Cross-Model Conjunctive Queries over Relation and Tree-structured Data

Chen, Yuxing

Springer
2022-04-08

Chen , Y , Uotila , V , Lu , J , Liu , Z H & Das , S 2022 , Cross-Model Conjunctive Queries over Relation and Tree-structured Data . in Database Systems for Advanced Applications. DASFAA 2022. . Lecture Notes in Computer Science , vol. 13245 , Springer , Cham , International Conference on Database Systems for Advanced Applications , Hyderabad , India , 11/04/2022 . https://doi.org/10.1007/978-3-031-00123-9_2

http://hdl.handle.net/10138/345969
https://doi.org/10.1007/978-3-031-00123-9_2

unspecified
acceptedVersion

Cross-Model Conjunctive Queries over Relation and Tree-structured Data

Yuxing Chen[1], Valter Uotila[1], Jiaheng Lu[1], Zhen Hua Liu[2], and Souripriya Das[2]

[1] University of Helsinki
{yuxing.chen,valter.uotila,jiaheng.lu}@helsinki.fi
[2] Oracle
{zhen.liu,souripriya.das}@oracle.com

Abstract. Conjunctive queries are the most basic and central class of database queries. With the continued growth of demands to manage and process the massive volume of different types of data, there is little research to study the conjunctive queries between relation and tree data. In this paper, we study of Cross-Model Conjunctive Queries (CMCQs) over relation and tree-structured data (XML and JSON). To efficiently process CMCQs with bounded intermediate results, we first encode tree nodes with position information. With tree node original label values and encoded position values, it allows our proposed algorithm *CMJoin* to join relations and tree data simultaneously, avoiding massive intermediate results. *CMJoin* achieves worst-case optimality in terms of the total result of label values and encoded position values. Experimental results demonstrate the efficiency and scalability of the proposed techniques to answer a CMCQ in terms of running time and intermediate result size.

Keywords: Cross-model join · Worst-case optimal · Relation and tree data.

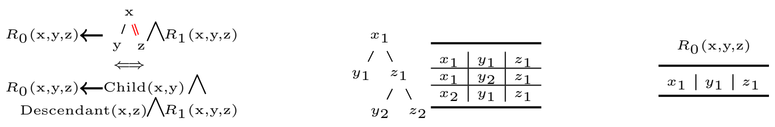
1 Introduction

Conjunctive queries are the most fundamental and widely used database queries [2]. They correspond to `project-select-join` queries in the relational algebra. They also correspond to non-recursive datalog rules [7]

$$R_0(u_0) \leftarrow R_1(u_1) \wedge R_2(u_2) \wedge \ldots \wedge R_n(u_n), \tag{1}$$

where R_i is a relation name of the underlying database, R_0 is the output relation, and each argument u_i is a list of $|u_i|$ variables, where $|u_i|$ is the arity of the corresponding relation. The same variable can occur multiple times in one or more argument lists.

It turns out that traditional database engines are not optimal to answer conjunctive queries, as all pair-join engines may produce unnecessary intermediate results on many join queries [26]. For example, consider a typical triangle conjunctive query $R_0(a, b, c) \leftarrow R_1(a, b) \wedge R_2(b, c) \wedge R_3(a, c)$, where the size of


R_0(x,y,z)← x / \\ y z ⋀R_1(x,y,z)
⟺
R_0(x,y,z)←Child(x,y) ⋀
Descendant(x,z)⋀R_1(x,y,z)
x_1 y_1 z_1 y_2 z_2
x_1 | y_1 | z_1
x_1 | y_2 | z_1
x_2 | y_1 | z_1
R_0(x,y,z)
x_1 | y_1 | z_1

(a) A CMCQ (b) Instances (c) Result

Fig. 1: An example of a CMCQ.

input relations $|R_1|= |R_2| =|R_3| = N$. The worst-case size bound of the output table $|R_0|$ yields $\mathcal{O}(N^{\frac{3}{2}})$. But any pairwise relational algebra plan takes at least $\Omega(N^2)$, which is asymptotically worse than the optimal engines. To solve this problem, recent algorithms (e.g. *NPRR* [26], *LeapFrog* [31], *Joen* [8]) were discovered to achieve the optimal asymptotic bound for conjunctive queries.

Conjunctive queries over trees have recently attracted attention [13], as trees are a clean abstraction of HTML, XML, JSON, and LDAP. The tree structures in conjunctive queries are represented using node label relations and axis relations such as *Child* and *Descendant*. For example, the *XPath* query $A[B]//C$ is equivalent to the conjunctive query:

$$\begin{aligned} R(z) \leftarrow Label(x, \text{``}A\text{''}) \wedge Child(x,y) \wedge Label(y, \text{``}B\text{''}) \\ \wedge Descendant(x,z) \wedge Label(z, \text{``}C\text{''}). \end{aligned} \tag{2}$$

Conjunctive queries with trees have been studied extensively. For example, see [13] on their complexity, [3] on their expressive power, and [4,13] on the satisfiability problem. While conjunctive queries with relations or trees have been studied separately in the literature, hybrid conjunctive queries have gained less attention.

This paper embarks on the study of a Cross-Model Conjunctive Query (CMCQ) over both relations and trees. Figure 1 depicts a CMCQ. CMCQs emerge in modern data management and analysis, which often demands a hybrid evaluation with data organized in different formats and models, e.g. data lake [16], multi-model databases [22], polystores [9], and computational linguistics [32].

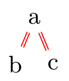
The number of applications that we have hinted at above motivates the study of CMCQs, and the main contributions of this paper are as follows:

1. This paper embarks on the study of the cross-model conjunctive query (CMCQ) and formally defines the problem of CMCQ processing, which integrates both relational conjunctive query and tree conjunctive pattern together.
2. We propose *CMJoin*-algorithm to process relations and encoded tree data efficiently. *CMJoin* produces worst-case optimal join result in terms of the label values as well as the encoded information values. In some cases, *CMJoin* is worst-case optimal join in the absence of encoded information.
3. Experiments on real-life and benchmark datasets show the effectiveness and efficiency of the algorithm in terms of running time and intermediate result size.

$$R_0(r_a,r_b,r_c) \longleftarrow \begin{array}{c} R_1(r_a,r_b) \bowtie R_2(r_a,r_c) \bowtie \\ \text{Descendant}(t_a,\, t_b) \bowtie \text{Descendant}(t_a,\, t_c) \bowtie \\ \text{Label}(t_a,r_a) \bowtie \text{Label}(t_b,r_b) \bowtie \text{Label}(t_c,r_c) \end{array}$$

(a) Conjunctive query form

$$R_0(a,b,c) \longleftarrow$$



$$\bowtie R_1(a,b) \bowtie R_2(a,c)$$

(b) Simplified expression

Fig. 2: Complete and simplified expressions of CMCQ

The remainder of the paper is organized as follows. In Section 2 we provide preliminaries of approaches. We then extend the worst-case optimal algorithm for CMCQs in Section 3. We evaluate our approaches empirically in Section 4. We review related works in Section 5. Section 6 concludes the paper.

