple nodes, MapReduce-based methods are not able to maintain global structures during computation. Computing nodes executing Map and Reduce jobs cannot communicate to have awareness of global status. Thus, given an RDF graph represented as a file of adjacent node lists stored on HDFS, passing information is only possible from a node to its adjacent nodes. Local information has to be passed multiple iterations to finally reach global answers.

A simple way to compute keyword queries is using edges as basic building blocks. For each iteration, if the two building blocks have the same node (two edges joined by one node), Reduce function joins them and emits a new building block (which could be a path or a tree structure). All keyword query answers can be obtained by iteratively using Reduce function on all building blocks until no more blocks can be joined.

However, this approach will lead to some serious problems. First, the amount of intermediate data generated in each iteration could be very huge especially for dense graphs. In fact, most of intermediate data has no contribution to generate the potential keyword query answers. Second, executing MapReduce jobs iteratively is not very efficient. If we build answer trees from edges, the possible number of iteration steps could be large, which will lead to low efficiency. Thus, we hope to find a way which could decrease the amount of intermediate data generated and the number of iteration steps. Meanwhile we also hope to maximally employ the index we build for the first search phase.

To avoid redundant computation and too much generation of intermediate data, our strategy is to divide

2 Apache Hadoop http://hadoop.apache.org.

the keyword query Q into some sub queries by grouping keywords to multiple disjoint sets, under the assumption that the keywords from the same set locate on one computing node. For example, given a sub query $Q_i(\{k_1, k_2\}, \{k_3\})$ *of* $Q(k_1, k_2, k_3)$, the answers of Q_i span on two computing nodes such that k_1, k_2 are on the same computing node, and k_3 is on another one.

For a keyword query Q with m keywords, the number N of sub queries can be computed as:

$$N = C_m^0 + C_m^1 + C_m^2 + \cdots + C_m^{\lfloor \frac{m}{2} \rfloor} \quad (2)$$

For example, the query with 3 keywords would have 4 sub queries.

For each sub query, we can dynamically generate a set of tree structures and paths instead of edges in each partition as basic building blocks, which have high granularities and are very useful for consisting the potential answer trees. We also define the join nodes for each building block to build new blocks. And by the index we build for the first search phase, these blocks can be computed efficiently. Three basic building blocks in each partition are defined as follows:

- **TYPE Ⅰ** *This type of building block is a rooted tree with a root node r connecting a set of keywords* $(k_1, \cdots, k_n)$ *and a set of out-portal nodes* E_o *to which r is reachable. We denote it by a 6-tuple* $\langle E_o, r, (k_1, \cdots, k_n), S_{id}, score, \rho \rangle$ *such that the join nodes of this building block is the nodes in* E_o; *and* S_{id}, *score*, ρ *indicates the partition id, score value and the path information of this block, respectively.*
- **TYPE Ⅱ** *This type of building block is a rooted tree with an in-portal node* e_i *as a root node connecting a set of keyword nodes* $(k_1, \cdots, k_n)$. *We denote it by a 6-tuple* $\langle e_i, e_i, (k_1, \cdots, k_n), S_{id}, score, \rho \rangle$ *such that the join node is* e_i.
- **TYPE Ⅲ** *This type of building block is a shortest path from an in-portal node* e_i *to an out-portal node* e_o, *denoted by* $\langle e_i, e_i, e_0, S_{id}, score, \rho \rangle$.

For example, in Figure 3, there exists a TYPE Ⅰ building block: $\langle \{e_1\}, a, (k_1), S_1, 2, \rho \rangle$ in partition S_1.

