Noname manuscript No.
(will be inserted by the editor)

Title

Subtitile

First Author · Second Author

Received: date / Accepted: date

Abstract Insert your abstract here. Include keywords, PACS and mathematical subject classification numbers as needed.

Keywords First keyword · Second keyword · More

1 Introduction

2 The Motivating Example

In this section, we consider a simple example of the context-free path query evaluation problem, using a small graph and the classical *same-generation query* [1], which cannot be expressed using regular expressions.

Suppose we have a graph database or any other object, which can be represented as a graph. The same-generation query can be used for discovering a vertex similarity, for example, gene similarity [2]. For graph databases or RDF data, the same-generation query is aimed at the finding all the nodes at the same hierarchy level. The language, formed by the paths between such nodes, is not regular and corresponds to the language of matching parentheses. Hence, this query can be formulated as a context-free grammar and cannot be formulated as a regular grammar.

Supported by the Russian Science Foundation grant 18-11-00100.

F. Author
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

For example, let us have a small double-cyclic graph (see Figure 1). One of the cycles has three edges, labeled with a, and the other has two edges, labeled with b. Both cycles are connected via a shared node 0.

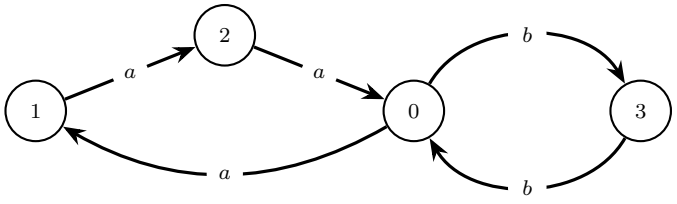

Fig. 1: An example graph.

For this graph, we have a same-generation query, formulated as a context-free grammar, which generates a context-free language $L = \{a^n b^n \mid n \geq 1\}$.

The result of context-free path query evaluation w.r.t. relational query semantics for this example is a set of all node pairs (m, n), such that there is a path from the node m to the node n, whose labeling forms a word from the language L. For example, the node pair $(0, 0)$ must be in this set, since there is a path from the node 0 to the node 0, whose labeling forms a string $w = aaaaaabbbbbb = a^6b^6 \in L$. This path is obtained by traversing twice the cycle with a labels and three times the cycle with b labels.

3 Preliminaries

3.1 Context-Free Path Querying

Let Σ be a finite set of edge labels. Define an *edge-labeled directed graph* as a tuple $D = (V, E)$ with a set of nodes V and a directed edge relation $E \subseteq V \times \Sigma \times V$. For a path π in a graph D, we denote the unique word, obtained by concatenating the labels of the edges along the path π as $l(\pi)$. Also, we write $n\pi m$ to indicate, that the path π starts at the node $n \in V$ and ends at the node $m \in V$.

Following Hellings [3], we deviate from the usual definition of a context-free grammar in *Chomsky Normal Form* [4] by not including a special starting non-terminal, which will be specified in the path queries for the graph. Since every context-free grammar can be transformed into an equivalent one in Chomsky Normal Form and checking, that empty string belongs to the language is trivial, it is sufficient to consider only grammars of the following type.

A *context-free grammar* is a triple $G = (N, \Sigma, P)$, where N is a finite set of non-terminals, Σ is a finite set of terminals, and P is a finite set of productions of the following forms:

- $A \to BC$, for $A, B, C \in N$,

– $A \to x$, for $A \in N$ and $x \in \Sigma$.

Note that we omit the rules of the form $A \to \varepsilon$, where ε denotes empty string. This does not restrict the applicability of our algorithm since only the empty paths $m\pi m$ correspond to empty string ε.

We use the conventional notation $A \xrightarrow{*} w$ to denote, that a string $w \in \Sigma^*$ can be derived from a non-terminal A by some sequence of production rule applications from P. The *language* of a grammar $G = (N, \Sigma, P)$ with respect to a start non-terminal $S \in N$ is defined by

$$L(G, S) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}.$$

Rules in our grammars will be enumerated. When we want to specify the number i of the applied rule, we use the conventional notation $w_1 \xrightarrow{i} w_2$ where $w_1, w_2 \in (N \cup \Sigma)^*$.

For a given graph $D = (V, E)$ and a context-free grammar $G = (N, \Sigma, P)$, we define *context-free relations* $R_A \subseteq V \times V$ for every $A \in N$, such that

$$R_A = \{(n, m) \mid \exists n\pi m \ (l(\pi) \in L(G, A))\}.$$

In order to find these context-free relations, we use a semiring without associativity of the product over the subsets of the set of nonterminals N. It consists of a product operation $(\ \cdot\ ) : 2^N \times 2^N \to 2^N$ and the union $(\cup) : 2^N \times 2^N \to 2^N$ as an associative and commutative sum operation. Also, the sum is distributive over the product.

We define a product operation $(\ \cdot\ )$ for arbitrary subsets N_1, N_2 of N with respect to a context-free grammar $G = (N, \Sigma, P)$ as

$$N_1 \cdot N_2 = \{A \mid \exists B \in N_1, \exists C \in N_2 \text{ such that } (A \to BC) \in P\}.$$

Using this semiring, we can define a *matrix multiplication*, $a \times b = c$, where a and b are matrices of size $n \times n$, that have subsets of N as elements, as

$$c_{i,j} = \bigcup_{k=1}^{n} a_{i,k} \cdot b_{k,j}.$$

Also, we use the element-wise union operation on matrices a and b with the same size: $a \cup b = c$, where $c_{i,j} = a_{i,j} \cup b_{i,j}$.