2 Preliminary

Cross-model conjunctive query Let $\mathbb{R}$ be a database schema and $R_1, \ldots, R_n$ be relation names in $\mathbb{R}$. A rule-based conjunctive query over $\mathbb{R}$ is an expression of the form $R_0(u_0) \leftarrow R_1(u_1) \wedge R_2(u_2) \wedge \ldots \wedge R_n(u_n)$, where $n \geq 0$, R_0 is a relation not in $\mathbb{R}$. Let $u_0, u_1, \ldots, u_n$ be free tuples, i.e. they may be either variables or constants. Each variable occurring in u_0 must also occur at least once in $u_1, \ldots, u_n$.

Let T be a tree pattern with two binary axis relations: `Child` and `Descendant`. The axis relations `Child` and `Descendant` are defined in the normal way [13]. In general, a cross-model conjunctive query contains three components: (i) the relational expression $\tau_1 := \exists r_1, \ldots, r_k\colon R_1(u_1) \wedge R_2(u_2) \wedge \ldots \wedge R_n(u_n)$, where $r_1, \ldots, r_k$ are all the variables in the relations $R_1, \ldots, R_n$; (ii) the tree expression $\tau_2 := \exists t_1, \ldots, t_k\colon \mathit{Child}(v_1) \wedge \ldots \wedge \mathit{Descendant}(v_n)$, where $t_1, \ldots, t_k$ are all the node variables occurring in v_i, for $i \geq 1$ and each v_i is a binary tuple (t_{i_1}, t_{i_2}); and (iii) the cross-model label expression $\tau_3 := \exists r_1, \ldots, r_k, t_1, \ldots, t_k\colon \mathit{label}_1(t_{i_1}, r_{j_1}) \wedge \ldots \wedge \mathit{label}_n(t_{i_n}, r_{j_n})$, where Σ denotes a labeling alphabet. Given any node $t \in T$, $\mathit{label}(t, s)$ means that the label of the node t is $s \in \Sigma$. The label relations bridge the expressions of relations and trees by the equivalence between the label values of the tree nodes and the values of relations.

By combining the three components together, we define a cross-model conjunctive query with the calculus of form $\{e_1, \ldots, e_m \mid \tau_1 \wedge \tau_2 \wedge \tau_3\}$, where the variables $e_1, \ldots, e_m$ are the return elements which occur at least once in relations. Figure 2a shows an example of a cross-model conjunctive query, which includes two relations and one tree pattern. For the purpose of expression simplicity, we do not explicitly distinguish between the variable of trees (e.g. t_a) and that of relations (e.g. r_a), but simply write them with one symbol (i.e. a) if $\mathit{label}(t_a, r_a)$ holds. We omit the label relation when it is clear from the context. Figure 2b shows a simplified representation of a query.

Revisiting relational size bound We review the size bound for the relational model, which Asterias, Grohe, and Marx (AGM) [2] developed. The AGM bound is computed with linear programming (LP). Formally, given a relational schema $\mathbb{R}$, for every table $R \in \mathbb{R}$ let A_R be the set of attributes of R and $\mathcal{A} = \cup_R A_R$. Then the worst-case size bound is precisely the optimal solution for the following LP:

$$\begin{aligned} &\text{maximize} && \Sigma_r^{\mathcal{A}} x_r && \\ &\text{subject to} && \Sigma_r^{A_R} x_r \leq 1 && \text{for all } R \in \mathbb{R}, \\ & && 0 \leq x_r \leq 1 && \text{for all } r \in \mathcal{A}. \end{aligned} \tag{3}$$

Let ρ denote the optimal solution of the above LP. Then the size bound of the query is N^ρ, where N denotes the maximal size of each table. The AGM bound can be proved as a special case of the discrete version of the well-known Loomis-Whitney inequality [20] in geometry. Interested readers may refer to the details of the proof in [2]. We present these results informally and refer the readers to Ngo et al. [27] for a complete survey.

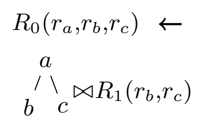
For example, we consider a typical triangle conjunctive query $R_0(a,b,c) \leftarrow R_1(a,b) \wedge R_2(b,c) \wedge R_3(a,c)$ that we introduced in Section 1. Then the three LP inequalities corresponding to three relations include $x_a + x_b \leq 1$, $x_b + x_c \leq 1$, and $x_a + x_c \leq 1$. Therefore, the maximal value of $x_a + x_b + x_c$ is $3/2$, meaning that the size bound is $\mathcal{O}(N^{\frac{3}{2}})$. Interestingly, the similar case for CMCQ in Figure 3, the query $Q = a[b]/c \bowtie R(b,c)$ has also the size bound $\mathcal{O}(N^{\frac{3}{2}})$.

3 Approach

In this section, we tackle the challenges in designing a worst-case optimal algorithm for CMCQs over relational and tree data. We briefly review the existing relational worst-case optimal join algorithms. We represent these results informally and refer the readers to Ngo et al. [27] for a complete survey. The first algorithm to have a running time matching these worst-case size bounds is the NPRR algorithm [26]. An important property in NPRR is to estimate the intermediate join size and avoid to produce the case which is larger than the worst-case bound. In fact, for any join query, its execution time can be upper bounded by the AGM [2]. Interestingly, *LeapFrog* [31] and *Joen* [8] completely abandon the "query plan" and propose to deal with one attribute at a time with multiple relations at the same time.