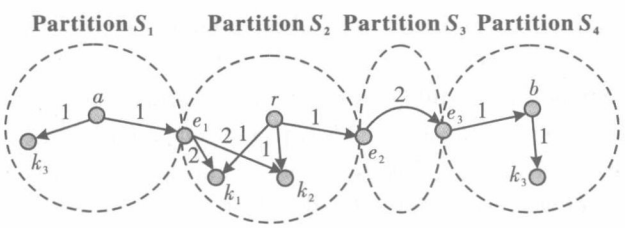

Figure 3 Answers span on multiple partition

TYPE Ⅰ and Ⅱ building blocks can be every efficiently online computed through our single-machine search algorithm and index. The number of these building blocks depends on the concrete keywords, but is normally limited to a small value. Thus, data copying across network during communication would not cost much.

TYPE Ⅲ building blocks are much more than TYPE Ⅰ and TYPE Ⅱ building blocks, but can be precomputed offline since they are independent of keywords. Storing them in HDFS will highly facilitate the block joining process as no communication overhead will occur.

Note that the root nodes in building blocks of TYPE Ⅰ are potential answers. Thus, we only consider joins between building blocks of TYPE Ⅰ and TYPE Ⅱ, TYPE Ⅲ building blocks. To join two building blocks, we define the join rules as follows:

Given two building blocks bk_1 (TYPE Ⅰ), bk_2 (TYPE Ⅱ or TYPE Ⅲ) with different partition IDs, keys(bk_1), keys(bk_2) denote the keyword set of bk_1 and bk_2. The join nodes set of bk_1 is E_o and the join node of bk_2 is e. If $keys(bk_1) \cap keys(bk_2 = \rho$ and $e \in E_o$, then we have:

- TYPE Ⅰ ∞ TYPE Ⅱ: $\langle E_o, r, (k_1, \cdots, k_m), S_{id1}, score_1, \rho_1 \rangle \infty \langle e, (k_{m+1}, \cdots, k_j), S_{id2}, score_2, \rho_2 \rangle = \langle E_o - \backslash \{e\}, r, (k_1, \cdots, k_j), S_{id1}, score_1 + score_2, \rho_{12} \rangle$.
- TYPE Ⅰ ∞ TYPE Ⅲ: $\langle E_o, r, (k_1, \cdots, k_m), S_{id1}, score_1, \rho_1 \rangle \infty \langle e, e, e', S_{id2}, score_2, \rho_2 \rangle = \langle E_o - \{e\} \cup \{e'\}, r, (k_1, \cdots, k_m), S_{id1}, score_1 + score_2, \rho_{12} \rangle$.

Obviously, if a new generated building block contains all keywords in query Q, it should be a potential answer for query Q.

For each sub query we iteratively use MapReduce function to compute potential keyword query answers. The Map function takes each building block and emits an intermediate *key/value* pair. The key is the join node of each building block and value is the whole building block. For TYPE Ⅰ building blocks, since they have multiple join nodes we select each join node as key and emit multiple *key/value* pairs. The second task of map function is to delete all building blocks with score values exceeding a threshold γ.

The pruning with threshold γ is very helpful as it limits the amount of data transferred across the network and therefore can speed up the moving towards final global answers. The threshold γ can be set to a user specified value or set to a suggested value generated from the first search phase.

The Reduce function executes the job of joining two building blocks to generate a new one. Reduce function also checks if there exists any redundant building block,

which will not be emitted.

The whole process is outlined in **Algorithm 1** and **Algorithm 2**.

Algorithm 1 Map Function

```
Input: Building block bk, threshold γ;
1: if bk.score <= γ then
2:     if bk is TYPE Ⅱ or TYPE Ⅲ then
3:         Node = bk.JoinNode;
4:         EMIT(Node, bk)
5:     else
6:         for all portal nodes e ∈ bk.Eo do
7:             bk.Eo = bk.Eo − e
8:             EMIT(e, bk)
9:         end for
10:    end if
11: end if
```

Algorithm 2 Reduce Function

```
Input: node, building blocks [bk1, bk2, ···, bkn];
1: for all bki do
2:     Node = bk.JoinNode;
3:     EMIT(Node, bk)
4: end for
5: for all bk, bj do
6:     if bk, bj are from different partitions AND bk ≠ bj then
7:         bkj = bk ∞ bj;
8:         node = bkj.JoinNode;
9:         EMIT(Node, bkj)
10:    end if
11: end for
```

4.2 Iteration Termination Condition

To achieve the global query answers, we have to iterate the MapReduce function. A problem would arise: when do we stop the iteration? Intuitively, the maximal number of iteration can be set as the number of partitions, since at each iteration we join building blocks from different partitions. However, this termination condition may allow unnecessary joining iterations and therefore delay the report of query answers, when the number of partition is big.

