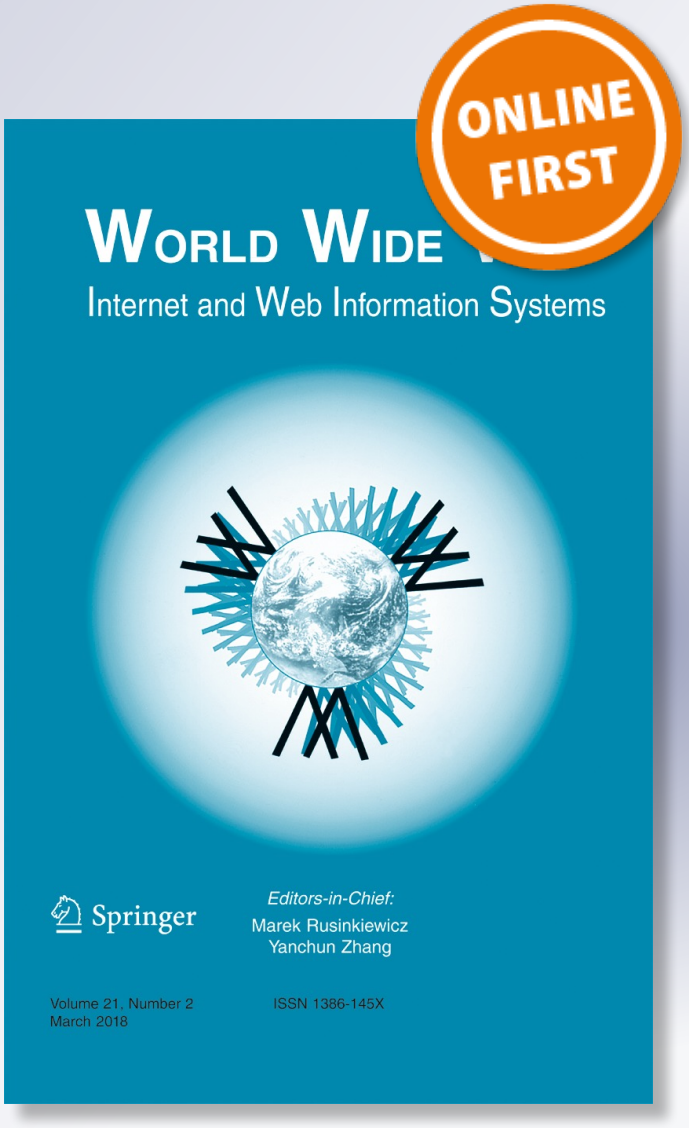
An index structure supporting rule activation in pervasive applications

Yi Qin, Xianping Tao, Yu Huang & Jian Lü

World Wide Web
Internet and Web Information Systems

ISSN 1386-145X

World Wide Web
DOI 10.1007/s11280-017-0517-2



Springer

World Wide Web
https://doi.org/10.1007/s11280-017-0517-2

An index structure supporting rule activation in pervasive applications

Yi Qin[1] · Xianping Tao[1] · Yu Huang[1] · Jian Lü[1]

Received: 22 July 2017 / Revised: 16 October 2017 / Accepted: 22 November 2017

Abstract Rule mechanism has been widely used in many areas, such as databases, artificial intelligent and pervasive computing. In a rule mechanism, rule activation decides which rules are activated, when the rules are activated, and which tuples can be generated through the activation. Rule activation determines the efficiency of rule mechanism. In this article, we define the semantic constraints, constant constraint and variable constraint, of the rule according to the semantics of Datalog rules. Based on the constraints, we propose an index structure, named Yield index, to support the rule activation effectively. Yield index consists of the data index and semantic index, and records the complete information of a rule, including the matching relationship among the tuples of different relations in rule body. The index integrates tuple insertion and rule activation to directly determine whether the matching tuples of new inserted tuple exist. Due to this character, we perform effective rule activation only, avoiding ineffective rule activation that cannot generate new tuples, so that the efficiency of rule activation is improved. The article describes the structure of Yield index, the construction and maintenance algorithms, and the rule activation algorithm based on Yield index. The experimental results show that Yield index has better performance and improves activation efficiency of one order of magnitude, comparing with other index structures. In addition, we also discuss the possible extensions of Yield index in other applications.

✉ Yi Qin
borakirchies@163.com

Xianping Tao
txp@nju.edu.cn

Yu Huang
yuhuang@nju.edu.cn

Jian Lü
lj@nju.edu.cn

[1] State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Xianling Road No.163, Nanjing, China

Published online: 19 February 2018

Keywords Index · Rule activation · Dummy tuples · Rule mechanism · Pervasive systems

1 Introduction

Rule mechanism is an important topic in database and artificial intelligent communities. It has got researchers' attention for a long time, and has achieved great success in both scientific study and real-life applications. Recently, rule mechanism has been applied in new scenarios of pervasive computing, for example, context-aware systems [4, 12, 29], wireless sensor networks [9] and coordination mechanisms [13]. Lim and Dey [28] report that 54% of context-aware decision-making models uses rule mechanism after investigating 114 context-aware applications. These rule mechanisms execute in a dynamic and interactive environment, which challenges the efficiency of the mechanisms. A pervasive system continuously gets all kinds of data as its surrounding environment changes, and its rule mechanism should be capable of performing data maintenance and rule activation in a high frequency.

The efficiency of a rule mechanism primarily depends on the efficiency of rule activation. *Rule activation* has been defined as determining which rules to activate, when to activate them, and which tuples can be generated through the activation [5]. When a new tuple t arrives, the task of rule activation is two-fold. Firstly, it finds the rule that can be activated by t. Secondly, it solves the rule through locating the matching tuples and generating the resultant tuples. Since the first step can be done through a naive mapping between rules and relations, we focus on how to solve a rule efficiently in this article.

We use *Datalog* as the rule representation, which is as follows.

$$R_0 : hp :- p_1, p_2, ..., p_n$$

where hp is the head predicate of rule R_0, and $p_1, p_2, ..., p_n$ are body predicates. A body predicate has some arguments, $(a_1, a_2, ..., a_m)$, in which a_i can be either a variable or a constant value. Each predicate p_i corresponds to a relation P_i, which can be classified as an *EDB* (Extensional DataBase) relation or an *IDB* (Intensional DataBase) relation.

A rule can be represented as a relation algebra expression, $P_1 \bowtie P_2 \bowtie ... \bowtie P_n$, that determines the matching tuples of a rule. When we try to solve rule R_0, we have to locate its matching tuples from body relations. We call this *tuple matching* in rule activation. For a non-recursive rule, rule activation performs tuple matching for one time, and then resultant tuples can be generated. For a recursive rule, one approach is that rule activation performs tuple matching iteratively, using the derived resultant tuples as the input in next iteration of tuple matching, until reaching a fixed point [38].

Due to the limited scale and comparative static nature of extensional database, past researchers focus on how to reduce the number of iterations [18, 40]. However, the boosting in data scale makes single iteration of tuple matching very time-consuming. Researchers find that it is difficult to establish large scale expert systems due to the complexity of facts and rules [5].

Index has been used to improve the efficiency of rule activation through quickly locating the matching tuples in each relation for mass data. Some rule engines [10, 19, 27] implement various index structures. They use conventional index structure, such as B tree, to build the separate indices for related relations. Some researchers also propose specific index structure to support the rule activation [5, 14, 25]. These approaches build the index on the join attribute of each relation that corresponds to body predicate, and use the value of join attribute as the index *KEY*.

All of the above approaches assume that database update is infrequent [5]. Once a new tuple t is inserted, the only way to know whether t has matching tuples is to run the activation algorithm. When a new tuple cannot find its matching tuples, the activation algorithm will return an empty result. Under this situation, the result of the activation algorithm, *the last index entry of search path*, is not stored in the index. We call this the *ineffective invoking* of rule activation, which contributes a considerable part to the poor-efficiency of existing rule activation approaches. Considering the pervasive environment that many rule systems execute in, the rule activation approach faces continuous updating (insertion of new tuples) to *EDB* relations, which further intensifies the problem of ineffective invoking.

Towards these challenges, we propose a new index structure, Yield index, to support effective rule activation, especially when new tuples are inserted in *EDB* relations at a high frequency. Yield index is established according to the rule semantics, which are defined and decomposed by *rule constraints*. We conclude two categories of rule constraints, *constant constraint* and *variable constraint*. The former expresses the constraints caused by the constants, and the latter expresses the constraints through the variable argument of other predicates. We take the following rule for example.

$$R_1 : h(a, b) : -s(3, x, a), t(x, b)$$

R_1 has two constraints. One is a constant constraint, which the first argument value of predicate s is 3. Another one is a variable constraint that the second argument of s has the same value as the first argument of predicate t. For different rules, the constant constraints are rule-independent and only describe the property of predicate arguments. However, the variable constraints are rule-dependent.

Yield index is a double-layer index structure. The lower level is called the data index, which is built on the variable constraint arguments of a rule. The upper level is called the semantic index, which conveys the information of the semantic constraints of the rule. The pointer of semantic index shifts on the data index, searching for its target value that matches the variable constraints. Data index enable us to locate specified tuples in a relation through *KEY*, which is similar to the conventional index structure. Semantic index represents the matching relationship between the tuples in different relations of the rule.

A Yield index covers all information of a rule. It integrates tuple insertion and rule activation to directly determine whether the matching tuples of an inserted tuple exist in Yield index. When a new tuple t arrives, we insert its *KEY* into the index, and can immediately know whether existing tuples can match tuple t, which guarantees that every time of rule activation is *effective*. Another important feature of Yield index is that we introduce *dummy tuple* in the process of tuple matching. A tuple t is a dummy tuple, which means that there is no matching tuple of t in related relations. In existing rule activation approaches, rule activation triggered by dummy tuple is simply discarded and results in ineffective invoking. However, in Yield index, the explicitly definition of dummy tuple enable us to give an incremental method of rule activation and index maintenance, which avoids ineffective invoking and greatly improves the efficiency of rule activation.

Our contributions in this article are as follows.

- We propose a two-layer index structure, Yield index, to support the rule activation effectively, which ensures that each call of activation algorithm is effective.
- We define dummy tuple as the result of unsuccessful tuple matching and give the incremental updating and activating methods of Yield index.

FATHER

Alice	--
Tom	Jim
Bill	Tom
Ellen	Jim
George	--
Cobe	George
Bob	--
Jim	--
Linda	--

MOTHER

Alice	Linda
George	Linda
Bill	Ellen
Tom	Alice
Ellen	Alice
Bob	--
Cobe	--
Jim	--
Linda	--

SEX

Alice	F
George	M
Bill	M
Tom	M
Ellen	F
Cobe	M
Bob	M
Jim	M
Linda	F
Mary	F

Figure 1 The tuples of *EDB* relations

– We show the extensibility of Yield index with presenting new types of semantic index to support novel *datalog*-based applications
– We conduct experiments to evaluate the performance of Yield index and other well-known index structures for rule activation. The results show that Yield index improves activation efficiency of one order of magnitude and has affordable overhead in both aspects of time and space for the construction and maintenance.

The remainder of this article is organized as follows. We give the motivation of Yield index in Section 2. In Section 3, we illustrate the basic ideas of the two-layer structure in Yield index. We then present our implementation of Yield index in Section 4. Some extensions and optimizations of the index structure are discussed in Section 5. Evaluation result is shown in Section 6. We review some related work in Section 7. Finally, Section 8 is the conclusion of this article.

2 Motivation

Since Yield index is based on the matching tuples among the relations of a rule body, we introduce the matching tuple and dummy tuple.

Definition 1 (*Matching Tuple*) For relations S and T in the body of rule R, if $\forall x, y, \exists z$, tuple $s(x, z) \in S$ and $t(z, y) \in T$, the tuple $t(z, y)$ in T is a *matching tuple* of the tuple $s(x, z)$ in S, and vice versa.

Definition 2 (*Dummy Tuple*) For relations S and T in R, if $\forall x, y, \exists z$, tuple $t(z, y) \in T$ and $s(x, z) \notin S$, the tuple $t(z, y)$ in T is a *dummy tuple* of relation S, and vice versa. The property of dummy tuple indicates that no tuple in S can match the tuple $t(z, y)$ of T, which is in the waiting state for matching with new tuples in S.

We give an example to demonstrate the motivation of Yield index. Let *father(x, y)*, *mother(x, y)*, *sex(x, MF)* be three *EDB* predicates, which show that y is x's father, y is x's mother, and x's gender is M or F respectively. The corresponding relations of three predicates are *FATHER*, *MOTHER* and *SEX* respectively. Figure 1 shows the tuples of three *EDB* relations, in which tuple *FATHER(Alice, —)* represents *Alice*'s father died.

We have the following rule.

$$R_2 : sonwithparent(z) :- father(z, x), mother(z, y), sex(z, M)$$

R_2 expresses that z is a son whose father and mother are alive, if x is z's father, y is z's mother, and z is a masculinity.

For rule R_2, the existing index methods build an index on join variable of each relation in rule body, such as *FATHER.z*, *MOTHER.z* and *SEX.z*. Some methods [14, 25] also build an index for matching tuples among relations of rule body. In a pervasive environment, new tuples of *EDB* relations may be arrived frequently, which results in frequently maintaining index and activating rule. However, the approaches of existing indices cannot directly know whether the new tuple t can trigger an effective rule activation or index adjustment. We observe the following two problems of these approaches.