3.1 Tree and relational data representation

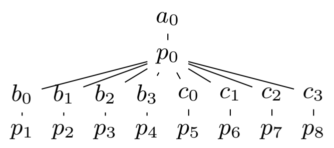
To answer a tree pattern query, a positional representation of occurrences of tree elements and string values in the tree database are widely used, which extends the classic inverted index data structure in information retrieval. There existed two common ways to encode an instance tree, i.e. Dewey encoding [23] and containment encoding [6]. These decodings are necessary as they allow us



(a) Tree query Q

r_b	r_c
b_0	c_0
b_0	c_1
b_1	c_0
b_1	c_1

(b) Table D_{R_1}


a_0
p_0
b_0 b_1 b_2 b_3 c_0 c_1 c_2 c_3
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8

(c) Encode Tree D_T

Fig. 3: A CMCQ (a) and its table instance (b) and tree instance (c).

to partially join tree patterns to avoid undesired intermediate result. After encoding, each attribute j in the query node can be represented as a **node table** in form of $t_j(r_j, p_j)$, where r_j and p_j are the label value and position value, respectively. Check an example from a encoded tree instance in Figure 3c. The position value can be added in $\mathcal{O}(N)$ by one scan of the original tree. Note that we use Dewey coding in our implementation but the following algorithm is not limited to such representation. Any representation scheme which captures the structure of trees such as a region encoding [6] and an extended Dewey encoding [23] can all be applied in the algorithm.

All the data in relational are label data, and all relation tables and node tables will be expressed by the Trie index structure, which is commonly applied in the relational worst-case optimal algorithms (e.g. [1,31]). The Trie structure can be accomplished using standard data structures (notably, balanced trees with $\mathcal{O}(\log n)$ look-up time or nested hashed tables with $\mathcal{O}(1)$ look-up time).

3.2 Challenges

In our context, tree data and twig pattern matching do make situation more complex. Firstly, directly materializing tree pattern matching may yield asymptotically more intermediate results. If we ignore the pattern, we may loose some bound constraints. Secondly, since tree data are representing both label and position values, position value joining may require more computation cost for pattern matching while we do not need position values in our final result.

Example 1. Recall that a triangle relational join query Q=$R_1(r_a, r_b) \bowtie R_2(r_a, r_c) \bowtie R_3(r_b, r_c)$ has size bound $\mathcal{O}(N^{\frac{3}{2}})$. Figure 3 depicts an example of a CMCQ Q with the table $R_1(r_b, r_c)$ and twig query $a[b]/c$ to return result $R_0(r_a, r_b, r_c)$, which also has size bound $\mathcal{O}(N^{\frac{3}{2}})$ since the PC paths a/b and a/c are equivalent to the constraints $x_a + x_b \leq 1$ and $x_a + x_c \leq 1$, respectively. Figure 3b and Figure 3c show the instance table D_{R_1} and the encoded tree D_T. The number of label values in the result $R_0(r_a, r_b, r_c)$ is only 4 rows which is $\mathcal{O}(N)$. On the other hand, the result size of only the tree pattern is 16 rows which is $\mathcal{O}(N^2)$, where N is a table size or a node size for each attribute. The final result with the position values is also $\mathcal{O}(N^2)$. Here, $\mathcal{O}(N^2)$ is from the matching result of the position values of the attributes t_b and t_c.

EmptyHeaded [1] applied the existing worst-case optimal algorithms to process the graph edge pattern matching. We may also attempt to solve relation-tree joins by representing the trees as relations with the node-position and the node-label tables and then reformulating the cross-model conjunctive query as a relational conjunctive query. However, as Example 1 illustrated, such method can not guarantee the worst-case optimality as extra computation is required for position value matching in a tree.

3.3 Cross-model join (CMJoin) algorithm

In this part, we discuss the algorithm to process both relational and tree data. As the position values are excluded in the result set while being required for the tree pattern matching, our algorithm carefully deals with it during the join. We propose an efficient cross-model join algorithm called *CMJoin* (cross-model join). In certain cases it guarantees the runtime optimality. We discover the join result size under three scenarios: with all node position values, with only branch node position value, and without position value.

Lemma 1. *Given relational tables $\mathcal{R}$ and pattern queries $\mathcal{T}$, let S_r, S_p, and S'_p be the sets of all relation attributes, all position attributes, and only branch node position attributes, respectively. Then it holds that*

$$\rho_1(S_r \cup S_p) \geq \rho_2(S_r \cup S'_p) \geq \rho_3(S_r). \tag{4}$$

Proof. $Q(S_r)$ is the projection result from $Q(S_r \cup S'_p)$ by removing all position values, and $Q(S_r \cup S'_p)$ is the projection result from $Q(S_r \cup S_p)$ by removing non-branch position values. Therefore, the result size holds $\rho_1(S_r \cup S_p) \geq \rho_2(S_r \cup S'_p) \geq \rho_3(S_r)$.

Example 2. Recall the CMCQ Q in Figure 3a, which is $Q{=}R_1(r_b, r_c) \bowtie a[b]/c$. Nodes a, b, c in the tree pattern can be represented as node tables (r_a, p_a), (r_b, p_b), and (r_c, p_c), respectively. So we have $Q(S_r \cup S_p) = R(r_a, r_b, r_c, p_a, p_b, p_c)$, $Q(S_r \cup S'_p) = R(r_a, r_b, r_c, p_a)$, and $Q(S_r) = R(r_a, r_b, r_c)$. By the LP constraint bound for the relations and PC-paths, we achieve $\mathcal{O}(N^2)$, $\mathcal{O}(N^{\frac{3}{2}})$, and $\mathcal{O}(N^{\frac{3}{2}})$ for the size bounds $\rho_1(S_r \cup S_p)$, $\rho_2(S_r \cup S'_p)$, and $\rho_3(S_r)$, respectively.

We elaborate *CMJoin* algorithm 1 more in the following. In the case of $\rho_1(S_r \cup S_p){=}\rho_3(S_r)$, *CMJoin* executes a generic relational worst-case optimal join algorithm [1,8] as the extra position values do not affect the worst-case final result. In other cases, *CMJoin* computes the path result of the tree pattern first. In this case, we project out all position values of a non-branch node for the query tree pattern. Then, we keep the position values of the only branch node so that we still can match the whole part of the tree pattern.