To terminate iterations at the right time, we introduce the concept of "valid join" as follows:

Definition 5 *Given a building block b_{ik} generated from block b_i and block b_k, if the score of b_{ik} is less than threshold γ and if b_{ik} is never generated before, we say that $b_i \infty b_k$ is a valid join.*

Obviously, if there is no "*valid join*", we can stop the iteration. One example of iterative joining process is shown in Figure 4.

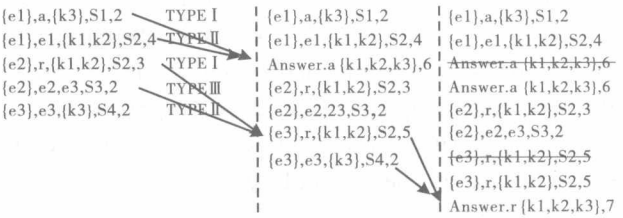

Figure 4 Iteration steps of joining building blocks

5 Experiments

In this section we describe the performance evaluation of the proposed methods.

5.1 Experimental Methodology

We ran experiments on 3 Dell T7500 workstations connected with a 1 GB network switch. Each Dell workstation has 6 Intel Xeon processor X5650 2.67 GHz with 12 cores, 24 GB of RAM, and a 1.4 TB hard disk. We virtualized 18 virtual machines. Each virtual machine is assigned 2 cores and 4 GB of RAM. We installed the Ubuntu 11.04, 64-bit, server edition operating system, Java 1.6 with a 64-bit server JVM, and Hadoop 0.20.1. We used an extra node to manage the Hadoop jobs and the Hadoop distributed file system. In order to maximize the parallelism and minimize the running time, we made the following changes to the default Hadoop configuration: we set the block size of the distributed file system to 128 MB, allocated 3 GB of virtual memory to each map/reduce task, set the replication factor to 1, and disabled the speculative task execution feature.

Data sets Both synthetic and real-world datasets are used in our experiments. In this paper, we present results on two datasets:

(1) LUBM[3] is developed by Lehigh University. It consists of a university domain ontology (containing 43 classes and 32 properties). We generated around 40 M distinct triples to form a synthetic LUBM data set.

(2) YAGO[4] consists of facts extracted from Wikipedia and integrated with the WordNet thesaurus and contains about 20 M triples. Compared with the former dataset, this real-world data set is relatively heterogeneous. More details of the two datasets are

3 http://swat.cse.lehigh.edu/projects/lubm/.

4 http://www.mpi-inf.mpg.de/yago-naga/yago/downloads\yago.html.

shown in Table 1.

Table 1 Statistics of datasets

#	LUBM	YAGO
# Edges	42,211,071	18,343,536
# Nodes	20,186,615	14,230,223
# Properties	43	93
# Keywords	4,233,534	8,278,931

Query Loads Query processing speed depends on not only the keyword search system, but also the content of query words.

A keyword system produces answers quickly for some queries, but for other queries it may take longer time or even fail to produce any answer after exhausting memory. Thus for fair comparison, we generated three kinds of query sets in our experiments: Q_2, Q_3, Q_4. Each query set has 30 keyword queries. Each query in query set Q_2, Q_3, Q_4 is composed of 2, 3, and more than 4 randomly selected keywords, respectively. Examples of keywords used to generate query sets are shown in Figure 5. Table 2 shows some statistic information about query sets used in the experiments.