According to Valiant [5], we define the *transitive closure* of a square matrix a as $a_+ = a_+^{(1)} \cup a_+^{(2)} \cup \cdots$, where $a_+^{(1)} = a$ and

$$a_+^{(i)} = \bigcup_{j=1}^{i-1} a_+^{(j)} \times a_+^{(i-j)}, \ i \geq 2.$$

Valiant proposed an algorithm for a context-free recognition for a linear input, which in graph terms corresponds to a directed chain of nodes. The algorithm enumerates the positions in the input string s from 0 to the length of s, constructs an upper-triangular matrix, and computes its transtive

closure. In the context-free path querying input graphs can be arbitrary, which removes the upper-triangularity limitation. For this reason, we introduce another definition of transitive closure for arbitrary square matrix a as $a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots$, where $a^{(1)} = a$ and

$$a^{(i)} = a^{(i-1)} \cup (a^{(i-1)} \times a^{(i-1)}),\ i \geq 2.$$

These two transitive closure definitions are in fact equivalent. To show the equivalence of a^{cf} and a_+ definitions of transitive closure, we introduce the partial order $\succeq$ on matrices with a fixed size that have subsets of N as elements. For square matrices a, b of the same size we denote $a \succeq b$ iff $a_{i,j} \supseteq b_{i,j}$, for every i, j. For these two definitions of transitive closure, the following lemmas and theorem hold.

Lemma 1 *Let $G = (N, \Sigma, P)$ be a grammar, let a be a square matrix. Then $a^{(k)} \succeq a_+^{(k)}$ for any $k \geq 1$.*

Proof (Proof by Induction)

Base case: The statement of the lemma holds for $k = 1$, since

$$a^{(1)} = a_+^{(1)} = a.$$

Inductive step: Assume that the statement of the lemma holds for any $k \leq (p-1)$ and show that it also holds for $k = p$ where $p \geq 2$. For any $i \geq 2$

$$a^{(i)} = a^{(i-1)} \cup (a^{(i-1)} \times a^{(i-1)}) \Rightarrow a^{(i)} \succeq a^{(i-1)}.$$

Hence, by the inductive hypothesis, for any $i \leq (p-1)$

$$a^{(p-1)} \succeq a^{(i)} \succeq a_+^{(i)}.$$

Let $1 \leq j \leq (p-1)$. The following holds

$$(a^{(p-1)} \times a^{(p-1)}) \succeq (a_+^{(j)} \times a_+^{(p-j)}),$$

since $a^{(p-1)} \succeq a_+^{(j)}$ and $a^{(p-1)} \succeq a_+^{(p-j)}$. By the definition,

$$a_+^{(p)} = \bigcup_{j=1}^{p-1} a_+^{(j)} \times a_+^{(p-j)}$$

and from this it follows that

$$(a^{(p-1)} \times a^{(p-1)}) \succeq a_+^{(p)}.$$

By the definition,

$$a^{(p)} = a^{(p-1)} \cup (a^{(p-1)} \times a^{(p-1)}) \Rightarrow a^{(p)} \succeq (a^{(p-1)} \times a^{(p-1)}) \succeq a_+^{(p)}$$

and this completes the proof of the lemma.

Lemma 2 *Let $G = (N, \Sigma, P)$ be a grammar, let a be a square matrix. Then for any $k \geq 1$ there is $j \geq 1$, such that $(\bigcup_{i=1}^{j} a_+^{(i)}) \succeq a^{(k)}$.*

Proof (Proof by Induction)

Base case: For $k = 1$ there is $j = 1$, such that

$$a_+^{(1)} = a^{(1)} = a.$$

Thus, the statement of the lemma holds for $k = 1$.

Inductive step: Assume that the statement of the lemma holds for any $k \leq (p-1)$ and show that it also holds for $k = p$ where $p \geq 2$. By the inductive hypothesis, there is $j \geq 1$, such that

$$(\bigcup_{i=1}^{j} a_+^{(i)}) \succeq a^{(p-1)}.$$

By the definition,

$$a_+^{(2j)} = \bigcup_{i=1}^{2j-1} a_+^{(i)} \times a_+^{(2j-i)}$$

and from this it follows that

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \succeq (\bigcup_{i=1}^{j} a_+^{(i)}) \times (\bigcup_{i=1}^{j} a_+^{(i)}) \succeq (a^{(p-1)} \times a^{(p-1)}).$$

The following holds

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \succeq a^{(p)} = a^{(p-1)} \cup (a^{(p-1)} \times a^{(p-1)}),$$

since

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \succeq (\bigcup_{i=1}^{j} a_+^{(i)}) \succeq a^{(p-1)}$$

and

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \succeq (a^{(p-1)} \times a^{(p-1)}).$$

Therefore there is $2j$, such that

$$(\bigcup_{i=1}^{2j} a_+^{(i)}) \succeq a^{(p)}$$

and this completes the proof of the lemma.

Theorem 1 *Let $G = (N, \Sigma, P)$ be a grammar, let a be a square matrix. Then $a_+ = a^{cf}$.*

Proof By the lemma 1, for any $k \geq 1$, $a^{(k)} \succeq a_+^{(k)}$. Therefore

$$a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots \succeq a_+^{(1)} \cup a_+^{(2)} \cup \cdots = a_+.$$

By the lemma 2, for any $k \geq 1$ there is $j \geq 1$, such that

$$(\bigcup_{i=1}^{j} a_+^{(i)}) \succeq a^{(k)}.$$

Hence

$$a_+ = (\bigcup_{i=1}^{\infty} a_+^{(i)}) \succeq a^{(k)},$$

for any $k \geq 1$. Therefore

$$a_+ \succeq a^{(1)} \cup a^{(2)} \cup \cdots = a^{cf}.$$

Since $a^{cf} \succeq a_+$ and $a_+ \succeq a^{cf}$,

$$a_+ = a^{cf}$$

and this completes the proof of the theorem.