Theorem 1. *Assume we have relations $\mathcal{R}$ and pattern queries $\mathcal{T}$. If either*

(1) $\rho_1(S_r \cup S_p) \leq \rho_3(S_r)$ or

Algorithm 1: *CMJoin*

```
   Input: Relational tables $\mathcal{R}$, pattern queries $\mathcal{T}$
1  $\mathcal{R}' \leftarrow \emptyset$                                    // Tree intermediate result
2  if $\rho_1(S_r \cup S_p) \leq \rho_3(S_r)$ then // Theorem 1 condition (1)
3      foreach $N \in \mathcal{T}$ do
4          $\mathcal{R}' \leftarrow \mathcal{R}' \cup C_N(r_N, p_N)$                        // Nodes as tables
5  else
6      $\mathcal{P} \leftarrow \mathcal{T}.getPaths()$
7      foreach $P \in \mathcal{P}$ do
8          $R_P(S_r \cup S_p) \leftarrow$ path result of $P$                // Paths as tables
9          $R'_P(S_r \cup S'_p) \leftarrow$ project out non-branch position values of $R_P(S_r \cup S_p)$
10         $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R'_P(S_r \cup S'_p)\}$
11 $Q(S_r \cup S'_p) \leftarrow generic\_join(\mathcal{R} \cup \mathcal{R}')$
12 $Q(S_r) \leftarrow$ project out all position values $Q(S_r \cup S'_p)$
   Output: Join results $Q(S_r)$
```

(2) (i) $\rho_1(S_r \cup S'_p) \leq \rho_3(S_r)$ and (ii) for each path P in $\mathcal{T}$ let S''_r and S''_p be the set of label and position attributes for P so that $\rho_4(S''_r \cup S''_p) \leq \rho_3(S_r)$.

Then, CMJoin is worst-case optimal to $\rho_3(S_r)$.

Proof. (1) Since the join result of the only label value $\rho_3(S_r)$ is the projection of $\rho_1(S_r \cup S_p)$, we can compute $Q(S_r \cup S_p)$ first, then project out all the position value in linear of $\mathcal{O}(N^{\rho_1(S_r \cup S_p)})$. Since $\rho_1(S_r \cup S_p) \leq \rho_3(S_r)$, we can estimate that the result size is limited by $\mathcal{O}(N^{\rho_3(S_r)})$.

(2) $\rho_1(S''_r \cup S''_p) \leq \rho_3(S_r)$ means that each path result with label and position values are under worst-case result of $\rho_3(S_r)$. We may first compute the path result and then project out all the non-branch position values. The inequality $\rho_2(r, p') \leq \rho_3(r)$ means that the join result containing all branch position values has a worst-case result size which is still under $\rho_3(S_r)$. Then by considering those position values as relational attribute values and by a generic relation join [26,1], *CMJoin* is worst-case optimal to $\rho_3(S_r)$.

Example 3. Recall the CMCQ query $Q = R_1(r_b, r_c) \bowtie a[b]/c$ in Figure 3a. Since $\rho_1(S_r \cup S_p) > \rho_3(r)$, directly computing all label and position values may generate asymptotically bigger result ($\mathcal{O}(N^2)$ in this case). So we can compute path results of a/b and a/c, which are (r_a, p_a, r_b, p_b) and (r_a, p_a, r_c, p_c) and in $\mathcal{O}(N)$. Then we obtain only branch node results (r_a, p_a, r_b) and (r_a, p_a, r_c). By joining these project-out result with relation R_1 by a generic worst-case optimal algorithm, we can guarantee the size bound is $\mathcal{O}(N^{\frac{3}{2}})$.

4 Evaluation

In this section, we experimentally evaluate the performance of the proposed algorithms and *CMJoin* with four real-life and benchmark data sets. We com-

Table 1: Intermediate result size (10^6) and running time (S) for queries. "/" and "-" indicate "timeout" ($\geq$ 10 mins) and "out of memory". We measure the intermediate size by accumulating all intermediate and final join results.

	Intermediate result size(10^6)					Running time (second)				
Query	**PG**	**SJ**	**VJ**	**EH**	***CMJoin***	**PG**	**SJ**	**VJ**	**EH**	***CMJoin***
Q1	7.87x	2.60x	2.00x	1.68x	0.15	18.02x	1.39x	1.51x	1.66x	3.22
Q2	/	-	3.75x	4.83x	0.08	/	-	4.52x	129x	1.96
Q3	86.0x	62.6x	3.63x	4.61x	0.08	21.3x	4.27x	1.99x	4.28x	3.06
Q4	/	1.96x	1.75x	1.64x	0.24	/	2.34x	2.63x	1.82x	3.55
Q5	/	-	1.86x	1.77x	0.22	/	-	4.75x	39.8x	3.11
Q6	/	2.24x	2.00x	1.85x	0.21	/	6.10x	3.30x	2.89x	3.00
Q7	133x	106x	-	35.1x	0.29	4.82x	9.05x	-	7.18x	8.36
Q8	350x	279.8x	-	/	0.11	4.36x	5.61x	-	/	13.8
Q9	8.87x	8.34x	-	2.01x	4.62	1.12x	2.13x	-	1.48x	35.0
Q10	110x	440x	4.86x	/	0.07	2.91x	12.7x	1.22x	/	5.62
Q11	110x	440x	4.86x	/	0.07	2.11x	10.5x	0.88x	/	6.84
Q12	110x	440x	4.86x	/	0.07	2.68x	9.99x	1.06x	/	7.25
Q13	1.04x	1.22x	1.22x	1.07x	43.2	1.37x	4.81x	4.79x	1.31x	34.2
Q14	19.7x	2.56x	2.56x	3.90x	0.39	2.04x	3.82x	3.79x	2.14x	2.73
Q15	14.2x	1.85x	1.85x	17.0x	0.54	1.68x	3.53x	3.54x	2.01x	2.87
Q16	1.24x	1.24x	6.81x	2.15x	0.37	2.88x	1.32x	1.96x	1.02x	12.9
Q17	1.59x	7.84x	2.28x	1.31x	0.32	7.03x	3.58x	3.38x	2.10x	5.08
Q18	1.59x	7.13x	1.59x	1.64x	0.32	14.1x	5.21x	5.02x	2.06x	2.98
Q19	/	5.47x	6.62x	1.77x	0.45	/	1.41x	1.94x	0.89x	14.7
Q20	7.80x	25.1x	7.30x	4.19x	0.10	12.1x	6.77x	6.39x	3.17x	3.37
Q21	12.0x	36.1x	18.4x	14.7x	0.10	14.6x	8.82x	8.75x	10.9x	2.89
Q22	1.00x	18.5x	18.5x	0.96x	0.57	1.16x	5.22x	4.31x	2.35x	12.7
Q23	18.5x	18.5x	18.5x	1.61x	0.57	1.92x	2.17x	1.83x	1.02x	15.8
Q24	14.3x	3.02x	4.02x	0.96x	0.57	>9kx	>11kx	>12kx	0.18x	0.01
AVG	5.46x	5.90x	1.92x	1.90x	2.24	4.37x	5.34x	3.33x	3.46x	8.54

prehensively evaluate *CMJoin* against state-of-the-art systems and algorithms concerning efficiency, scalability, and intermediate cost.