(1) *Ineffective activation of new tuples*

For a new tuple t, existing approaches must invoke activation algorithm to know whether t has matching tuples in other relations of rule body. If activation algorithm for a new tuple t cannot find some matching tuples, this invoking of activation algorithm is actually ineffective. For example, when tuple *MOTHER(Mary, Jane)* is inserted, activation algorithm will search relation *FATHER* with *KEY* "*Mary*", but no matching tuple can be found. Before the arriving of a matching tuple, e.g., *FATHER(Mary, ***)*, the insertion of tuples like *MOTHER(Mary, ***)* will cause ineffective invoking, where "***" is a wildcard. This kind of tuples without matching tuples may account for a considerable proportion, which greatly harm the efficiency of rule activation. The more this situation appears, the lower the efficiency of index mechanism is.

(2) *Inefficient index maintenance of new tuples*

The index between the matching tuples in different relations of a rule body brings big overload in index maintenance. The frequent arriving of new tuples further increases the overload of existing approaches since each new tuple results in searching the index. When tuple *MOTHER(Mary, Jane)* is inserted and there is no matching tuple in relation *FATHER*, existing approaches will terminate this index search and discard the search result (last index entry). They do not store the unsuccessful information of searching index into the index. When tuple *FATHER(Mary, George)* is inserted in *FATHER*, existing approaches cannot use the information of anteriorly searching tuple *MOTHER(Mary, Jane)* to build the matching relationship between the tuples, *FATHER(Mary, George)* and *MOTHER(Mary, Jane)*. They have to search relation *MOTHER* with *KEY* "*Mary*" again, which increases the cost to build and maintain the index.

Our purpose is to make full use of the information of searching index every time, and store it in the most appropriate location in the index. And for any tuple t, we can know whether t has the matching tuples in other relations based on our index. These methods can improve the efficiency of building index, maintaining index and activating rule.

We use Figures 2 and 3 to further explain the intuition of Yield index using the example described in Figure 1. When constructing the index, we first build *data index* to index the tuples in relation *FATHER* and *MOTHER*, respectively (illustrated as the linear table in Figure 2). Data index can be built using traditional index structures, such as linear table, B tree, and B^+ tree. We use linear table here for the easy of explanation, and use B^+ tree in latter evaluation for efficiency considerations. Based on the data index, Yield index also builds

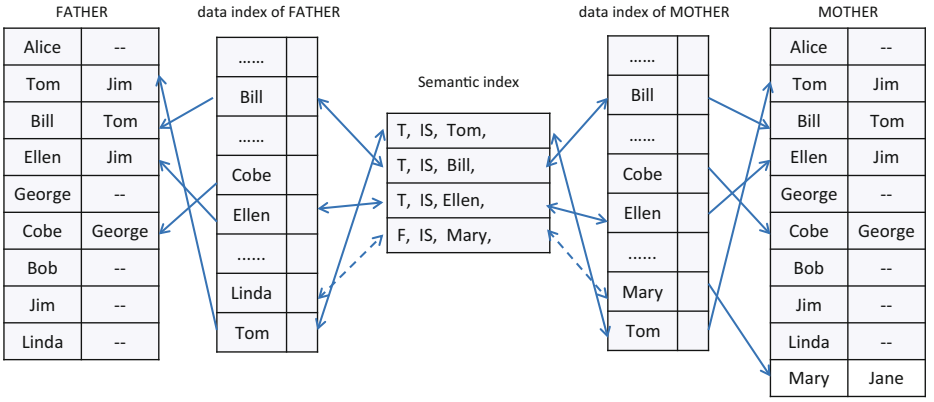


Figure 2 Yield Index on relation *MOTHER* in rule R_2 after inserting *MOTHER(Mary, Jane)*

semantic index between *FATHER* and *MOTHER*, as illustrated in the middle of Figure 2. For tuple *FATHER(Bill, Tom)*, a semantic index entry *(T, IS, Bill)* links itself and its matching tuple *MOTHER(Bill, Ellen)*'s data index. Here the first argument of the semantic index entry indicates whether the entry connects to a matching tuple (as *T*) or a dummy tuple (as *F*). The second argument describes the types of the entry and the third argument records the value of the constrained argument in matching tuples. Semantic index is built after the construction of data index and it connects the matching/dummy tuples between the relations under the data index. When performing rule activation, one can easily know if a tuple has a matching tuple and directly locate the matching tuples using semantic index.

The dummy tuple further improves the efficiency of activating rule when the index is updated due the insertion of a new tuple. When a tuple *MOTHER(Mary, Jane)* is inserted in relation *MOTHER*, as illustrated in Figure 2, there is no matching tuple for it in the relation of *FATHER*. As a result, *MOTHER(Mary, Jane)* is as a dummy tuple of *FATHER* and a semantic index entry *(F, IS, Mary)* is built. The dummy tuples are pointed by the dotted line. The entry is pointed by the last index entry of search path in *FATHER*'s data index based on *"Mary"* value, which is an index entry of nearest real position. Later, when tuple

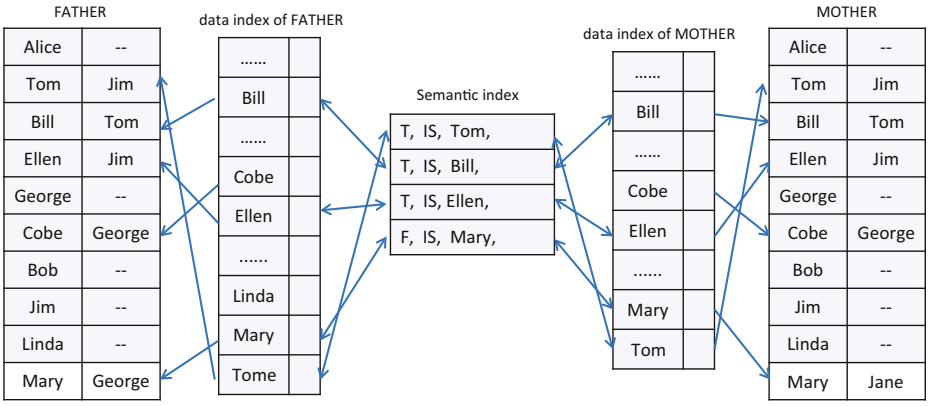


Figure 3 Yield Index on relation *FATHER* in rule R_2 after inserting *FATHER(Mary, George)*

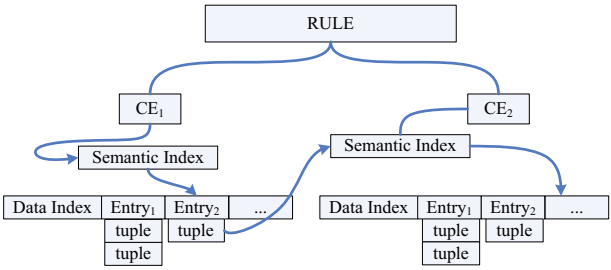


Figure 4 A Sketch of Yield Index Structure

FATHER(Mary, George) arrives and *"Mary"* is inserted in the data index of *FATHER*, as illustrated in Figure 3, *MOTHER(Mary, Jane)* is directly adjusted for a matching tuple (This is only a pointer adjustment). Finally semantic index entry between relations *MOTHER* and *FATHER* is directly used to locate matching tuple *MOTHER(Mary, Jane)* in *MOTHER*, without the need to search *MOTHER*'s index. Thus, Yield index can significantly improve the efficiency of activation algorithm when the relations are updated.

Figure 4 gives a schematic demonstration of Yield Index. The double-layer structure brings multiple benefits to Yield index. Firstly, data index functions as an indicator of the relation tuples, which enables the incremental shifting of the semantic index on data index. This results in a significant efficiency improvement in rule activation. Secondly, independent data index means that it would be shared by different rules, which reduces the overhead brought by index maintenance and the index file size. Last but not least, the structure has good extensibility since the semantic constraints are only related to the semantic index. Additional semantic information of extended *Datalog* can be indexed through new types of semantic index without modifying other part of the structure.

3 Yield index

In this section, we give an overall introduction on Yield index. We first define some important concepts concerning rule semantics and constraint elements. Then an overview of the index structure is given. After that, we focus on some design issues of Yield index, including how to derive constraint elements from a rule, how to handle multi-variable constraints and how to choose data index.

3.1 Constraint elements

Definition 3 (*Constraint Element* (***CE***)) A *constraint element*(***CE***) on a predicate p is a binary restriction that forms as $x = y$, where x is a variable of p, and y can either be a constant value or another variable. If y is a constant value, the constraint element is a *constant constraint*. If y is a variable of another predicate, the constraint element is a *variable constraint*. Apparently, the syntax of *Datalog* rule has only these two kinds of constraints [5].

A constant constraint can be represented as a built-in predicate. For example, the meaning of rule R_3 is the same as rule R_1:

$$R_3 : h(a, b) : -s(z, x, a), t(x, b), z = 3$$

The variable constraint represents the relationship between the variables of different predicates and restricts the tuples that can be matched with the tuples in different relations. We use *matching tuple* and *dummy tuple* to define whether the tuples satisfy a variable constraint.

Based on constraint elements, we can establish the matching relationship between all involved relations through either matching tuples or dummy tuples. Therefore, rule activation algorithm can use such matching relationships to accomplish tuple matching among the relations. Constant constraints work as a filter in Yield index. A tuple that does not satisfy constant constraints will not be inserted into the index. As a result, the activation algorithm will not consider this kind of tuples because the matching relationship between the tuples is neither originated from the tuple, nor pointed to the tuple.

The dummy tuple is one of the key features of Yield index. Through indexing dummy tuple along with matching tuple, successful matching relationship and failed matching relationship between the relations can be indexed and maintained. In Yield index, the information of a dummy tuple t is stored at the anticipated position where t's matching tuple should appear. Recording failed matching relationship enables Yield index to know whether a coming tuple triggers some successful matching tuples without performing an all-around rule activation. With dummy tuple, Yield index also achieves an incremental maintenance manner for the index, which greatly improves its maintenance efficiency.

We use R_1 to further explain the aforementioned concepts. Consider a tuple $t(x, b)$ in relation T. For the variable constraint on variable x between S and T, we need to search the value of x in relation S. If we find a matched index entry e that builds on relation S, which indicates that t is a matching tuple of some tuples in S, the information of t is then inserted into the entry e. Otherwise, t is a dummy tuple of S and the information of t is inserted into an *anticipate* position of x's value. Intuitively, the anticipate position is the index entry that the value of x should be inserted in the corresponding index, which will be detailed discussed in Section 3.5. In the latter situation, when a new tuple t_n of S is inserted, we decide whether t's information remains on the *anticipate* place, according to the x value of t and t_n. If not so, the position of t's information will be adjusted. We repeat the procedure along with the insertion of tuples, until t becomes a matching tuple of a new tuple that is inserted in S.

3.2 An overview of yield index

Yield Index consists of data index and semantic index. The data index is used to filter and quickly locate tuples, depending on the constant and variable constrain attributes. Different rules can share one data index if they have the constraints on the same attribute of same relation simultaneously. The semantic index conveys the matching relationship among the relations according to the corresponding variable constraints. We record the information about the matching tuples and dummy tuples as the entries of semantic index. Semantic index can locate the matching tuples of a specified key stored in the data index, and can be regarded as a special index entry of the data index.

Figures 5 and 6 illustrate the structure of Yield index of an example. The rule R_{11} is decomposed to three constraint elements which are depicted as $CE.1$, $CE.2$ and $CE.3$. Data index is built on the restricted arguments that appear in the constraint elements, for example $W.\$2$ and $S.\$1$. Semantic index is built according to the derived constraint elements. $CE.1$, $W.\$1 = 3$, is a constant constraint, which is used to filter the tuples in the data index built on $W.\$2$. The other two constraint elements are variable constraint, and their semantic indices

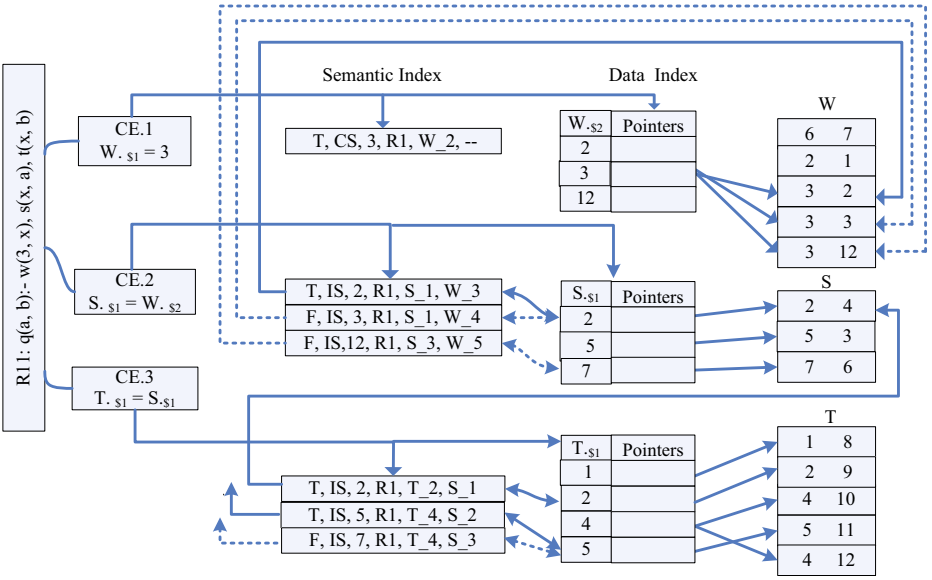


Figure 5 The Yield Index Structure of Figure 6

convey the matching relationship between the relations. Taking $CE.3$, $S._{\$1} = T_{\$1}$, as an example, the semantic index connects the matching tuples from relations S and T, as well as the dummy tuples between relations S and T.

