Hindawi
Scientific Programming
Volume 2022, Article ID 3943442, 13 pages
https://doi.org/10.1155/2022/3943442

Research Article

Concept Tree-Based Event Matching Algorithm in Publish/Subscribe Systems

Zhi Yuan Zhang,[1] **Yu Jie Wang**,[2] **Xue Hu Huang**,[1] **and Kai Leung Yung**[3]

[1]*School of Computer Science and Technology, Civil Aviation University of China, Tianjin 300300, China*
[2]*State Key Laboratory of Air Traffic Management System and Technology, Nanjing 210014, China*
[3]*Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University, Hong Kong 999077, China*

Correspondence should be addressed to Zhi Yuan Zhang; zyzhangcauc@163.com

Received 8 March 2022; Revised 23 June 2022; Accepted 12 August 2022; Published 31 August 2022

Academic Editor: Daniel Mo

Semantic-based publish/subscribe system has attracted a lot of attention in recent years due to its powerful description ability in message dissemination scenarios. As a key part of semantic-based publish/subscribe systems, event matching needs to understand the semantic meaning of subscriptions, especially the hierarchy of concepts. However, existing event matching algorithms are severely affected by the complexity of concept hierarchy trees, some even cannot run due to high memory occupation. This article proposes an event matching algorithm called CTPS (concept tree-based publish/subscribe system) to address it; specifically, we build subscription indexes on concept hierarchy trees for the first time, employ bit arrays to avoid unnecessary matches, and use faster bit operations to accelerate the matching speed. Experiments show that, compared with previous algorithms, CTPS has less memory occupation and shorter event matching time, and its performance is not sensitive to the height of the concept hierarchy tree.

1. Introduction

Sending and receiving messages are very common in many scenarios, such as pushing restaurant or hotel information to online users in location-based service systems. Different from peer-to-peer message delivery methods, publish/subscribe (pub/sub) systems use a decoupled messaging architecture [1], in which a broker collects subscribers' subscription interests and publishers' published messages (often referred to as events in pub/sub systems) and route the events to corresponding subscribers automatically. In this way, an event can be received by multiple subscribers at the same time. Pub/sub systems can be classified into three types according to the subscription model, namely topic-based, content-based, and semantic-based. In topic-based pub/sub systems, events need to carry topic information when publishing, and subscribers also need to submit their interesting topics, and the broker matches them simply through topic information. IBM's MQ [2] and Apache's Kafka [3] are popular topic-based pub/sub system products. Event matching in topic-based pub/sub systems is highly efficient, but its subscription granularity is coarse and its description ability is weak. In content-based pub/sub systems, subscribers are no longer confined to topics but can specify constraints on the content of events. For example, if one wants to purchase a computer with a price less than 400$, in content-based pub/sub systems, he may send a subscription message of (<string, target, =, "computer"> ∧ <string, currency, =, "$"> ∧ <int, price, ≤, 400>) to the broker, where each constraint has a form of <*data type, attribute name, operator, value*>, and ∧ is a conjunction (logical and), which means every constraint must be satisfied. Content-based pub/sub systems such as SIENA [4, 5], REIN [6], GEM [7, 8], H-Tree [9], and BE-Tree [10]greatly enhance the description ability and make the system more flexible and have been a research focus in recent years.

Content-based pub/sub systems can only match subscriptions according to the structure information of events,

which is not enough to understand their semantic meanings, and thus may deviate from users' original intentions. For example, if there is a provider who is selling laptops with a price of 350$, he may publish an event message of (<string, target, "laptop"> <string, currency, "$"> ∧ <int, price, 350>) to the broker. However, in content-based pub/sub systems, without knowing the semantic relation between *computer* and *laptop*, the broker only finds that *laptop ! = computer*, thus will inevitably fail to deliver this message to the above subscriber. To address this problem, some researchers try to incorporate semantic information into pub/sub systems, such as S-ToPSS [11], which uses a very intuitive three-step process to solve the semantic matching problem: At first, it translates all attributes with different names but same meaning to a *root* attribute. Then, it uses a concept hierarchy tree to organize the hypernym-hyponym relations of concepts, that is to say, if the attribute/value in the subscription is the ancestor of the corresponding attribute/value in the event, the matching condition is also satisfied. And finally, it uses functions to express more flexible semantic equivalence relations, such as "professional experience = present date – graduation year." S-ToPSS replaces each attribute of an event with all its possible hypernym; in this way, one event may become multiple new events when matching. Obviously, this method artificially increases matching times and is inefficient. In subsequent research of semantic-based pub/sub systems, ontology is often used to model events, and RDF (resource description framework) graph, which is similar to ontology querying, is often used to model subscriptions. As ontology is a general framework for knowledge and concepts, the description ability of semantic-based pub/sub systems such as OPS [12], G-ToPSS [13], MIC [14], and iBroker [15]is further improved, and the users' subscription intention is better understood; therefore, it has got much attention in recent years. However, all these methods are severely affected by the complexity of the concept hierarchy tree: S-ToPSS [11] expands one event to many events with all attributes' hypernym (ancestors), thus increasing matching times; MIC [14] expands one subscription to many subscriptions with all attributes' hyponym (descendants), thus increasing the memory usage of subscription index and cannot manage a large number of subscriptions; and G-ToPSS [13] and iBroker [15] traverse the hierarchy tree and check them one by one when matching, thus the matching efficiency is low.

To address these problems, this article proposes CTPS (concept tree-based pub/sub system) model, which makes the matching efficiency effectively improved. The main contributions of this article are as follows: (1) We propose a new event matching algorithm called CTPS, which is the first time to build a subscription index on concept hierarchy trees. By CTPS index structure, a subscription does not need to store multiple times, thus memory occupation is dramatically reduced. Also by CTPS index structure, an event does not need to match multiple times either, and unmatched subscriptions can be quickly filtered, thus event matching efficiency is greatly improved. (2) Instead of using complex RDF graphs, we use simple <*attribute*, *value*> and <*attribute*, *constraint*> pairs as event and subscription models, respectively, and together with the concept hierarchy tree, the description ability is powerful enough for semantic-based pub/sub systems.