4.1 Evaluation setup

Datasets and query design Table 2 provides the statistics of datasets and designed CMCQs. These diverse datasets differ from each other in terms of the tree structure, data skewness, data size, and data model varieties. Accordingly, we designed 24 CMCQs to evaluate the efficiency, scalability, and cost performance of the *CMJoin* in various real-world scenarios.

Comparison systems and algorithms *CMJoin* is compared with two types of state-of-the-art cross-model solutions. The first solution is to use one query to retrieve a result without changing the nature of models [25,34]. We implemented queries in PostgreSQL (*PG*), that supports cross-model joins. This enables the usage of the *PG*'s default query optimizer.

The second solution is to encode and retrieve tree nodes in a relational engine [5,29,35,1]. We implemented two algorithms, i.e. structure join (*SJ*) (pattern matching first, then matching the between values) and value join (*VJ*) (label value matching first, then matching the position values). Also, we compared to a worst-case optimal relational engine called EmptyHeaded (*EH*) [1].

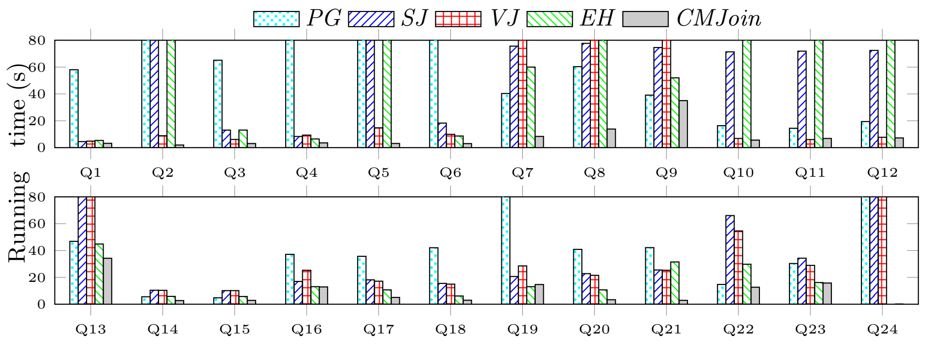


Fig. 4: **Efficiency:** runtime performance for all queries by *PG*, *SJ*, *VJ*, *EH*, and *CMJoin*. The performance time of more than 80s is cut for better presentation.

Experiment Setting We conducted all experiments on a 64-bit Windows machine with a 4-core Intel i7-4790 CPU at 3.6GHz, 16GB RAM, and 500GB HDD. We implemented all solutions, including *CMJoin* and the compared algorithms, in memory processing by Python 3. We measured the computation time of joining as the main metric excluding the time used for compilation, data loading, index presorting, and representation/index creation for all the systems and algorithms. We employed the Dewey encoding [23] in all experiments. The join order of attributes is greedily chosen based on the frequency of attributes. We measured the intermediate cost metric by accumulating all intermediate and final join results. For *PG* we accumulated all sub-query intermediate results. We repeated five experiments excluding the lowest and the highest measure and calculated the average of the results. Between each measurement of queries we wiped caches and re-loaded the data to avoid intermediate results.

Efficiency Figure 4 shows the evaluation of the efficiency. In general, *CMJoin* is 3.33-13.43 times faster in average than other solutions as shown in Table 1. These numbers are conservative as we exclude the "out of memory" (OOM) and "time out" (TO) results from the average calculation. Algorithms *SJ*, *VJ*, and *EH* perform relatively better compared to *PG* in the majority of the cases as they encoded the tree data into relation-like formats, making it faster to retrieve the tree nodes and match twig patterns.

Specifically in queries *Q*1-*Q*6, *CMJoin*, *SJ*, *VJ*, and *EH* perform better than *PG*, as the original tree is deeply recursive in the TreeBank dataset [32], and designed tree pattern queries are complex. So, it is costly to retrieve results directly from the original tree by *PG*. Instead, *CMJoin*, *SJ*, and *VJ* use encoded structural information to excel in retrieving nodes and matching tree patterns in such cases. In *Q*2 and *Q*5, *EH* performs worse. The reason is that it seeks for a better instance bound by joining partial tables and sub-twigs first and then aggregates the result. However, the separated joins yield more intermediate result in such cases in this dataset. In *Q*1 and *Q*4, which deal with a single table, *SJ* and *VJ* perform relatively close as no table joining occurs in these cases. However, in $Q2$-$Q3$ and $Q5$-$Q6$, *SJ* performs worse as joining two tables first leads to huge intermediate results in this dataset.

In contrast to the above, *SJ* outperforms *VJ* in $Q7$-$Q9$. The reason is that in the Xmark dataset [30], the tree data are flat and with less matching results in twig queries. The data in tables are also less skew. Therefore, *SJ* operates table joins and twig matching separately yielding relatively low results. Instead, *VJ* considers tree pattern matching later yielding too many intermediate results (see details of $Q7$ in Figure 5 and Figure 6) when joining label values between two models with non-uniform data. *PG*, which implements queries in a similar way of *SJ*, performs satisfactorily as well. The above comparisons show that compared solutions, which can achieve superiority only in some cases, and can not adapt well to dataset dynamics.

Queries $Q10$-$Q12$ have more complex tree pattern nodes involved. In these cases *VJ* filters more values and produces less intermediate results. Thus it outperforms *SJ* ($\sim$10x) and *PG* ($\sim$2x). For queries $Q10$-$Q12$ *EH* also yields huge intermediate results with more connections in attributes. The comparison between $Q7$-$Q9$ and $Q10$-$Q12$ indicates that the solutions can not adapt well to query dynamics.

Considering queries $Q13$-$Q15$ and $Q22$-$Q24$, *PG* performs relatively well since it involves only JSON and relational data. *PG* performs well in JSON retrieving because JSON documents have a simple structure. In $Q14$-$Q15$ most of the solutions perform reasonably well when the result size is small but *SJ*, and *VJ* still suffer from a large result size in $Q13$. With only JSON data, *SJ* and *VJ* perform similarly, as they both treat a simple JSON tree as one relation. In contrast in $Q16$-$Q21$, it involves XML, JSON and relational data from the UniBench dataset [34]. *CMJoin*, *SJ*, *VJ*, and *EH* perform better than *PG*. This is again because employing the encoding technique in trees accelerates node retrieval and matching tree patterns. Also, *CMJoin*, *SJ*, *VJ*, and *EH* are able to treat all the data models together instead of achieving results separately from each model by queries in *PG*.