The entry of data index has the following form.

(*KEY, MatchPointer, DummyPointer, KeyPointer*)

Here *KEY* is the value of key. *MatchPointer* points to the matching tuples which satisfy variable constraint. *DummyPointer* points to dummy tuples. *KeyPointer* points to the tuples with *KEY* value.

The entry of semantic index is the following form:

(*Fire, Type, KEY, RuleName, Root, Pointer*)

The *Fire* indicates whether the index is fired. Its value is T or F. T means to point to matching tuples. Otherwise, it points to dummy tuples. The *Type* suggests the type of the index. Different semantic index types express different kinds of semantic constraints. In Figure 5, ***CS*** denotes the semantic index of constant constraint, and ***IS*** denotes the semantic index of variable constraint. *KEY* stores the value of the attribute with *variable constraint* in another relation. *Rulename* indicates rule name. *Root* shows who brings the constraints

Figure 6 The Example of Rule and Instances

IDB
R11: q(a, b) :- **w**(3, x), s(x, a), t(x, b)

EDB	
W	(6,7) (2,1) (3,2) (3,3) (3,12)
T	(1,8) (2,9) (4,10) (5,11) (4,12)
S	(2,4) (5,3) (7,6)

and *Pointer* shows who is bound by the constraint. In the evaluation in Section 6, *Root* and *Pointer* are used to locate the corresponding entries of data index. We can get the tuples through the entries of data index.

Semantic index shifts their pointers on data index, searching for their target values. The shifting enables Yield index to be aware of the status of the rule activation process since data index functions as indicators of the current relation. Unlike other kinds of index structures, the semantic index is updated even when we cannot return a successful search result in the index. If a tuple t is a *matching tuple* of other tuples, Yield index quickly locates its matching tuples, and *yields* (means to generate) the resultant tuples. If no tuple matches t, it locates at the last position of searching path and will *yield* a semantic entry of dummy tuple for tuple t. Therefore, we call our index structure Yield index. The "yield" property of semantic index implies its incremental modification ability, which enable Yield index to maintain the index and perform rule activation simultaneously. This ensures that each rule only accesses its tuple no more than one time to complete the index maintenance and rule activation, which improves the efficiency of rule activation.

3.3 Constraint decomposition

In the construction of Yield index, we first decompose the rule constraints into *constraint elements* in order to recognize the semantic constraints of the rule. Once the ***CEs*** are derived, data index and semantic index are built accordingly. Here we present a constraint decomposition method to derive the *CEs* from a rule.

Definition 4 (*matching Set of Constraint Elements(SCE)*) Given a set of constraint elements $sc = \{ce_1, ce_2, ..., ce_n\}$ and a rule R, sc is a *SCE* of R *iff* an arbitrary set of tuples that satisfy the constraints of R also satisfies every constraint element in sc, and vice versa.

The constraint elements of a rule should be included in the *SCE* of the rule. We use the following method to construct the *SCE* for the rule R.

1) If a join relationship exists between relations P_i and P_j of a rule body, we create a variable constraint that $P_{i\cdot\$l} = P_{j\cdot\$m}$, where l and m are the positions of join variables of two relations respectively.
2) If a built-in predicate $P_{k\cdot\$n} = \mathrm{c}$ exists, we create a constant constraint directly, such as $P_{k\cdot\$n} = \mathrm{c}$.

Let rule R_4 be as follows.

$$R_4 : h(a, b, c) : -s(3, x, a), t(x, b), w(x, c)$$

We can get the *SCE* of rule R_4.

$$SCE = \{S_{\cdot\$2} = T_{\cdot\$1}, S_{\cdot\$2} = W_{\cdot\$1}, T_{\cdot\$1} = W_{\cdot\$1}, T_{\cdot\$1} = S_{\cdot\$2}, W_{\cdot\$1} = S_{\cdot\$2}, W_{\cdot\$1} = T_{\cdot\$1}, S_{\cdot\$1} = 3\}$$

The *SCE* can be used to construct data index and semantic index directly. But this will result in a big space-consuming since the *SCE* may contain some reduplicated constraint elements, like $S_{\cdot\$2} = T_{\cdot\$1}$ and $T_{\cdot\$1} = S_{\cdot\$2}$. The scale of an index is dependent on the size of *SCE*. A small *SCE* not only saves the space, but also saves time.

Algorithm 1 is used to simplify the *SCE* for the *Minimum Constraint Element set*, *MCE*, so that we can eliminate the redundant constraint elements and generate a compact *SCE*. For constant constraint elements, we simply move them from the input *SCE* to the output *MCE*

since they cannot be redundant (lines 4-6). For variable constraint elements, we not only move the current constraint elements into the output *MCE* (lines 8-9), but also transform all the depended constraint elements to reduce redundancy (lines 10-11). For example, in rule R_4, we have a subset of constraint elements, $\{S._{\$2} = T._{\$1}, S._{\$2} = W._{\$1}, T._{\$1} = W._{\$1}, T._{\$1} = S._{\$2}, W._{\$1} = S._{\$2}, W._{\$1} = T._{\$1}, S._{\$1} = 3\}$, in that all of the individual constraint element depends on each other. Thus, we can reduce the set of constraint elements to two constraint elements $\{T._{\$1} = S._{\$2}, W._{\$1} = S._{\$2}\}$. Applying Algorithm 1 on rule R_4, we can generate the *MCE* of rule R_4 as $MCE = \{S._{\$1} = 3, T._{\$1} = S._{\$2}, W._{\$1} = S._{\$2}\}$.

Algorithm 1 *SCE* Simplification

Input:
A Datalog rule $R : head :- p_1, p_2, ..., p_n$;
S: The *SCE* of R;
Output:
M: The *MCE* of R;

```
1:  M = NULL;
2:  while S is not empty
3:      get a constraint element CE_i(x = y) from S;
4:      if y is a constant
5:          S = S − {CE_i};
6:          M = M ∪ {CE_i};        //move CE_i from S to M
7:      else
8:          S = S − {CE_i};
9:          M = M ∪ {CE_i};        //move CE_i from S to M
10:         S = S − {CE_j(w = v)|w ≡ x ∨ v ≡ x};
                    //remove all CEs that are depended on with CE_i
11:         M = M ∪ {CE(w = x)|w ≡ x};
                    //reform and add the depended constraint elements into M
12: return M;
```

3.4 Multi-variable constraint

A predicate may involve in one constraint element or multiple constraint elements. If all predicates of a constraint element are not bound by other constraint elements, we call the constraint element a *single-variable constraint*. *Multi-Variable constraint* means that more than one arguments of a predicate are bound by the constraints of the rule. For example,

$$R_5 : t(x, b) :- s(x, 5, 1), w(x, b)$$

Relation $S(x, 5, 1)$ in rule R_5 is bound by multi-variable constraints, one variable constraint and two constant constraints. A tuple of relation S should match two constant constraints simultaneously, $S._{\$2} = 5$ and $S._{\$3} = 1$, as well as a variable constraint, $S._{\$1} = W._{\$1}$. In some works, rule-specific index is used to solve this problem [5, 30]. A composite attribute is used, $S._{\$2} + S._{\$3}$ in rule R_5 for example. As a result, every rule in the system has an independent index structure, which increases the overhead and maintenance cost of the index structure.

We build one data index for each variable with *variable constraint* in Yield index. For example rule R_5, we build one data index on attribute $S_{\$1}$. Two constant constraints, $S_{\$2}$=5

and $S_{\S 3}$=1, are used to filter the tuples during building the data index. But this is suitable only when relation S appears in one rule. If relation S appears in two or more than two rules, the constant constrain on S can not filter the tuples in building the data index. This makes the data index independent of rules, which enables different rules to share the same data index when the rules put constraints on the same attribute of a relation. We use the semantic index to build the matching tuples between S and W in rule R_5. If a predicate p is bound by multi-variable constraints, the matching tuples in relation P are determined based on these constraints simultaneously. Each constraint is used to generate a set of candidate tuples in relation P. The set of matching tuples in relation P is the intersection of the sets of derived candidate tuples. As a result, all founded tuples in the relation P satisfy all constraint elements in the predicate p.

3.5 Position of dummy tuple

For dummy tuples, Yield index needs to determine their semantic index pointers based on anticipating the position of the *KEY* value in the data index. The choice of data index puts influence on the correctness and the performance of Yield index. The data index of Yield index should satisfy the following property.

Definition 5 (*Search Path*) Given an index structure I and a relation S, for a tuple t in S, let $P^t_{I\_S}$ be the search path of t in S based on I. If S' be an updating version of relation S, the new search path of t in S' based on I is $P^t_{I\_S'}$.

Definition 6 (*Search Invariant Property*) Given an index structure I and a relation S, I owns search invariant property *iff* for arbitrary tuple t and S', $P^t_{I\_S}$ and $P^t_{I\_S'}$ satisfy:

1) Both $P^t_{I\_S}$ and $P^t_{I\_S'}$ are deterministic and pass each node no more than once.
2) If the end index entry e of $P^t_{I\_S}$ exists in the data index of S', then e exists in $P^t_{I\_S'}$.

The searching invariant property enables us to maintain the semantic index incrementally once a new tuple is inserted into a relation. It guarantees that the past shifting of the pointers of semantic index is valid during the rule activation. This can dramatically decrease the cost in index maintenance and rule activation. However, searching invariant property does not solve everything in the maintenance of semantic index. When a new index entry is inserted, we have to shift the pointer toward an appropriate index entry on the data index. Figure 7 shows an update process of semantic index. Due to the searching invariant property, the pointer which aims at value **6** does not start from the index head at every time when the data index is updated. It can remain the correction of data index incrementally.

When a new tuple t is inserted, we can update the semantic index incrementally without searching for the *KEY* value of the matching tuples. Specifically, when the insertion of t creates a new data index entry e_n after an old data index entry e_o, we update the semantic index under e_o. For all semantic index entries under the dummy pointer of e_o, we first check whether any dummy tuple t_d can match t. If so, we need to move the index entry of t_d from the *DummyPointer* of e_o to the *MatchPointer* of e_n, a new data index entry. After that, we check whether the remaining semantic index entries under the *DummyPointer* of e_o need to be moved. If the new data index entry can extend the search path of dummy *KEY* of a semantic index entry, the semantic index entry is moved to the *DummyPointer* of e_n. Otherwise, the semantic index entry remains in the position under the *DummyPointer* of e_o.

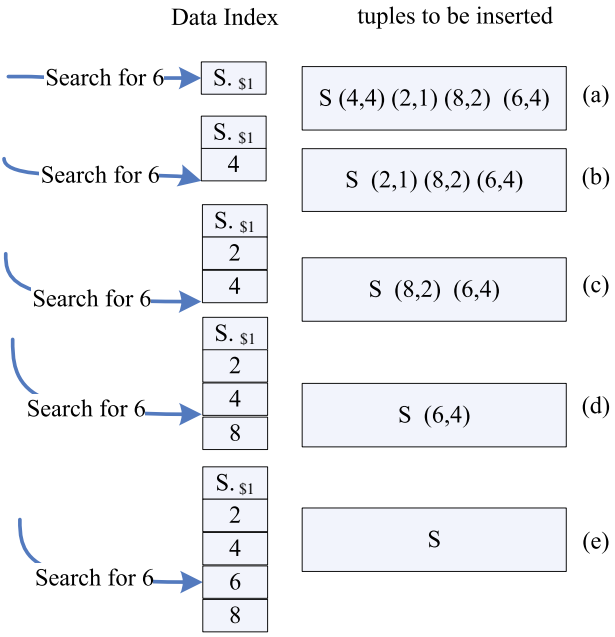


Figure 7 Illustration of searching invariant property and incremental shifting

Many existing storage structures have the searching invariant property, including binary sort tree and B-tree. The index approaches with searching invariant property can be used as the data index structure of Yield index.

4 Implementation of yield index

In this section, we give the implementation of Yield Index, including how to construct Yield index on existing relations, how to update Yield index when a new tuple is inserted, and how to perform rule activation using Yield index.

4.1 Index construction

4.1.1 Data index construction

Data index is build on every relation bound by constraint elements in the *SCE* of a rule. Some strategies of establishing data index are as follows.

(1) Concerning constraint priority. If the relation with *constant constraint* only appears in one rule, the attribute with *constant constraint* is first chosen to filter tuples. The tuples that do not satisfy the constant constraint are not inserted into the data index.
(2) Concerning constraint type. For the relations with single-variable constraint, we build a data index for each relation on the attribute bound by variable constraints; for the relations of multi-variate constraint, we build a data index for each variable bound by variable constraint.