2. Related Works

In semantic-based pub/sub systems, events and subscriptions are usually modeled by RDF graphs. OPS [12] takes event matching as a subgraph isomorphism problem, in which all nodes in the subscription graph are thought to be variables, so the algorithm complexity is very high. In G-ToPSS [13], only ?*x* in form of (?*x*, *op*, *v*) is regarded as a variable, where *op* may be a relational operation used for literal value comparison or may be a hypernym-hyponym relational operation used for concept/class comparison. For example, "laptop" < "computer" means the former is a hyponym or descendant concept of the latter. G-ToPSS builds a two-level hash index for all edges in the subscription graphs. The first level is the start and end vertices of an edge, and the second level is the edge label, together with a subscription list. G-ToPSS applies two processing stages for event matching: In the first stage, a completed graph is constructed for each edge (*s*, *p*, *o*) in the event graph, which means no matter the corresponding secondary index contains p or not, G-ToPSS will always search the following pairs in the index space:so, s*i, *io, and *i *j (*i and *j are variables), and the matched subscriptions are added to a processing set. In the second stage, for each subscription in the processing set, the variables are set with proper values to check whether the constraints are satisfied. If the checks are about hypernym-hyponym constraints, it will take a lot of time to traverse the concept hierarchy tree. MIC [14] (multidimensional index matching count) algorithm builds a three-level hash index for subscriptions, different from G-ToPSS, the edge labels are stored in the first level, and the vertex pairs are stored in the second level. If the second-level index contains variable vertices, the variable range will be divided into multiple intervals organized by a binary tree as the third level. For hypernym-hyponym constraints, MIC uses the hyponym expansion operation, that is, for any edge (*s*, *p*, *o*) in the subscription graph, new edges in the form of (*Ts*, *Tp*, *To*) will also be added to the index, where *Ts, Tp,* and *To* are descendants of *s*, *p*, and *o*, respectively. Therefore, there is no need to worry about the hypernym-hyponym problems when matching events; however, the index space increases dramatically, especially when the concept tree is high. In [16], subscriptions are translated into SPARQL queries, and the query condition is also extended as in MIC. In iBroker [15], both classes and attributes are stored in the first-level hash index, and the second-level index stores the next class or attribute to be matched, thus pulling the constraints of each subscription into a linked list. In essence, it merges identical vertices/edges in different subscription graphs into the same node, and the node value is then set to be true or false in the subsequent matching process. Finally, the list is traversed to get the matching result.

The above pub/sub systems are all centralized, that is, there is a broker who manages all subscriptions and is responsible for event message routing. There are also

decentralized peer-to-peer pub/sub systems, such as OpenPubSub [17], which proposes a hybrid event routing model that combines rendezvous routing and gossiping over a structured peer-to-peer network. The network is built based on a high-dimensional semantic vector space. This article only focuses on centralized pub/sub systems.

3. CTPS Model

To better understand the relation of concepts involved in events and subscriptions, concept trees must be defined at first, then events and subscriptions can use terms defined in these concept trees. Thereafter, the matching algorithm can use the concept tree to say whether an event's attribute/value is satisfied with a subscription constraint. The concept model is used to express concept relations, and the event/subscription model is used to express the event/subscription message structure sent to the broker.

3.1. Concept Model. The concept model reflects the hierarchical relation of classes and attributes, such as "computer" is the parent of "laptop PC," as shown in Figure 1. Similarly, attributes may also have hierarchical relations. For example, both "*cellphone number*" and "*email*" are subattributes of contact information. If there is only one node in the concept tree, it is called an independent concept, such as "MoneyValue" in Figure 2. In this way, each attribute could be an independent attribute or a hierarchical attribute, and also the attribute value could be an independent class or a hierarchical class. The xsd:string and xsd:decimal, which are frequently used in ontology, are regarded as independent classes. We specify that the attribute name is unique, and the attribute value type is determined by the attribute name, which can be obtained by domain(attribute) function, for example domain(target) = product, domain(contact information) = xsd:string. In addition, root(name) function can be used to obtain the corresponding root node's name, for example root (computer) = product. As for independent concepts, the root node's name is itself, for example root(MoneyValue) = MoneyValue.

3.2. Event Model. In semantic-based pub/sub systems, events are usually represented as RDF graphs. Figure 2 shows an example, indicating that John is selling a HP desktop PC with a 40G IBM hard disk for $450. This article uses a more concise event model, which is composed of <*attribute*, *value*> pairs, where *attribute* is the attribute name defined by the attribute hierarchy tree, and *value* is the attribute value, which can be a class name defined by the class hierarchy tree or a literal value. With a specification that attribute name is unique and it can determine its value type, for example price.value and hardDisk.size.value are both numeric, and *target* is of "*product*" type (see Figure 1), we can then use the following nine pairs to represent the event graph in Figure 2. Note that if the value type is an independent class, which means that we can exactly know its value type by the attribute name, then the pair for describing its value type can be omitted, such as < price.type, MoneyValue>. It can be seen that, compared with the original RDF event graph, our model is more concise. In addition, because event data are often drawn from relational databases in most cases, our model reduces the workload for there is no need to additionally convert events into RDF graphs.

<seller.name, John>
<seller.cellPhoneNumber, 123456789>
<price.value, 450>
<price.currency, units:$>
<target, Desktop PC>
<target.manufacture.name, IBM>
<target.hardDisk.size.value, 40>
<target.hardDisk.size.units, units:*G*>
<target.hardDisk.manufacture.name, IBM>

3.3. Subscription Model. In semantic-based pub/sub systems, subscriptions are often represented as graphs. As an example, Figure 3 shows a subscription graph about purchasing a computer with a price less than $400. In the graph, each vertex consists of 2 or 3 components; the first is the vertex ID, the second is the value type, and the third is an optional constraint. The subscription model in this article is also < attribute, constraint > pairs, where attribute is the attribute name and constraint is the constraint condition. There are two kinds of constraint: one is literal value constraint on basic data types such as numeric or string, and the other is type value constraint on class data types. For type value constraints, it is specified that the constraint conditions are only satisfied when the event value is the constraint value itself or is its subclass. For example, let "Desktop PC" and "computer" be event and constraint value, respectively, since "Desktop PC" is a subclass of "computer," the constraint is satisfied. The subscription graph in Figure 3 can be represented as the following four pairs, where interval [*lower*, *upper*] is used to denote numeric constraints. As we assume that the attribute name can be used to determine its value type, so if the attribute value type is unique, its type value constraint will be excluded.

<target, Computer>
<price.currency, units:$>
<price.value, [0,400]>

4. Index Structure

There may be several ways to express one thing, such as "Desktop PC" and its Chinese term "台式机," or "US" for short of "United States." To translate attributes and values in different names but with the same meaning, a synonym hash table in the form of <key, value> is designed, where "value" is the root words and "key" is all possible statements of them. In this way, the attribute names/values in events and subscriptions are easy to be replaced with their corresponding root words before processing. In addition, the unit types in events and subscriptions are also unified before processing,

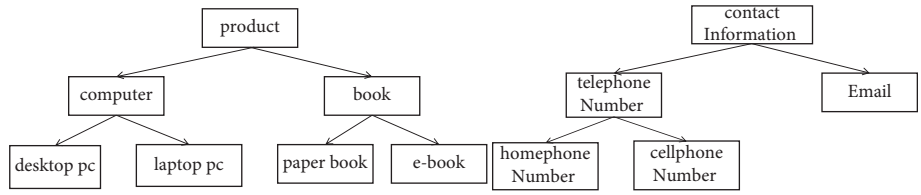

Figure 1: Example of hierarchy tree for classes (a) and attributes (b).