Furthermore, in this paper we use the transitive closure a^{cf} instead of a_+, and the algorithm for computing a^{cf} also computes Valiant's transitive closure a_+.

3.2 GPU

4 Related Work

Traditionally, query languages for graph databases use regular expressions to describe paths to find [6,7,8,9,10], but there are some other useful queries, which cannot be expressed by regular expressions. For example, there are classical *same-generation queries* [1], which can be used for finding all the nodes at the same level in some hierarchy, and are useful for discovering vertex similarity. The context-free path querying algorithms can be used to evaluate such types of queries since this queries can be represented by context-free grammars.

In [19], the comparative experimental investigation of several state of the art context-free path query evaluation methods w.r.t. relational query semantics is provided.

There are a number of solutions [3,2,11] for context-free path query evaluation w.r.t. relational query semantics, which make use of such parsing algorithms as CYK [12,13] or Earley [14].

Hellings [3] presented an algorithm for context-free path query evaluation using relational query semantics. According to Hellings, for a given graph

$D = (V, E)$ and a grammar $G = (N, \Sigma, P)$ the context-free path query evaluation w.r.t. relational query semantics reduces to a calculation of a set of context-free relations R_A. Thus, in this paper, we focus on the calculation of these context-free relations. The worst-case time complexity of this algorithm is $O(|N||E| + (|N||V|)^3)$. Additionally, the algorithm of Hellings was implemented [11] in the context of RDF processing.

In [15], a bottom-up algorithm for a context-free path querying over graph databases is presented. This algorithm is inspired by the LR [16] parsing technique and uses a variant of the GSS introduced in [17] to encompass several derivations at a time. Also, this algorithm was implemented in the context of RDF processing.

An algorithm for the context-free path query evaluation and its implementation over RDF graph databases are proposed in [18]. This algorithm is inspired by the LL [14] parsing technique, has $O(|V|^3|P|)$ worst-case time complexity, and has $O(|V|^2|N|)$ worst-case space complexity, where V is the set of vertices, N is a set of non-terminal symbols, and P is a set of production (or derivation) rules of the context-free grammar.

In [20] an algorithm for context-free path query evaluation with Dyck and semi-Dyck grammars is presented. Dyck and semi-Dyck context-free languages are important due to the close relationship between transitive closure, Boolean and algebraic matrix multiplication, and context-free grammar recognition. This algorithm has $O(|V|^{\omega} log^3(|V|))$ time complexity where ω is the best exponent for matrix multiplication.

Given any two vertices s and t, and the output of Nykänen and Ukkonen's [21] exact integer path length algorithm that costs $O(n^3)$. An algorithm in [22] gives a minimal-cost point-to-point Dyck shortest path result in $O(n^2 log(n))$ additional operations. This paper assumes the graphs have no cycles and the Dyck languages have a single parenthesis type.

Firstly, the paper [23] addresses the problem of context-free path querying w.r.t. relation semantics with Dyck languages on bidirected graphs. Given a bidirected graph with $|V|$ nodes and $|E|$ edges: (i) an algorithm with worst-case running time $O(|E|+|V|\cdot\alpha(|V|))$ is presented, where $\alpha(|V|)$ is the inverse Ackermann function; (ii) provided a matching lower bound that shows that this algorithm is optimal w.r.t. worst-case complexity; and (iii) presented an optimal average-case upper bound of $O(|E|)$ time. Secondly, presented the proof that combinatorial algorithms for context-free path querying with Dyck languages on general graphs with truly sub-cubic bounds cannot be obtained without obtaining sub-cubic combinatorial algorithms for Boolean Matrix Multiplication, which is a long-standing open problem. This means that the existing combinatorial algorithms for context-free path querying with Dyck languages are (conditionally) optimal for general graphs.

Other examples of path query semantics are *single-path* and *all-path query semantics* [24]. The all-path query semantics requires a finding of all possible paths from a node m to a node n whose labelings are derived from a non-terminal A. The single-path query semantics requires presenting only one such path (usually, the shortest one).

Hellings [24] presented some algorithms for context-free path query evaluation using single-path and all-path query semantics. If a context-free path query w.r.t. all-path query semantics is evaluated for cyclic graphs, then the query result can be an infinite set of paths. For this reason, in [24] annotated grammars were proposed as a way to represent the results.

In [25] a distributed context-free path query evaluation algorithm w.r.t. the single-path query semantics is proposed. This algorithm provides a shortest path between each pair of nodes according to the context-free grammar and a weight function on graph edges. This algorithm can be efficiently distributed on up to $O(|V||N|)$ compute nodes.

An algorithm for context-free path query evaluation w.r.t. all-path query semantics is proposed in [26]. This algorithm is based on the generalized top-down parsing algorithm (GLL) [27]. For the result representation, this solution uses derivation trees, which is more native for grammar-based analysis. The worst-case time complexity of this algorithm is $O(|V|^3 max_{v \in V}(deg^+(v)))$, where $deg^+(v)$ is the outdegree of vertex v. For complete graphs, the time complexity of this algorithm can reach $O(|V|^4)$.

The algorithms [26, 24] for context-free path query evaluation w.r.t. single-path and all-path query semantics can also be used for query evaluation using relational semantics.

Our work is inspired by Valiant [5], who proposed an algorithm for general context-free recognition in less than cubic time. This algorithm computes the same parsing table as CYK algorithm but does this by offloading the most intensive computations into calls to the Boolean matrix multiplication procedure. This approach not only provides an asymptotically more efficient algorithm but also allows us to effectively apply GPGPU computing techniques. Valiant's algorithm computes the transitive closure a^+ of a square upper-triangular matrix a. Valiant also has shown, that the matrix multiplication operation ($\times$) is essentially the same as $|N|^2$ Boolean matrix multiplications, where $|N|$ is the number of non-terminals in the given context-free grammar in Chomsky normal form.