Algorithm 2 shows the process to build the data index of Yield index. The algorithm takes a Datalog rule R and its minimal constraint element set M as its input, and build Yield index on the tuples in the related relations. The algorithm mainly consists of two steps. In the first step, we build Yield index's data index for the predicates that are constrained in R, and record the tuples that semantic index needs to be built on. In the second step, we build semantic index for the recorded tuples (using Algorithm 3 in Section 4.1.2). Specifically, for each of R's body predicates P_i, we go through the constraint elements in M to decide if index needs to be built on P_i's tuples (lines 3-5). For constant constraints, only the tuples that satisfy the constant constraint are kept for index construction (lines 6-9). For variable constraints, we keep all the involved tuples and build corresponding data index for P_i if the data index has not been built yet (lines 10-14). Finally, we use Algorithm 3 to build the semantic index for each tuple we collected in the set of $ToIndex$ (lines 15-16).

Algorithm 2 Index Construction

Input:
A Datalog rule $R : head :- p_1, p_2, ..., p_n$;
M : The MCE of R;

1: $Current = \emptyset$; //current processing tuples
2: $ToIndex = \emptyset$; // the tuples need to be build semantic index on
3: **for each** $P_i, 1 \leq i \leq n$
4: $Current = \{t_k \mid t_k \in P_i\}$; //the tuples of P_i needed to be indexed
5: **for each** $ce \in M$
6: **if** ce is a constant constraint that "$P_i.x = c$"
//filter the tuples using constant constraints
7: **for each** tuple $t_k \in Current$
8: **if** $t_k.x \neq c$
9: $Current = Current - \{t_k\}$;
10: **for each** $ce \in M$
11: **if** ce is a variable constraint that "$P_i.x = P_j.y$" or "$P_j.y = P_i.x$"
//build data index for predicate arguments that are not indexed yet
12: **if** $P_i.x$ is not indexed
13: build a data index entry on $P_i.x$ of t_k that $t_k \in current$;
14: $ToIndex = ToIndex \cup Current$;
//record the tuples needed to be indexed in $ToIndex$
15: **for each** $t \in ToIndex$ that t belongs to relation P
16: $BuildSemanticIndex(t, P, M)$;

4.1.2 Semantic index construction

Based on the data index, we build the semantic index of Yield index to get the matching relationship among the body relations according to the *SCE* of the rule. The strategies of building semantic index are as follows.

(1) For each element in *SCE*, we will build the corresponding semantic index, which directly defines the tuple matching relationship between two relations involved by the *SCE* element.

(2) The semantic index of each variable constraint, such as *S.x=W.x*, in *SCE* is successively built in the form of increasing cardinalities of the relations. Let the cardinalities of relations *S* and *W* be *N(S)* and *N(W)*. We choose the relation *S* as main relation to build the semantic index between *S* and *W* if *N(S) > N(W)*.

Algorithm 3 $BuildSemanticIndex(t, P, M)$

Input:

A tuple $t(x, y)$, *R* : rule name, *TD* : the address of t;
The relation P that t belongs to;
M: a set of constraint elements;

```
1: for each ce ∈ M
2:   if ce is a variable constraint that "P.x = P_i.y" or "P_i.y = P.x"
                    //search for the place that the entry of sematic index should be built on
3:     search for value t.x in the data index on P_i.y, end on an index entry e;
4:     if search successes              // build a matching index entry for matching tuples
5:       create a matching index entry JI(T, SI, y, R, TD, e);
6:       insert JI into e.MatchPointer;
7:     else              // no matching tuple exists and build a dummy index entry
8:       create a dummy index entry JD(F, SI, y, R, TD, e);
9:       insert JD into e.DummyPointer;
```

Algorithm 3 describes how to build the semantic index entry for a given tuple t using the variable constraint in a *SCE* of the rule. For a variable constraint $P.x = P_i.y$, semantic index entries are built for the tuples in relation P which satisfy the constraint. The value of attribute y is used as the *KEY* of semantic index entry and the address of the tuple is used as the tuple address *TD* of semantic index entry. Finally, we insert the semantic index entry under the *MatchPointer* or *DummyPointer* of an appropriate data index entry in P_i.

Data index is also used to determine the position to insert the semantic index entry. We search for the index entry with *KEY*, which is denoted as $t.x$ (line 3). If such an index entry e is found, a semantic index entry *(T, SI, KEY, R, TD, e)* is inserted under the *MatchPointer* of e (lines 4-6). Otherwise, the search mechanism locates at the last index entry e on the search path of $t.x$ in the data index. We insert the semantic index entry *(F, SI, KEY, R, TD, e)* under the *DummyPointer* of e, indicating the dummy tuple that it points to (lines 7-9).

4.2 Index maintenance

4.2.1 Tuple insertion

When a new tuple t is inserted, Yield index not only updates the data index and semantic index for tuple t, but also adjusts existing semantic index entries which are affected by t. Algorithm 4 describes how to update Yield index when a tuple t is inserted. We first check whether the new tuple t satisfies all constant constraints on the corresponding relation. We simply discard tuple t and terminate the algorithm if the check fails (lines 1-5). If the check successes, then the *KEY* of t is inserted into the data index. If a corresponding data index entry exists for t, we insert the information of t under the entry (lines 6-11). If no such entry exists, we insert t as a new data index entry (lines 12-14). After that, we update the semantic index of the involved data index entry, including the index entries under the *MatchPointer* and *DummyPointer* (lines 15-21). Finally, the semantic index entry of t is built using Algorithm 3 (line 22).

Algorithm 4 Insertion of New Tuple

Input:
A Datalog rule $R : head :- p_1, p_2, ..., p_n$;
M : The *SCE* of R;
A tuple t of relation P_i;

```
1:  for each ce ∈ M
2:    if ce is a constant constraint "P_i.attr_i = c"
            //check constant constraints on t and filter the arguments that are only constrained
            //by constant constraints
3:      U = U ∪ {attr_i};
4:      if t.attr_i ≠ c
5:        return;
6:  for each attr_i of P_i ∉ U
7:    if no data index is built on P_i.attr_i
8:      continue;            //no data index indicates that the current argument is irrelevant
9:    search value t.attr_i in the data index on P_i.attr_i,
            ends on an index entry e;
10:   if search successes      // directly insert t under data index and semantic index
11:     insert t into KeyPointer of e;
12:   else          //build new data index entry and adjust existing dummy index entries
13:     create a new index entry ne for value t.attr_i;
14:     insert t into KeyPoniter of ne;
15:     for each JD ∈ e.DummyPointer
16:       if JD.KEY > t.attr_i
                  // the dummy index entries should be moved to the new index entry
17:         move JD from e.DummyPointer to ne.DummyPointer;
18:       if JD.KEY == t.attr_i
                  // tuple t is a matching tuple for the dummy index entries
19:         create an index entry JI(T, SI, KEY, JD.rule, JD.tuple, ne);
20:         remove JD from e.DummyPointer;
21:         insert JI into ne.MatchPointer;
22: BuildSemanticIndex(t, P_i, M);
```

Updating the semantic index entries along with the insertion of new tuple enables Yield index to perform index maintenance in an incremental manner. When the insertion of a new tuple t results in a new data index entry ne in the data index (lines 13-14), we have to update the semantic index entries connected to the precursor index entry e of ne. Assuming the *KEY* value of ne is K_1 ($t.x$ in the algorithm), and the *Dummy KEY* value of a semantic index entry *JD* is K_2 (*JD.KEY* in the algorithm), the update strategies are as follows.

1. If $K_1 < K_2$, which means that the semantic index entry *JD* cannot be matched by t and ne is on the searching path of K_2 in the data index, then *JD* is moved under the *DummyPointer* of ne (lines 16-17).
2. If $K_1 = K_2$, then the semantic index entry *JD* is moved under the *MatchPointer* of ne (lines 18-21). Meanwhile, we can perform rule activation on the tuple t.

3. If $K_1 > K_2$, which means that the semantic index entry *JD* cannot be matched by t and ne is not on the searching path of K_2 in the data index, we will not update *JD*.

The successful establishment of semantic index entry of new tuple t means that the matching tuples of tuple t already exist according to Yield index. At this moment, activation algorithm should be called, which ensures that ineffective activation is not called for the new tuple t. This process integrates the insertion and activation for a new tuple t, which improves the efficiency of rule activation.

4.2.2 Tuple deletion

When a new tuple t is deleted, Yield index will also update the data index and semantic index affected by tuple t.

Algorithm 5 Deletion of a Tuple

Input:

A Datalog rule $R : head :- p_1, p_2, ..., p_n$;
A tuple *t(x, y)* of relation P_i which will be deleted;
M : The *MCE* of R;

1: **for each** $ce \in M$ // check M to collect constrained arguments to a set U
2: **if** ce is a variable constraint "$P_i.attr_1 = P_j.attr_1$"
3: $U = U \cup \{P_j.attr_1\}$;
4: search value $P_i.x$ in the data index on P_i and end on an index entry e;
5: locate the *KeyPointer* and *MatchPointer* in e, which $P_i.attr_2$ = t.y;
6: $e.KeyPointer$ = NULL; // remove t from data index
7: **for each** $P_j.attr \in U$ // check if t is connected to any semantic index entries
8: **if** $e.MatchPointer \neq$ NULL
9: search value $P_j.attr$ in the data index on P_j and end on an index entry ne;
10: locate *MatchPointer* in *ne* ,which is equal to *e.MatchPointer*;
//find the matching tuples of t
11: *ne.MatchPointer* = NULL;
//remove the semantic index entries connecting t and its matching tuples
12: Delete $P_j.x$ in the data index of P_j;
13: Delete $P_i.x$ in the data index of P_i;
14: Delete the semantic index entry pointed by *e.MatchPointer*;

Algorithm 4 describes how to update Yield index when a tuple t is deleted. First, we use the *MCE* to find the constrained attributes related to relation P_i and store the result in set U (lines 1-3). Then we locate t's index entry e in data index through *key* (lines 4-5). Based on e, we use U to check if there is semantic index that connects to e. We set the corresponding pointers of the found semantic index to *NULL* and delete the key of t from data index of P_j (lines 7-12). Finally, we delete the data index entry of t and the related semantic index entries (lines 13-14).

4.3 Rule activation using yield index

Rule activation can be triggered by either a query on existing tuples, or the insertion of a new tuple t. In the first situation, Yield index is used to find all combinations of matching tuples, and then generate the resultant tuples. In the second situation, Yield index is used

to locate all matching tuples of the new tuple t and generate the resultant tuples. Unlike existing rule activation approaches, Yield index itself supports directly locating of matching tuples from different relations. As a result, the set of matching tuples is derived through the semantic index.

Algorithm 6 Rule Activation for a Tuple, $Act(t)$

Input:

A Datalog rule $R : head :- p_1, p_2, ..., p_n$;
M : The *SCE* of R ;
t : A tuple of relation P_i ;

Output:

MT : A set of matching tuples based on t and Rule R;

1: $MT = \{t\}$;
2: $last = P_i$; // current processing predicate
3: $Used = \{ce_i | ce_i$ is related to predicate in $last\}$;
//processed constraint elements
4: S = $\{t.x\}$;
// the target value of t's matching tuples
5: **while** $Used \neq M$
6: $TS = S$'s matching tuple set in P_k with $last.x = P_k.y$;
// locate matching tuples of relation $last$
7: $MT = MT \bowtie TS$;
// use join operation to derive the set of matching tuples
8: ce_i = search next constraint element;
9: $Used = Used \cup \{ce_i\}$;
10: $S = \pi_{last.x}(MT)$;
// move to the next constraint elements
11: **return** MT;

Algorithm 6 illustrates how to perform rule activation for a new tuple t, which joins successively with the rest relations of rule body based on each constraint element and Yield index. We start from adding t to the matching tuples set MT (line 1). Then we use set $Used$ to store all constraint elements, ce_i, related to *last.x* (line 3). Let each ce_i be a variable constraint $last.x = P_k.y$, which *last.x* is an attribute in MT and $P_k.y$ is an attribute in relation P_k. Set TS stores the tuples which satisfy variable constraint, $last.x = P_K.y$, in relation P_k, which is directly solved based on the *matchPointer* of data index and *Root* of semantic index (line 6). The number of attributes in MT is increased step by step through the join operation between MT and new relation P_K (line 7). Algorithm 6 is an iterative process, in which each loop needs to determine the join attributes of MT and next relation P_k through the constraint element $last.x = P_k.y$. Set S stores the attribute values of MT, which will join with relation P_k in next loop (line 10). Since line 7 involves the join operation among multiple relations in the loop, we adopt single tuple technique to avoid the intermediate result files. Algorithm 6 can ensure that each tuple is accessed only once based on Yield index.

Algorithm 7 illustrates the rule activation for all existing tuples. We select a relation and use Algorithm 6 to activate the rule for every tuple in the relation.

Algorithm 7 Whole Relations Activation

Input:
A Datalog rule $R : head :- p_1, p_2, ..., p_n$;
M : The *SCE* of R;
Output:
MT : A set of matching tuples based on t;
1: let P be a relation that belongs to R;
2: $S = \emptyset$;
3: **for each** t_i that belongs to P
4: $MT = MT \cup \{Act(t_i)\}$;
5: **return** S;

5 Optimization and extension

In this section, we discuss some optimization and extension techniques based on Yield index.