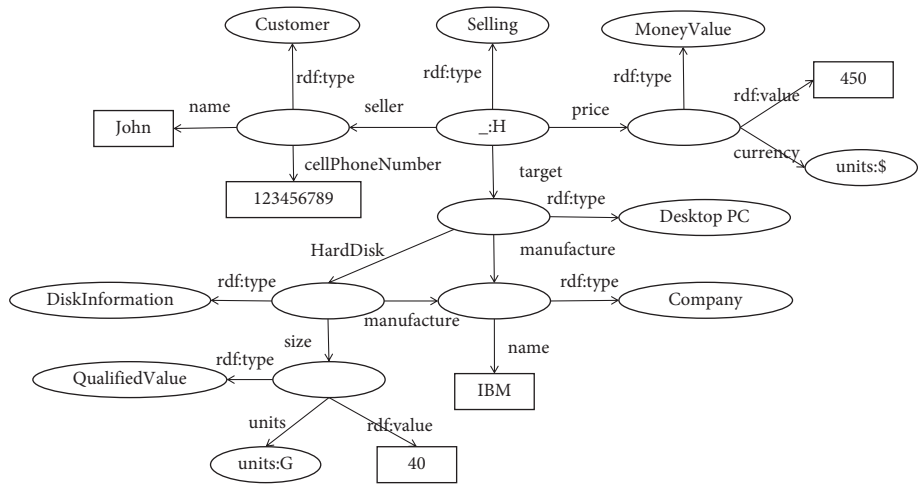

Figure 2: Example of an RDF event graph.

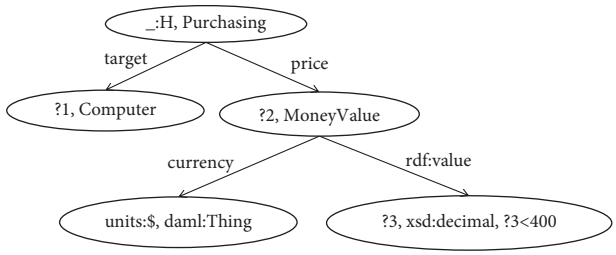

Figure 3: Example of a subscription graph.

such as "target.hardDisk.size.units" is transformed to *G*. By doing this, pairs about attribute unit can also be omitted, such as <price.currency, units:$> and <target.hardDisk.size.units, units:*G*> in the above event and <price.currency, units:$> in the above subscription. At the same time, a unique ID is assigned to each subscription.

4.1. Hierarchy Tree Index. An index is built on each concept hierarchy tree. As shown in Figure 4, the index consists of three parts: The left part is a hash table, where the *key* is attribute name/value and the *value* is linked to its corresponding node of the tree; the right part is a hierarchy tree, where each node consists of an attribute name/value and a list of subscription IDs, and each node is linked to its parent node; and the last part is the total subscription list of the hierarchy tree, as shown in the figure, *SID*: {1,4,5,10}. The subscription lists of nodes and the total subscription list *SID* are all stored in bit arrays, like *BitSet* in Java. Different from *integer list* which needs to be compared one by one, the bit array uses bit operations and greatly speeds up the matching speed. It should also be noted that, unlike the child node representation used in Figure 1, we use the parent node representation here, because only ancestor searching is needed in our algorithms.

Data Structure:

```
TreeNode{
    String name;
    TreeNode parentNode;
    BitSet sid;
}
TreeIndex{
    Map < String, TreeNode > hashTable;
    BitSet SID;
};
Map < String, TreeIndex > m1, m2;
```

The hierarchy index tree is built by Algorithm 1: *buildTreeIndex*. For each concept hierarchy tree, if it is an attribute tree, such as the right part of Figure 1, we build and initialize the corresponding index tree, and put it into $m1$ so we can quickly find it by the attribute name (2–6). If it is an attribute value tree, such as the left part of Figure 1, we build and initialize the corresponding index tree, and put it into $m2$ for subsequent search (7–11). The algorithm *initTreeIndex* recursively traverses the concept

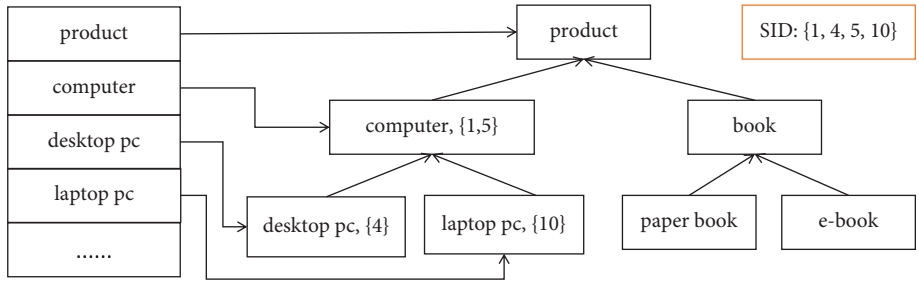

Figure 4: Example of a hierarchy tree index.

```
    buildTreeIndex( ){
(1)   for each hierarchy tree{
(2)     if it is an attribute tree pt{
(3)       TreeIndex ptree = new TreeIndex();
(4)       initTreeIndex(pt, null, ptree);
(5)       for each node name pname of pt, m1.put(pname, ptree);
(6)     }
(7)     if it is an attribute value tree vt{
(8)       TreeIndex vtree = new TreeIndex();
(9)       initTreeIndex(vt, null, vtree);
(10)      for each node name vname of vt, m2.put(vname, vtree);
(11)    }
(12)  }
    }
    initTreeIndex(ct, pNode, tree){
(1)   node = new TreeNode( );
(2)   node.name = ct.name;
(3)   node.parent = pNode;
(4)   tree.hashtable.put(node.name, node);
(5)   for each childNode of ct{
(6)     initTreeIndex(childNode, node, tree);
(7)   }
    }
```

Algorithm 1: Build and initialize a hierarchy index tree.

hierarchy tree, uses the parent node representation to initialize the index tree, and adds hash table entries as shown in the left part of Figure 4 so as to quickly access the corresponding index node and its ancestors by the attribute name.