Yannakakis [28] analyzed the reducibility of various path querying problems to the calculation of transitive closure. He formulated a problem of Valiant's technique generalization for the context-free path query evaluation w.r.t. relational query semantics. Also, he conjectured, that this technique cannot be generalized for arbitrary graphs, though it does for acyclic graphs.

5 Context-free Path Querying by Transitive Closure Calculation

In this section, we show, how the context-free path query evaluation using relational query semantics can be reduced to the calculation of matrix transitive closure a^{cf}, prove the correctness of this reduction, introduce an algorithm for computing the transitive closure a^{cf}, and provide a step-by-step demonstration of this algorithm on a small example.

5.1 Reducing Context-Free Path Querying to the Calculation of Transitive Closure

In this section, we show, how the context-free relations R_A can be calculated by computing the transitive closure a^{cf}.

Let $G = (N, \Sigma, P)$ be a grammar and $D = (V, E)$ be a graph. We enumerate the nodes of the graph D from 0 to $(|V| - 1)$. We initialize the elements of the $|V| \times |V|$ matrix a with $\varnothing$. Further, for every i and j we set

$$a_{i,j} = \{A_k \mid ((i, x, j) \in E) \wedge ((A_k \to x) \in P)\}.$$

Finally, we compute the transitive closure

$$a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots$$

where

$$a^{(i)} = a^{(i-1)} \cup (a^{(i-1)} \times a^{(i-1)}),$$

for $i \geq 2$ and $a^{(1)} = a$. For the transitive closure a^{cf}, the following statements hold.

Lemma 3 *Let $D = (V, E)$ be a graph, let $G = (N, \Sigma, P)$ be a grammar. Then for any i, j and for any non-terminal $A \in N$, $A \in a^{(k)}_{i,j}$ iff $(i, j) \in R_A$ and $i\pi j$, such that there is a derivation tree of the height $h \leq k$ for the string $l(\pi)$ and a context-free grammar $G_A = (N, \Sigma, P, A)$.*

Proof (Proof by Induction)

Base case: Show that the lemma holds for $k = 1$. For any i, j and for any non-terminal $A \in N$, $A \in a^{(1)}_{i,j}$ iff there is $i\pi j$ that consists of a unique edge e from the node i to the node j and $(A \to x) \in P$, where $x = l(\pi)$. Therefore $(i, j) \in R_A$ and there is a derivation tree of the height $h = 1$, shown on Figure 2, for the string x and a context-free grammar $G_A = (N, \Sigma, P, A)$. Thus, it has been shown that the lemma holds for $k = 1$.

Fig. 2: The derivation tree of the height $h = 1$ for the string $x = l(\pi)$.

Inductive step: Assume that the lemma holds for any $k \leq (p - 1)$ and show that it also holds for $k = p$, where $p \geq 2$. For any i, j and for any non-terminal $A \in N$,

$$A \in a^{(p)}_{i,j} \text{ iff } A \in a^{(p-1)}_{i,j} \text{ or } A \in (a^{(p-1)} \times a^{(p-1)})_{i,j},$$

since

$$a^{(p)} = a^{(p-1)} \cup (a^{(p-1)} \times a^{(p-1)}).$$

Let $A \in a_{i,j}^{(p-1)}$. By the inductive hypothesis, $A \in a_{i,j}^{(p-1)}$ iff $(i,j) \in R_A$ and there exists $i\pi j$, such that there is a derivation tree of the height $h \leq (p-1)$ for the string $l(\pi)$ and a context-free grammar $G_A = (N, \Sigma, P, A)$. The statement of the lemma holds for $k = p$ since the height h of this tree is also less than or equal to p.

Let $A \in (a^{(p-1)} \times a^{(p-1)})_{i,j}$. By the definition of the binary operation $(\cdot)$ on arbitrary subsets, $A \in (a^{(p-1)} \times a^{(p-1)})_{i,j}$ iff there are r, $B \in a_{i,r}^{(p-1)}$ and $C \in a_{r,j}^{(p-1)}$, such that $(A \to BC) \in P$. Hence, by the inductive hypothesis, there are $i\pi_1 r$ and $r\pi_2 j$, such that $(i,r) \in R_B$ and $(r,j) \in R_C$, and there are the derivation trees T_B and T_C of heights $h_1 \leq (p-1)$ and $h_2 \leq (p-1)$ for the strings $w_1 = l(\pi_1)$, $w_2 = l(\pi_2)$ and the context-free grammars G_B, G_C respectively. Thus, the concatenation of paths π_1 and π_2 is $i\pi j$, where $(i,j) \in R_A$ and there is a derivation tree of the height $h = 1 + max(h_1, h_2)$, shown on Figure 3, for the string $w = l(\pi)$ and a context-free grammar G_A.

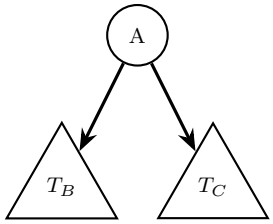

Fig. 3: The derivation tree of the height $h = 1 + max(h_1, h_2)$ for the string $w = l(\pi)$, where T_B and T_C are the derivation trees for strings w_1 and w_2 respectively.

The statement of the lemma holds for $k = p$ since the height $h = 1 + max(h_1, h_2) \leq p$. This completes the proof of the lemma.