Though compared systems and algorithms possess their advantages of processing and matching data, they straightforwardly join without bounding intermediate results, thus achieving sub-optimal performance during joining. *CMJoin* is the clear winner against other solutions, as it can wisely join between models and between data to avoid unnecessary quadratic intermediate results.

Scalability Figure 5 shows the scalability evaluation. In most queries, *CMJoin* performs flatter scaling as data size increases because *CMJoin* is designed to control the unnecessary intermediate output.

As discussed, *CMJoin*, *SJ*, *VJ*, and *EH* outperform *PG* in most of the queries, as the encoding method of the algorithms speeds up the twig pattern matching especially when the documents or queries are complex. However, *PG* scales better when involving simpler documents (e.g. in $Q15$ and $Q23$) or simpler queries (e.g. in $Q7$). Comparing to processing XML tree pattern queries, *PG* processes JSON data more efficiently.

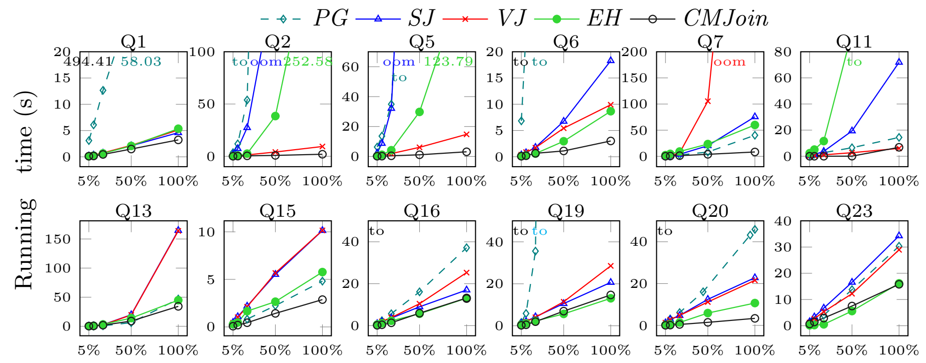


Fig. 5: **Scalability:** runtime performance of *PG*, *SJ*, *VJ*, *EH*, and *CMJoin*. The x-axis is the percentage of data size. "oom" and "to" stand for "out of memory" error and "timeout" ($\geq$ 10 mins), respectively.

Interestingly in $Q2$, *SJ* and *PG* join two relational tables separately from twig matching, generating quadratic intermediate results, thus leading to the OOM and TO, respectively. In $Q7$ *VJ* joins tables with node values without considering tree pattern structural matching and outputs an unwanted non-linear increase of intermediate results, thus leading to OOM in larger data size. Likewise evaluating *EH* between $Q2$ and $Q7$, it can not adapt well with different datasets. Performing differently in diverse datasets between *SJ*/*PG* and *VJ*/*EH* indicates that they can not smartly adapt to dataset dynamics. While increasing twig queries in $Q11$ compared to $Q7$, *VJ* filters more results and thus decreases the join cost and time in $Q11$. The comparison between *SJ*/*EH* and *VJ* shows dramatically different performance in the same dataset with different queries that indicates they can not smartly adapt to query dynamics.

In $Q11$, both *CMJoin* and *VJ* perform efficiently as they can filter out most of the values at the beginning. In this case, *CMJoin* runs slightly slower than *VJ*, which is reasonable as *CMJoin* maintains a tree structure whereas *VJ* keeps only tuple results. Overall, *CMJoin* judiciously joins between models and controls unwanted massive intermediate results. The evaluation shows that it performs efficiently and stably in dynamical datasets, with various queries and it also scales well.

Cost analysis Table 1 presents the intermediate result sizes showing that *CMJoin* outputs 5.46x, 5.90x, 1.92x, and 1.90x less intermediate results on average than *PG*, *SJ*, *VJ*, and *EH*, respectively. Figure 6 depicts more detailed intermediate results for each joining step. In general, *CMJoin* generates less intermediate results due to its designed algorithmic process, worst-case optimality, as well as join order selections. In contrast, *PG*, *SJ*, and *VJ* can easily yield too many (often quadratic) intermediate results during joining in different datasets or queries. This is because they have no technique to avoid undesired massive intermediate results.

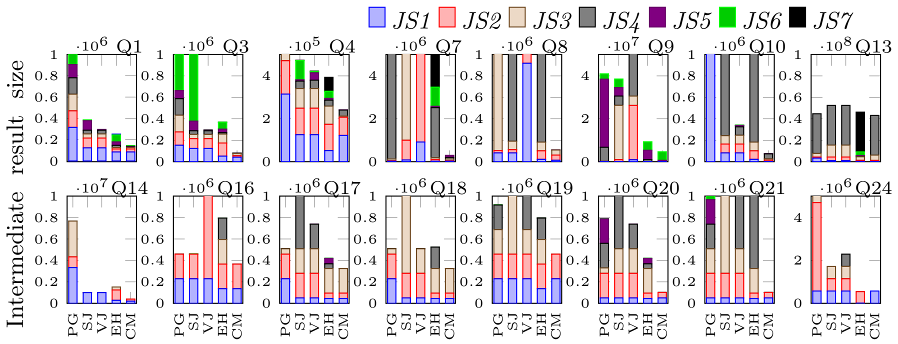


Fig. 6: **Cost:** intermediate result size. CM: *CMJoin*. JS: Joining step.

PG and *SJ* suffer when the twig matching becomes complex in datasets (e.g. *Q*3 and *Q*10), while *VJ* suffers in the opposite case of simpler twig pattern matching (e.g. *Q*7 and *Q*16). More specifically in *Q*3, *PG* and *SJ* output significant intermediate results by joining of two relational tables. In turn, *VJ* controls intermediate results utilizing the values of common attributes and tags between two models. On the other hand, in *Q*7, *Q*9, and *Q*16 *VJ* does not consider structural matching at first yielding unnecessary quadratic intermediate results. The above two-side examples indicate that solutions considering only one model at a time or joining values first without twig matching produce an undesired significant intermediate result.

EH suffers when the queries and attributes are more connected that leads to larger intermediate results during join procedures. The reason is that *EH* seeks a better instance bound so that it follows the query plan based on the GHD decomposition [1]. Our proposed method, *CMJoin*, by wisely joining between models, avoids an unnecessary massive intermediate output from un-joined attributes.

Summary We summarize evaluations of *CMJoin* as follows:

1. Extensive experiments on diverse datasets and queries show that averagely *CMJoin* achieves up to 13.43x faster runtime performance and produces up to 5.46x less intermediate results compared to other solutions.
2. With skew data *CMJoin* avoids undesired huge intermediate results by wisely joining data between models. With uniform data *CMJoin* filters out more values by joining one attribute at a time between all models.
3. With more tables, twigs, or common attributes involved *CMJoin* seems to perform more efficiently and scale better.