Algorithm 2: insertSubscribe adds each constraint of subscription s into the index structure. First, we add the attribute name into the attribute index tree (2–3), and then add it into different index structures according to the type of the attribute value: string and numeric types are added to the literal constraint index (5–10), and other types are added into the attribute value index tree (11–13). Algorithm 3: *insertTreeConstraint* adds subscription ID into the subscription list of the node according to the attribute name/value and also adds it into the total subscription list in the index tree (2–4). As an example, for constraint <*target*, c*omputer*>, there is no need to add an index for *target* because it is an independent attribute (line 1), while "c*omputer*" is not, so we find its node in the *hashtable* (line 2), and then for the node subscription list and the total subscription list of the tree, we set its corresponding element to be 1(3–4).

4.2. Literal Constraint Index. We only consider two literal types: string and numeric. For string types, we only consider the equivalent constraint, and its index is a two-level hash table, where the first level is the attribute name, and the second level is the string value of the attribute with a subscription list, which is also stored in a bit array, as shown in Figure 5. A special item $*$ is added to the second-level index to match all possible string values of the attribute name. Algorithm 4 shows how to insert string constraints. Although there may be many nodes in an attribute hierarchy tree, they have the same value type, so we build only one string index by its root node.

Data structure is as follows:

Map < String, Map < String, BitSet >> $m3$;

```
    insertSubscribe(s){
(1)   for each pair < attribute, constraint> of s{
(2)     ptree = m1.get(attribute);
(3)     insertTreeConstraint(ptree, attribute, s.id);
(4)     switch(domain(attribute)){
(5)     case xsd:string:
(6)       insertStringConstraint(attribute, constraint, s.id);
(7)       break;
(8)     case xsd:decimal:
(9)       insertGEMConstraint(attribute, constraint, s.id);
(10)      break;
(11)    default:
(12)      vtree = m2.get(attribute);
(13)      insertTreeConstraint(vtree, constraint, s.id);
(14)    }
(15)  }
    }
```

Algorithm 2: Insert subscription.

```
    insertTreeConstraint(tree, pvName, id){
(1)   if tree is null, return;
(2)   node = tree.hashtable.get(pvName);
(3)   node.sid.set(id);
(4)   tree.SID.set(id);
    }
```

Algorithm 3: Insert attribute name/value class constraint.

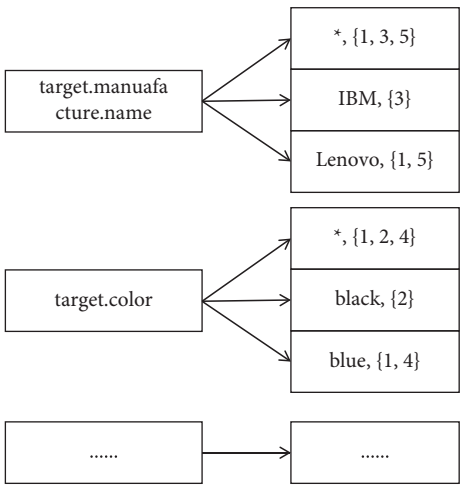

Figure 5: Example of string constraint indexes.

The numerical constraint index is also a hash table, in which the *key* is an attribute name, and the *value* is a GEM [7] index, as shown in Figure 6. Each cell of the GEM index has a subscription list, which is also stored in a bit array. The x-axis of the GEM index is the lower bound of interval constraints, and the y-axis is the upper bound of interval constraints. Since the lower bound must be less than or equal to the upper bound, constraints satisfying this condition all fall into the upper triangular region in the plane. The value range R_m is divided evenly into T boxes both on the x-axis and y-axis, and in this way, a constraint will only fall into one *cell*. Each *cell* is denoted as a coordinate (y, x), where y and x are calculated as follows:

$$y = \begin{cases} \dfrac{\text{upper}}{R_m/T}, & \text{upper} \neq R_m \\ T-1, & \text{otherwise} \end{cases} \quad x = \begin{cases} \dfrac{\text{lower}}{R_m/T}, & \text{lower} \neq R_m \\ T-1, & \text{otherwise} \end{cases}. \tag{1}$$

Assuming a constraint of [0, 20], $R_m = 100$, and $T = 5$, then the *cell* index is (1,0). Since the calculation of the *cell* index is independent of the number of constraints, the subscription insertion speed of GEM is a constant.

Data structure is as follows:

```
GEMCell{
  BitSet sid;
  Map < id, constraint > list;
}
GEMIndex{
  int T, Rm; GEMCell cell[T][T]; }
  Map < String, GEMIndex > m4;
```

```
    insertStringConstraint(attribute, value, id){
(1)   name = root(attribute);
(2)   map = m3.get(name);
(3)   if map is null{
(4)     map = new Map < String, BitSet>( );
(5)     m3.put(name, map);
(6)   }
(7)   sid = map.get(value);
(8)   if sid is null{
(9)     sid = new BitSet( );
(10)    map.put(value, sid);
(11)  }
(12)  sid.set(id);
(13)  aid = map.get(*);
(14)  if aid is null{
(15)    aid = new BitSet( );
(16)    map.put(*, aid);
(17)  }
(18)  aid.set(id);
    }
```

Algorithm 4: Insert string constraint.

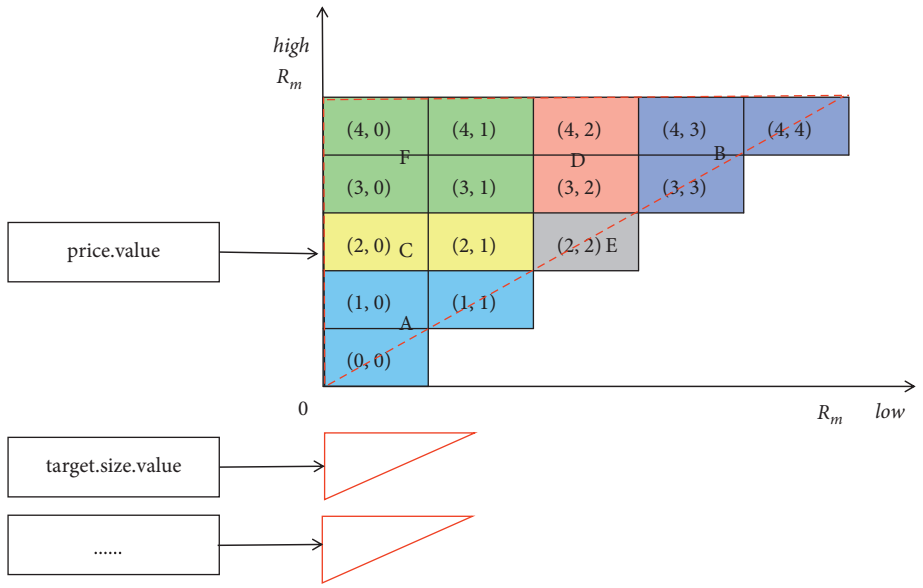

Figure 6: Example of a numerical constraint index.