Theorem 2 *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a grammar. Then for any i, j and for any non-terminal $A \in N$, $A \in a_{i,j}^{cf}$ iff $(i,j) \in R_A$.*

Proof Since the matrix $a^{cf} = a^{(1)} \cup a^{(2)} \cup \cdots$, for any i, j and for any non-terminal $A \in N$, $A \in a_{i,j}^{cf}$ iff there is $k \geq 1$, such that $A \in a_{i,j}^{(k)}$. By the lemma 3, $A \in a_{i,j}^{(k)}$ iff $(i,j) \in R_A$ and there is $i\pi j$, such that there is a derivation tree of the height $h \leq k$ for the string $l(\pi)$ and a context-free grammar $G_A = (N, \Sigma, P, A)$. This completes the proof of the theorem.

We can, therefore, determine whether $(i,j) \in R_A$ by asking whether $A \in a_{i,j}^{cf}$. Thus, we show how the context-free relations R_A can be calculated by computing the transitive closure a^{cf} of the matrix a.

5.2 The algorithm

In this section, we introduce an algorithm for calculating the transitive closure a^{cf} which was discussed in section 5.1.

Let $D = (V, E)$ be the input graph and $G = (N, \Sigma, P)$ be the input grammar.

Algorithm 1 Context-free recognizer for graphs

```
1:  function contextFreePathQuerying(D, G)
2:    n ← the number of nodes in D
3:    E ← the directed edge-relation from D
4:    P ← the set of production rules in G
5:    T ← the matrix n × n in which each element is ∅
6:    for all (i, x, j) ∈ E do                          ▷ Matrix initialization
7:      T_{i,j} ← T_{i,j} ∪ {A | (A → x) ∈ P}
8:    while matrix T is changing do
9:      T ← T ∪ (T × T)                                 ▷ Transitive closure T^{cf} calculation
10:   return T
```

Note, the matrix initialization in lines **6-7** of the Algorithm 1 can handle arbitrary graph D. For example, if a graph D contains multiple edges (i, x_1, j) and (i, x_2, j) then both the elements of the set $\{A_1 \mid (A_1 \to x_1) \in P\}$ and the elements of the set $\{A_2 \mid (A_2 \to x_2) \in P\}$ will be added to $T_{i,j}$.

We need to show that the Algorithm 1 terminates in a finite number of steps. Since each element of the matrix T contains no more than $|N|$ non-terminals, the total number of non-terminals in the matrix T does not exceed $|V|^2|N|$. Therefore, the following theorem holds.

Theorem 3 *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a grammar. The Algorithm 1 terminates in a finite number of steps.*

Proof It is sufficient to show, that the operation in the line **9** of the Algorithm 1 changes the matrix T only finite number of times. Since this operation can only add non-terminals to some elements of the matrix T, but not remove them, it can change the matrix T no more than $|V|^2|N|$ times.

Denote the number of elementary operations executed by the algorithm of multiplying two $n \times n$ Boolean matrices as $BMM(n)$. According to Valiant, the matrix multiplication operation in the line **9** of the Algorithm 1 can be calculated in $O(|N|^2 BMM(n)(|V|))$. Denote the number of elementary operations, executed by the matrix union operation of two $n \times n$ Boolean matrices as $BMU(n)$. Similarly, it can be shown that the matrix union operation in the line **9** of the Algorithm 1 can be calculated in $O(|N|^2 BMU(n))$. Since the line **9** of the Algorithm 1 is executed no more than $|V|^2|N|$ times, the following theorem holds.

Theorem 4 *Let $D = (V, E)$ be a graph and let $G = (N, \Sigma, P)$ be a grammar. Then the Algorithm 1 calculates the transitive closure T^{cf} in $O(|V|^2|N|^3(BMM(|V|) + BMU(|V|)))$.*

We also find the worst-case example taken from [24], for which the time complexity in terms of the graph size provided by Theorem 4 cannot be improved. This example is based on the context-free grammar $G = (N, \Sigma, P)$ where:

- the set of non-terminals $N = \{S\}$;
- the set of terminals $\Sigma = \{a, b\}$;
- the set of production rules P is presented on Figure 4.

$$0 : S \to a\ S\ b$$
$$1 : S \to a\ b$$

Fig. 4: Production rules for the worst-case example.

Let the size $|N|$ of the grammar G be a constant. The worst-case time complexity is reached by running this query on the double-cyclic graph where:

- one of the cycles having $u = 2^k + 1$ edges labeled with a;
- another cycle having $v = 2^k$ edges labeled with b;
- the two cycles are connected via a shared node m.

A small example of such graph with $k = 1$, $u = 3$, $v = 2$, and $m = 0$ is presented on Figure 1.

The shortest path π from the node m to the node m, whose labeling forms a string from the language $L(G, S) = \{a^n b^n | n \geq 1\}$, has a length $l = 2 * u * v$, since $u = 2^k + 1$ and $v = 2^k$ are coprime, and string s, formed by this path, consists of $u * v$ labels a and $u * v$ labels b. The string $s = l(\pi)$ has a derivation tree, according to a context-free grammar G and starting nonterminal S, of the minimal height $h = 2 * u * v$ among all the paths from the node m to the node m in this double-cyclic graph. Therefore, if we run the worst-case example query on this graph, then the operation in the line **9** of the Algorithm 1 changes the matrix T at least $h = 2 * u * v$ times. Hence, the Algorithm 1 computes this query in $O(|V|^2(BMM(|V|) + BMU(|V|)))$, since $|V| = (u + v - 1) = 2 * v$ and $h = 2 * u * v > 2 * v * v = |V|^2/4 = O(|V|^2)$.

5.3 An Example

In this section, we provide a step-by-step demonstration of the proposed algorithm. For this, we consider the example with the worst-case time complexity.

The **example query** is based on the context-free grammar $G = (N, \Sigma, P)$ of the worst-case example query which was discussed in section 5.2. The set of production rules for this grammar is shown on Figure 4.