5.1 Further optimization

The *SCE* decomposition method given in Section 3.3 is a pessimistic method that takes the worst situation into consideration in order to guarantee the correctness of the derived *SCE*. However, an optimistic method can be applied to improve the performance of Yield index.

$$R_7 : q(x, y) :- p_1(x, z), p_2(y, z), p_3(w, z)$$

$$R_7 : q(x, y) :- p_1(x, y), p_2(x, z), p_3(y, w), p_4(y, u)$$

The *SCE* of rule R_6, $\{P_{1\cdot\$2} = P_{2\cdot\$2}, P_{2\cdot\$2} = P_{3\cdot\$2}, P_{1\cdot\$2} = P_{3\cdot\$2}\}$, is a rule with one constraint variable z. For these constraint elements, original Yield index will build three data indices for $P_{1\cdot\$2}$, $P_{2\cdot\$2}$, and $P_{3\cdot\$2}$ respectively. However, only one data index is sufficient to perform rule activation. We can choose the relation with the largest cardinality to build the data index. Assuming that P_1 has the largest cardinality among the three relations, the data index is built on $P_{1\cdot\$2}$ only. For a tuple $t(y, z)$ in relation P_2 (or P_3), we use z to search the data index, and use Algorithm 3 to add the index entry of t into the semantic index between relations P_1 and P_2 (or P_3). This enables us to find all the tuples with value z, regardless that the found tuple is in relation P_1, P_2, or P_3. Therefore, we can find all matching tuples of t. This optimization can dramatically decrease the space overhead of Yield index through limiting the number of data indices. For the rule which has one constraint variable for example rule R_6, the more the predicates of a rule are, the more effective this method is.

The above optimization can be extended to a rule with multiple variable constraint. In rule R_7, we have two variables with *variable constraint*, x and y, among P_1, P_2, and P_1, P_3, P_4, respectively. Only two data indices are enough according to the above optimization. We select the relation with the largest cardinality to build the data index for each variable with *variable constraint*. If a data index has already existed in the selected relation, we choose the relation with the second largest cardinality until the relation has not been used before. Let the cardinalities of relations P_1, P_2, P_3 and P_4 be $N(P_1)$, $N(P_2)$, $N(P_3)$ and $N(P_4)$, respectively, where $N(P_1) > N(P_2) > N(P_3) > N(P_4)$. We select relation P_1 to build a data index for variable x. Then for variable y, an ideal selection is relation

P_1. Since P_1 has been used for x, we select P_3 to build a data index for variable y. The space complexity for constructing the data index and semantic index can be greatly reduced using this. We have to process one index first and generate an intermediate tuple set that contains the matching tuples bound by the variable with *variable constraint*. Then, for the tuples in the intermediate tuple set, we use the remaining index to find the matching tuples bound by other variable with *variable constraint*.

5.2 Extensibility of yield index

The extended *Datalog* brings new types of constraints so that we require new types of semantic index to express their additional semantic information. Two types of specific semantic index are presented in order to demonstrate how to use Yield index to deal with the additional constraints from the extended *Datalog* languages. We also discuss the possibility of solving transitive closure using Yield index.

5.2.1 Extension on inequality constraint

Inequality is a very useful relationship in various applications, for example "*Find all conference participants that leave after five p.m*" and "*Collect temperature information if the temperature difference is more than 40*". Inequality is processed as built-in predicate in existing works [9].

The constant constraint and variable constraint discussed in Section 3 are equivalent constraint, which means the relationship between the constraint variables is equivalent. However, the relationship between the constraint variables is more complex in inequality constraint. Yield index can be extended to support the rule activation with inequality constraint. We introduce a new type of semantic index, *InequalityS*, to deal with inequality constraints. *InequalityS* includes the following five Inequality types to specify the relationship between the variables in an inequality constraint.

- Equivalent type (EQJ): $P_{i\cdot\$1} = P_{j\cdot\$2}$, which is discussed in original Yield index.
- Greater type (GJ): $P_{i\cdot\$1} > P_{j\cdot\$2}$.
- Greater/equivalent type(GEJ): $P_{i\cdot\$1} \geq P_{j\cdot\$2}$.
- Less type (LJ): $P_{i\cdot\$1} < P_{j\cdot\$2}$.
- Less/equivalent type(LEJ): $P_{i\cdot\$1} \leq P_{j\cdot\$2}$.

Figures 8 and 9 give the index structure with the *InequalityS* of an example. The intuition of *InequalityS* is to index the boundary of inequality constraints in order to restrict the matching tuples within the interval bound by the semantic index entries. Due to the searching invariant property, we can decide if a tuple matches the inequality constraint by checking whether its searching path encounters the *InequalityS* index entry or not. We use two semantic subtypes, *InequalityS-l* and *InequalityS-r*, to denote the left and right boundary of restricted interval respectively.

Apparently, the structure of Yield index remains the same in dealing with the new semantic index type. Owing to the difference between equality constraint and inequality constraint, the implementation in Section 4 should be modified. A difference is that a satisfied tuple of *InequalityS* does not require directly under a fired semantic index entry. Whether a tuple matches an inequality constraint is determined by the encounters of the *InequalityS* index entries during its searching path. The modification of rule activation should be made due to the lack of explicit semantic index entries indicating who puts the inequality constraint on the tuple.

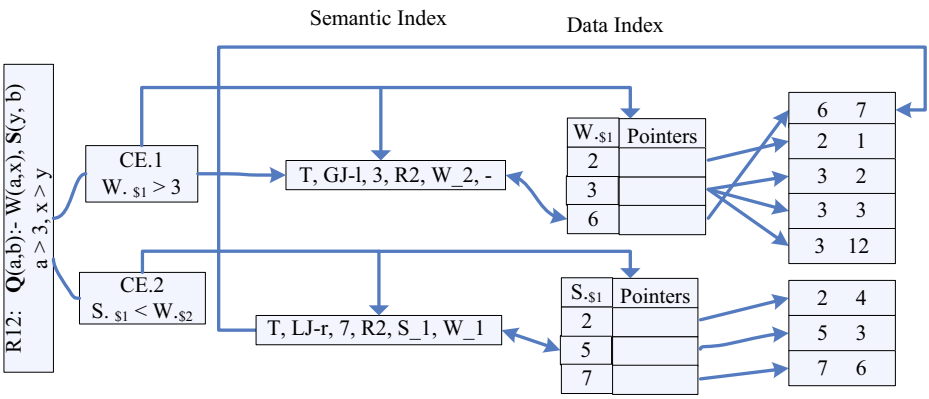


Figure 8 The Extended Yield Index Structure of Figure 9

5.2.2 *Yield index with Snlog*

Snlog is a specific *Datalog*-based declarative language [9], which is developed for the design and implementation of declarative sensor networks. To manage and execute logic languages on sensor networks, *Snlog* requires that every *EDB* predicate must have an argument that indicates its stored address. A location specificity symbol of an underlined address field is introduced to denote the location of the tuple. Some rule-level constraints of Snlog are presented in [22]. We assume that the *Datalog*-like rules satisfy the definition of *Snlog*. A typical *Snlog* rule is as follows.

$$path(\underline{s}, d, p, c) \ :- \ loc(s, \underline{z}, c_1), \ path(\underline{z}, d, p_2, c_2), c = c_1 + c_2, \\ p = concatpath(loc(s, \underline{z}, c_1), p_2)$$

The denotation of location is the major difference between the syntaxes of *Snlog* and *Datalog*, which brings new semantic information. Location constraint guarantees that all tuples which participate in current rule activation and new tuple are stored at the same node so that it can guarantee the validation of the execution of *Snlog* [9]. In order to build Yield index structure for *Snlog*, a new type of semantic index should be designed for the location constraint.

Together with the local information, we would translate the location constraint into some constraint elements. In the above *Snlog* rule, the constraint elements are:

$$\{LOC._{\$2} = local\ address, PATH._{\$1} = local\ address\}$$

Although these constraint elements have the similar form as constant constraint elements, their semantics are different because they are the preconditions of other constraints. These

Figure 9 The Example with Inequality Constraints

IDB

R12: q(a,b) :- w(a,x), s(y, b)
a > 3, x > y

EDB

W (6,7) (2,1) (3,2) (3,3) (3,12)
S (2,4) (5,3) (7,6)

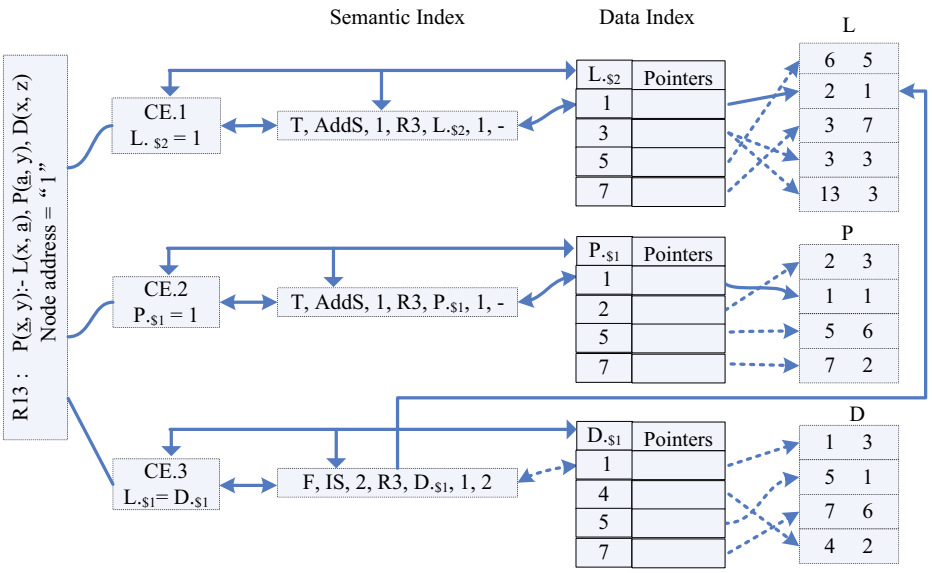


Figure 10 The Extended Yield Index Structure of Figure 11

constraint elements have the higher priority than the others. We introduce a semantic index type of *AddressS* to express this constraint. Figures 10 and 11 give the index structure of Yield index with *AddressS* of an example.

Due to their higher priority, the construction and maintenance of *AddressS* index entries are independent from the normal semantic index entries. Once a location constraint is identified, we build the *AddressS* index entry before the construction or other operations of Yield index. The procedure of building *AddressS* index entries is the same as the ***CS*** index entries. When an unsuccessful searching for index entry occurs during the construction and maintenance of *AddressS* index entries, we halt the whole process of index construction until some inserted tuples fire the unsuccessful *AddressS* index entries. The rule activation of *AddressS* index entries remains the same as normal semantic index types, such as *CS* and *IS*, since the evaluation of *Snlog* is nearly the same as *Datalog* besides the introduce of the location specificity symbol.

5.3 Recursive rule with yield index

Previously, we focus on single iteration activation of a datalog rule with Yield index. However, datalog rule is sometimes recursive in modeling a concrete application. Let *PARENT*(*x*,

Figure 11 The Example with Inequality Constraints

IDB

R13 : p(x, y):- l(x, a), p(a, y), d(x, z)
Node address = "1"

EDB

L (6,5) (2,1) (3,7) (3,3) (13,3)
P (2,3) (1,1) (5,6) (7,2)
D (1,3) (5,1) (7,6) (4,2)

y) be one *EDB* relation, which represents y is x's parent. Then the ancestor relationship can be represented by the following rules.

$$RA : ancestor(x, y) \ :- \ parent(x, y)$$
$$ancestor(x, y) \ :- \ parent(x, z), ancestor(z, y)$$

Rule *RA* is a typical recursive rule. Agrawa et al. [2] show that recursive rules can be converted to solving the transitive closure on the *EDB* relation, which is called as α operator. For the query that *"finds the common ancestor of Jim and Tom"*, for example, we can get the following relational algebra expression.

$$\pi_{pname}(\sigma_{name=Jim}(\alpha(PARENT(x, y)))) \ \cap$$
$$\pi_{pname}(\sigma_{name=Tom}(\alpha(PARENT(x, y))))$$

The following relational algebra expression can be used to solve the rule *RA*.

$$\begin{aligned} ANCESTOR(x, y) &= PARENT(x, y) \cup TC(PARENT(x, y)) \\ &= TC(PARENT(x, y)) \end{aligned}$$

Here *TC* is an operator of transitive closure, which solves the transitive closure on *EDB* relation *PARENT*.

Yield index can be used to solve the transitive closure of the *EDB* relation. We use the above example to show how Yield index can support solving transitive closure.

(1) Build Yield index for *EDB* relation *PARENT*

In order to build the Yield index of the relation *PARENT*, we need to build semantic index that links *PARENT* itself, which means both the *Root* and *Pointer* of the index locate on *PARENT*. We first build a data index on the attribute y of relation *PARENT*. Then we name *PARENT* as an alias *PARENT'(y,z)*. Using the tuples of *PARENT'(y,z)*, we can establish the semantic index between *PARENT* and *PARENT'*. Since *PARENT'* is only an alias of *PARENT* logically, the index pointers to the tuples in *PARENT'* actually point to the tuples in *PARENT*, which means that the semantic index connects relation *PARENT* itself.