Algorithm 5: insertGEMConstraint inserts a numerical constraint in the form of [*lower*, *upper*] into the *cell* of GEM. First, we need to find the corresponding GEM index by the root name of the attribute in *m*4 (1–2). If it is not found, a new hash table entry should be inserted (3–6). Then, we calculate *cell* index according to the formula and set the corresponding ID of the subscription list to be 1(7–8).

5. Event Matching Algorithm

In event matching, we delete unmatched subscriptions as early as possible. First, we initialize a bit array with a length of subscription number and set all its elements to be 1. Then we match each pair of the event in turn; if a constraint is not satisfied, then the location of its subscription ID in the bit array is set to be 0, which means it would not be matched anymore, and, finally, those subscriptions that are 1 in the bit array are matched. Since subscriptions are no longer matched after being set to 0, lots of invalid matching operations are avoided, so the matching efficiency is improved. As shown in Algorithm 6, for each pair <*attribute*, *value*> of an event, at first we find the index tree by the root name of the attribute (3–4), then we match the *attribute* if the index tree is not null (5–7), and then we match the *value* according

```
    insertGEMConstraint(attribute, constraint, id){
(1)   name = root(attribute);
(2)   gem = m4.get(name);
(3)   if gem is null{
(4)     gem = new GEMIndex( );
(5)     m4.put(name, gem);
(6)   }
(7)   compute index y, x by [lower, upper] of the constraint
(8)   gem.cell[y][x].set(id);
    }
```

Algorithm 5: Insert numeric constraint.

```
    match(e){
(1)   initial a BitSet sid with a length of subscription number, and set all its elements to be 1
(2)   for each pair <attribute, value> of e{
(3)     name = root(attribute);
(4)     ptree = m1.get(name);
(5)     if ptree is not null{
(6)       sid = treeMatch(ptree, attribute, sid);
(7)     }
(8)     switch(domain(attribute)){
(9)     case xsd:string:
(10)      sid = stringMatch(name, value, sid);
(11)      break;
(12)    case xsd:decimal:
(13)      sid = gemMatch(name, value, sid);
(14)      break;
(15)    default:
(16)      vtree = m2.get(name);
(17)      if vtree is not null{
(18)        sid = treeMatch(vtree, value, sid);
(19)      }
(20)    }
(21)  }
(22)  return sid;
    }
```

Algorithm 6: Event matching algorithm.

to the value type of the attribute (8–20), and, finally, the matching result is returned (line 22).

5.1. Hierarchy Tree Matching Algorithm. If the constraint's attribute name is the child node of the event's attribute name, they cannot match. Similarly, if the constraint's attribute value is the child node of the event's attribute value, they cannot match either. Algorithm 7: treeMatch performs event matching on attribute names/values. First, we find the corresponding node of the attribute name/value in the index tree and save the subscription list of the node and its ancestors in a temporary bit array tmpid (1–6). Then we perform bit operations and get the matching result sid (7–9). The resize function on line 7 sets tree.sid to the same length as sid by padding zeros. For example, tree.sid = {0,1,1}, sid = {0,1,1,1}, which means there are 4 subscriptions and only the first and second subscriptions (index starting from 0) are indexed in the tree, and after resize function, tree.sid = {0,1,1,0}. Supposing that tmpid = {0,1}, that is, only the first subscription matches, so the second subscription should be set to 0, and the result of the calculation is (~{0,1,1,0} | {0,1,0,0}) & {0,1,1,1} = {0,1,0,1}. If the length alignment is not performed, the system will take the ~ operation first, which may get a wrong result.

5.2. String Matching Algorithm. Algorithm 8: stringMatch is used for attribute values of string type. First, we get the hash table from $m3$ according to the attribute name, and if we get nothing, further operation will not be required, for there is no string constraint on it at all (lines 1–2). Then, we get subscriptions matching the attribute value (line 3) and subscriptions matching all possible values of the attribute

```
    treeMatch(tree, key, sid){
(1)     node = tree.hashtable.get(key);
(2)     tmpid = new BitSet( );
(3)     while node is not null{
(4)         tmpid = tmpid | node.sid;
(5)         node = node.parentNode;
(6)     }
(7)     resize(tree.sid, sid.length);
(8)     sid = (~tree.sid | tmpid) & sid;
(9)     return sid;
    }
```

Algorithm 7: Tree matching algorithm.

```
    stringMatch(key, value, sid){
(1)     map = m3.get(key);
(2)     if map is null, return sid;
(3)     tmpid = map.get(value);
(4)     aid = map.get(*);
(5)     resize(aid, sid.length);
(6)     sid = (~aid | tmpid) & sid;
(7)     return sid;
    }
```

Algorithm 8: String matching algorithm.

(line 4) and perform bit operations similar to algorithm 7 (lines 5–7).

5.3. Numeric Matching Algorithm. Algorithm 9: gemMatch is used for attribute values of numeric type. First, we get the GEM index according to the root name of the attribute (line 1). If it does not exist, which means that there is no numeric constraint on this attribute at all, we just return without doing anything (line 2). Otherwise, unmatched subscriptions should be removed. Using formula in 4.2, the cell index r of the *value* is calculated (2–4), and the index structure is divided into 5 parts by r, labeled as A to E as shown in Figure 6. As an example, let $R_m = 100$, $T = 5$, and $value = 45$, it is easy to know that r is 2.

(1) For cells in part A, $y < 2$, that is, $upper < 40$, it is impossible to match the event, so subscriptions in these cells will be removed (5–7);

(2) For cells in part B, $x > 2$, that is, $lower > 60$, it is impossible to match the event, so subscriptions in these cells will also be removed (8–10);

(3) For cells in part C, $x < 2$, that is, $lower < 40$, there is no need to check the lower bound, so only subscriptions with constraint whose upper bound is less than 45 will be removed (11–17);

(4) For cells in part D, $y > 2$, that is, $upper > 60$, there is no need to check the upper bound, so only subscriptions with constraint whose lower bound is greater than 45 will be removed (18–24);

(5) There is only one cell ($x = y = 2$) in part E, each constraint in it should be checked, and if it does not meet the condition, its corresponding subscription will be removed (25–30);

(6) For cells in part F, $y > 2$ and $x < 2$, that is, $upper > 60$ and $lower < 40$, the event is certainly matched, so there is no need to deal with it.