Table 2 Statistics of query datasets

#	Q_2	Q_3	Q_4
# keywords	2	3	>=4
avg. # keyword nodes (LUBM)	3312	4467	5313
avg. # keyword nodes (YAGO)	1678	1567	3215

Figure 5 shows the running time when boosting global query answers. The execution time is dominated by the Map/Reduce iterations actually. For LUBM dataset, the execution time reported is less than 20 seconds, because the data graph of LUBM is relatively simple and the distances between nodes are small so that the number of Map/Reduce iterations are limited. On the contrary, the data graph of Yago is more complex and some shortest paths are longer than 20 steps.

Thus, more number of iterations are required to execute queries on it. Short queries, like the ones in Q_2 including less number of keywords than Q_3 and Q_4, can be processed quickly in less than 20 seconds. More number of computing nodes can speed up the process even more.

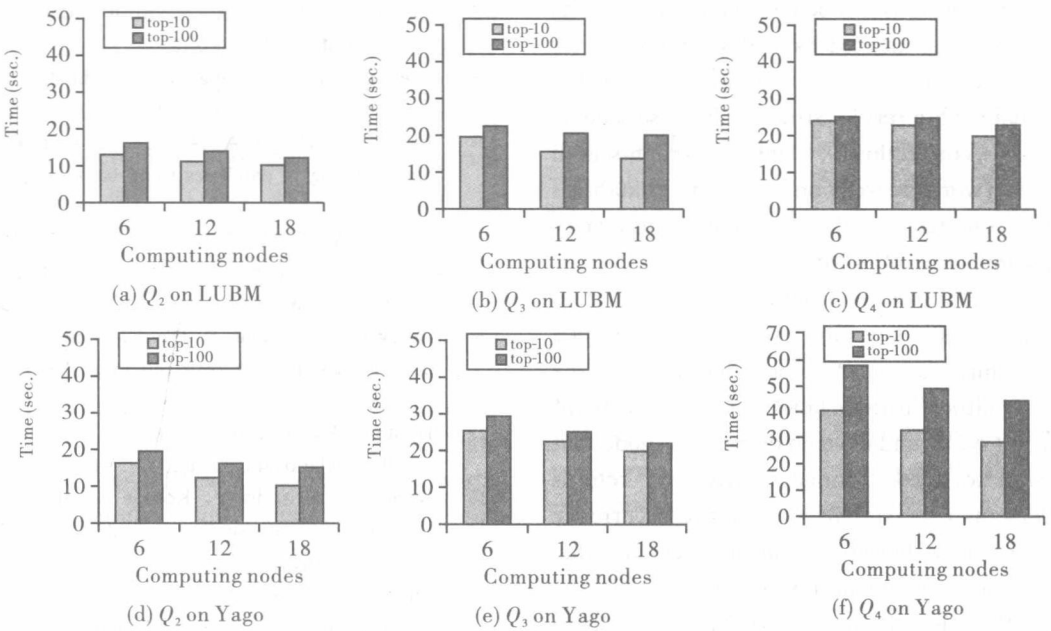

Figure 5 The search performance

6 Related Work

There are some related work on keyword search over RDF data graph. [6] proposed the method that aims at mapping the keyword query into one or more structured query. They assume that the user keyword-query is an implicit representation of a structured triple-pattern query. They try to infer such structured query using the RDF graph and retrieve the top-k most relevant structured queries. They then provide the user with the retrieved queries and let her choose the most appropriate structured query to be evaluated. Different with them, we compute the answers through traversing the RDF data graph instead of generating candidate

SPARQL queries. In fact, our problem is more related to keyword search over general graphs. In [7], the authors are concerned with the ranking function for RDF keyword search, and they use statistic language models to rank the results of keyword search. In our work, we focus on obtaining keyword query answers efficiently.