(2) Solve the transitive closure based on Yield index There have been a lot of transitive closure algorithms based on graph theory, such as *BTC* [17], *MEMTC* [15] etc. The aforementioned example can also be solved in the similar manner. We can consider relation *PARENT(x, y)* as a graph G and the transitive closure on G can be derived using Yield index. The data index entry in Yield index represents a vertex in the graph, and semantic index represents the information of reachable edges from the vertex. Yield index contains the information of adjacency table of G, which enables us to calculate its transitive closure efficiently. We briefly describe the algorithm steps as follows.

a) Traverse each entry of the data index in turn, and invoke *Tarjan* algorithm [35] to solve all the strongly connected components on the graph G.
b) Compress the original graph. One strongly connected component is compressed into a vertex so that graph G_1 is constituted.
c) Derive the transitive closure based on the compressed G_1.

5.4 Rules with negation

Consider a rule with negation as follows.

$$RN : q(x, y) : -p(x, y), \neg s(x, y)$$

Here $\neg s(x, y)$ is a predicate with negation. As known from the literature [36], a negation operation in the rule body can be transferred to a set difference operation. The corresponding relational algebra expression of rule *RN* is as follows.

$$Q(x, y) = P(x, y) - S(x, y)$$

Its semantic is that $\forall t(x, y), t(x, y) \in P(x, y) \wedge t(x, y) \notin S(x, y)$. We call the relations without $\neg$ operation and with $\neg$ operation as *positive relation* and *negated relation* respectively.

The original Yield index links the tuples between two relations *P* and *S* using the equal values between join attributes. However, the set difference operation is solved for all tuples of a relation. As a result, we have to establish the relationship of the tuples using tuple equivalence. We thus introduce a new semantic index type *TS* to express the equivalence of two tuples.

The lexical analysis mechanism of rules can easily identify whether a rule is with negation operation. Since set difference is to judge the equivalence for the tuples, Yield index will establish data index for first attribute of the relations in rule body in order to quickly locate arbitrary tuple in the relation. The data structure of semantic index remains the same, but the meanings of some attributes are changed.

(*Fire, Type, KEY, RuleName, Root, Pointer*)

Here *Type* is *TS*. *KEY* is the value of attribute, which says that the tuples in corresponding two relations are equal. It represents that tuple values are equal rather than the attribute values are equal. The meanings of the other components remain the same.

Algorithm 8 Build the Index of Negation Rule

Input:
A Datalog rule $RN : head :- p(x, y), \neg s(x, y)$;
A tuple *t(x, y)* of relation *P* which will be inserted;

```
1:  Insert t.x into the data index PB of relation P;
2:  Search t.x in the data index SB of relation S;
3:  if no found
4:     Return;        //There is no equivalent tuple of t(x, y) in the S
5:  else
6:     Get the tuple s(x, y) from relation S according to the pointer;
7:    if t.y ≠ s.y
8:       Return;      //There is no equivalent tuple of t(x, y) in the S
9:     Construct a semantic index entry (T, TS, key, RN, XXX, YYY);
10:    Insert the index entry (T, TS, key, RN, XXX, YYY) into semantic index;
```

Algorithm 8 describes how to update Yield index with negation rule when a tuple *t* is inserted. The insertion of a tuple $t(x, y)$ results in updating the corresponding to the entries of data index and semantic index. First, we update the data index of relation *P* (line 1). Then Key *t.x* is used to search the relation with negation (line 2). The search result is used to judge whether the tuple *t(x, y)* is in relation *S* (lines 3-8). If there is the tuple *t(x, y)* in relation *S*, a semantic index entry *(T, TS, key, RN, XXX, YYY)* is constructed between relation *P* and *S* (line 9). Here *XXX* and *YYY* are two pointers, which respectively point the tuples in relation *P* and *S*. Finally, the semantic index entry *(T, TS, key, RN, XXX, YYY)* is insert into the semantic index between relation *P* and *S* (line 10).

Based on Yield index, we can easily solve the rules with negation since it is converted to the operation of the set difference and the tuple equivalent information between the relations of rule body has been stored in Yield index. Algorithm 9 describes how to solve a negation rule based on Yield index. First, result set is set to *NULL* (line 1). Then we traverse all entries of data index of *positive relation* (line 2). If *MatchPointer* of an entry is *null*, it represents current tuple *t(x, y)* does not belong to relation with negation (line 3). The tuple *t(x, y)* is added to result set (lines 4-5).

Algorithm 9 Solve the Rule with Negation Based on Yield Index

Input:

A Datalog rule $RN : head :- p(x, y), \neg s(x, y)$;
Relations P and S;

```
1: result = NULL;        //result set is set to null
2: for each index entry e of P's data index
3:    if e.MatchPointer = NULL
4:       Get tuple t(x, y) from P according to e.KeyPointer;
5:       result = result ∪ t(x, y);
```

When a new tuple *t(x, y)* is inserted into the relation *P*, *t.x* is used to search the data index of relation *P*. if not found, tuple *t(x, y)* will be stored in result set and Algorithms 8 is executed; Otherwise, we judge whether the tuple *t(x, y)* belongs to relation with negation. If it is not in the *negated relation*, *t(x, y)* is added in result set.

At the same time, Yield index with type *TS* can effectively support operations intersection and union between the relations.

5.5 Activation on a set of rules

Yield index can be extended to support the activation of a set of rules. According to Ullman et al. [36], a set of rules is regarded as a logic program. The semantic of a logic program can be stored in a data dictionary. We can also use data dictionary to store the abstract information of the rule's Yield index. When the rule *R* is activated, we first search the data dictionary to find the corresponding Yield index of the rule *R* and then perform rule activation based on Yield index.

For the rules in a logic program, dependency exists between different *IDB* predicates. An order for *IDB* predicates in a logic program can be derived based on their dependency, which is further used to determine the order of solving the *IDB* predicates as well as solving the rules. When a rule is solved, some tuples may be inserted into the relation, for example *Q*, corresponding a rule head. The relation *Q* may be a predicate of the rule body in another rule R_i. Therefore, solving a rule will result in updating the Yield index related to other rules, and then activate another rule according to an order of *IDB* predicates. Both rule activation and Yield index update alternate until all rules of the logic program are solved.

6 Evaluation

We implement Yield index on a Core 2, 2.66GHz computer with 16GB main memory and evaluate it on various *Datalog* rules with different data-sets. Our aim is to answer the following questions: *How does Yield Index (YI) compare with existing index structures in*

supporting rule activation? We select four baseline index structures, first-attribute-index (*FAI*), which builds the index on the first attribute of each relation [10], all-attributed index, which builds the index on all attributes of each relation [19, 27], *AMEM* index [24], and filter index (*FI*) [5].

6.1 Time complexity

Let R_0 be a rule which contains n relations, $P_1, \cdots, P_n$. Let N_i be the cardinality of relation P_i and m_{ij} be the number of matching tuples between relations P_i and P_j.

We analyze the time complexity in Yield index according to the order of the P_1, P_2, P_3, $\cdots$. Our main concern is the time complexity of accessing the index based on the tuple matching probability between P_i and P_j.

Let $N_i \approx N_j = N$ and $m_{ij} \approx m_{sl} = m$. The probability that a tuple of P_i matches any tuple of P_j is as follows.

$$pr_{ij} = \frac{m_{ij}}{N_i} = \frac{m}{N}$$

6.1.1 Time complexity of rule body

The data index of Yield Index is a B^+ tree of order f, the time complexity of which consists of two parts, searching inside the node and searching along the B^+ branches.

$$Log_2 f * log_f N_i = log_2 N_i$$

When a tuple t of P_i is inserted, the time complexity of rule activation, T, is the sum of searching the data index, T_1, and accessing the semantic index, T_2, of Yield index.

The time complexity of accessing the data index is as follows.

$$\begin{aligned}
T_1 &= log_2 N_1 + log_2 N_2 * \frac{m_{12}}{N_1} + log_2 N_3 * \frac{m_{12}}{N_1} * \frac{m_{23}}{N_2} + \cdots \\
&= \sum_{i=0}^{n-1} log_2 N_{i+1} * \left(\frac{m_{i,i+1}}{N_i}\right)^i \\
&\approx \sum_{i=0}^{n-1} log_2 N * \left(\frac{m}{N}\right)^i \\
&= log_2 N * \sum_{i=0}^{n-1} \left(\frac{m}{N}\right)^i \\
&= log_2 N * \left(\frac{1 - (\frac{m}{N})^n}{1 - \frac{m}{N}}\right) \\
&\approx log_2 N * \left(\frac{1}{1 - \frac{m}{N}}\right) \qquad \because m \ll N \qquad \therefore \left(\frac{m}{N}\right)^n \ll 1 \\
&= log_2 N * \left(\frac{N}{N - m}\right) \\
&\approx O\left(log_2 N * \frac{N}{N - m}\right)
\end{aligned}$$

Semantic index directly connects the matching tuples of different relations, which enables a constant time to locate the matching tuples of a tuple. As a result, the time complexity of accessing the semantic index is as follows.

$$T_2 = \sum_{i=1}^{n-1} m_{ij} \qquad i \in [1,2,...,n\text{-}1],\ j = j + 1$$
$$\approx \sum_{i=1}^{n-1} m \approx O(n * m)$$

Therefore,

$$T(N) = T_1(N) + T_2(N)$$
$$\approx O\left(\frac{N}{N-m} * log_2 N + n * m\right) \qquad m \ll N$$

Obviously, the less the matching tuples between the relations of rule body are, the higher the efficiency of YI index is. And the more the number of relations in rule body is the more obvious the efficiency of YI index is.

6.1.2 Time complexity of join

Yield index can effectively support the join operation since Yield index contains all matching tuples among the relations. If we know the join operation among the relations in advance, we can extend *SQL* statement to build Yield index, which is used to store all matching tuples on specified attributes in the relations. When a *SQL* statement executes a join operation with Yield index, it will find corresponding semantic index *YI-SI* (a subset of semantic index), directly locate the tuples in different relations through scanning all entries of *YI-SI*, and complete the join operation.

The time complexity of join operation between two relations is as follows.

$$T(n) = O(m)$$

Here m is the number of matching tuples between two relations. In general, m is far less than N which is the number of tuples in the relations.

6.1.3 Time complexity of negation rule

Known from Section 5.4, a negation operation is converted to a operation of set difference. For a rule with negation, for example rule *RN* in Section 5.4, we can use type *TS* to build Yield index. Its time complexity consists of two parts: one is to scan all the tuples in *positive relation P* and another is to deal with semantic index. Because the data index of relation P includes the keys of all tuples, we in turn scan all leaf nodes in data index of relation P.

$$T_1(n) = O(N)$$

Based on Yield index with type *TS*, there is the relationship of the tuple equivalence between the data index and semantic index. Each index entry in data index of relation P make clear whether a tuple in P is also in the relation S. So, the time complexity of dealing with semantic index is as follows.

$$T_2(n) = O(m)$$

Therefore, the time complexity of negation operation is:

$$T(n) = O(N) + O(m) = O(N + m)$$

6.2 Experiment setup

Evaluated Datalog rules. We use four *Datalog* rules to evaluate the effectiveness of *YI*. The rules are representative since they cover different scenarios and different constraint types between the relations.

1) *Requesting channel problem* [29]

$$UC : canusechannel(x, y) : -near(x, z), use(x, z), request(z, y)$$

Assume that there are many wireless devices in a large Mall. People request access to and uses a channel through some devices such as mobile phone. People keep on moving, close to or leave the wireless device. But only part of people that register and close to a device can access and use the channel.

This rule says that a person x can use a channel y if x is near a wireless device z, x can use the z, and z requests the channel y. Tuples in relations *NEAR*, *USE* and *REQUEST* are randomly generated using 500,000 distinguished values. We generate two data sets in which one contains 50,000 tuples and the other contains 100,000 tuples, denoted by UC_s and UC_l, respectively.

2) *Coordinated flight booking problem* [13]

$$CF : reserve(Tom, y) : -friend(s, x), city(s, c), city(x, c), ticket(x, y), s = Tom$$

This rule is proposed by Gupta et al. to find a flight that a person can book with his/her friends simultaneously. The rule says that *Tom* will book a flight y if x is a friend of *Tom*, both x and *Tom* live in a city c, and x books a flight y. Tuples in the relations are randomly generated using 100,000 person names, 1,000 city names and 100 flights. We generate two data sets, one with 50,000 tuples in each relation, and another with 100,000 tuples in each relation, which are denoted as CF_s and CF_l, respectively.

3) *WSN routing problem* [9]

$$RT : path(x, y) : -link(x, y), located(x)$$

This rule is proposed by Chu et al. to establish routing paths between *WSN* nodes. The rule says that a routing path is from node x to node y if there is a media-level link from node x to node y and the link information is stored on node x. Tuples in the relations are randomly generated using 20,000 node names. We generate two data sets, one with 50,000 tuples in each relation, and another with 100,000 tuples in each relation, which are denoted as RT_s and RT_l, respectively.

4) *Directly Graph problem* [36]

$$\begin{aligned} RG: \ & (1) path(x, y) :- arc(x, y) \\ & (2) path(x, y) :- arc(x, z), path(z, y) \end{aligned}$$

This is a recursive rule, which is used to test the efficiency of the recursive rule. Rule (1) says that there is a path from vertex x to y if there is an arc (directly edge) from vertex x to y. Rule (2) shows that there is a path from vertex x to y if there are an arc from vertex x to z and a path from vertex z to y.

Relation *ARC* is an *EDB* relation. The tuples in relation *ARC* are randomly generated using 3,000 different vertices. We select the 200 vertices from 3,000 vertices as the vertices with out-degree and respectively construct the directed graphs with different sizes. The number of tuples in relation *ARC* are 6,000, 8,000, 10,000, 12,000 and 15,000 respectively, which correspond five test data sets, RG_1, RG_2, RG_3, RG_4 and RG_5.