Different from the original GEM event matching algorithm, when processing parts A and B, the original algorithm needs to traverse all subscriptions in each *cell*, while it is changed to bit operations in this algorithm to speed up matching. And also, when dealing with parts C, D, and E, the original algorithm needs to check every subscription in each *cell*, regardless of whether it is invalid or not, while this algorithm first finds the intersection of *sid* and the subscription list of the current *cell* (lines 12, 19, 26), and then only valid subscriptions are checked. Thus, the number of comparisons is reduced and the matching efficiency is improved.

5.4. Algorithm Analysis. Index maintenance complexity analysis: For any subscription, *attribute* of every constraint should be inserted into the hierarchy tree index $m1$, and *value* of every constraint should be inserted into $m2$ (for class value), $m3$ (for string value), or $m4$ (for numeric value). For $m1$ and $m2$, each insertion includes 1 hash operation, 2 bit operations; for $m3$, each insertion includes 3 hash operations and 2 bit operations; for $m4$, each insertion includes

```
    gemMatch(key, value, sid){
(1)   gem = m4.get(key);
(2)   if gem is null, return sid;
(3)   if value = gem.Rm, r = gem.T - 1;
(4)   else r = value/(gem.Rm/gem.T);
(5)   for each pair (x,y) that 0 ≤ x<=y < r{
(6)     sid = sid & ~gem.cell[x][y].sid;
(7)   }
(8)   for each pair (x,y) that r + 1 ≤ x<=y < gem.T{
(9)     sid = sid & ~gem.cell[x][y].sid;
(10)  }
(11)  for (x = 0; x< r; x++){
(12)    tmpid = sid & gem.cell[x][r].sid;
(13)    for each bit 1 index of tmpid{
(14)      if gem.cell.[x][r].list.get(index).upper < value
(15)        sid.set(index, 0);
(16)    }
(17)  }
(18)  for(y = r + 1; y < gem.T; y++){
(19)    tmpid = sid & gem.cell[x][r].sid;
(20)    for each bit 1 index of tmpid{
(21)      if gem.cell.[r][y].list.get(index).lower > value
(22)        sid.set(index, 0);
(23)      }
(24)  }
(25)  tmpid = sid & gem.cell[r][r].sid;
(26)  for each bit 1 index of tmpid{
(27)    c = gem.cell[r][r].list.get(index);
(28)    if c.lower > value ||c.upper < value
(29)      sid.set(index, 0);
(30)  }
(31)  return sid;
    }
```

Algorithm 9: Numeric matching algorithm.

1 hash operation, 1 GEM index computation, and 1 bit operation. Operations are same for subscription deletion. Note that none of these operations are proportional to subscription numbers, and the maintenance complexity is nearly $O(1)$.

Space complexity analysis: It mainly includes the hierarchy tree index of attribute names $m1$, the hierarchy tree index of attribute values $m2$, the string index $m3$, and the numeric index $m4$. Suppose the number of root attributes is a. There are a_1 of them whose attribute name has a hierarchical class, a_2 of them whose attribute value has a hierarchical class, a_3 of them whose attribute value type is a string, and a_4 of them whose attribute value type is a numeric. There is $a_1 \le a$ and $a_2 + a_3 + a_4 \le a$. We suppose that there are n subscriptions in total. Each hierarchy tree has at most k nodes, and each string attribute takes at most b values. Then the space complexity is $a_1kn + a_2kn + a_3(b+1)n + a_4T^2n = ((a_1 + a_2)k + a_3(b+1) + a_4T^2)n = O(n)$, which is proportional to the number of subscriptions.

Time complexity analysis: Suppose the height of the hierarchy tree is h. Matching an attribute name of hierarchical class requires 2 hash operations, and a maximum of $h+1$ "|" operations, one "~"operation, and one "&" operation, that is, a total of 2 hash operations and $h+3$ bit operations are required to match the attribute name hierarchical class. The match of an attribute value hierarchical class is similar to this procedure. Matching a string value requires 3 hash operations and 3 bit operations. For numeric index matching, if the constraints are evenly distributed, each *cell* contains about $2n/T^2$ constraints: for each *cell* of sections A and B, 2 bit operations are required. Considering that A and B account for half of all *cells* on average, a total of $2\,T^2/4 = T^2/2$ bit operations are required. In the worst case, one comparison operation is required for the constraints in each *cell* of sections C and D, and two comparison operations are required for the constraints in each *cell* of section E, that is, a total of $(T-1+2)*2n/T^2 = 2(T+1)n/T^2 \approx 2n/T$ comparison operations. Supposing that the cost of one hash operation is p, the cost of one bit operation to is βn (the longest bit array is n, so the cost of bit operations is proportional to n), and the cost of one comparison operation is q. Hence, the time cost is $(a_1 + a_2)\ (2p + (h+3)\beta n) + a_3(3p + 3\beta n) + a_4(\beta n * T^2/2 + q * 2n/T) = (2a_1 + 2a_2 + 3a_3)p + [(a_1 + a_2)\ (h+3)\beta + 3a_3\beta + a_4T2\beta/2 + 2a_4q/T]n$, and the

Table 1: Parameter settings.

Parameter name	Parameter value	Memo
Attribute number with concept tree	6	
Attribute value number with concept tree	4	
Attribute value number with basic type	2	xsd:decimal
Height of concept tree	1~5	Tree with only root node has height 0
Width of concept tree	2	Each nonleaf node has 2 child nodes
Nodes (edges) of subscription/event graph	7(6)	7 nodes, 6 edges
Subscription number	500 ~ 10000	
Event number	1000	