5 Related work

Worst-case size bounds and optimal algorithms Recently, Grohe and Marx [15] and Atserias, Grohe, and Marx [2] estimated size bounds for conjunctive joins using the fractional edge cover. That allows us to compute the worst-case size bound by linear programming. Based on this worst-case bound, several worst-case optimal algorithms have been proposed (e.g. *NPRR* [26], *LeapFrog* [31], *Joen* [8]). Ngo et al. [26] constructed the first algorithm whose running time is worst-case optimal for all natural join queries. Veldhuizen [31] proposed an optimal algorithm called *LeapFrog* which is efficient in practice to implement. Ciucanu et al. [8] proposed an optimal algorithm *Joen* which joins each attribute at a time via an improved tree representation. Besides, there exist research works on applying functional dependencies (FDs) for size bound estimation. The initiated study with FDs is from Gottlob, Lee, Valiant, and Valiant (GLVV) [14], which introduces an upper bound called GLVV-bound based on a solution of a linear program on polymatroids. The follow-up study by Gogacz et al. [11] provided a precise characterization of the worst-case bound with information theoretic entropy. Khamis et al. [19] provided a worst-case optimal algorithm for any query where the GLVV-bound is tight. See an excellent survey on the development of worst-case bound theory [27].

Multi-model data management As more businesses realized that data, in all forms and sizes, are critical to make the best possible decisions, we see a continuing growth of demands to manage and process massive volumes of different types of data [21]. The data are represented in various models and formats: structured, semi-structured, and unstructured. A traditional database typically handles only one data model. It is promising to develop a multi-model database to manage and process multiple data models against a single unified backend while meeting the increasing requirements for scalability and performance [21,24]. Yet, it is challenging to process and optimize cross-model queries.

Previous work applied naive or no optimizations on (relational and tree) CMCQs. There exist two kinds of solutions. The first is to use one query to retrieve the result from the system without changing the nature of the model [25,34]. The second is to encode and retrieve the tree data into a relational engine [1,5,29,35]. Even though the second solution accelerates twig matching, they both may suffer from generating large, unnecessary intermediate results. These solutions or optimizations did not consider cross-model worst-case optimality. Some advances are already in development to process graph patterns [28,17,1]. In contrast to previous work, this paper initiates the study on the worst-case bound for cross-model conjunctive queries with both relation and tree structure data.

Join order In this paper, we do not focus on more complex query plan optimization. A better query plan [12,10] may lead to a better bound for some instances [1] by combining the worst-case optimal algorithm and non-cyclic join optimal algorithm (i.e. Yannakakis [33]). We leave this as the future work to continue optimizing CMCQs.

6 Conclusion and future work

In this paper, we studied the problems to find the worst-case size bound and optimal algorithm for cross-model conjunctive queries with relation and tree structured data. We provide the optimized algorithm, i.e. CMJoin, to compute the

Table 2: Dataset statistics and designed queries (m=10^6, k=10^3).

Dataset	**Statistics**	**Query**	**Relational table**	**XML or JSON path query**	**LP**	**#Result**
D1:TreeBank[32] (Linguistic data)	Zipfian Tables: 1m rows XML: 2.4m nodes	Q1	R1(NP,VP)	S[NP]/VP//PP[IN]//NNP	N^3	7.6k
		Q2	R1(NP,VP) R2(NP,PP)		N^3	4.6k
		Q3	R1(NP,VP) R3(NP,NNP)		N^3	<0.1k
		Q4	R1(NP,VP)	S[NP]/VP//PP[IN]//NNP	N^3	1.4k
		Q5	R1(NP,VP) R2(NP,PP)	S/VP/PP/IN	N^2	0.8k
		Q6	R1(NP,VP) R3(NP,NNP)		N^3	<0.1k
D2:Xmark[30] (Auction data)	Normal Tables: 1m rows XML: 1.6m nodes	Q7	R1(incategory,quantity,email) R2(item,incategory,email) R3(item,quantity,email)	T7=Item[incategory]/quantity	N^2	91k
		Q8		T8=Item[incategory][localtion][quantity]//email	N^3	0.4k
		Q9		T9=Item[location]//email	N^3	2.4m
		Q10		T7, T8	N^3	0.7k
		Q11		T7, T9	N^3	0.7k
		Q12		T7, T8, T9	N^3	0.7k
D3:UniBench[34] (E-commerce)	Uniform Tables: 1m rows JSON: 2m-4m nodes	Q13	R1(asin,productID,orderID)	$.[orderID,personID]	N^3	37.0m
		Q14	R2(personID,lastname)	$.[orderID,personID,orderline[productID]]	N^3	<0.1k
		Q15	R3(productID,product_info)	$.[personID,orderline[productID, asin]]	N^3	0.1k
D3:UniBench[34] (E-commerce)	Uniform Tables: 1m rows JSON: 4m nodes XML: 1.4m nodes	Q16	R1(asin,orderID) R2(personID,lastname) $.[orderID,personID,orderline[asin]]	OrderLine[asin]/price	N^3	1.1k
		Q17		T17=Invoice[orderID]/orderline[asin]/price	N^3	<0.1k
		Q18		T18=Invoice[orderID]//asin	N^3	<0.1k
		Q19		orderline/asin, orderline/price	N^3	1.1k
		Q20		Invoice(I)/orderID, I/orderline(O)/asin, I/O/price	N^3	<0.1k
		Q21		T17, T18	N^3	<0.1k
D4:MIMIC-III[18] (Clinical data)	Uniform Tables:0.5-10m rows JSON: 10m nodes	Q22	R1(RowID,ICUstayID,ItemID,CGID), R2(RowID,SubjectID,ICUstayID,ItemID)	T22=$.[RowID,SubjectID,HADMID]	N	<0.1k
		Q23	R1,R3(SubjectID,ItemID)	T22	$N^{\frac{3}{2}}$	<0.1k
		Q24	R1,R2,R4(RowID,SubjectID,HADMID)	T22, T23=$.[RowID,ICUstayID,ItemID,CGID], T24=$.[RowID,SubjectID,HADMID,ICUstayID,ItemID,CGID]	N	<0.1k

worst-case bound and the worst-case optimal algorithm for cross-model joins. Our experimental results demonstrate the superiority of proposal algorithms against state-of-the-art systems and algorithms in terms of efficiency, scalability, and intermediate cost. Exciting follow-ups will focus on adding graph structured data into our problem setting and designing a more general cross-model algorithm involving three data models, i.e. relation, tree and graph.