Configurations Our major concern is the effectiveness and overhead of the index structures. Regarding effectiveness, we compare the time cost of performing rule activation on the rules with the support of different index structures. We perform rule activation on the aforementioned data sets, in which all tuples are already stored in the relations. We also perform rule activation under the situation that tuples are continuously inserted into the relations, which is common for applications in dynamic and pervasive environment. In rule UC, we use 50% of tuples in relation *NEAR* as inserted tuples. In rule CF, we use 50% of tuples in relation *TICKET* as inserted tuples. In rule RT, we use 50% of tuples in relation *LINK* as inserted tuples. Regarding overhead, we compare the time and space cost to build the different index structures.

Measurements We use B^+ tree in main memory as the data index of Yield index. All tests of various index structures are executed on main memory. Since the time cost is closely related to the implementation of index structure, we use *index access times*(#), the number of index accesses, and *tuple access times*(#), the number of tuple accesses, to measure the time cost of rule activation supported by different index structures. Concerning the time cost in index construction, we use *index entry access times*(#) as the metrics, since time of index access may be different when the index size increases. For testing on the recursive rule, beside the *index entry access times*, we also compare the entire processing time (*time(ms)*) of Yield index with semi-naive algorithm on each data sets.

6.3 Result analysis

6.3.1 Rule activation on base relations

Table 1 gives an overview of rule activation. The result shows that rule activation is extremely time-consuming without index support. *YI* has the smallest index access times as well as tuple access times among the five index structures. *AAI* performs better than *FAI* and *FI* performs better than *AMEM*. In the following discussion, we focus on the three index structures, *AAI*, *FI* and *YI*, since *AAI* and *FI* are based on *FAI* and *AMEM*, respectively.

Figure 12 gives a comparison of AAI, FI, and YI on index access times. *YI* performs better than other two index structures, $2.1\% - 67.8\%$ less than AAI and $7.7\% - 70.5\%$ less than FI. Since rule activation has to access index entries for each tuple at least one time, the effect of avoiding ineffective rule activation is not obvious. As a result, the difference among *YI* and other methods is comparatively small, but it is sufficient to show the outperforming of *YI* on this metrics.

Figure 13 shows a comparison of AAI, FI, and YI on tuple access times. The performance of *YI* is much better than the other methods, decreasing the one order of magnitude on average. *YI* never wastes any tuple access since every tuple access indicates a successful rule activation. We notice that FI performs nearly as well as YI in rule CF. This is probably because FI is effective for selective constant constraint, such as relation *CITY(Tom, x)* in CF.

Table 1 Comparison of the index/tuple access number in rule activation

Index Structure	Access Target	UC_s	UC_l	CF_s	CF_l	RT_s	RT_l
No Index	index	0	0	0	0	0	0
	tuple	1.25E+14	1E+15	1.25E+14	1E+15	2.50E+09	1E+10
FAI	index	50,126	108,327	53,224	153,556	89,359	190,723
	tuple	1.19E+14	4.90E+13	52,183	145,428	65,994	147,498
AAI	index	55,277	123,790	51,114	141,917	74,305	147,017
	tuple	55,277	123,790	51,115	131,786	63,294	132,743
$AMEM$	index	0	0	63,657	124,248	52,634	104,274
	tuple	1.25E+14	1E+15	6.80E+06	2.48E+07	1.32E+08	4.27E+08
FI	index	55,277	123,790	73,756	174,248	79,355	160,715
	tuple	55,277	123,790	117	342	60,564	32,282
YI	index	51,026	108,327	50,086	100,268	52,634	104,274
	tuple	1,026	8,327	34	106	2,634	4,274

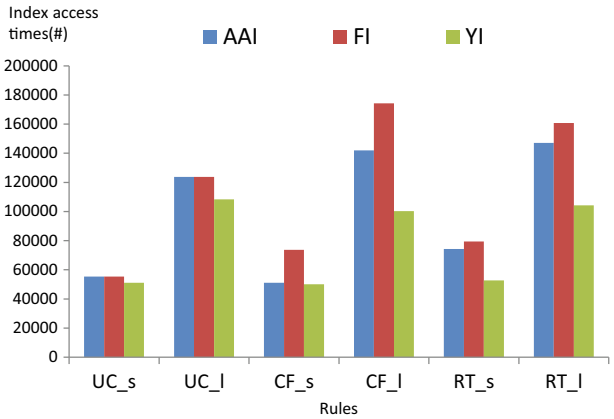


Figure 12 Index Access Times of AAI, FI, and YI on Base Relation

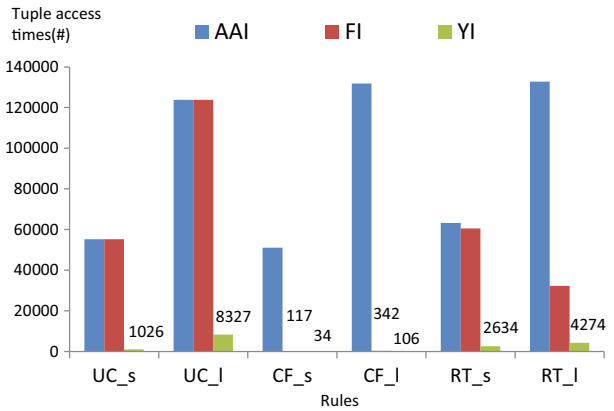


Figure 13 Tuple Access Times of AAI, FI, and YI on Base Relation

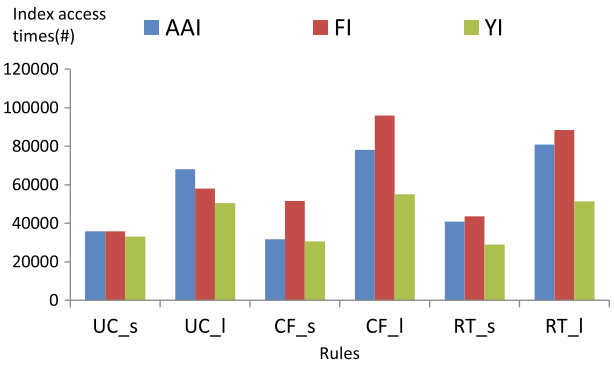


Figure 14 Index Access Times of AAI, FI, and YI on Inserted Tuples

6.3.2 *Rule activation for inserted tuples*

One of YI's advantages is integrating tuple insertion and rule activation. For existing methods, rule activation for inserted tuples includes two steps, updating the index and performing rule activation. However, YI only takes one step. When we update the index structure, rule activation is performed as shifting semantic index. Figures 14 and 15 show the result of rule activation on inserted tuples. Apparently, YI performs the best. It has $3.9\% - 36.5\%$ less index access times than AAI, $7.7\% - 42.5\%$ less than FI. Regarding tuple access times, YI is $93.3\% - 99.91\%$ less than AAI and $44.1\% - 98.7\%$ less than FI.

6.3.3 *Construction efficiency and space overhead*

YI performs well in rule activation efficiency, but we have to show that the time and space overhead of *YI* is affordable. The cost to build an index contributes the major time overhead of an index structure. Figure 16(a) to (f) show index access times during the index construction. Besides the above three index structures, we give a separate measurement on the semantic index(*YI_SI*) in *YI*. The result shows that the major overhead of *YI* is the construction of the data index, namely the construction of B^+ tree. The construction of the semantic index in

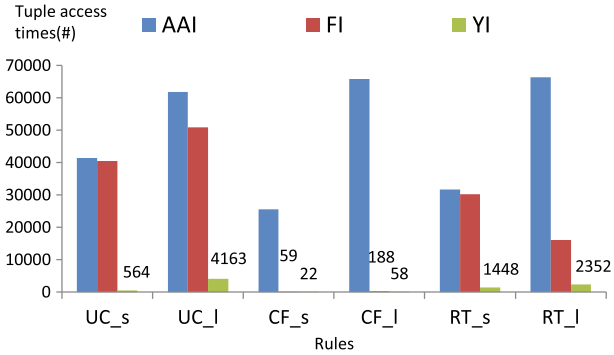
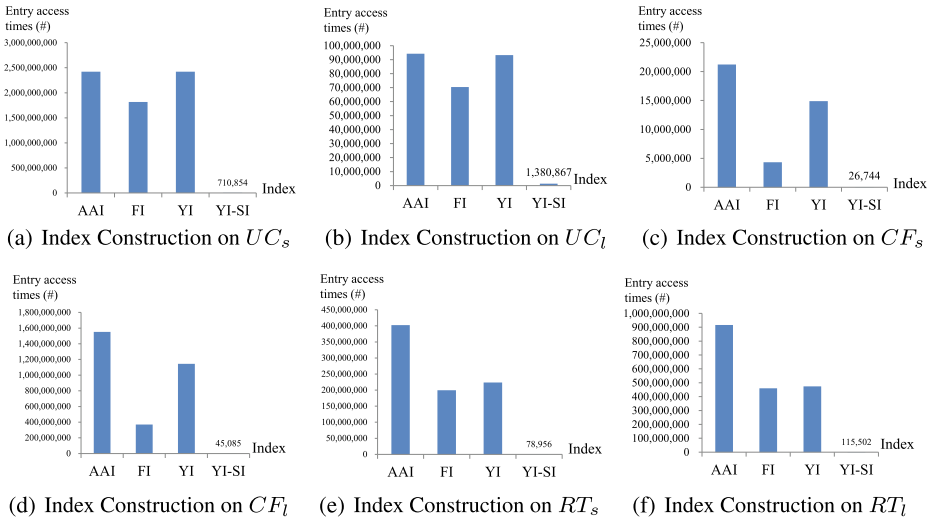


Figure 15 Tuple Access Times of AAI, FI, and YI on Inserted Tuples



(a) Index Construction on UC_s (b) Index Construction on UC_l (c) Index Construction on CF_s

(d) Index Construction on CF_l (e) Index Construction on RT_s (f) Index Construction on RT_l

Figure 16 Index Access Times During Construction

YI brings a much smaller overhead. The performance of *YI* varies a lot for different rules. In the test sets of rule *UC*, which has a multi-variable constraint, *YI* spends lots of time to build the data index. In the test sets of rule *CF*, *YI* performs much better, which is nearly as the same as *FAI*. In the test sets of rule *RT*, which has a highly selective constant constraint, the overhead of *YI* drops dramatically, just a litter larger than the overhead of *FI*. However, unlike *FI*, the major overhead of *YI* is actually overtaken by all rules that share the same predicates.

Another issue of index overhead is the size of index file. Figure 17 shows the number of index entries of *AAI*, *FI* and *YI*. *YI* and *FI* are better than other index structures in most tests. However, the major part of overhead in *YI* is shared by different rules, while *FI* is a rule-specific method. *YI* does not bring more overhead than other methods in terms of time and space. Considering the overhead and rule activation efficiency, we conclude that Yield index can support efficiently the rule activation with affordable overhead.

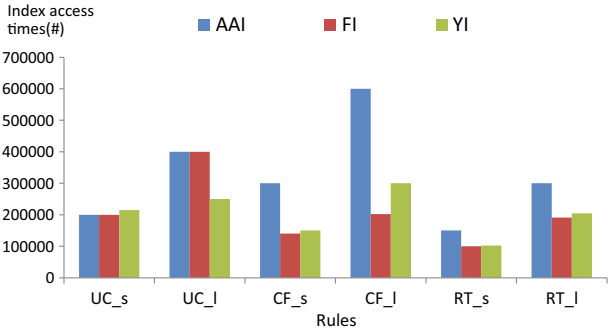


Figure 17 Comparison of the Number of Index Entries

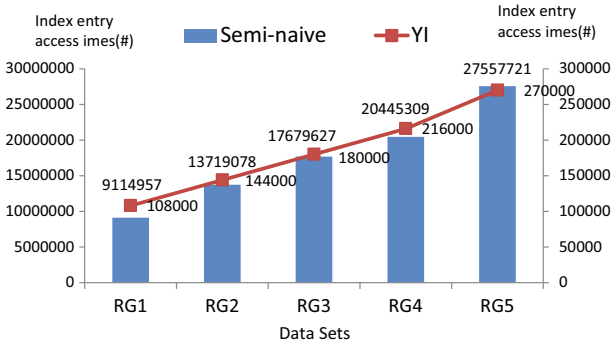


Figure 18 Comparison of the Number of Index Entries

6.3.4 Comparison with semi-naive algorithm

In order to validate the efficiency of the recursive rule, we compare the performance of Yield index with *semi-naive* algorithm [36], and measure the processing time and index entry access times of both methods. For semi-naive algorithm, we build B^+ tree index to search the wanted tuples efficiently. Figure 18 compares the index entry accessing times between Yield index and *semi-naive* algorithm. Yield index greatly reduces the index entry accessing times in all five data sets, and it achieves an average two order of magnitude improvement in efficiency. Since both two approaches use B^+ tree as the basic data structure, Yield index's advantage is due to its semantic index that links the matching tuples and dummy tuples directly.