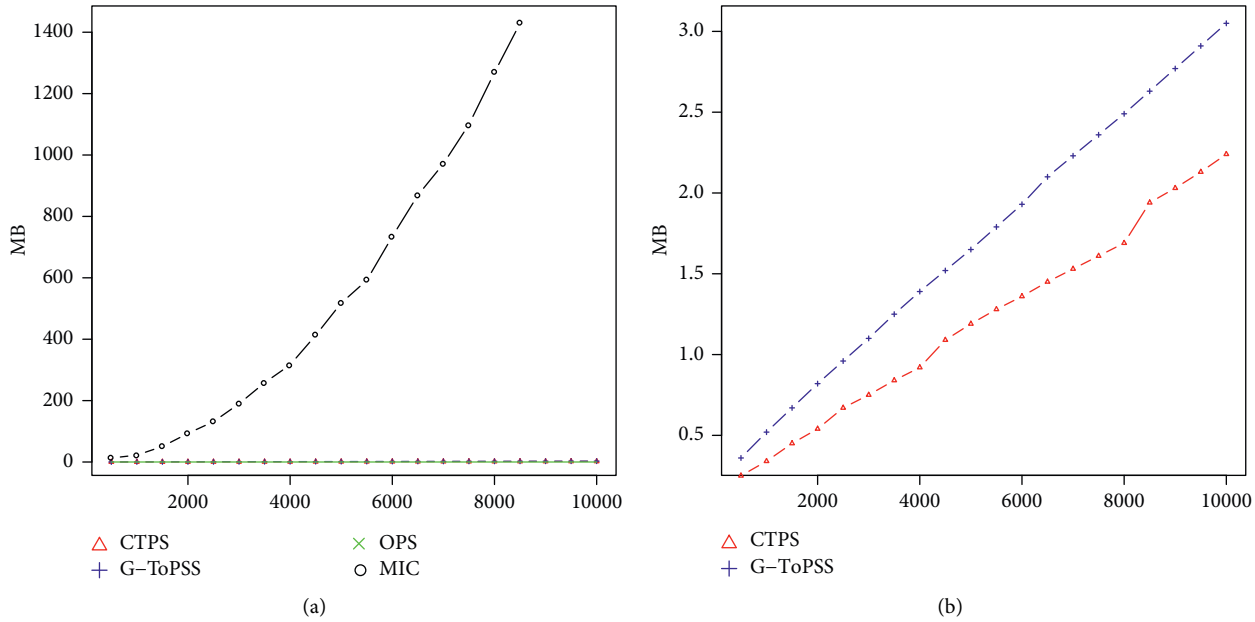

Figure 7: Comparison of memory usage: (a) subscribe (height = 1); (b) subscribe (height = 5).

complexity is $O(n)$. Due to the fast speed of the bit operation, the β value is small and the matching time will grow relatively slowly with the number of subscriptions.

6. Experiments

The CTPS prototype system is developed using JDK1.8. Experiments are run on a Windows10 laptop with an Intel i5-6200 2.40 GH CPU and 8G memory, and the experiment data are synthetic. Parameter settings are listed in Table 1.

In Figure 7(a), we compare the memory occupation of the four algorithms CTPS, G-ToPSS [13], OPS [12], and MIC [14] and the height of tree is 1 (one root node with two child nodes). Since MIC algorithm expands the original subscription, even when the tree height is only 1, it will expand $3*3*3 = 27$ times. It can also be seen that with the increase in subscription number, the memory space of the MIC increases sharply. When the subscription number is greater than 8500, the memory of MIC overflows, while the growth of CTPS and G-ToPSS is relatively slow. It should be pointed out that the OPS index only stores all possible vertices and edges, and the memory occupation has nothing to do with the subscription number and only grows rapidly with the height of the tree. When the tree height is 4, OPS occupies about 505 MB (Mega Byte), and it overflows when the tree height is 5. As the tree height increases, the number of subscriptions that MIC can handle also decreases rapidly. When the tree height is 4, only 1000 subscriptions can be handled in MIC, and when the tree height is 5, this number drops to less than 500. Therefore, in Figure 7(b), we only compare the memory occupation of CTPS and G-ToPSS when the tree height is 5. It can be seen that the memory occupation of the two increases linearly with the number of subscriptions. The growth rate of CTPS is smaller than that of G-ToPSS, and the total amount is relatively small. When the number of subscriptions is 10000, the memory is less than 3 MB.

In Figure 8, we compare the matching time of the four algorithms CTPS, G-ToPSS [13], OPS [12], and MIC [14]. The number of events is 1000 and the number of subscriptions varies from 500 to 10000. In Figures 8(a) and 8(b), we compare the matching time of the four algorithms when the tree height is 1. Since OPS matches subscriptions one by one, its matching time is so much longer than others, so graph 8(a) is drawn separately. It can be seen from Figure 8(b) that when the number of subscriptions is greater