Since the proposed algorithm processes only grammars in Chomsky normal form, we first transform the grammar G into an equivalent grammar $G' = (N', \Sigma', P')$ in normal form, where:

- the set of non-terminals $N' = \{S, S_1, A, B\}$;
- the set of terminals $\Sigma' = \{a, b\}$;
- the set of production rules P' is presented on Figure 5.

$$
\begin{array}{l}
0: \; S \to A \; B \\
1: \; S \to A \; S_1 \\
2: \; S_1 \to S \; B \\
3: \; A \to a \\
4: \; B \to b
\end{array}
$$

Fig. 5: Production rules for the example query grammar in normal form.

We run the query on a graph, presented on Figure 1. We provide a step-by-step demonstration of the work with the given graph D and grammar G' of the Algorithm 1. After the matrix initialization in lines **6-7** of the Algorithm 1, we have a matrix T_0, presented on Figure 6.

$$
T_0 = \begin{pmatrix}
\varnothing & \{A\} & \varnothing & \{B\} \\
\varnothing & \varnothing & \{A\} & \varnothing \\
\{A\} & \varnothing & \varnothing & \varnothing \\
\{B\} & \varnothing & \varnothing & \varnothing
\end{pmatrix}
$$

Fig. 6: The initial matrix for the example query.

Let T_i be the matrix T, obtained after executing the loop in the lines **8-9** of the Algorithm 1 i times. The calculation of the matrix T_1 is shown on Figure 7.

$$
T_0 \times T_0 = \begin{pmatrix}
\varnothing & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \varnothing \\
\varnothing & \varnothing & \varnothing & \{S\} \\
\varnothing & \varnothing & \varnothing & \varnothing
\end{pmatrix}
$$

$$
T_1 = T_0 \cup (T_0 \times T_0) = \begin{pmatrix}
\varnothing & \{A\} & \varnothing & \{B\} \\
\varnothing & \varnothing & \{A\} & \varnothing \\
\{A\} & \varnothing & \varnothing & \{\boldsymbol{S}\} \\
\{B\} & \varnothing & \varnothing & \varnothing
\end{pmatrix}
$$

Fig. 7: The first iteration of computing the transitive closure for the example query.

When the algorithm at some iteration finds new paths in the graph D, then it adds corresponding nonterminals to the matrix T. For example, after

the first loop iteration, non-terminal S is added to the matrix T. This non-terminal is added to the element with a row index $i = 2$ and a column index $j = 3$. This means, that there is $i\pi j$ (a path π from the node 2 to the node 3), such that $S \xrightarrow{*} l(\pi)$. For example, such a path consists of two edges with labels a and b, and $S \xrightarrow{0} A\ B \xrightarrow{3} a\ B \xrightarrow{4} a\ b$.

The calculation of the transitive closure is completed after k iterations, when a fixpoint is reached: $T_{k-1} = T_k$. For the example query, $k = 13$ since $T_{13} = T_{12}$. The remaining iterations of computing the transitive closure are presented on Figure 8 (new matrix elements on each iteration are shown in bold).

$$T_2 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \{A, \boldsymbol{S_1}\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_3 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \{\boldsymbol{S}\} & \varnothing & \{A\} & \varnothing \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_4 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \{S\} & \varnothing & \{A\} & \{\boldsymbol{S_1}\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_5 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B, \boldsymbol{S}\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_6 = \begin{pmatrix} \{\boldsymbol{S_1}\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_7 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1, \boldsymbol{S}\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_8 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, \boldsymbol{S_1}\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_9 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1, \boldsymbol{S}\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_{10} = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S, \boldsymbol{S_1}\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_{11} = \begin{pmatrix} \{S_1, \boldsymbol{S}\} & \{A\} & \varnothing & \{B, S\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_{12} = \begin{pmatrix} \{S_1, S\} & \{A\} & \varnothing & \{B, S, \boldsymbol{S_1}\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_{13} = \begin{pmatrix} \{S_1, S\} & \{A\} & \varnothing & \{B, S, S_1\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Fig. 8: Remaining states of the matrix T.

Thus, the result of the Algorithm 1 for the example query is the matrix $T_{13} = T_{12}$. Now, after constructing the transitive closure, we can construct the context-free relations R_A. These relations for each non-terminal of the grammar G' are presented on Figure 9. The example graph with the result of the context-free path querying is presented on Figure 10.

In the context-free relation R_S, we have all node pairs corresponding to paths, whose labeling is in the language $L(G, S) = \{a^n b^n | n \geq 1\}$. This conclusion is based on the fact, that the grammar G' with the starting nonterminal

$$
\begin{aligned}
R_S &= \{(0,0),(0,3),(1,0),(1,3),(2,0),(2,3)\},\\
R_{S_1} &= \{(0,0),(0,3),(1,0),(1,3),(2,0),(2,3)\},\\
R_A &= \{(0,1),(1,2),(2,0)\},\\
R_B &= \{(0,3),(3,0)\}.
\end{aligned}
$$

Fig. 9: Context-free relations for the example query.

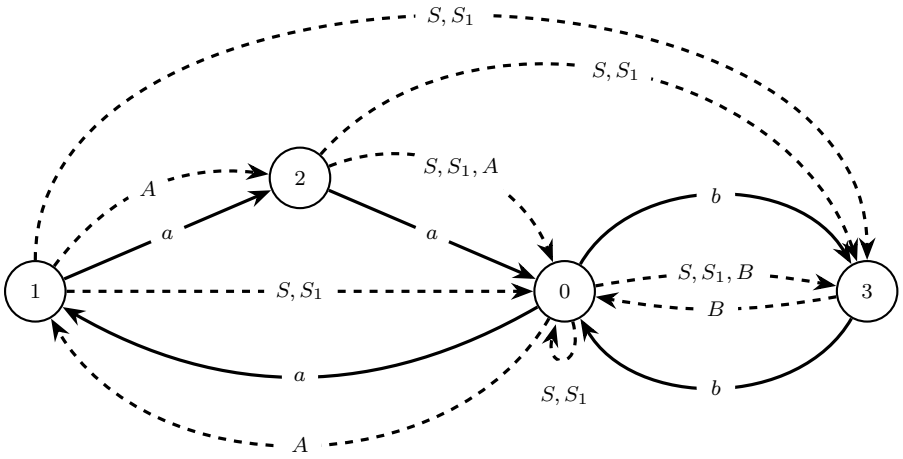

Fig. 10: The example graph with the result of CFPQ.