7 Acknowledgment

This paper is partially supported by Finnish Academy Project 310321 and Oracle ERO gift funding.

References

1. Aberger, C.R., Tu, S., Olukotun, K., Ré, C.: Emptyheaded: A relational engine for graph processing. In: SIGMOD Conference. pp. 431–446. ACM (2016)
2. Atserias, A., Grohe, M., Marx, D.: Size bounds and query plans for relational joins. In: FOCS. pp. 739–748. IEEE Computer Society (2008)
3. Benedikt, M., Fan, W., Kuper, G.: Structural properties of xpath fragments. Theoretical Computer Science **336**(1), 3 – 31 (2005), database Theory
4. Björklund, H., Martens, W., Schwentick, T.: Conjunctive query containment over trees. In: Proceedings of the 11th International Conference on Database Programming Languages. DBPL'07 (2007)
5. Bousalem, Z., Cherti, I.: Xmap: A novel approach to store and retrieve XML document in relational databases. JSW **10**(12), 1389–1401 (2015)
6. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD Conference. pp. 310–321. ACM (2002)
7. Chaudhuri, S., Vardi, M.Y.: On the equivalence of recursive and nonrecursive datalog programs. In: PODS. pp. 55–66. ACM Press (1992)
8. Ciucanu, R., Olteanu, D.: Worst-case optimal join at a time. Tech. rep., Technical report, Oxford (2015)
9. Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S.B.: The bigdawg polystore system. SIGMOD Record **44**(2), 11–16 (2015)
10. Fischl, W., Gottlob, G., Pichler, R.: General and fractional hypertree decompositions: Hard and easy cases. In: PODS. pp. 17–32. ACM (2018)
11. Gogacz, T., Toruńczyk, S.: Entropy bounds for conjunctive queries with functional dependencies. arXiv preprint arXiv:1512.01808 (2015)
12. Gottlob, G., Grohe, M., Musliu, N., Samer, M., Scarcello, F.: Hypertree decompositions: Structure, algorithms, and applications. In: International Workshop on Graph-Theoretic Concepts in Computer Science. pp. 1–15. Springer (2005)
13. Gottlob, G., Koch, C., Schulz, K.U.: Conjunctive queries over trees. J. ACM **53**(2), 238–272 (2006)
14. Gottlob, G., Lee, S.T., Valiant, G., Valiant, P.: Size and treewidth bounds for conjunctive queries. J. ACM **59**(3), 16:1–16:35 (2012)
15. Grohe, M., Marx, D.: Constraint solving via fractional edge covers. ACM Trans. Algorithms **11**(1) (Aug 2014). https://doi.org/10.1145/2636918, `https://doi.org/10.1145/2636918`

16. Hai, R., Geisler, S., Quix, C.: Constance: An intelligent data lake system. In: SIGMOD Conference. pp. 2097–2100. ACM (2016)
17. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: ISWC (1). Lecture Notes in Computer Science, vol. 11778, pp. 258–275. Springer (2019)
18. Johnson, A.E., Pollard, T.J., Shen, L., Li-wei, H.L., Feng, M., Ghassemi, M., Moody, B., Szolovits, P., Celi, L.A., Mark, R.G.: Mimic-iii, a freely accessible critical care database. Scientific data **3**, 160035 (2016)
19. Khamis, M.A., Ngo, H.Q., Suciu, D.: Computing join queries with functional dependencies. In: PODS. pp. 327–342. ACM (2016)
20. Loomis, L.H., Whitney, H.: An inequality related to the isoperimetric inequality. Bulletin of the American Mathematical Society **55**(10), 961–962 (1949)
21. Lu, J., Holubová, I.: Multi-model data management: What's new and what's next? In: EDBT. pp. 602–605. OpenProceedings.org (2017)
22. Lu, J., Holubová, I.: Multi-model databases: A new journey to handle the variety of data. ACM Comput. Surv. **52**(3), 55:1–55:38 (Jun 2019)
23. Lu, J., Ling, T.W., Chan, C.Y., Chen, T.: From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In: VLDB. pp. 193–204. ACM (2005)
24. Lu, J., Liu, Z.H., Xu, P., Zhang, C.: UDBMS: road to unification for multi-model data management. CoRR **abs/1612.08050** (2016)
25. Nassiri, H., Machkour, M., Hachimi, M.: One query to retrieve XML and relational data. In: FNC/MobiSPC. Procedia Computer Science, vol. 134, pp. 340–345. Elsevier (2018)
26. Ngo, H.Q., Porat, E., Ré, C., Rudra, A.: Worst-case optimal join algorithms. J. ACM **65**(3), 16:1–16:40 (Mar 2018)
27. Ngo, H.Q., Ré, C., Rudra, A.: Skew strikes back: new developments in the theory of join algorithms. SIGMOD Record **42**(4), 5–16 (2013)
28. Nguyen, D.T., Aref, M., Bravenboer, M., Kollias, G., Ngo, H.Q., Ré, C., Rudra, A.: Join processing for graph patterns: An old dog with new tricks. In: GRADES@SIGMOD/PODS. pp. 2:1–2:8. ACM (2015)
29. Qtaish, A., Ahmad, K.: Xancestor: An efficient mapping approach for storing and querying XML documents in relational database using path-based technique. Knowl.-Based Syst. **114**, 167–192 (2016)
30. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: Xmark: A benchmark for XML data management. In: VLDB. pp. 974–985. Morgan Kaufmann (2002)
31. Veldhuizen, T.L.: Leapfrog triejoin: A simple, worst-case optimal join algorithm. arXiv preprint arXiv:1210.0481 (2012)
32. Xue, N., Xia, F., Chiou, F.D., Palmer, M.: The penn chinese treebank: Phrase structure annotation of a large corpus. Natural language engineering **11**(2), 207–238 (2005)
33. Yannakakis, M.: Algorithms for acyclic database schemes. In: VLDB. pp. 82–94. IEEE Computer Society (1981)
34. Zhang, C., Lu, J., Xu, P., Chen, Y.: Unibench: A benchmark for multi-model database management systems. In: Technology Conference on Performance Evaluation and Benchmarking. pp. 7–23. Springer (2018)
35. Zhu, H., Yu, H., Fan, G., Sun, H.: Mini-xml: An efficient mapping approach between XML and relational database. In: ICIS. pp. 839–843. IEEE Computer Society (2017)