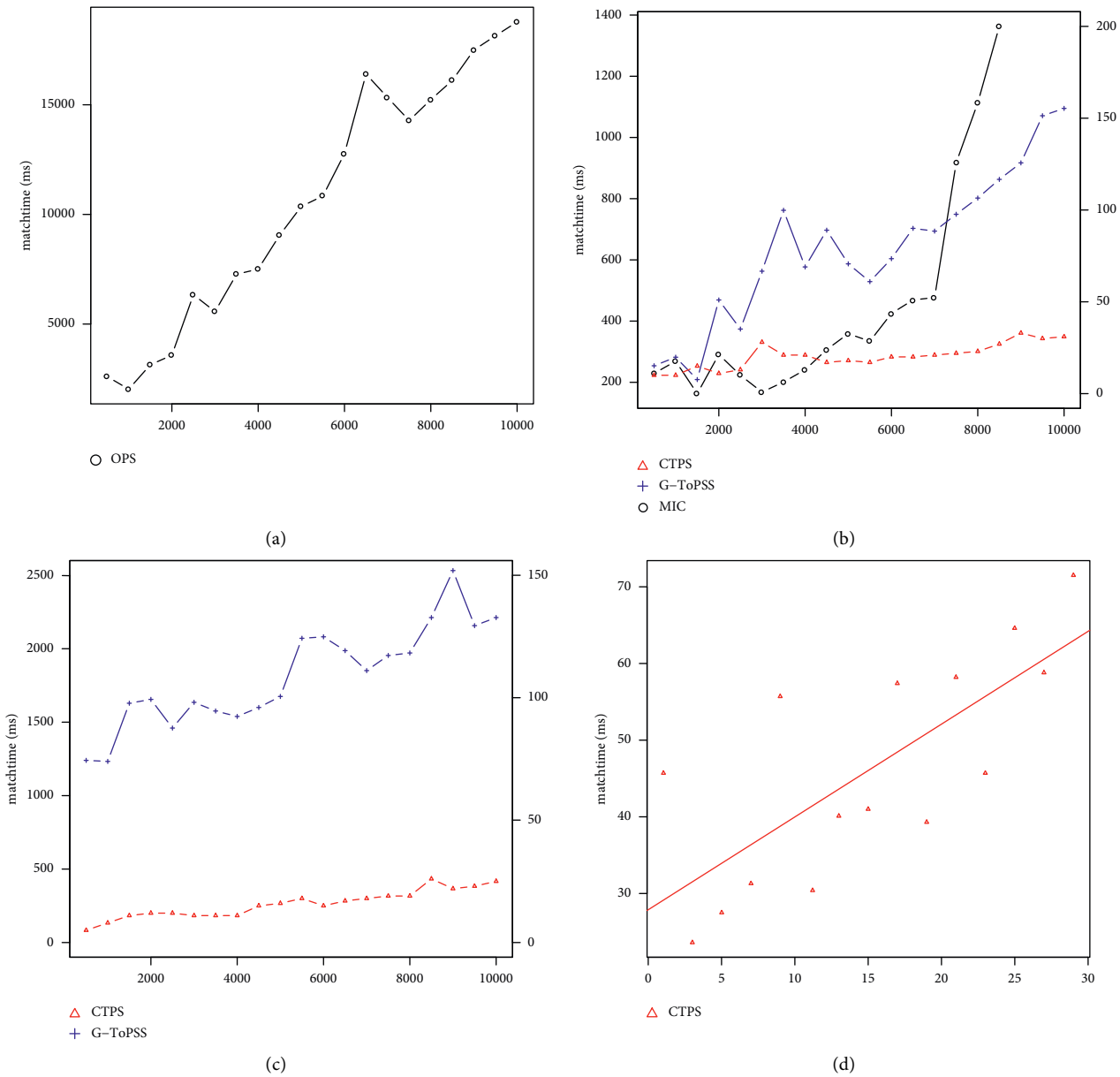

Figure 8: Comparison of matching time (number of events = 1000): (a) subscribe(height = 1); (b) subscribe(height = 1); (c) subscribe(height = 5); (d) matchrate(%) and subscribenum = 10000.

than 7000, the matching time of the MIC rises sharply, possibly due to the sharp increase in its memory usage. The matching time of G-ToPSS grows slowly with the number of subscriptions, while CTPS does not change much. It should be noted that since the matching time of CTPS is much shorter than other algorithms, Figure 8(b) sets a right vertical axis for CTPS alone. In Figure 8(c), we compare the matching time of CTPS and G-ToPSS algorithms when the tree height is 5, where G-ToPSS uses the blue vertical axis on the left and CTPS uses the red vertical axis on the right, and we can see that CTPS is about 70 times faster than G-ToPSS. Compared with Figure 8(b), the matching time of G-ToPSS increases as the tree height increases, while CTPS remains basically unchanged. In Figure 8(d), we compare the time used for CTPS under different matching rates. It can be seen that the matching time of the CTPS algorithm grows slowly with the increase of the matching rate, but the overall matching time is still very short. To sum up, CTPS is superior to other algorithms in terms of memory occupation and matching time, and the performance is not affected by the height of the concept tree.

7. Conclusion

Semantic-based pub/sub system has powerful description abilities, but its event matching algorithm is severely affected by the complexity of concept trees. We propose a new index structure called CTPS, which builds indexes on the concept hierarchy tree, though which unmatched subscriptions can be quickly filtered. By using bit arrays in the index, invalid

matches are avoided, and by fast bit operations, the matching time is further shortened. Experiments show that the algorithm in this article is significantly superior to existing algorithms in terms of memory occupation and matching time.

Data Availability

Data are available on request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors' Contributions

Zhiyuan zhang proposed CTPS algorithm and completed the manuscript; Yujie Wang performed the experiments; Xuehu Huang completed the coding of G-ToPSS algorithm; and Kai Leung Yung improved the editing quality of this article.

Acknowledgments

This work was supported by the States Key Laboratory of Air Traffic Management System and Technology of China (grant no. SKLATM201902).

References

[1] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[2] I. B. M. RedBook, *Internet application development with MQSeries and Java*, IBM, Armonk, NY.USA, 1997.

[3] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a distributed messaging system for log processing[C]," *6th International Workshop on Networking Meets Databases (NetDB)*, vol. 11, 2011.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.

[5] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network[C]," *Proceedings of ACM SIGCOMM*, pp. 163–174, ACM, Karlsruhe, Germany, 2003.

[6] S. Qian, J. Cao, Y. Zhu, and L Minglu, "REIN: a fast event matching approach for content-based publish/subscribe systems," in *Proceedings of the IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, April 2014.

[7] W. Fan, Y. Liu, and B. Tang, "GEM: an analytic geometrical approach to fast event matching for multi-dimensional content-based publish/subscribe services," in *Proceedings of the IEEE INFOCOM 2016*, April 2016.

[8] W. Fan, P. Xiong, F. Wu, and Y Liu, "GEM-tree: tree-based analytic geometrical multi-dimensional content-based event matching," *IEEE Access*, vol. 7, pp. 164089–164101, 2019.

[9] S. Qian, J. Cao, and Y. Zhu, "H Tree an efficient index structure for event matching in publish/subscribe systems," in *Proceedings of the IFIP Networking Conference, 2013*, April 2013.

[10] M. Sadoghi and J. H. A. . BE-Tree, "An index structure to efficiently match Boolean expressions over high-dimensional discrete space," *Acm Sigmod International Conference on Management of Data*, ACM, Athens, Greece, 2011.

[11] M. Petrovic, I. Burcea, H. A. . S Jacobsen, and S. S. ToP, "Semantic Toronto Publish/Subscribe System," *VLDB 2003*, pp. 1101–1104, Berlin, Germany, 2003.

[12] J. Wang, B. Jin, L. I. Jing, and S Dan Hua, "Data model and matching algorithm in an ontology-based publish/subscribe system," *Journal of Software*, vol. 16, no. 9, pp. 1625–1635, 2005.

[13] M Petrovic, L Haifeng, and J Hans Arno, "G-ToPSS: Fast filtering of graph-based metadata," *WWW 2005*, pp. 539–547, ACM, Chiba, Japan, 2005.

[14] X. Hu, "Matching algorithm for semantic-based publish/subscribe system[J]," *Journal of Zhejiang University*, vol. 43, no. 1, pp. 63–68, 2009.

[15] M. J. Park and C. W. Chung, "iBroker: an intelligent broker for ontology based publish/subscribe systems," *2009 IEEE 25th International Conference on Data Engineering*, in *Proceedings of the*, pp. 1255–1258, Shanghai, China, March 2009.

[16] H. Zhang, X. Zhang, K. Ding, and L Ming, "A fuzzy matching with reasoning publish/subscribe system based on ontology," in *Proceedings of the 2022 2nd international conference on consumer electronics and computer engineering (ICCECE)*, pp. 150–156, Guangzhou, China, January 2022.

[17] T. Zaarour, A. Bhattacharya, and E. Curry, "OpenPubSub: supporting large semantic content spaces in peer-to-peer publish/subscribe systems for the internet of multimedia things," *IEEE Internet of Things Journal*, vol. 1, 2022.