S is equivalent to the grammar G with the same starting nonterminal and $L(G', S) = L(G, S)$.

6 Evaluation

Since our algorithm works with graphs, each RDF file from a dataset was converted to an edge-labeled directed graph as follows. For each triple (o, p, s) from an RDF file, we added an edge (o, p, s) to the graph. In addition, we added an edge (s, p^{-1}, o) to the graph, if p corresponded to the terminals *subClassOf* and *type* of the query grammars.

All tests were run on a PC with the following characteristics:

- OS: Microsoft Windows 10 Pro
- System Type: x64-based PC
- CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Core(s), 4 Logical Processor(s)

- RAM: 16 GB
- GPU: NVIDIA GeForce GTX 1070
  - CUDA Cores: 1920
  - Core clock: 1556 MHz
  - Memory data rate: 8008 MHz
  - Memory interface: 256-bit
  - Memory bandwidth: 256.26 GB/s
  - Dedicated video memory: 8192 MB GDDR5

We evaluated two classical *same-generation queries* [1] which, for example, can be used in bioinformatics.

Query 1 is based on the grammar G_1 for retrieving the concepts on the same layer, where:

- the grammar $G_1 = (N_1, \Sigma_1, P_1)$;
- the set of non-terminals $N_1 = \{S, S_1, S_2, S_3, S_4, S_5, S_6\}$;
- the set of terminals

$$\Sigma_1 = \{subClassOf, subClassOf^{-1}, type, type^{-1}\}$$

- the set of production rules P_1 is presented on Figure 11.

$$
\begin{aligned}
&0: \ S \rightarrow S_1 \ S_5\\
&1: \ S \rightarrow S_3 \ S_6\\
&2: \ S \rightarrow S_1 \ S_2\\
&3: \ S \rightarrow S_3 \ S_4\\
&4: S_5 \rightarrow S \ S_2\\
&5: S_6 \rightarrow S \ S_4\\
&6: S_1 \rightarrow subClassOf^{-1}\\
&7: S_2 \rightarrow subClassOf\\
&8: S_3 \rightarrow type^{-1}\\
&9: S_4 \rightarrow type
\end{aligned}
$$

Fig. 11: Production rules for the Query 1 grammar.

Table 1: Initial graph characteristics for Query 1

Ontology	E	$\#S$	$\#S_1$	$\#S_2$	$\#S_3$	$\#S_4$	$\#S_5$	$\#S_6$
funding	1480	0	0	0	0	0	0	0
wine	2450	0	0	0	0	0	0	0
pizza	2604	0	0	0	0	0	0	0
core	4342	0	0	0	0	0	0	0
pathways	18598	0	0	0	0	0	0	0
enzyme	109695	0	0	0	0	0	0	0
eclass_514en	523727	0	0	0	0	0	0	0
go	534311	0	0	0	0	0	0	0
go-hierarchy	980218	0	0	0	0	0	0	0

Table 2: Evaluation characteristics for Query 1

Ontology	time mean (ms)	time std (ms)	iterations
funding	0.0	0.0	8
wine	3.0	0.0	10
pizza	1.0	0.0	8
core	0.0	0.0	16
pathways	1.0	0.0	10
enzyme	4.0	0.0	10
eclass_514en	50.0	0.0	10
go	1063.0	0.0	42
go-hierarchy	198.0	0.0	8

Table 3: Graph characteristics after CFPQ evaluation for Query 1

Ontology	$\#S$	$\#S_1$	$\#S_2$	$\#S_3$	$\#S_4$	$\#S_5$	$\#S_6$
funding	17634	90	90	304	304	6555	2375
wine	66572	126	126	485	485	8172	16261
pizza	56195	259	259	365	365	23044	720
core	316	178	178	706	1412	82	239
pathways	884	3117	3117	3118	3118	882	882
enzyme	396	8163	8163	14989	14989	393	393
eclass_514en	90994	90962	90962	72517	72517	35505	30330
go	304070	90512	90512	58483	58483	278610	39642
go-hierarchy	588976	490109	490109	0	0	324016	0

Query 2 is based on the grammar G_2 for retrieving concepts on the adjacent layers, where:

- the grammar $G_2 = (N_2, \Sigma_2, P_2)$;
- the set of non-terminals $N_2 = \{S, S_1, S_2, S_3\}$;
- the set of terminals $\Sigma_2 = \{subClassOf, subClassOf^{-1}\}$;
- the set of production rules P_2 is presented on Figure 12.

$$
\begin{aligned}
&0: \ S \ \rightarrow S_1 \ S_3 \\
&1: \ S \ \rightarrow subClassOf \\
&2: S_3 \rightarrow S \ S_2 \\
&3: S_1 \rightarrow subClassOf^{-1} \\
&4: S_2 \rightarrow subClassOf
\end{aligned}
$$

Fig. 12: Production rules for the Query 2 grammar.