Among other approaches of keyword search over graphs[1, 2, 8-10], focusing on computing rooted trees as keyword search answers. BANKS[9] uses a backward search algorithm searching backwards from the nodes that contain keywords. BANKS-Ⅱ[10] is proposed to overcome the drawbacks of BANKS. The main idea of BANKS-Ⅱ is to start forward searches from potential roots. BLINKS[2] is proposed as a bi-level index to speed up BANKS-Ⅱ, as no index (except the keyword-node index) is used in BANKS-Ⅱ. The single-level index precomputes and indexes all the distances from the nodes to keywords, but this will incur very large index size. In this paper, we solve this problem by using bloom filter techniques and design our index which is able to fit a large number of keyword-node pairs in main memory.

Keyword search on relational databases[9, 11-14] is related to keyword search on graph. Conceptually, a database can be viewed as a labeled graph where tuples in different tables are treated as nodes connected via foreign-key relationships. Note that a graph constructed in this way usually has a regular structure because schema restricts node connections. Unlike graph-search approaches, keyword search on relational databases heavily relies on the database schema and query processing techniques in RDBMS.

Keyword search over XML data is also relevant to but different from keyword search on graph. XKSearch[15] returns a set of nodes that contain the query keywords either in their labels or in the labels of their descendant nodes and have no descendant node that also contains all keywords. Similarly, XRank[16] returns the set of elements that contain at least one occurrence of all of the query keywords, after excluding the occurrences of the keywords in sub elements that already contain all of the query keywords. However, all these techniques assume a tree-structure and thus cannot be directly applied to graph-structured data such as RDF graphs.

7 Conclusion

In this paper, we presented our method for efficient keyword search over internet of things which can be represented as large RDF data graphs on multiple computing nodes. It obtains global top-k results through iteratively joining the building blocks generated by each node under MapReduce framework. The experimental results verify the efficiency. In the future work, we will consider more efficient and better partition methods for large RDF graphs.

References

[1] Li G, Ooi B C, Feng J, et al. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data [C]. SIGMOD Conference, 2008: 903-914.

[2] He H, Wang H, Yang J, et al. Blinks: Ranked keyword searches on graphs [C]. SIGMOD Conference, 2007: 305-316.

[3] Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters[C]. OSDI, 2004:137-150.

[4] Guo Y, Pan Z, Heflin J. Lubm: A benchmark for owl knowledge base systems[J]. J. Web Sem., 2005, 3(2/3): 158-182.

[5] Suchanek F M, Kasneci G, Weikum G. Yago: A core of semantic knowledge[C]. WWW, 2007:697-706.

[6] Tran T, Wang H, Rudolph S, et al. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data[C]. ICDE, 2009: 405-416.

[7] ElbassuoniS, Blanco R. Keyword search over rdf graphs [C]. CIKM, 2011: 237-242.

[8] Ding B, Yu J X, Wang S, et al. Finding top-k min-cost connected trees in databases[C]. ICDE, 2007:836-845.

[9] Bhalotia G, Hulgeri A, Nakhe C, et al. Keyword searching and browsing in databases using banks[C]. ICDE, 2002: 431-440.

[10] Kacholia V, Pandit S, Chakrabarti S, et al. Bidirectional expansion for keyword search on graph databases [C]. VLDB, 2005:505-516.

[11] Agrawal S, Chaudhuri S, Das G. Dbxplorer: A system for keyword-based search over relational databases[C] ICDE, 2002:5-16.

[12] Hristidis V, Papakonstantinou Y. Discover: Keyword search in relational databases[C]. VLDB, 2002:670-681.

[13] Balmin A, Hristidis V, Koudas N, et al. A system for keyword proximity search on xml databases[C]. VLDB, 2003:1069-1072.

[14] Liu F, Yu C T, Meng W, et al. Effective keyword search in relational databases [C]. SIGMOD Conference, 2006: 563-574.

[15] Xu Y, Papakonstantinou Y. Efficient keyword search for smallest lcas in xml databases[C]. SIGMOD Conference, 2005:537-538.

[16] Guo L, Shao F, Botev C, et al. Xrank: Ranked keyword search over xml documents [C]. SIGMOD Conference, 2003:16-27.