Figure 19 shows the execution time of Yield index and *semi-naive* algorithm. Yield index is 1,506-4,115 times faster than *semi-naive* algorithm on the five test data sets. The main reason is that our approach is first to find the strongly connected components and then to solve the recursive rule on the basis of strongly connected components. Yield index improves the efficiency of solving the recursive rule since it can effectively support the solution of the strongly connected components and transitive closure. We also notice that Yield index's leading in execution time is much larger than its leading in index entry access times. We believe that this difference shows that besides the proposed semantic index, Yield index also uses the information acquired from the data index in a more effective manner.

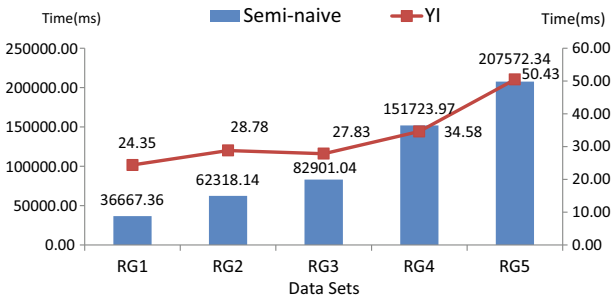


Figure 19 Comparison of running time

7 Related work

Rule activation has been extensively studied for decades. Reference [33] lists three prevailing rule activation techniques in principle, brute force, the discrimination networks [11, 24], and marking method [32, 34]. The discrimination networks are proposed by researchers in A.I. community, in which rules, predicates, and tuples are connected through a network that can assist the activation of rules. *Rete* [11] adopts a rule-centric fashion, which uses the rules to construct a network that efficiently locates tuples based on the constant constraints between rules and tuples. *TREAT* [24] uses a tuple-centric fashion to record all potential useful tuples in a network. Our rule activation method is similar with discrimination networks, in which Yield index connects the matching tuples through its semantic index. However, Yield index organizes its semantics index based on data index, which can be updated in a more effective manner and brings much lower space overhead.

Marking method can be view as a primitive form of specific index structure that supports rule activation. References [34] and [32] use some well-designed records to keep track of tuples that may be involved in rule activation. Since the records are less-structured, marking method performs worse than the specific index structures.

A well-chosen index may mean the difference between hours and a few seconds in rule activation [20]. In most systems, conventional index structure is directly applied to the rule mechanism. Some systems use a brute force method, which builds the index on predefined attributes of *EDB* relations. *DLV* [10] builds an index on the first attribute of each *EDB* relation. *Jena* [19] and *OWLIM* [27] build the indices on each attribute of each *EDB* relation. Other systems determine the *KEY* of the index through analyzing the activation rule, such as *Ontobroker* [26] and *Yap* [39]. *XSB* provides a more flexible index mechanism [30].

Some works use specific index structure to support rule activation. *LEAPS* [25] uses *AMEM* index to record the static part of extensional databases, which is bound by the constant constraints of the rules. The similar technique is used in Ariel System [14]. As an improved version of *LEAPS*, *DATEX* [5] presents *FI* index structure, Filter Index, which mixes the idea of *TREAT* with the marking method. In *DATEX*, the constant constraints, like $S._{\$1}=3$, are used to build the *AMEM* index in the same way as *LEAPS*. Meanwhile, the semantic information of variable constraint, e.g., $S._{\$2} = T._{\$1}$, is expressed by *FI* index, which records all tuples with the same value on the equivalent arguments.

All above index structures, either conventional or specific, improve the efficiency of rule activation through quickly locating the matching tuples in each relation. These indices are built on the *EDB* relations separately, and a well-design algorithm is applied to solve the rules. But they provide little information about the rule's semantics, especially the semantic constraint information between the body predicates that shapes matching tuples among the relations. These structures cannot tell whether an inserted tuple has matching tuples without performing activation algorithm. As a result, these index structures suffer from ineffective invoking. Facing the dynamic and interactive nature of current applications, frequent ineffective invoking is one of the major causes of rule activation's poor-efficiency.

Segev et al. [31] propose a join pattern indexing that records both complete and incomplete join information of base relations for rule activation. The recorded join information for join pattern index is similar with the semantic index of Yield index, but the join pattern index is built as an independent relation, which makes its updating less efficient. In Yield index, we build the semantic index based on the data index and use dummy index to update existed semantic index in a much more efficient manner. Meanwhile, Yield index also provides the navigation among the relations, which is the basis of solving a recursive rule.

Wang et al. [37] introduce belief rules as an extension to traditional *'If-THEN'* rules which is the form of '*If P THEN Q*'. The consequent (*Q*) is believed to be 100% true given that the antecedent (*P*) has happened. Previous research has shown that such strict knowledge representation scheme leaves no room for uncertain or incomplete judgements [1]. In recent years, the rule activation is also discussed in *Belief Rule Bases*. Some researches focus on an advanced belief rule-based decision model and propose a dynamic rule activation (*DRA*) method to address data incompleteness and inconsistency issues in data-driven decision models [3, 7], in which the activated rules are selected in a dynamic way to search for a balance between the incompleteness and inconsistency. But the rules used in these pieces of works are not a kind of *Datalog* rules.

Some pieces of work focus on the computational complexity of *Datalog*-like logic programs. Liu et al. [21] propose a method for transforming any set of *Datalog* rules into an efficient specialized implementation with guaranteed worst-case time and space complexities. Calvanese et al. [6] present algorithms for handling usual reasoning tasks, as well as answering the unions of conjunctive queries, on the proposed family of description logics. Their works and our work are complementary, since Yield Index improves the efficiency of answering a single *Datalog* rule at implementation level, while their works improve the efficiency of answering a set of *Datalog* rule at query-planning level.

The intuition of the extended semantic index, *InequalityS*, is similar to the work on the region monitoring query(*RMQ*) [8, 16]. The difference is that our inequality constraint not only concerns about the evaluation of region query, but also focuses on how to link the query result with other matching tuples to generate the matching tuple set of the rule. As a result, the semantic index of Yield index can connects the matching tuples and support rule activation directly.

In other research areas, the rule activation is also discussed. Mandl et al. [23] discuss the multi-context systems with activation rules. If an activation rule is fired, its context is active and shall be taken into account. If a context is not activated by any activation rule, it is not considered. But the efficiency of the rule activation is not discussed.

8 Conclusion

The application scenarios of dynamic and pervasive computing environment, in which the system will interact with the environment intensively, request high efficiency of logic programming. In the face of the poor-efficiency of rule activation, we propose an index structure, Yield index, to support rule activation effectively. The index mechanism sets up the data index and semantic index for constraint elements in rules to contain the information of rule semantics and tuple matching between the relations. Our approach integrates tuple insertion and rule activation to avoid ineffective invoking of rule activation. The article introduces the index structure of Yield index, the construction and maintenance algorithms, as well as the activation algorithm. We also discuss the good extensibility of Yield index. The experimental results show that Yield index has better performance and improves activation efficiency of one order of magnitude, comparing with other index structures. Yield index is suitable for application scenarios with frequent interaction in dynamic and pervasive computing environment.

Acknowledgments This work was supported by National Natural Science Foundation of China(Grant Nos. 91318301, 61073031, 61321491, 61373011 and 61772258).

References

1. AbuDahab, K., Xu, D., Chen, Y.: A new belief rule base knowledge representation scheme and inference methodology using the evidential reasoning rule for evidence combination. Expert Syst. Appl. **51**, 218–230 (2016)
2. Agrawal, R.: Alpha: an extension of relational algebra to express a class of recursive queries. IEEE Trans. Softw. Eng. **14**, 879–885 (1988)
3. Alberto, C., Liu, J., Wang, H., Kashyap, A.: A new dynamic rule activation for extended belief rule bases. Proc. Int. Conf. Mach. Learn. Cybern. **18**, 1836–1841 (2013)
4. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathanf, A., Riboni, D.: A survey of context modelling and reasoning techniques. Perv. Mob. Comput. **6**(2), 161–180 (2010)
5. Brant, D.A., Miranker, D.P.: Index support for rule activation. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data(SIGMOD,'93), pp. 42–48. Washington, D.C. (1993)
6. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The dl-lite family. J. Autom. Reason. **39**(3), 385–429 (2007)
7. Calzada, A., Liu, J., Wang, H., Kashyap, A.: A new dynamic rule activation method for extended belief rule-based systems. ACM Trans. Knowl. Data Eng. **27**, 880–893 (2015)
8. Chandrasekaran, S., Franklin, M.J.: Psoup: A system for streaming queries over streaming data. VLDB J. **12**, 140–156 (2003)
9. Chu, D., Popa, L., Tavakoli, A., Hellerstein, J.M., Levis, P., Shenker, S., Stoica, I.: The design and implementation of a declarative sensor network system. In: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems (SenSys'07), pp. 175–188. Sydney (2007)
10. Dlv: [Online]. Available: http://www.dbai.tuwien.ac.ar/proj/dlv/
11. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artif. Intell. **19**(1), 17–37 (1982)
12. Gu, T., Pung, H., Zhang, D.: Toward an osgi-based infrastructure for context-aware applications. Perva. Comput. IEEE **3**(4), 66–74 (2004)
13. Gupta, N., Kot, L., Roy, S., Bender, G., Gehrke, J., Koch, C.: Entangled queries: enabling declarative data-driven coordination. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11), pp. 673–684. Athens (2011)
14. Hanson, E.: The design and implementation of the ariel active database rule system. IEEE Trans. Knowl. Data Eng. **8**(1), 157–172 (1996)
15. Hirvisalo, V., Nuutila, E., Soisalon-Soininen, E.: Transitive closure algorithm memtc and its performance analysis. Discret. Appl. Math. **110**, 77–84 (2001)
16. Hu, H., Xu, J., Lee, D.L.: A generic framework for monitoring continuous spatial queries over moving objects. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data(SIGMOD,'05), pp. 479–490. Baltimore (2005)
17. Ioannidis, Y., Ramakrishnan, R., Winger, L.: Transitive closure algorithms based on graph traversal. ACM Trans. Database Syst. **18**, 512–576 (1993)
18. Jagadish, H.V., Agrawal, R., Ness, L.: A study of transitive closure as a recursion mechanism. In Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD'87) (1987)
19. Jena: [Online]. Available: http://www.jena.sourceforge.net/
20. Liang, S., Fodor, P., Wan, H., Kifer, M.: Openrulebench: an analysis of the performance of rule engines. In: Proceedings of the 18th International Conference on World wide Web (WWW'09), pp. 601–610. Madrid (2009)
21. Liu, Y.A., Stoller, S.D.: From datalog rules to efficient programs with time and space guarantees. ACM Trans. Program. Lang. Syst. **31**(6), 1–38 (2009)
22. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: language, execution and optimization. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data(SIGMOD'06), pp. 97–108. Chicago (2006)
23. Mandl, S., Ludwig, B.: Multi-context systems with activation rules. In: Proceedings of the 33rd Annual German Conference on AI, LNAI vol. 6359, pp. 135–142. Karlsruhe (2010)
24. Miranker, D.P.: Treat: a better match algorithm for ai production systems. In: Proceedings of the Sixth National Conference on Artifical Intelligence (AAAI'87), pp. 42–47. Seattle (1987)
25. Miranker, D., Lofaso, B.: The organization and performance of a treat-based production system compiler. IEEE Trans. Knowl. Data Eng. **3**(1), 3–10 (1991)
26. Ontobroker: [Online]. Available: http://www.ontoprise.de/de/en/home/products.html/

27. Owlim: [Online]. Available: http://www.ontotext.com/owlim/index.html/
28. Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context aware computing for the internet of things: A survey. IEEE Commun. Surv. Tutor. **16**(1), 414–454 (2014)
29. Perttunen, M., Riekki, J., Lassila, O.: Context representation and reasoning in pervasive computing: A review. Int. J. Multimed. Ubiq. Eng. **4**(4), 1–28 (2009)
30. Sagonas, K., Swift, T., Warren, D.S.: Xsb as an efficient deductive database engine. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data(SIGMOD,'94), pp. 442–453. Minneapolis (1994)
31. Segev, A., Zhao, J.: Rule activation techniques in database systems. J. Intell. Inf. Syst. **7**, 173–194 (1996)
32. Sellis, T., Lin, C.C., Raschid, L.: Implementing large production systems in a dbms environment: Concepts and algorithms. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data(SIGMOD'88), pp. 404–423. Chicago (1988)
33. Stonebraker, M.: The integration of rule systems and database systems. IEEE Trans. Knowl. Data Eng. **4**(5), 415–423 (1992)
34. Stonebraker, M., Jhingran, A., Goh, J., Potamianos, S.: On rules, procedure, caching and views in data base systems. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data(SIGMOD,'90), pp. 281–290. Atlantic City (1990)
35. Tarjan, R.: Depth first search and linear graph algorithms. SIAM J. Comput. **1**, 146–160 (1972)
36. Ullman, J.D.: Principles of Database and Knowledge-base Systems, vol. I. Computer Science Press Inc., New York (1988)
37. Wang, Y.M., Yang, J.B., Xu, D.L.: Environmental impact assessment using the evidential reasoning approach. Eur. J. Oper. Res. **174**, 1885–1913 (2006)
38. Wolfson, O., Silberschatz, A.: Distributed processing of logic programs. In: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD,'88), pp. 329–336. Chicago (1988)
39. Yap: [Online]. Available: http://www.dcc.fc.up.pt/vsc/Yap/
40. Zhang, W., Yu, C.T.: A necessary condition for a doubly recursive rule to be equivalent to a linear recursive rule. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD'87). San Francisco (1987)