Query 3 is based on the grammar G_3 for retrieving ...TODO..., where:

- the grammar $G_3 = (N_3, \Sigma_3, P_3)$;
- the set of non-terminals $N_3 = \{S, S_1, BT, BTR\}$;
- the set of terminals $\Sigma_3 = \{bt, bt^{-1}\}$;
- the set of production rules P_3 is presented on Figure 13.

Table 4: Initial graph characteristics for Query 2

Ontology	E	$\#S$	$\#S_1$	$\#S_2$	$\#S_3$
funding	1480	0	0	0	0
wine	2450	0	0	0	0
pizza	2604	0	0	0	0
core	4342	0	0	0	0
pathways	18598	0	0	0	0
enzyme	109695	0	0	0	0
eclass_514en	523727	0	0	0	0
go	534311	0	0	0	0
go-hierarchy	980218	0	0	0	0

Table 5: Evaluation characteristics for Query 2

Ontology	time mean (ms)	time std (ms)	iterations
funding	0.0	0.0	7
wine	3.0	0.0	3
pizza	0.0	0.0	8
core	0.0	0.0	4
pathways	1.0	0.0	2
enzyme	3.0	0.0	2
eclass_514en	53.0	0.0	4
go	594.0	0.0	23
go-hierarchy	281.0	0.0	4

Table 6: Graph characteristics after CFPQ evaluation for Query 2

Ontology	$\#S$	$\#S_1$	$\#S_2$	$\#S_3$
funding	1158	90	90	566
wine	133	126	126	35
pizza	1262	259	259	384
core	214	178	178	108
pathways	3117	3117	3117	3010
enzyme	8163	8163	8163	8156
eclass_514en	96163	90962	90962	60633
go	334850	90512	90512	327628
go-hierarchy	738937	490109	490109	422848

$$
\begin{aligned}
0: \quad & S \quad \to BT\ S_1 \\
1: \quad & S_1 \quad \to S\ BTR \\
2: \quad & S \quad \to BT\ BTR \\
3: \quad & BT \quad \to bt \\
4: \quad & BTR \to bt^{-1}
\end{aligned}
$$

Fig. 13: Production rules for the Query 3 grammar.

Table 7: Initial graph characteristics for Query 3

Ontology	E	$\#S$	$\#S_1$	$\#BT$	$\#BTR$
geospeices	2311461	0	0	0	0

Table 8: Evaluation characteristics for Query 2

Ontology	time mean (ms)	time std (ms)	iterations
geospeices	13980.0	0.0	11

Table 9: Graph characteristics after CFPQ evaluation for Query 2

Ontology	$\#S$	$\#S_1$	$\#BT$	$\#BTR$
geospeices	226669749	21361542	20867	20867

7 Conclusion

References

1. S. Abiteboul, R. Hull, V. Vianu, *Foundations of databases: the logical level* (Addison-Wesley Longman Publishing Co., Inc., 1995)
2. P. Sevon, L. Eronen, Journal of Integrative Bioinformatics **5**(2), 100 (2008)
3. J. Hellings, in *Proceedings of ICDT'14* (2014), pp. 119–130
4. N. Chomsky, Information and control **2**(2), 137 (1959)
5. L.G. Valiant, Journal of computer and system sciences **10**(2), 308 (1975)
6. J.L. Reutter, M. Romero, M.Y. Vardi, Theory of Computing Systems **61**(1), 31 (2017)
7. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on* (IEEE, 2011), pp. 39–50
8. S. Abiteboul, V. Vianu, in *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (ACM, 1997), pp. 122–133
9. M. Nolé, C. Sartiani, in *Proceedings of the 28th International Conference on Scientific and Statistical Database Management* (ACM, 2016), p. 13
10. A. Mendelzon, P. Wood, SIAM J. Computing **24**(6), 1235 (1995)
11. X. Zhang, Z. Feng, X. Wang, G. Rao, W. Wu, in *International Semantic Web Conference* (Springer, 2016), pp. 632–648
12. T. Kasami, An efficient recognition and syntaxanalysis algorithm for context-free languages. Tech. rep., DTIC Document (1965)
13. D.H. Younger, Information and control **10**(2), 189 (1967)
14. D. Grune, C.J.H. Jacobs, *Parsing Techniques (Monographs in Computer Science)* (Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006)
15. F.C. Santos, U.S. Costa, M.A. Musicante, in *International Conference on Web Engineering* (Springer, 2018), pp. 225–233
16. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. Compilers: Principles, techniques, and tools second edition (2007)
17. M. Tomita, Computational linguistics **13**(1-2), 31 (1987)
18. C.M. Medeiros, M.A. Musicante, U.S. Costa, Journal of Computer Languages **51**, 75 (2019)
19. J. Kuijpers, G. Fletcher, N. Yakovets, T. Lindaaker, in *Proceedings of the 31st International Conference on Scientific and Statistical Database Management* (ACM, 2019), pp. 121–132
20. P.G. Bradford, in *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)* (IEEE, 2017), pp. 247–253
21. M. Nykänen, E. Ukkonen, Journal of Algorithms **42**(1), 41 (2002)

22. P.G. Bradford, V. Choppella, in *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)* (IEEE, 2016), pp. 1–7
23. K. Chatterjee, B. Choudhary, A. Pavlogiannis, Proceedings of the ACM on Programming Languages **2**(POPL), 30 (2017)
24. J. Hellings, arXiv preprint arXiv:1502.02242 (2015)
25. C.B. Ward, N.M. Wiegand, P.G. Bradford, in *2008 37th International Conference on Parallel Processing* (IEEE, 2008), pp. 373–380
26. S. Grigorev, A. Ragozina, in *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia* (ACM, 2017), p. 10
27. E. Scott, A. Johnstone, Electronic Notes in Theoretical Computer Science **253**(7), 177 (2010)
28. M. Yannakakis, in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (ACM, 1990), pp. 230–242